# EFSM-based Test Suite Generation for MC/DC Compliant Systems: Tool Design

**Amine Rahj**

**A Thesis**

**in**

**The Department**

**of**

**Concordia Institute for Information Systems Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Master of Applied Science in Quality Systems Engineering**

**Concordia University**

**Montreal, Quebec, Canada**

**January**
**2023**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Amine Rahj**

Entitled: **EFSM-based Test Suite Generation for MC/DC Compliant Systems: Tool Design**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science in Quality Systems Engineering**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
*Dr. Amr Youssef*

______________________________ Examiner
*Dr.* Nizar Bouguila

______________________________ Examiner
*Dr. Amr Youssef*

______________________________ Supervisor
*Dr. Rachida Dssouli*

______________________________ Co-supervisor
*Dr. Jamal Bentahar*

Approved by ______________________________
Dr. Zachary Patterson Graduate Program Director

____________ 2023 ______________________________

Dr. Mourad Debbabi Dean of Gina Cody School of Engineering and Computer Science

# Abstract

EFSM-based Test Suite Generation for MC/DC Compliant Systems: Tool Design

Amine Rahj

As Model-based Testing (MBT) approaches mature, they become a promising prospect for Safety-critical software systems testing. It is necessary to abide by RTCA DO-178C regarding requirement coverage, structural coverage, and traceability. The satisfaction of Modified Condition/Decision Coverage (MC/DC) is a must for avionics software certification. This thesis proposes a tool design for a test generation approach that satisfies the Modified Condition/Decision Coverage (MC/DC) and addresses path feasibility issues using constraints solving. The proposed methodology has several steps. It starts by transforming Low-Level Requirements (LLR), modelled as Extended Finite State Machines (EFSM), into a data-flow graph and a control-flow graph. Then, we highlight MC/DC information on both graphs, using graph labelling, before applying SMT-constraint solving to generate an executable test suite. Throughout, we keep records of the transformation between the models to prepare for requirements traceability as per RTCA DO-178C. The approach is based on the EFSM model, meaning that the assessment of MC/DC and other structural coverage criteria are on the model under the assumptions that the predicates are the same in the code and the model, and the model is valid.

# Acknowledgments

I would like to express my deepest gratitude to my thesis supervisors, Dr. Rachida Dssouli and Dr. Jamal Bentahar, for their guidance, encouragement, support and help throughout the course of my master's thesis. Their knowledge, expertise and dedication have been a constant source of inspiration for me.

I would also like to extend my appreciation to the members of my thesis committee, Dr. Amr Youssef and Dr. Nizar Bouguila, for their valuable feedback and suggestions.

I would like to thank my family, friends, and colleagues for their unwavering support and encouragement throughout my studies.

I am grateful to Concordia University for providing me with the opportunity to conduct this research, and for the facilities and resources that have been made available to me during the course of my studies.

Finally, I would like to acknowledge NSERC, CMC Electronics Inc., CRIAQ, and CS Canada for funding this research under the Collaborative Research and Development Grant – Project (CRDPJ 463076 - 14) entitled "Specification and verification of design models for certifiable avionics software".

Thank you all for being a part of this journey.

# Table of Content

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| CC | Condition Coverage |
| CEFSM | Communicating Extended Finite State Machine |
| CFG | Control Flow Graph |
| CSP | Constraint Satisfaction Problem |
| DC | Decision Coverage |
| DFG | Data Flow Graph |
| EFSM | Extended Finite State Machine |
| FIFO | First In First Out |
| HLR | High Level Requirement |
| IDAL | Item Development Assurance Level |
| IUT | Implementation Under Test |
| IEEE | Institute of Electrical and Electronics Engineers |
| LLR | Low Level Requirement |
| MBT | Model-Based Testing |
| MCC | Multiple condition coverage |
| MDD | Model-Driven Development |
| MC/DC | Modified Condition / Decision Coverage |
| RTCA | Radio Technical Commission for Aeronautics |
| SMT | Satisfiability Modulo Theory |
| SPO | Sigle Pushout |
| TGT | Test Generation Tool |
| UIO | Unique Input Output |
| UML | Unified Model Language |

# Chapter 2

# Introduction

The Institute of Electrical and Electronics Engineers (IEEE) defines safety-critical software as “software whose use in a system can result in unacceptable risk. Safety-critical software includes software whose operation or failure to operate can lead to a hazardous state, software intended to recover from hazardous states, and software intended to mitigate the severity of an accident” [IEEE Standard Glossary of Software Engineering Terminology, IEEE Std-610-1990, Los Alamitos, CA: IEEE Computer Society Press, 1990). This thesis is concerned with testing such software in avionics, namely from the perspective of RTCA DO-178C [1]. In fact, in the avionics industry, software products must be certified according to the RTCA standards [1]. The RTCA DO-178C standard promotes requirement-based testing and provides guidance on addressing safety in software development, i.e. tests are written/generated and executed to prove requirements are fulfilled, and safety concerns are addressed. From a testing viewpoint, there are two requirement types - high-level requirements (HLR) and low-level requirements (LLR), and five safety levels (A, B, D, C, and E), also known as Item Development Assurance Level (IDAL) as per APA4754 [2] - where Level A software are software items in which failure is deemed catastrophic, and Level E software have no effect on safety. DO178 summarizes the objectives of test case development and execution as follows:

- Objective 1: “Executable Object Code complies with high-level requirements”.
- Objective 2: “Executable Object Code is robust with high-level requirements”.
- Objective 3: “Executable Object Code complies with low-level requirements”.
- Objective 4: “Executable Object Code is robust with low-level requirements”.

- Objective 5: “Executable Object Code is compatible with target computer”.

DO-178C defines a *test case* as “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement” and promotes the development of two types of test cases to comply with aforementioned objectives. Two test cases are distinguished: normal test cases and robustness test cases. Normal test cases aim to find errors in the software under normal conditions and/or receiving expected inputs and robustness test cases show how the software behaves under abnormal conditions and/or receiving unexpected inputs. The type of testing required by DO-178C depends on the IDAL. For instance, for levels A and B, normal LLR are required with independence, while they are not required for the other levels. DO-178C uses the term independence to refers to “separation of responsibilities which ensures the accomplishment of objective evaluation”. For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool may be used to achieve equivalence to the human verification activity. For the software quality assurance process, independence includes the authority to ensure corrective action.

Testing is required to validate avionics software systems. Testing is a labor intensive and an expensive activity. It can count for more than 50% of the development cost [3,4]. The avionics industry is looking for ways to reduce the cost of testing and improve the effectiveness of tests by automating the testing process. Despite the availability of commercial tools, there are categories of complex systems that do not benefit from the current level of automation. The testing process includes several activities. The most challenging one is test case generation. This activity requires adequate coverage criteria that can show the effectiveness and the efficiency of the derived tests and guide the test case generation. The proposed research in this thesis looks at the automatic generation of LLR test cases for level A software. The thesis introduces a Model-Based Testing (MBT) approach, presents a tool design for this approach and shows how our MBT approach addresses testing requirements from RTCA DO-178C for Level A software. Moreover, we integrate model-based verification and MBT within one framework. We present thus a new methodology for the verification and testing of parallel communicating agents based on formal models. Properties are extracted from requirements and formally verified at the design level, while the verified properties are propagated to the implementation level via testing. DO-331 [1] defines a model as “An abstract representation of a given set of aspects of a system that is used for analysis, verification, simulation, code generation, or any combination thereof. A model should be unambiguous, regardless of its level of abstraction”. Our MBT approach requires the selection of a modelling scheme, deciding on test generation criteria, and designing a test generation approach. First, we present a model-based test generation technique guided by Modified Condition / Decision Coverage (MC/DC) and du-path coverage criteria. Du-path with respect to a graph $x$ is a simple path where the initial node of the path is the only defining node of $x$ in the path. The proposed approach combines two coverage criteria to improve the efficiency of the derived tests and to enhance their error detection power. The criteria are selected to satisfy the industrial needs for avionics software certification. The MC/DC criterion is required by the RTCA DO-178 C standard [1]. Second, we design of an automated Test Generation Tool (TGT) for the proposed approach.

# Chapter 3

# Background Information and Related Work

## 2.1 Research Context

This research work is part of an industrial research project funded by the Collaborative Research and Development Grant – Project (CRDPJ 463076 - 14), by NSERC, CMC Electronics Inc., CRIAQ, CS Canada. It is entitled "Specification and Verification of Design Models for Certifiable Avionics Software (see Figure 2.1). This project addresses the issues of specification, testing based on models and verification of avionics software to produce some artifacts that can be used for software certification in compliance with RTCA DO 187C standards [1].

The stated objectives of the project are to:

1. Specify, develop, and verify software using design models;

2. Enable verification techniques in the context of Model Driven Development (MDD);

3. Develop Low Level Requirement (LLR) testing techniques in conformity with avionics software standards RTCA DO-178C [1]; and

4. Enable low level requirement-based automatic test sequence generation.

The project is composed of several activities. The first activity is concerned with the requirements specification for avionics systems with UML and SIMULINK. The focus is to develop new UML profiles and Simulink Design standards. The second activity uses models' transformation to perform testing and verification. All specifications should lead to an output or a pivot that is in this project an Extended Finite State Machine (EFSM). The third activity transforms the EFSM and the communicating EFSMs (CEFSM) into specific models or graphs that can be used for verification and testing. As an example, the obtained model for testing can be transformed into a control flow graph and data flow graph that will be used in automatic test case generation for MC/DC. The transformation of an EFSM to a graph is done using the graph rewriting technique.

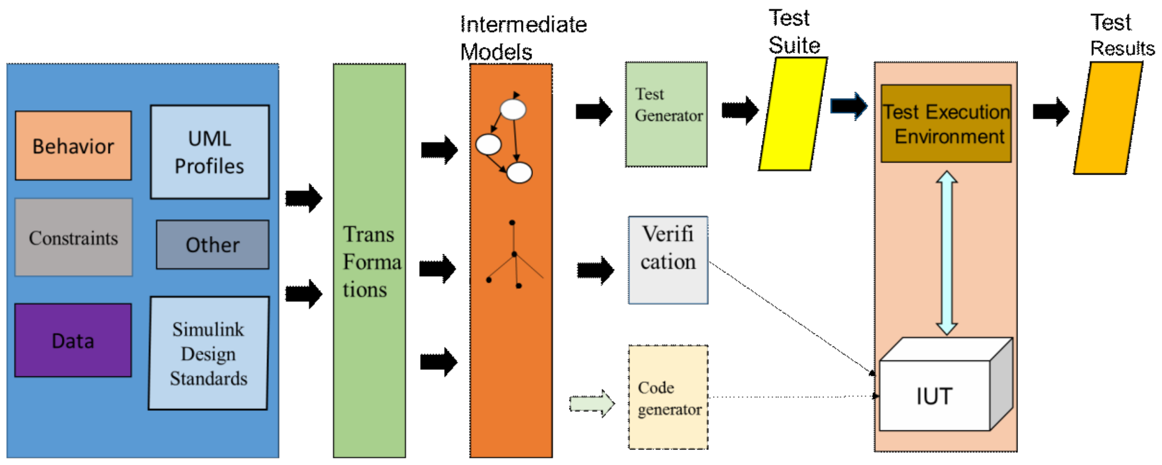


Figure 2.1: General overview of the Research Project

## 2.2 Background Information

### 2.2.1 RTCAD O178C

The avionics industry has developed a set of standards to prevent catastrophic events from occurring in their systems. The RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification is the main stands document by which the certification authorities approve all commercial software-based aerospace systems [1]. The RTCA DO 178C was approved in 2011 and replaced the previous document Do 178B. The document is published by RTCA. It became available for sale and use in January 2012. The DO 178C has more clarifications in comparison with Do178B. It also has 3 supplement documents: 1) DO-

331 "Model-Based Development and Verification" that addresses Model-Based Development (MBD) and verification and the capacity to use modeling techniques to improve development and verification in the development cycle and minimizes the pitfalls of using models [2]; 2) DO-332 "Object-Oriented Technology and Related Techniques" that addresses object-oriented software and the conditions under which it may be used. And 3) DO-333 "Formal Methods" addresses the use of formal methods to complement (but not replace) testing.

Figure 2.2 shows the testing in RTCA DO 178C. Our work is related to the last 3 objectives of RTCA DO 178C that are listed below, where * means that the objectives are partially or not met.

The executable code complies with the high-level requirements.

The executable code complies with the specification (low-level requirements).

Test coverage of high-level requirements is achieved. *

Test coverage of specification (low-level requirements) is achieved. *

Test coverage of the executable code is achieved. *

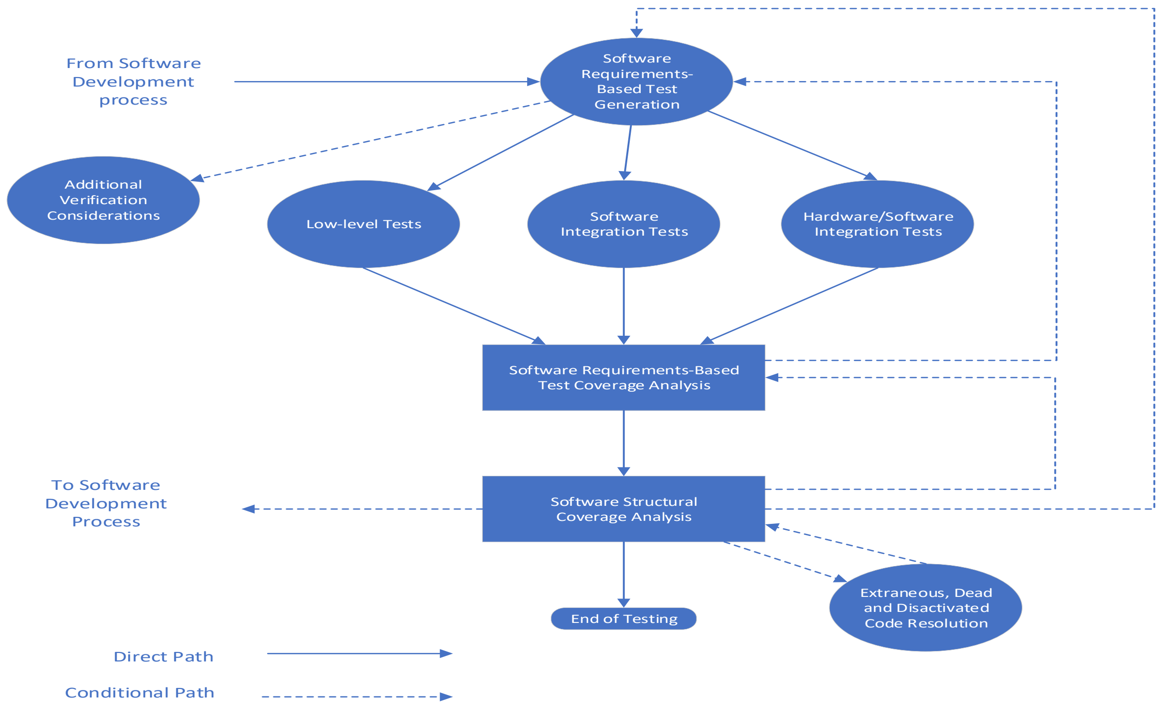

From Software Development process
Software Requirements-Based Test Generation
Additional Verification Considerations
Low-level Tests
Software Integration Tests
Hardware/Software Integration Tests
Software Requirements-Based Test Coverage Analysis
To Software Development Process
Software Structural Coverage Analysis
End of Testing
Extraneous, Dead and Disactivated Code Resolution
Direct Path
Conditional Path

Figure 2.2: Testing in DO-178C [1]

The defined Software Levels by ARP4754 [2], also known as Item Development Assurance Levels (IDAL), are mentioned in RTCA DO-178C and are from the safety assessment process and hazard analysis that examines the effects of a failure condition in the system. The failure conditions are categorized by their effects on the aircraft, crew, and passengers and are quoted as follows:

A- Catastrophic - Failure may cause deaths, usually with loss of the airplane.

B- Hazardous - Failure has a large negative impact on safety, performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload or causes serious or fatal injuries among the passengers.

C- Major - Failure significantly reduces the safety margin or significantly increases crew workload. May result in passenger discomfort (or even minor injuries).

D- Minor - Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change.

E- No Effect - Failure has no impact on safety, aircraft operation, or crew workload.

Figure 2.3 illustrates the required tracing between certification artifacts, as required by the RTCA DO-178C standard. Red-colored traces are required only for level A. Purple-colored traces are required for levels A, B, and C. Green-colored traces are for levels A, B, C, and D. Our work is related to level A.

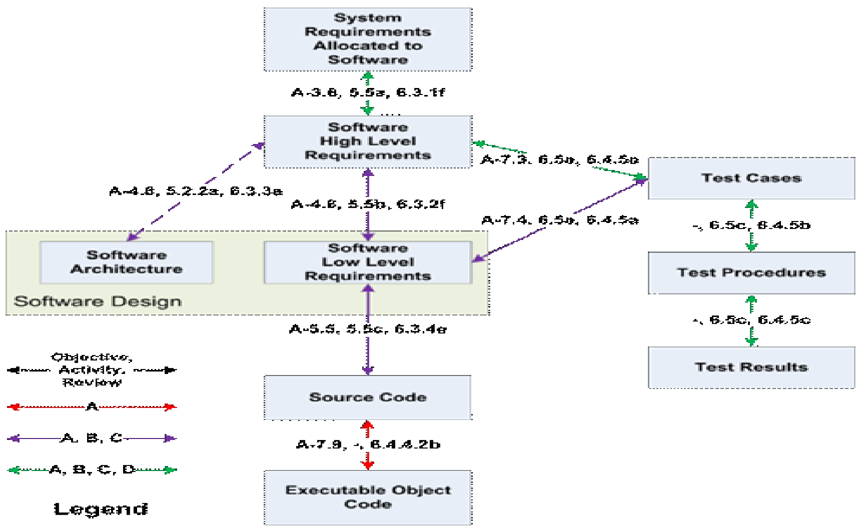


Figure 2.3: RTCA Do-178C required traceability, from [1]

## 2.3 Related Work

### 2.3.1 Model-based Testing

In this research, the target model is the extended finite state machine (EFSM) that it models control and data. When EFSMs include communication between components, they form Communicating Finite State Machines (CEFSM). Model-based testing tries to establish a conformance relationship between the specification and its implementations. Models are extracted from specifications and used for test cases generation. More precisely, it establishes a relationship between the model specification and an assumed abstract model of the implementation under test. Testers should be cautious not to extrapolate those results to the entire implementation under test (IUT). For example, if we test the control aspect, then only that aspect can be the subject of inference relations under certain assumptions. The model for testing the control aspect is limited, it cannot express data and time aspects. If the control aspect of the IUT after testing is error free, this does not allow the tester to declare that the IUT is error free. In the following, we define an EFSM, a CEFSM, and a global system. In the rest of this thesis, we will focus on EFSM model for test case generation for the MC/DC coverage.

**Definition 1**. An EFSM is formally represented as an 8-tuple $< S, s_0, I, O, T, A, \delta, V>$ where

1. S is a non-empty set of states,
2. $s_0$ is the initial state,
3. I is a non-empty set of input interactions,
4. O is a nonempty set of output interactions,
5. T is a nonempty set of transitions,
6. A is a set of actions,
7. δ is a transition relation: $\delta: S \times A \rightarrow S$,
8. V is the set variables.

Each element of A is a 5-tuple t = (initial state, final state, input, predicate, block). Here “initial state” and “final state” are the states in S representing the starting state and the tail state of t, respectively. “input” is either an input interaction from I or empty. “predicate” is a predicate expressed in terms of the variables in V, the parameters of the input interaction and some constants. “block” is a set of assignment and output statements.

**Definition 2**. A CFSM is a 2k-tuple ($C_1$, $C_2$, ..., $C_k$, $F_1$, $F_2$, ..., $F_k$) where

- $C_i$ = <S, $s_0$, I, O, T, A, V> is an agent’s model
- $F_i$ is a First In First Out (FIFO) list for $C_i$, i=1, …, k.

Suppose an agent system consists of k communicating CEFSMs: $C_1$, $C_2$, ..., $C_k$. Then its state is a k-tuple <s(1), s(2),..., s(k), $m_1$, $m_2$,...,$m_k$> where s(j) is a state of $C_j$ and $m_j$, j=1..k are set of messages contained in $F_1$, $F_2$,...,$F_k$ respectively. The CEFSMs exchange messages through bounded storage input FIFO channels. We suppose that a FIFO list exists for each CEFSM and that all messages to a CEFSM go through its list. We suppose in that case that an internal message identifies its sender and receiver. An input interaction for a transition may be internal (if it is sent by another CEFSM) or external (if it comes from the environment). The model obtained from a communicating system via reachability analysis is called a global model. This model is a directed graph G = (V, E) where V is a set of global states and E corresponds to the set of global transitions.

**Definition 3**. A global state of G is a 2k-tuple <s(1), s(2), ..., s(k), $m_1$, $m_2$, ..., $m_k$> where $m_j$, j = 1, …, k are set of messages contained in $F_1$, $F_2$, ..., $F_k$ respectively.

**Definition 4**. A global transition in G is a pair t = (i, α) where $\alpha \in A_i$ (set of actions). t is firable in s = <s(1), s(2), ..., s(k), $m_1$, $m_2$, ..., $m_k$> if and only if the following two conditions are satisfied where = (*input*, *predicat*e, *output*, *compute-block*).

- A transition relation $\delta_i$(S, α) is defined
- *input* = *null* and *predicate* = *True* or *input*= α and $m_i$= $\alpha W$,

where W is a set of messages to $C_i$, and *predicate* = *True*.

After t is fired, the system goes to s’ = <s’(1), s’(2), ..., s’(k),$m'_1$, $m'_2$, ..., $m'_k$> and messages contained in the channels are $m'_j$ where

- $s'(i) = \delta(s(i),\alpha)$ and $s'(j) = s(j)\ \forall\ (j \neq i)$
- if *input* = Ø and *output* = Ø, then $m'_j = m_j$
- if *input* = Ø and *output* = *b*, then $m'_k = m_k b$ ($C_k$ is the agent which receives *b*)
- if *input* ≠ Ø and *output* = Ø, then $m'_i = W$ and $m'_j = m_j\ \forall\ (j \neq i)$

- if *input* $\neq \emptyset$ and *output* = *b*, then $m'_{i} = W$ and $m'_{k} = m_k^b$

**Definition 5**. A test case in composed <preamble, target, post-amble, verdict> where the preamble is a sequence of transitions that start at the initial state and ends at the target, it might be empty. A post-amble is the sequence of transition that starts at the ending state of the target and end at the final state, this might be empty. The target is the element to test. The verdict is in the set {pass, fail, inconclusive}.

**Definition 6**. A test sequence is a set of test cases.

### 2.3.2 Test Coverage Criteria

The notion of coverage is important in test case generation. It characterizes the quality of a test case and test suite. It also helps determine the efficiency of test cases. There are several coverage criteria, among them requirement coverage, structural coverage, data flow coverage, input domain coverage, and fault coverage. Test coverage is very often used to measure how thoroughly software is tested. It is also used by software developers and vendors to indicate their confidence in the quality of their software product. Despite decades of research on coverage criteria and metrics, the traditional coverage notions that are used in software testing, such as statement coverage, branch coverage, and path coverage [5, 6, 7, 8, 9, 10], are not sufficient to ensure that a tested software satisfying a coverage criterion is error free. The only way to ensure that a software is error free via testing is to perform exhaustive testing which is very often very expensive or impossible due to very large or infinite input set. Coverage criteria provide a cost trade-off in testing. This research focusses on coverage criteria that are important to avionics software systems such as requirement coverage, and MC/DC [3, 11, 12, 13, 14, 15]. Coverage criteria were widely studied. In the following, we will focus on du-paths and MC/DC criteria (see Figure 2.4). These two criteria are not comparable. They can be used separately or integrated in a test sequence generation algorithm to enhance the quality of test cases.

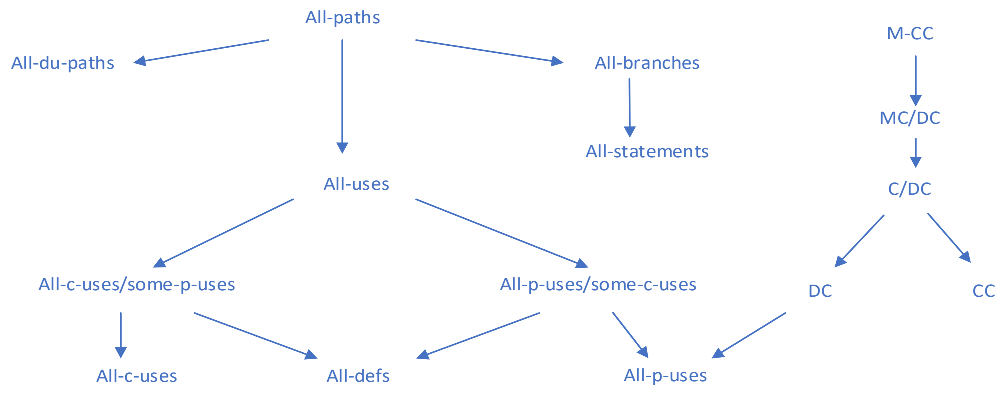


Figure 2.4: Coverage Criteria [5]

All-du-paths criterion is a dataflow coverage criterion that links the definition and usages of variables. A definition of a variable is any statement that modifies the value of a variable. It is equivalent to “write” in the memory zone associated with the variable name such as assignment and read input that modifies the value of the variable. A usage of a variable is all operations that read the value of a variable without modification such as computation use (C-use) and Predicate use (P-Use) and output use (O-use). A du-path is a definition-usage path that links the definition of a variable to its usage.  It is desirable that the path is definition clear, meaning that there is no redefinition of the variable within the path. All-du-paths is a criterion less strong but manageable than all-paths [5].

### 2.3.3 Modified Condition/Decision Coverage

The objective of Modified Condition/Decision Coverage (MC/DC) criterion is to demonstrate that all conditions involved in an expression (decision) can influence the result of that expression. All safety critical systems have decisions that need testing. Some of the specified decisions are complex and need specific techniques to address them. MC/DC criterion is stronger that condition and decision criteria. The satisfaction of MC/DC criterion is required by DO-178C standards for software avionics systems [1]. More details about MC/DC and challenges of its testing are provided in [3, 15]. A decision is a Boolean expression that is composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. A condition is in fact a leaf-level Boolean expression. It is atomic and cannot be broken down into a simpler Boolean expression. MC/DC is a structural coverage criterion, developed as a trade-off between Multiple-Condition Coverage criterion and Condition/Decision Coverage criterion that has a lower number of test cases . MC/DC was used for code testing with the following requirements:

(1) Every decision in the program must be tested for all possible outcomes at least once.

(2) Every condition in a decision within the program must be tested for all possible outcomes at least once.

(3) Every condition in a decision must be shown to independently affect that decision's outcome. This requirement ensures that the effect of each condition is tested relative to the other conditions; and

(4) Every exit and entry point in the program (or model) should be invoked at least once.

Several test sequence generations with the MC/DC coverage exist and all of them are dedicated to code testing [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. The proposed techniques are based on one of the following categories: Binary Trees [16], truth table for each Boolean expression [12, 22, 18, 23], n-cube graph [26], and constraints solving [10, 27]. Each of the proposed techniques, if not combined with other techniques that optimize the well-known issues, suffers from scalability and state explosion problem. The techniques that are based on graphs suffer from state explosion and decidability issues in relation with the number of variables. There are very few approaches to MC/DC test cases generation for system's specifications that consist of complex decisions. Most of them assume that each decision is independent. TING SU et al. [10] presented 3 different approaches, essentially based on binary trees and constraint solvers to generate MC/DC test cases for decisions. For each approach, we discuss its advantages and limitations [17, 18, 23, 24, 27]. Most of the proposed techniques do not address multiple decisions and cycles in test case generation. Very few automatic tests case generation tools exist and suffer from the above challenges [17, 18, 23, 24, 27].

### 2.3.4 Test Cases Generation based on EFSM

EFSM model extends the Finite State Machine (FSM) model that represents the control aspect with variables and predicates (data flow aspects). Test sequence generation is thus more complex for an EFSM.

The challenges are:

1. The state explosion problem, which leads to incomplete coverage.

2. Data selection, which can often be undecidable; and
3. Path feasibility (executability), which is very expensive.

Extensive research has been carried in model-based testing (MBT). The first comprehensive survey was published by Bourhfir et al. [7, 28, 29]. In 2015, Yang et al published a more recent and exhaustive survey entitled "EFSM-based Test Case Generation:  Sequences, Data and Oracle" [8]. Additional information on control and data flow-based test generation techniques can also be found in a book chapter published in 2017 [9]. Another approach for the test sequence generation for MC/DC was developed in the case of Communicating EFSM [30].

In data flow testing, all test strategies aim at selecting complete paths using the control flow to link the definitions of variables to their respective usages. Choosing the paths when executing is assumed to stimulate errors and detect faults related to the data aspect of a program/specification. To determine the quality of a test sequence, metrics or coverage criteria are used for two purposes:

1. In test generation to satisfy, by design, the desired criteria; and
2. In the evaluation of the existing test sequences and of the software.

Coverage criteria are often used for the assessment of software test data. A coverage criterion for software, if integrated within a test generation algorithm, will measure the amount of testing to be performed by executing the generated set of tests. It also provides a notion of a test sequence's quality. Testing based on the EFSM has been searched extensively. The EFSM is a model that extends the FSM model with variables and predicates that appear within condition statements. Test sequence generation is more complex in EFSM, and very often faces a state explosion problem that leads to incomplete coverage [8, 9, 10]. The data selection that is needed is often undecidable and path feasibility/executability is not cost efficient.

Bourhfir et al. proposed an EFSM-based test sequence generation method that generates executable test sequences [7 , 27, 28, 29]. A complete test sequence is obtained in five steps. First, the technique transforms an EFSM model into a dataflow graph. Second, it selects input values for the input parameter that affects the control flow. Third, executable sequences are generated using du-paths and removing any sub path inclusion, while appending the state identification sequence and post-amble to each du-path [29]. The executability of each path is verified in the fourth step. They used cycle analysis, symbolic execution, and CSP techniques to solve the path executability problem. Fifth, relevant paths are added to cover any uncovered transitions. This technique verifies the executability of a path during its generation. It uses optimized IO-df-chains [29] criterion and multiple UIO with Wp as the state identification method. Table 2.1 presents a summary and comparison criteria of the relevant EFSM-based methods [20, 23, 24, 25, 26, 27]. A more exhaustive list can be found in [8, 19, 30].

Table 2.1: EFSM-based test sequences generation approaches

**(- means not given or not addressed)**

| Authors and Date | Model Transformation | Coverage Criteria | Signature | Data Selection | Path Executability Technique |
|---|---|---|---|---|---|
| Ural 1991, 1993 [31, 32] | EFSM to Control graph and Data Flowgraph | IO-df-chain | - | - | - |
| Bourhfir 1997, 2001 [27, 29] | EFSM to Data Flowgraph | IO-df chains | M-UIO Wp | Rando m | Cycle analysis, Symbolic execution, CLP-BNR technique |
| Heirons 2002 [34] | SDL-EFSM to NF-EFSM to EEFSM/PEFSM | all-uses | - | - | Path splitting, state decomposition, Predicate decomposition simplex algorithm |
| Wong 2008, 2009 [35] | EFSM | all-nodes,all-edges Hot spots | - | Symbolic execution | Conflict detection Possibility to use CSP |

### 2.3.5 Test Data Generation Techniques

Testing based on EFSM requires test data generation. To test a system, both variables and parameters need values that must be selected from their domain definition in combinatorial manner. The selected data has the important role of stimulating the path and revealing any errors. The selected data should simultaneously satisfy all the predicates along the path for its feasibility (executability). The difficulty is that the input domain that combines all the variables and parameters domains is too large or infinite to consider its complete coverage. It is known that test data generation is an undecidable problem [37]. Several techniques have been explored for test data generation and selection. One of the techniques is exhaustive testing, which refers to using every input sequence from the input domain that is a combinatorial set of all variables and parameters domains. Exhaustive testing can only be considered for very small models. To cope with large input domains, partition testing is preferred, as it consists of dividing the input domain into several equivalence classes from which only one test data is chosen. The challenge is to define the equivalence relation that can best meet the requirements. Another technique used for software testing is boundary testing, which tests boundaries' limits [39]. Test data generation and selection techniques can be grouped in the following categories: symbolic execution [37, 38, 39, 40, 41], random, mutation, linear regression to narrow intervals, and search-based techniques [26].

# Chapter 4

# Test Generation Approaches

## 3.1 Overview

In this chapter, we describe the design of our MC/DC-TGT by investigating different views, each governed by a design viewpoint and outlining pertinent design elements and design relationships. We describe the tool along four design views: context view, process and algorithm view, architecture view, and data view. The context view provides for automatic test case generation and generates global test sequences. In the process view, we provide a high-level description of the approach and the rationale behind the theoretical decisions. The architecture view presents how the approach is structured into modules, recursively establishing the roles and interactions of the constituent submodules. In the data view, we describe the substantial persistent data and the data management strategies when applicable. Finally, in the algorithmic view, we wrap up the design views by detailing select routines and justifying some of the design choices in term of performance.

An automatic model-based test case generation approach that generates local test cases based on an existing EFSM test generation technique was modified to handle the MC/DC criterion in addition to the all-du-paths and published in [30] by Elqortobi et al. The approach also generates global test sequences for an integrated system. It uses parallel communicating agents for modeling the system to be tested. The approach also uses model checking for the verification of properties at the design level and validates their propagation to the product level via testing. For the context view, the proposed MC/DC-TGT will operate as an alternate solution for EFSM-based local test case generation.

## 3.2 Model-based Verification and Testing Methodology

In this thesis, we overview the issue of safety-critical software verification and testing that are key requirements for achieving RTCA DO-178C [1] regulatory compliance for airborne systems. As argued in [30], formal verification and testing are considered two different activities within the airborne standards, and they belong to two different levels in avionics software development cycle. The objective is to integrate model-based verification and model-based testing within one framework. This objective is achieved by proposing a new methodology for the verification and testing of parallel communicating agents based on formal models. In this methodology, properties are extracted from requirements and formally verified at the design level, while the verified properties are propagated to the implementation level via testing. The methodology is composed of five steps as depicted in Figure 3.1.:

1) modeling behaviors and specifying properties for formal verification at the design stage.

2) performing verification using and extending existing tools.

3) transforming the verification model to testing model using refinement.

4) generating test case automatically for testing individual agents in their context (conformity); and

5) generating test case automatically for the integration of all agents based on partial reachability graph, and finally checking that the verified properties hold at the implementation level via testing.

The results of formal verification and testing can be used as evidence for avionics software certification.

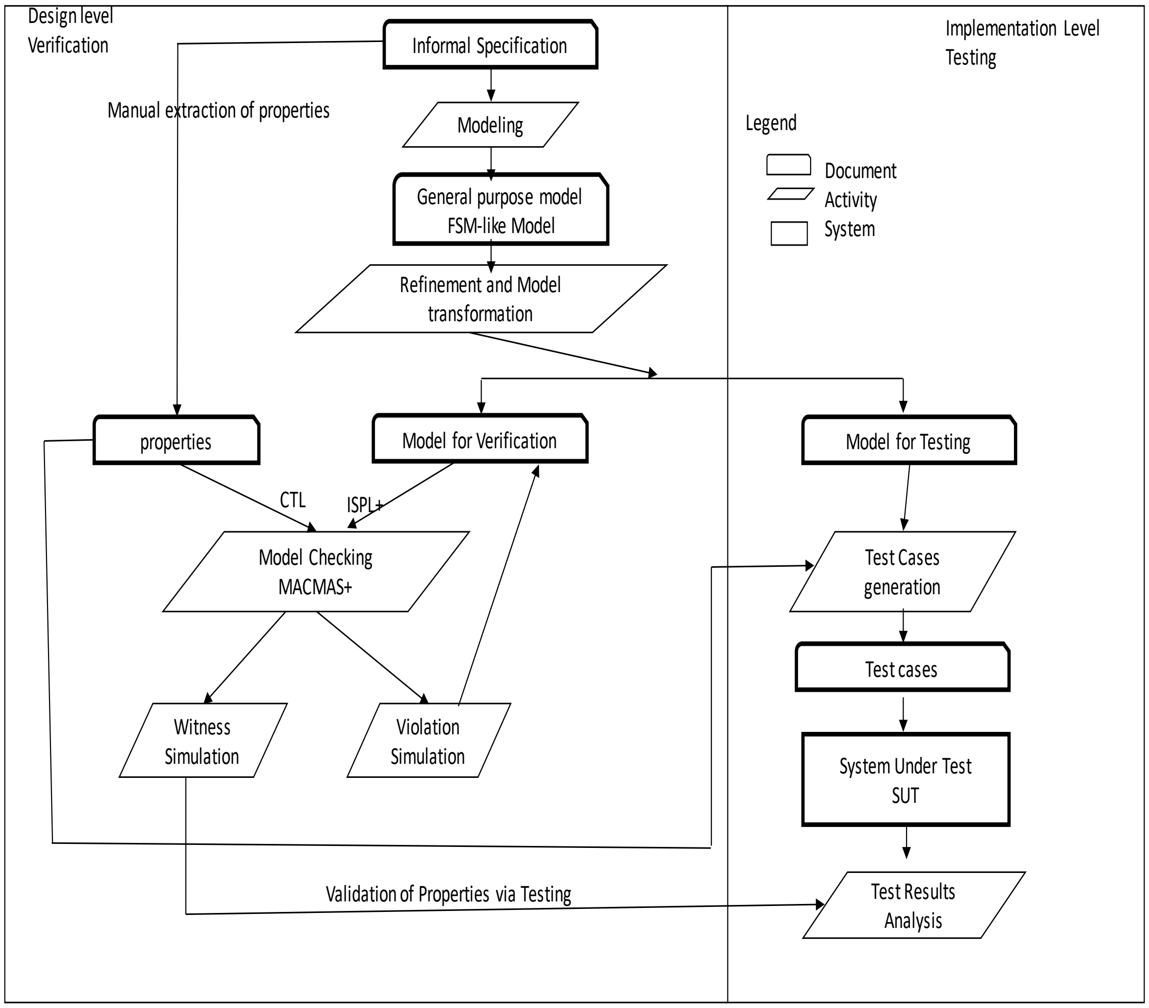


Figure 3.1: Overview of the methodology (from [12])

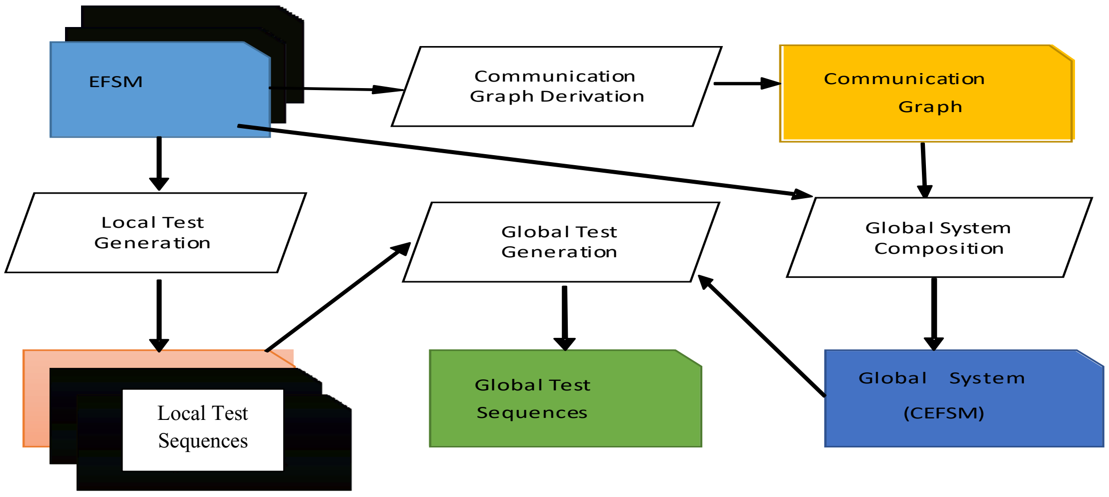


Figure 3.2: Test generation process (from [12])

The integration of verification and testing in our methodology is designed in compliance with RTCA DO-178C standards [1]. This design follows the following steps:

- ✓ Verification of properties at the design stage.
- ✓ Low-level requirements model-based testing. The test generation process, depicted in Figure 3.2, generates:
  - ✓ local test cases based on EFSM model of LLR;
  - ✓ global test cases based on CEFSM model, that is the integration of components (agents).
- ✓ Handling the mandatory coverage criterion MC/DC in addition to all du paths.
- ✓ Checking the propagation of properties at the implementation via testing as required by the standard [1].
- ✓ Addressing forward traceability (HLR $\rightarrow$ LLR $\rightarrow$ test cases) by construction.
- ✓ Producing some of the certification artifact, such as models, local test cases, global tests cases, and their relationship with LLR.

The proposed approach extends an existing test generation technique [30]. The extension is important as it modifies the coverage criteria that guides the test case generation. Bourhfir et al. proposed an approach that automatically and incrementally generates executable test sequences for Communicating Extended Finite State Machines CEFSMs model [27, 28, 29]. The communication mode is asynchronous. The approach does not compute the product of all communicating machines. It only generates test sequences by incrementally computing a partial product for each CEFSM, which mean, considering only transitions that influence (or are influenced by) the considered CEFSM and generating test sequences for it. The partial product for an CFSM represents its behavior when composed with parts of the other CEFSMs, the communicating transitions. They generate test sequences using the Extended Finite state machine Test Generator EFTG tool that was published in [29]. The tool generates executable test cases for EFSM specified protocols covering both control and data flow. The control flow criterion used is the UIO (Unique Input Output) sequence [33, 42] and the data flow criterion is the all-def-uses criterion [30]. Their approach is incremental and suitable for testing large systems. The objective is not to cover all transitions in the cross product of all CEFSMs, but to cover all transitions in all CEFSMs and all global transitions as well as all data-flow paths in each partial product. State explosion problem is possible in this technique [30, 31, 32, 34, 35, 36, 43, 44].

## 3.3 Model-based Testing Methodology with Constraint Solving

### 3.3.1 Approach Overview

The proposed approach addresses the generation of local test cases with complementary features. This test generation technique (see Figure 3.3) is an alternative solution to local test case generation presented in Figure 3.2. The focus here is to advance transition path feasibility and preparatory work to show requirements' traceability.

The main objective is to develop a methodology that generates test cases for local components and design a tool for its support. The idea is to build on our previous solution and improve test case generation based on models that handles path selection using both control flow and data flow graphs, and the required MC/DC coverage criterion**.** For path feasibility we use the constraints solving technique.

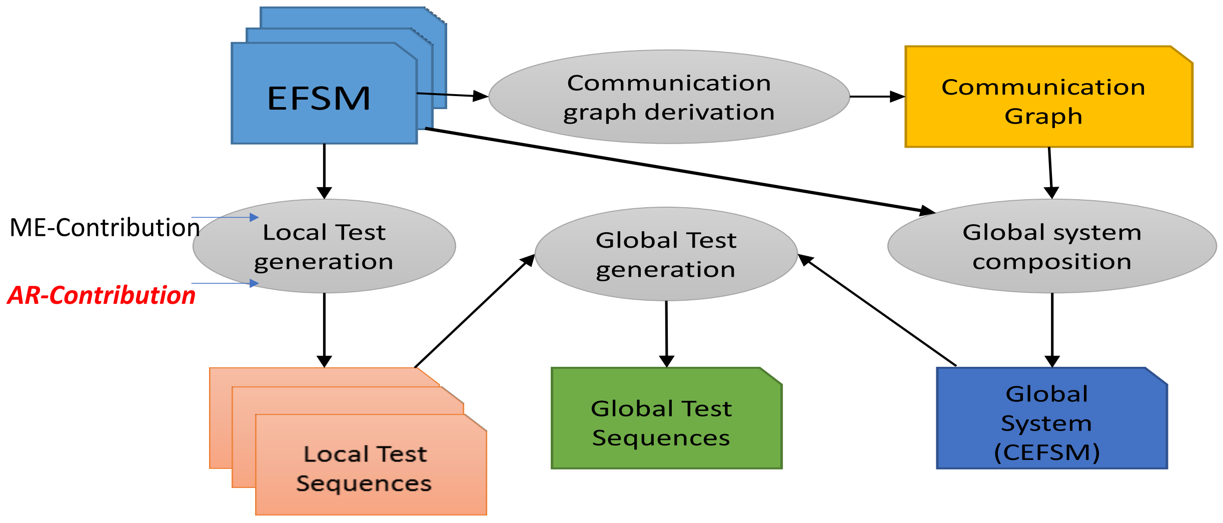


Figure 3.3: Alternative test generation process from [30]

### 3.3.2 Process Steps

We start by defining exactly how a test suite satisfies MC/DC for a given decision. After the desired MC/DC configuration is given in the form of an MC/DC table, we need to generate a test sequence for each row. The execution path on the sequence must include the control point where the decision is being evaluated (to satisfy the first MC/DC requirement). This is done while ensuring that the constituent conditions of the decision are allocated the desired outcome by checking that:

1. The variable definitions influencing the said conditions are coherent (to satisfy the second MC/DC requirement); and
2. The variable definitions are independent from each other, i.e. ensuring that different variables are involved in changing the values of the conditions in terms of i-use, d-use and c-use (to satisfy the third MC/DC requirement).

To handle MC/DC, we are faced with a challenge that requires analyzing data flow and control flow aspects to be solved. We therefore generate the necessary graphs for test generation. MC/DC-TGT outputs an executable test suite along with the coverage data, given user-provided input as an EFSM model, MC/DC tables and complementary test criteria. Figure 3.4 shows how MC/DC- TGT gradually constructs the executable test suit. Our approach is designed to combine several test criteria while reducing their search space by introducing the coverage element construct. In the following, we focus on the MC/DC criterion alone.

The first step of our approach is to formally generate the Data Flow Graph (DFG) and the Control Flow Graph (CFG) from the EFSM. The primary goal is to separate the data flow aspects from the control flow aspects, thereby simplifying the task of finding the targeted information. This information is obtained from the MC/DC tables (or coverage elements), and is used to label the EFSM, DFG and CFG. Selecting an executable path will become a matter of finding a path on a labeled graph. The path selection is guided by path feasibility using constraints-solving. We create an abstract test suite by associating the selected paths to the coverage elements they potentially satisfy if the proper test data is selected. Test data selection follows and gives us an executable test suite. We ran a coverage analysis to verify that the target coverage has been achieved and compile a test report in terms of expected outcomes as a verdict on the test sequence execution.

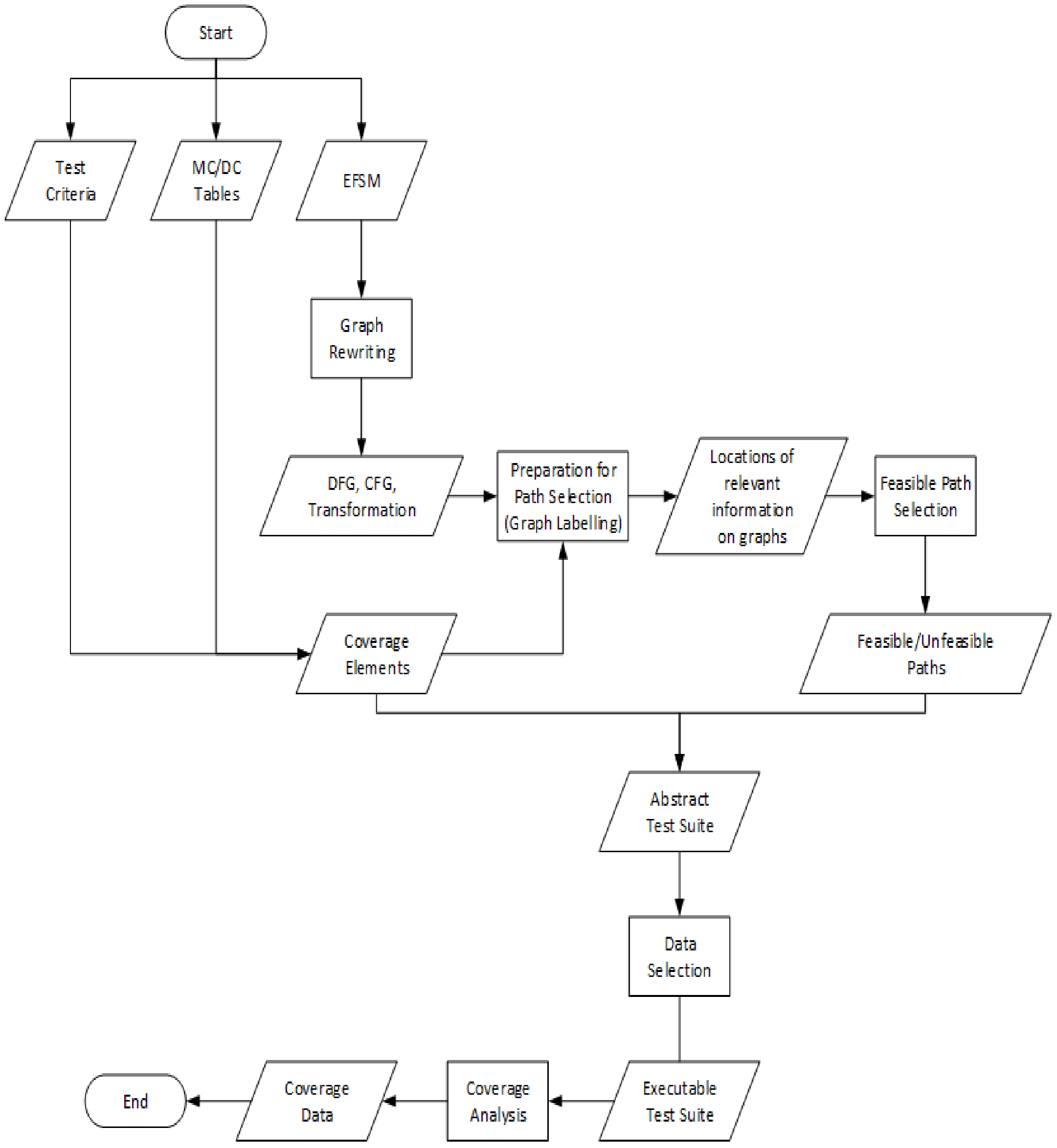


Figure 3.4: EFSM test generation process

The key steps of the approach are:

1. Automatic model transformation to obtain control and data flow graphs using graph rewriting (EFSM → graphs).

2. Preparation for path selection using graph labeling to obtain location of the relevant information on graphs.

3. Feasible path selection to obtain feasible paths and unfeasible paths that compose a set of abstract test sequences.

4. Data selection to obtain executable test sequences.

5. Coverage analysis where all MC/DC requirements are assessed again.

### 3.3.3 Step 1: Graph Rewriting

Here we provide a simplified explanation of the Single-Pushout (SPO) approach for graph rewriting and the concepts involved, and what it means in practice. Rigorous mathematical semantics based on category theory are provided in Rozenberg's book [45]. First, we define the following: Grammar, Rule Graph, State Graph, Match, Rule Morphism, Rules and Rule Application. Figure 3.5 provides an overview of the graph rewriting approach.

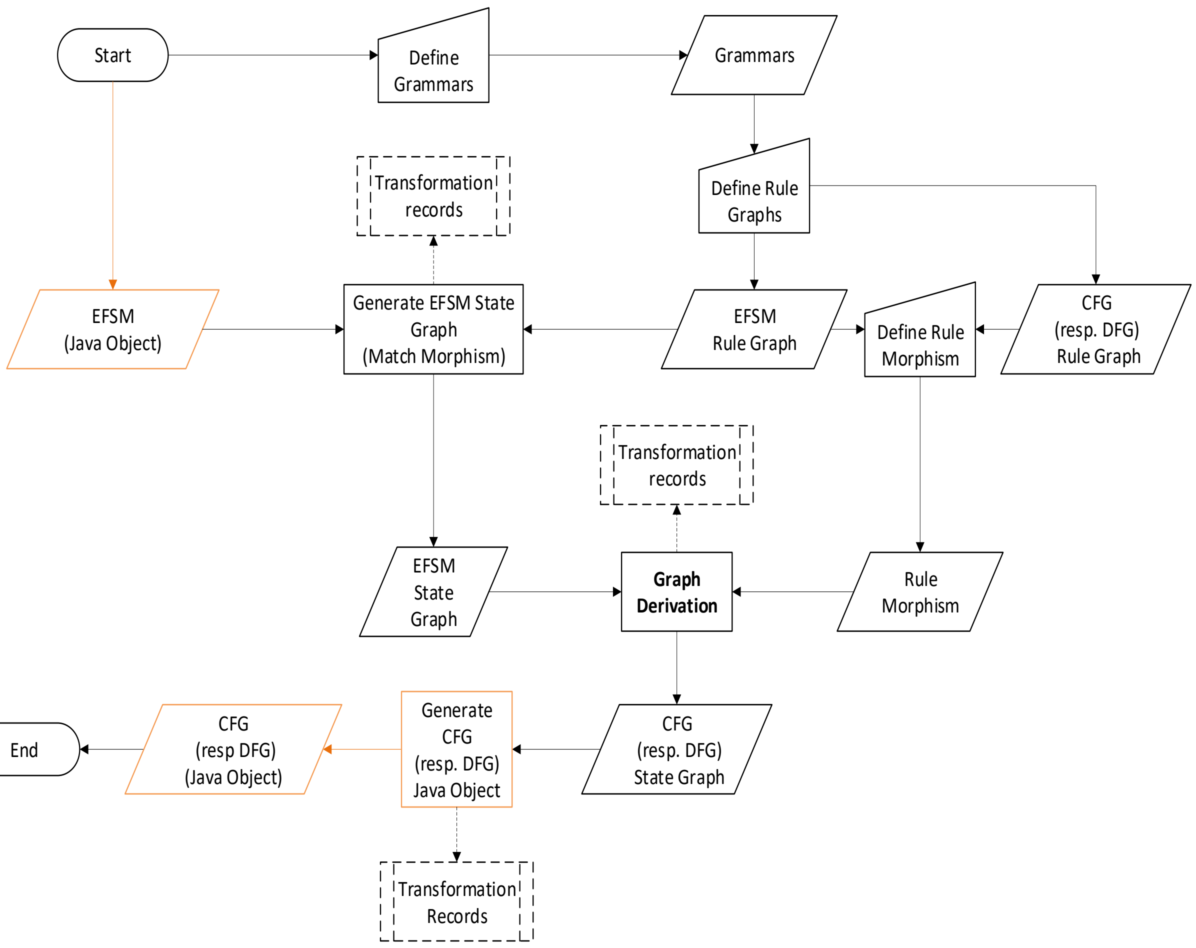


Figure 3.5: Graph rewriting overview

A Grammar is the formal definition of the graph elements. In an attributed grammar, vertices and edges are assigned types and may have attributes. Defining an attributed graph grammar amounts to defining a set of vertex types, a set of edge types, and for each type specify whether it has attributes (and whether those attributes have typed on untyped members).

A Rule Graph is a graph (usually a type graph) that uses a predefined grammar. It defines the vertices types each edge type can link, as well as the multiplicities. A State Graph is a graph G that adheres to the rules defined in a given rule graph R. In practice, the graph we aim to transform is a state graph. It is defined by a graph morphism m: R→G called Match. A Rule Morphism is a graph morphism between two graph Rule Graphs L and R. In practice, a rule morphism is a set of Rules, where a Rule is a graph morphism between a subgraph of L and a subgraph of R.

A graph transformation is a series of rule applications. A Rule Application (a.k.a rewriting step) in the SPO approach is defined by the pushout diagram depicted in Figure 3.6 [45, 46].

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "m"'] & R \arrow[d] \\
G \arrow[r] & G'
\end{tikzcd}$$

Figure 3.6: Pushout diagram

In the Pushout diagram, G (resp. G’) is the source (resp. target) state graph adhering to the Rule Graph L (resp. R), and r: L → R is a rule morphism and m: R → G a match. We say that we derive G’ from G via rule application of r at a match m. A rule can only be applied if its conditions (subset L’ of L) are satisfied in G. Checking whether a condition is satisfied equates to checking whether m(L’) is empty. In the SPO approach, a match has to be total. This will ensure that all the rules can be applied.

Via the SPO, we transform a graph without the need to construct an isomorphism (or morphism and a reverse morphism) between the elements of the source and target state graphs. In practice we must define Grammars and Rule Graphs for each formalism used (EFSM, Data Flow Graph and Control Flow graph.). EFSM, being the source Model, defining a match is as simple as defining an EFSM instance. For each Transformation (EFSM to DFG and EFSM to CFG) we have to define a rule morphism  [45, 46]..

We use SMTlib [47] to express the guard content on the EFSM. The main goal is to simplify the parsing of the guards and their content, otherwise we would have needed to define the variables and inputs used in the formal definition of an EFSM, as part of the grammar. The Input values on the Input Edge in the Control Flow Graph grammar would be one from the Input List attribute of the Input Points preceding it. An input could be either a simple control instruction or values meant to be assigned to a variable, i.e. they potentially affect both control and data flow.

Next, we define the rule graphs for each grammar as shown in Figure 3.6. The graph outlines the relationships between node types by means of arc. The arcs used to link the nodes are typed from the grammars in Table 3.1, thus defining how components of the grammar are related to each other.

Table 3.1: Grammars used in MC/DC-TGT

| Grammar | Nodes/Arcs | Type | Attributes | Members | Member Type |
|---|---|---|---|---|---|
| EFSM | Nodes | State | Yes | Name | String |
| | | | | ID | Integer |
| | Arcs | Transition | Yes | Input | SMTLib Expression |
| | | | | Predicate | SMTLib Expression |
| | | | | Computation Bloc | SMTLib Expression |
| Control Flow Graph | Nodes | Merge Point | Yes | ID | Integer |
| | | Input Point | Yes | Input List | Enumeration |
| | | Decision Point | Yes | Predicate | SMTlib Expression |
| | | Computation Bloc | Yes | Computations | SMTlib Expression |
| | Arcs | Simple Edge | No | N/A | N/A |
| | | Boolean Edge | Yes | Decision Value | Boolean |
| | | Input Edge | Yes | Decision Value | Input Value |
| Data Flow Graph | Nodes | Computation | Yes | Computations | SMTlib Expression |
| | Arcs | Simple Edge | No | N/A | N/A |
| | | Predicated Edge | Yes | Decision Value | SMTlib |
| | | Input Edge | Yes | Decision Value | Input Value |

The final activity is to define a state graph for the EFSM.

## *A: Graph rewriting – Grammars (EFSM to CFG)*

EFSM and CFG grammars are defined as follows:

**EFSM Grammar**

**Node Types**

- State
  - Has attributes: Yes
    - Name: String
    - ID: Integer

**Arc Types**

- Transition
  - Has Attribute: Yes
    - Input: SMTLib Expression
    - Predicate: SMTLib Expression
    - Computation Bloc: SMTLib Expression

**CFG Grammar**

**Node Types**

- Merge Point
  - Has attributes: Yes
    - ID: Integer
- Decision Point Type “input”
  - Has attributes: Yes
    - Input List: Enumeration
- Decision Point Type “Predicate:
  - Has attributes: Yes
    - Predicate: SMTLiB expression
- Computation Bloc
  - Has attributes: Yes
    - Predicate: SMTLiB expression

**Arc Types**

- Simple Edge
  - Has attributes: No
- Decision Edge Type “input”

Has attributes: Yes

Decision Value: Input Value (From enum)

Decision Edge Type “Boolean”

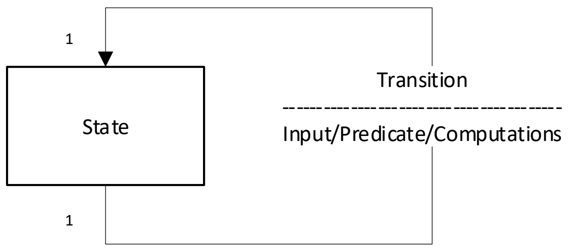
Has attributes: Yes

Decision Value: Boolean

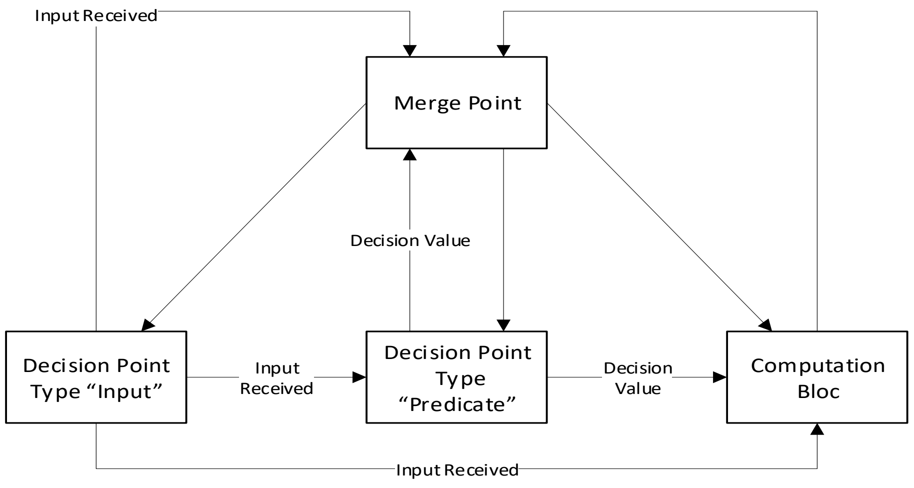
***B: Graph rewriting – Rule graphs and rule morphisms (EFSM to CFG)***

The EFSM (source) and CFG (target) rule graphs are as follows:

**EFSM Rule Graph:**

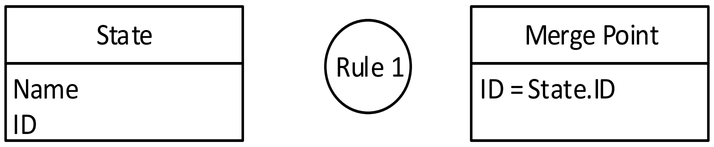


**CFG Rule Graph:**

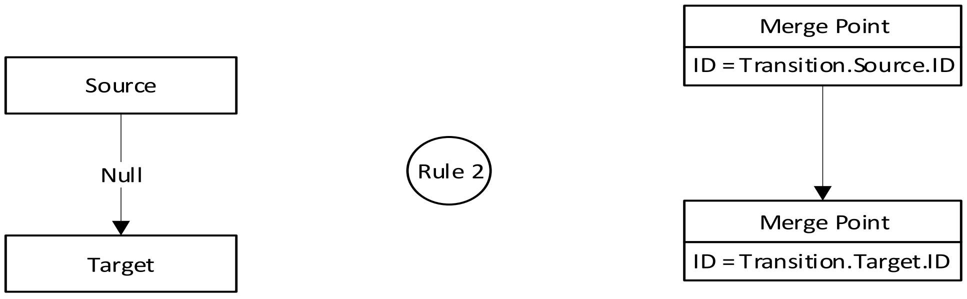


The EFSM to CFG rules are self-explanatory. Graphically, they are as follows:

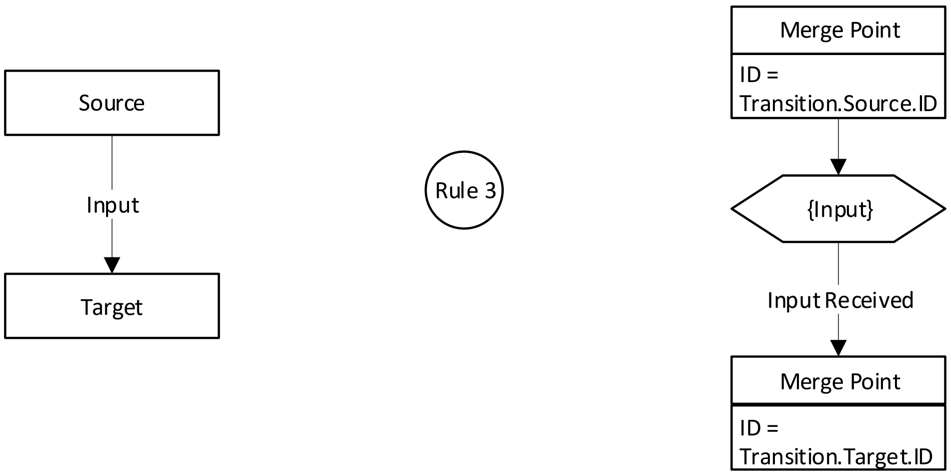
***Rule 1***: States are mapped to merge points with conservation of ID



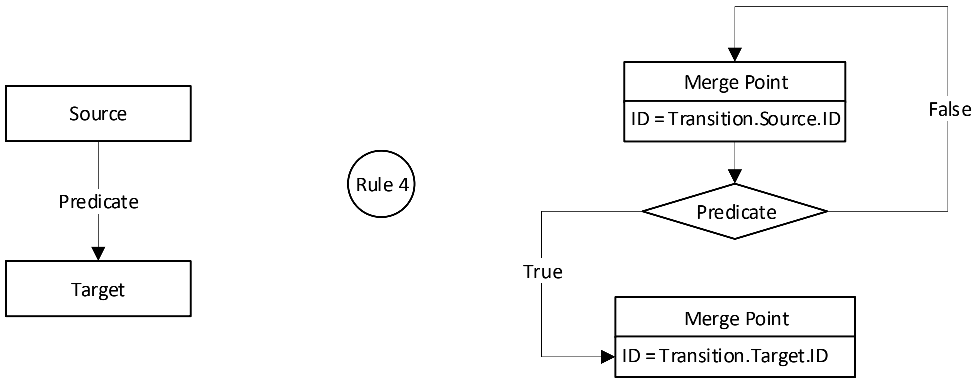
***Rule 2***: (Source-Null-Target mapping)



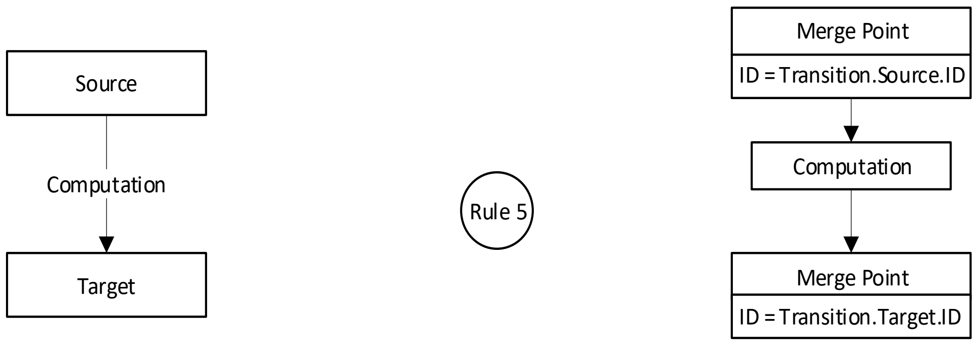
***Rule 3***: (Source-Input-Target mapping)



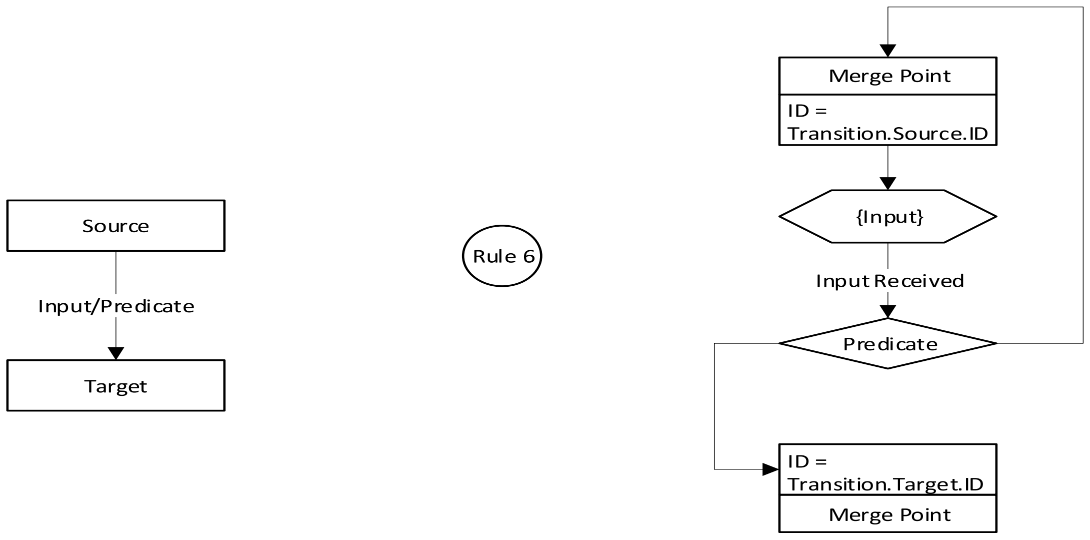
***Rule 4***: (Source-Predicate-Target mapping)



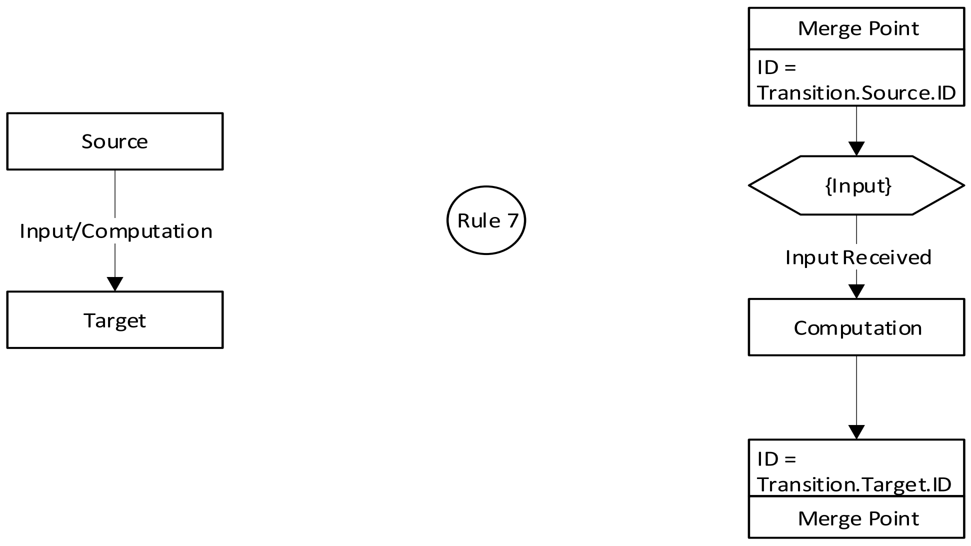
***Rule 5***: (Source-Computation-Target mapping)



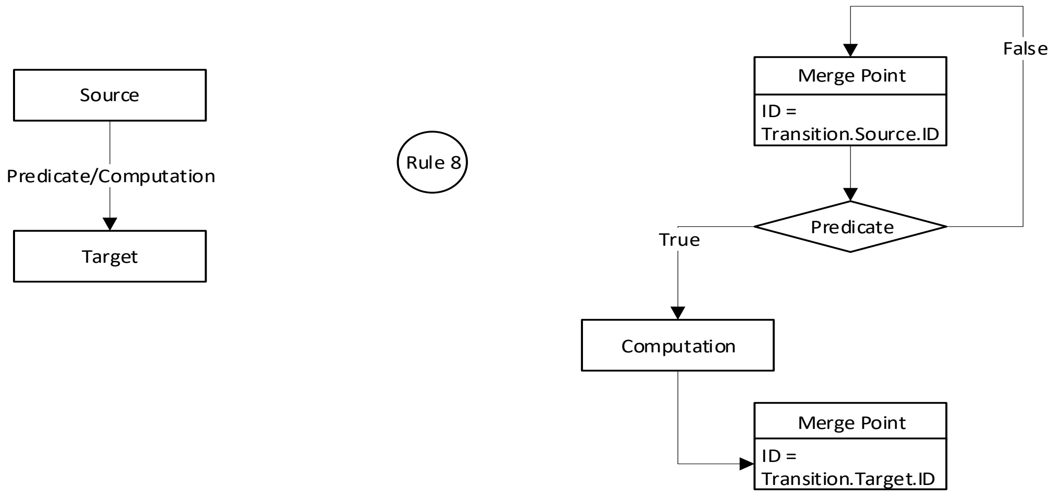
***Rule 6***: (Source-Input/Predicate-Target mapping)



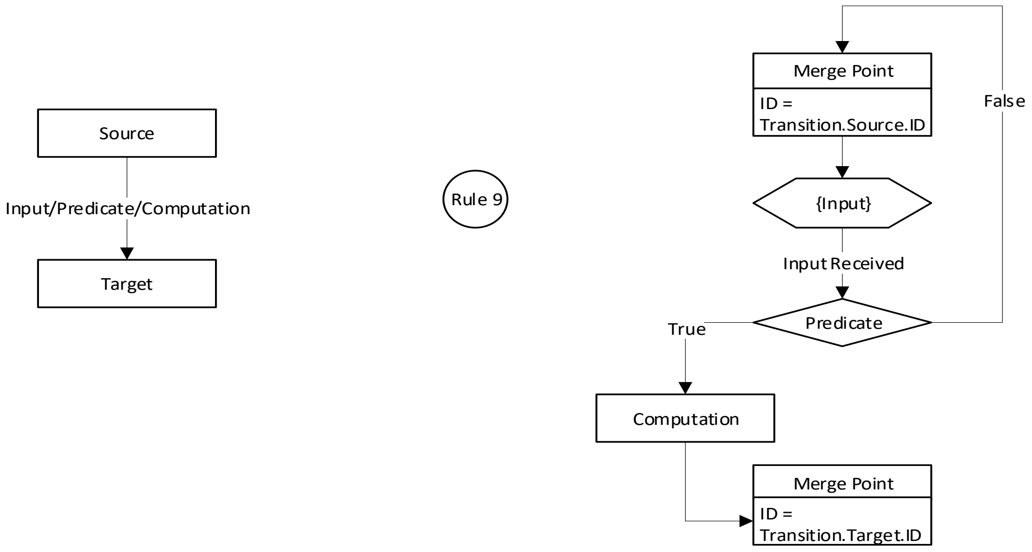
***Rule 7***: (Source-Input/Computation-Target mapping)



***Rule 8***: (Source-Predicate/Computation-Target mapping)



***Rule 9***: (Source-Input/Predicate/Computation-Target mapping)



***Rule 10***: (Source-Input|Input'-Target|Target' mapping)

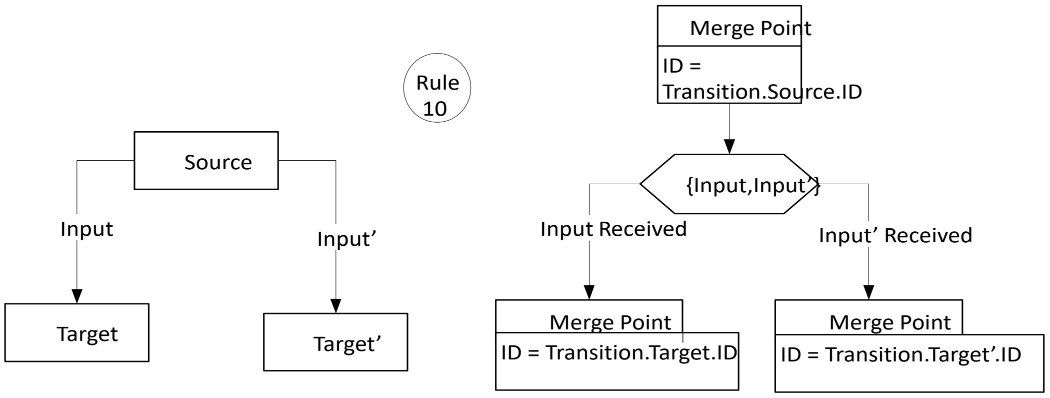


Figure 3.7 shows an example of the use of the aforementioned rules to rewrite an EFSM into a CFG.

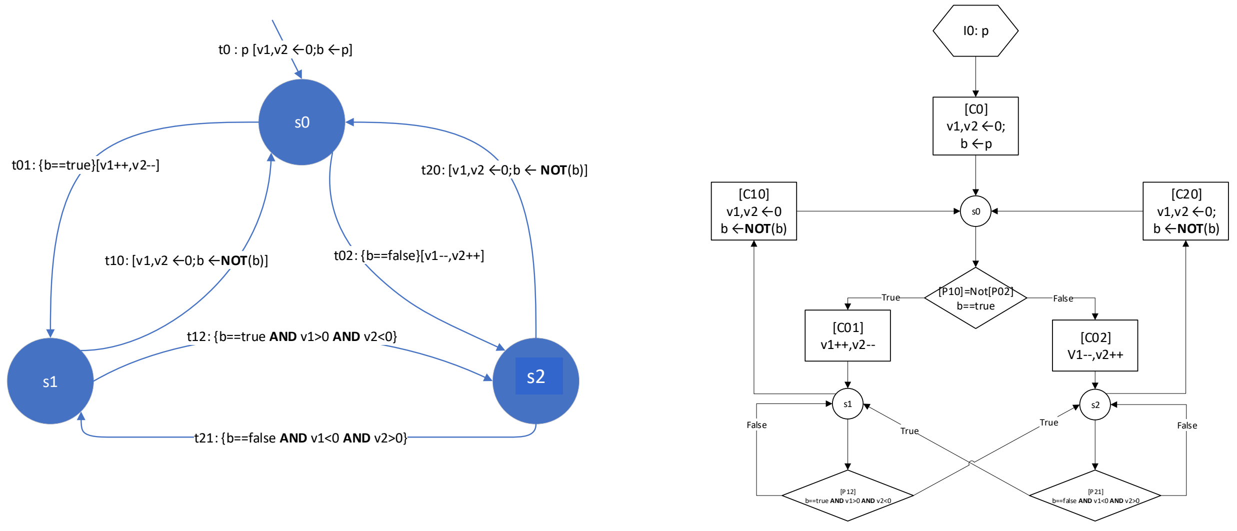

t0 : p [v1,v2 ←0;b ←p]
s0
t01: {b==true}[v1++,v2--]
t20: [v1,v2 ←0;b ← NOT(b)]
t10: [v1,v2 ←0;b ←NOT(b)]
t02: {b==false}[v1--,v2++]
t12: {b==true AND v1>0 AND v2<0}
s1
s2
t21: {b==false AND v1<0 AND v2>0}
I0: p
[C0]
v1,v2 ←0;
b ←p
[C10]
v1,v2 ←0
b ←NOT(b)
s0
[C20]
v1,v2 ←0;
b ←NOT(b)
True
[P10]=Not[P02]
b==true
False
[C01]
v1++,v2--
[C02]
V1--,v2++
s1
s2
False
True
True
False
[P12]
b==true AND v1>0 AND v2<0
[P21]
b==false AND v1<0 AND v2>0

Figure 3.7: Example of graph rewriting through CFG rules derivation

### *C: Graph rewriting – Grammars (EFSM to DFG)*

**EFSM Grammar**

**Node Types**

- State
    - Has attributes: Yes
        - Name: String
        - ID: Integer

**Arc Types**

- Transition
    - Has Attribute: Yes
        - Input: SMTLib Expression
        - Predicate: SMTLib Expression
        - Computation Bloc: SMTLib Expression

**DFG Grammar**

**Node Types**

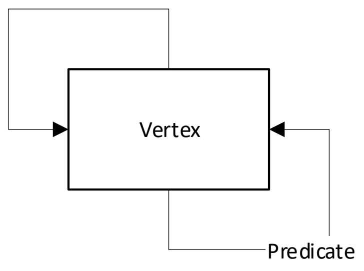
- Vertex
    - Has attribute: Yes
        - Definition/Computation: SMTLIB Expression

**Arc Types**

- Simple Edge
    - Has attributes: No
- Predicated Edge
    - Has attributes: Yes
        - Predicate: Input Value (From enum)

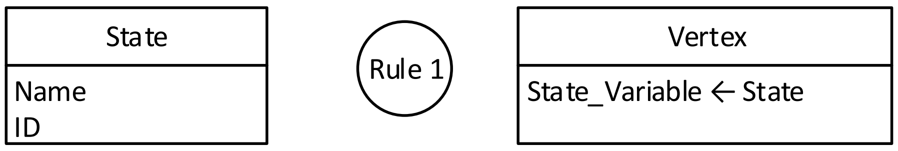
***D: Graph rewriting – Rule graphs and rule morphisms (EFSM to DFG)***

The EFSM (source) rule graph is already given, and the DFG (target) rule graph is as follows:

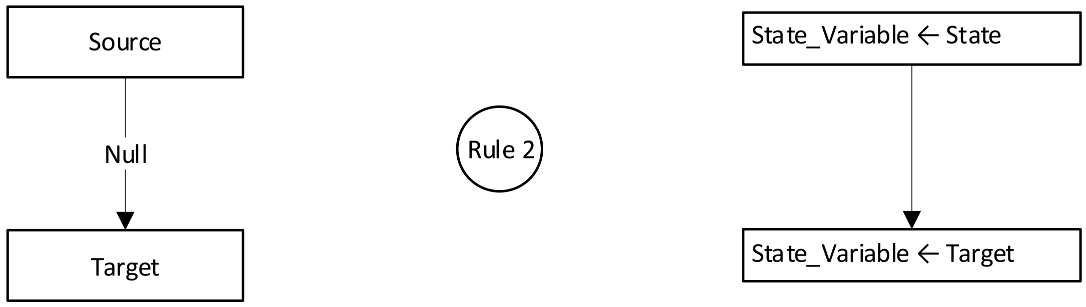


The EFSM to DFG rules are self-explanatory. Graphically, they are as follows:

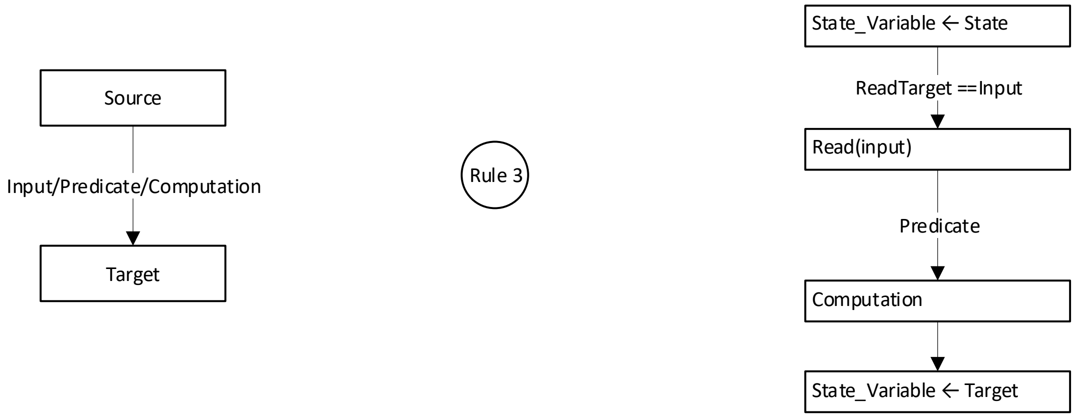
***Rule 1*:** State are mapped to vertex with state variable



***Rule 2***: (Source-Null-Target mapping)



***Rule 3***: (Source-Input-Target mapping)



***Rule 4***: (Source-Predicate-Target mapping)

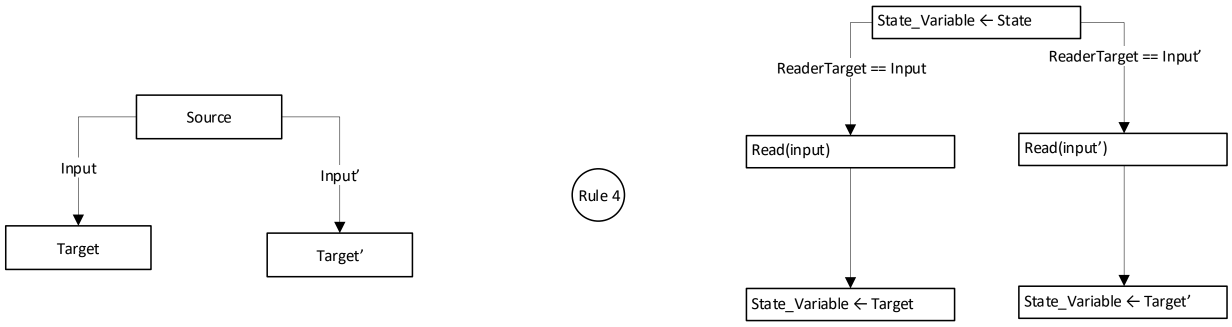


Figure 3.8 shows an example of the use of the aforementioned rules to rewrite an EFSM into a DFG.

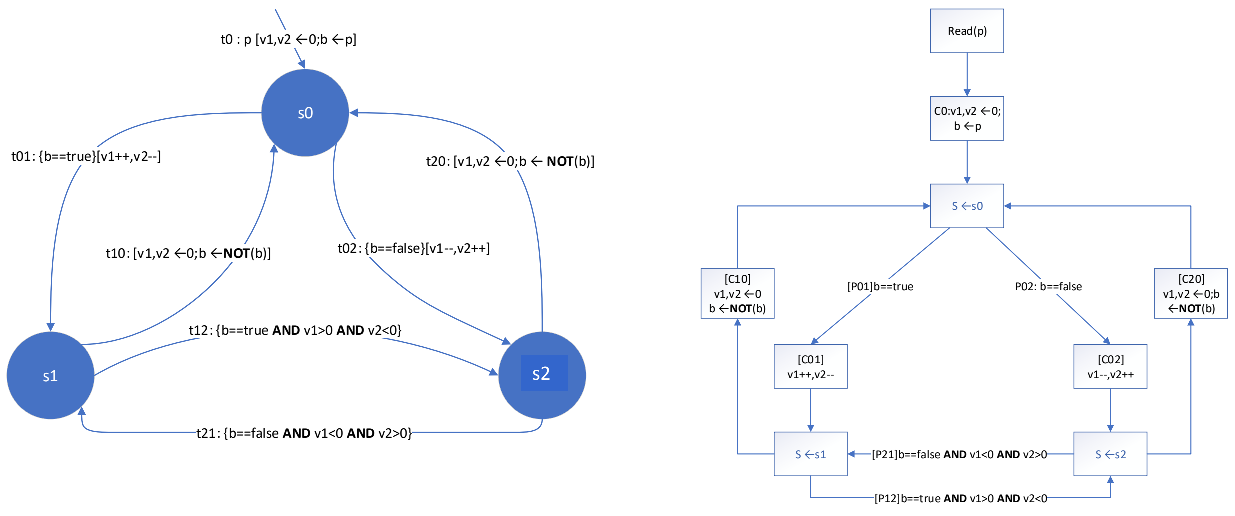

t0 : p [v1,v2 ←0;b ←p]
s0
t01: {b==true}[v1++,v2--]
t20: [v1,v2 ←0;b ← NOT(b)]
t10: [v1,v2 ←0;b ←NOT(b)]
t02: {b==false}[v1--,v2++]
t12: {b==true AND v1>0 AND v2<0}
s1
s2
t21: {b==false AND v1<0 AND v2>0}
Read(p)
C0:v1,v2 ←0; b ←p
S ←s0
[C10] v1,v2 ←0 b ←NOT(b)
[P01]b==true
P02: b==false
[C20] v1,v2 ←0;b ←NOT(b)
[C01] v1++,v2--
[C02] v1--,v2++
S ←s1
[P21]b==false AND v1<0 AND v2>0
S ←s2
[P12]b==true AND v1>0 AND v2<0

Figure 3.8: Example of graph rewriting through DFG rules derivation

### 3.3.4 Step 2: Preparation for the Path Selection

The goal of the approach is to generate executable test cases that satisfy the MC/DC criterion. To derive an executable test sequence, we need a feasible path along which the MC/DC requirements are satisfied. In other terms, we have two governing criteria: the values of Decisions and the Conditions (and thus the values of the variables influencing these Conditions), and the feasibility of the path being considered for selection. The goal of this step is to annotate the graphs using information from the MC/DC tables, which reduces the risk of state explosion when performing a multi-objective search on the EFSM/DFG/CFG by reducing the search space.

Figure 3.9 goes through the steps of the labelling. There are four information we want to pinpoint on the graphs' elements. The MC/DC Tables (or Decision) affected by the graph element, the Rows, The conditions and the values of the Conditions.  Thus, the final label would depend on each graph element as shown in Figure 3.9. We start by labelling the decision points from the CFG with the MC/DC tables ids as each MC/DC table is associated with one Decision (and thus with one predicate). Then for each table, we label the outgoing branches from the decision points with the row id that match the Decision outcome of that row. We also label the predicate edges from the DFG by means of the transformation records. And finally, for each condition we move to labeling the d-use for all variables affecting that Condition on the DFG.

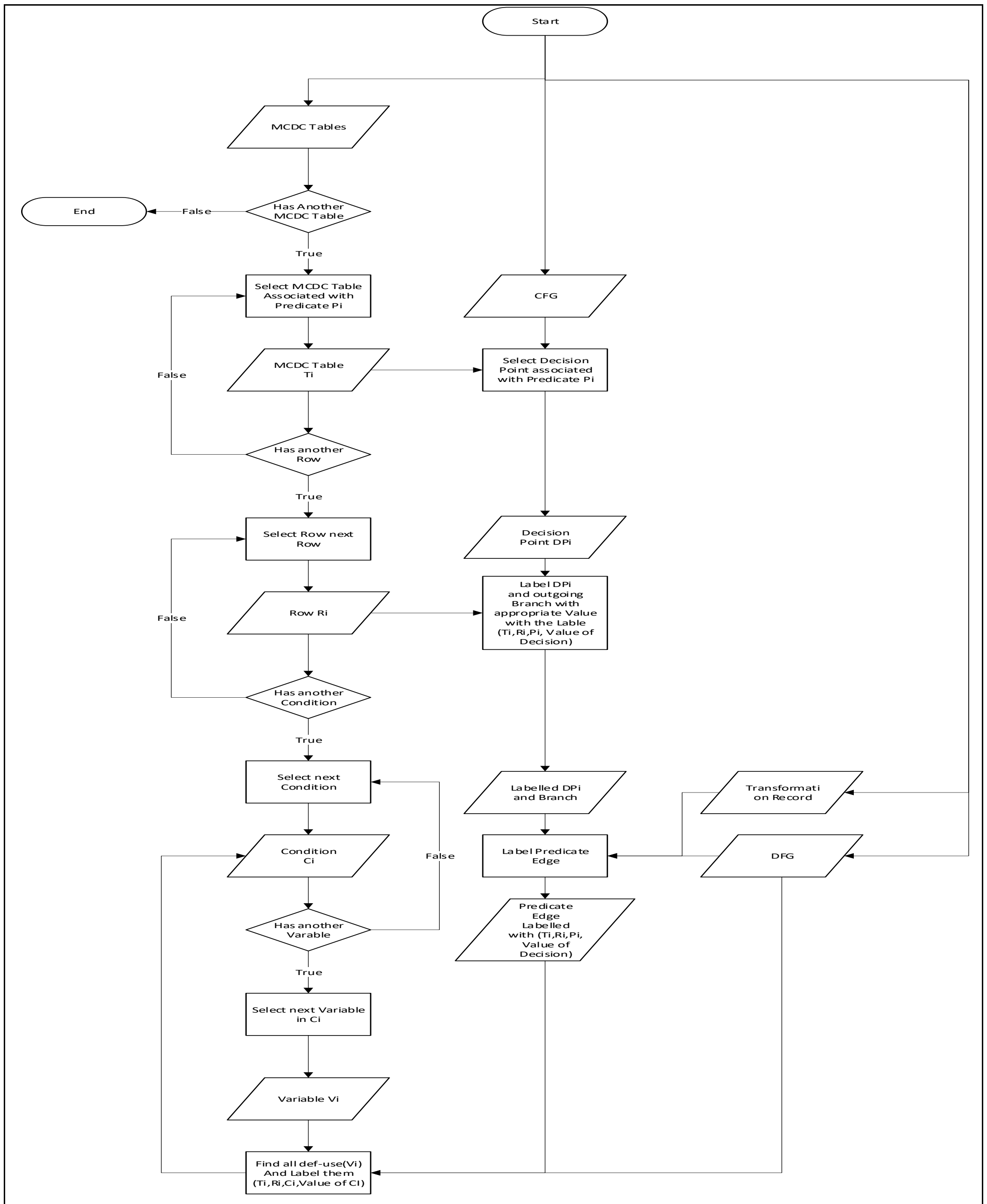
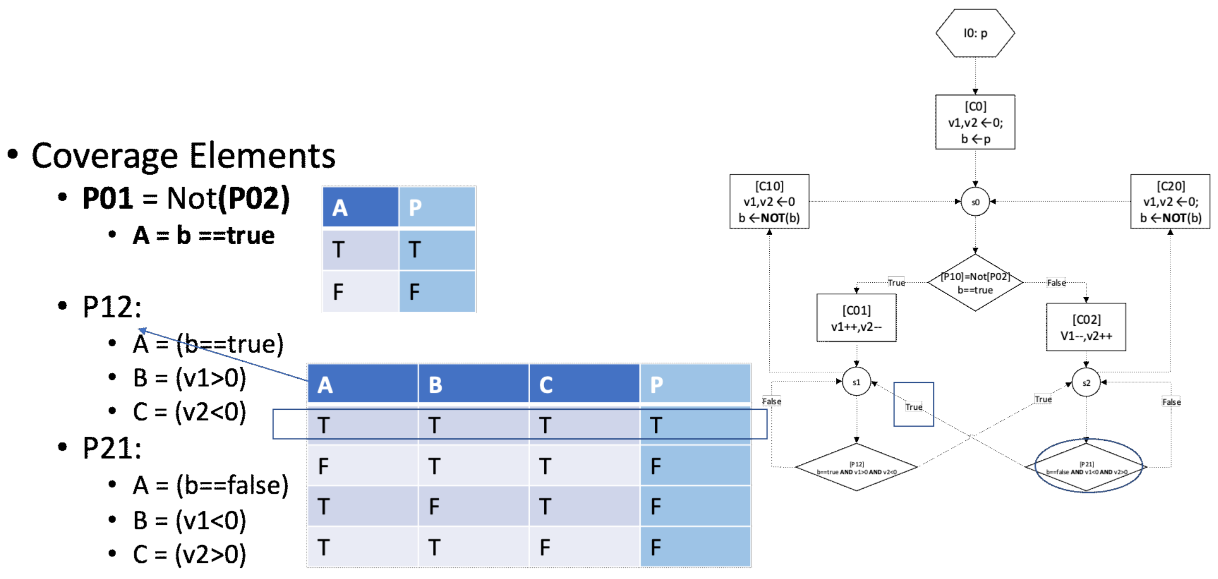

Start
MCDC Tables
End
False
Has Another MCDC Table
True
Select MCDC Table Associated with Predicate Pi
CFG
MCDC Table Ti
Select Decision Point associated with Predicate Pi
False
Has another Row
True
Select Row next Row
Decision Point DPi
Label DPi and outgoing Branch with appropriate Value with the Lable (Ti,Ri,Pi, Value of Decision)
Row Ri
False
Has another Condition
True
Select next Condition
Labelled DPi and Branch
Transformati on Record
Condition Ci
False
Label Predicate Edge
DFG
Has another Varable
True
Predicate Edge Labelled with (Ti,Ri,Pi, Value of Decision)
Select next Variable in Ci
Variable Vi
Find all def-use(Vi) And Label them (Ti,Ri,Ci,Value of Ci)

Figure 3.9: Preparation for path selection (graph labeling)

Figure 3.10 (a and b) shows an example of the application of Step 2: preparation for the path selection from the example shown in Figure 3.7.

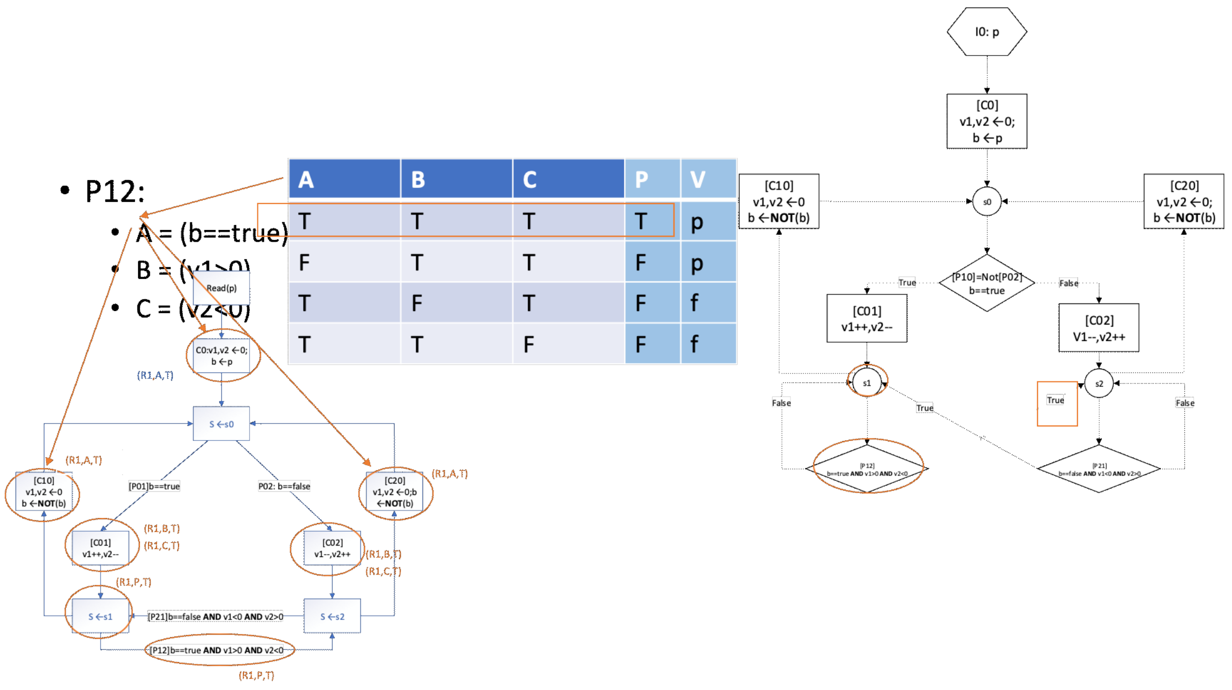


- Coverage Elements
  - **P01** = Not(**P02)**
    - **A = b ==true**
  - P12:
    - A = (b==true)
    - B = (v1>0)
    - C = (v2<0)
  - P21:
    - A = (b==false)
    - B = (v1<0)
    - C = (v2>0)

| A | P |
|---|---|
| T | T |
| F | F |

| A | B | C | P |
|---|---|---|---|
| T | T | T | T |
| F | T | T | F |
| T | F | T | F |
| T | T | F | F |

a



- P12:
  - A = (b==true)
  - B = (v1>0)
  - C = (v2<0)

| A | B | C | P | V |
|---|---|---|---|---|
| T | T | T | T | p |
| F | T | T | F | p |
| T | F | T | F | f |
| T | T | F | F | f |

b

Figure 3.10: Application of Step 2 from graph rewriting of Figure 3.7

### 3.3.5 Step 3: Path Selection

The aim is to select paths that have the potential to produce executable test cases and decide on their feasibility [55]. In TGT, we use jSMTLIB for parsing SMTLIB expressions and using the solvers with test generation tool. In the following we describe briefly how we address the selection of feasible paths. A search algorithm A* is used for finding the “shortest” path and a multi-objective search algorithm. Short path is expressed in terms of feasibility and uses of the involved variables. A* is used between the “nearest” def-use and the p-use. We also use a multi-objective search algorithm based on [52].

For SMT-constraint solving, any SMT-LIB solver can be used. For this step, the following data is required: Labelled DFG, Transformation records, Heuristics, Temporal Logic, and Theory. We obtain Feasible/unfeasible Paths (see Algorithm 1). The following is the Feasibility Analysis approach:

```
Precondition: MC/DC tables, labeled DFG
Labels applied during the previous steps
<T, R, C, Value of C> for def-uses
<T, R, P, Value of P> for p-uses
Where:
T: table
R: Row of MC/DC table
C: Condition
P: Predicate/ decision
For each table T in the set of MC/DC table
  For each (Row) R in T
    Find p-use in labeled <T, R, P, Value of P> in labeled DGF
      For each C in R find def-use with label <T, R, C, value of C>
Link p-use(C) and min-def-use(C)* with a def-clear-path**,
Add feasible preamble and post-amble to form a complete path.
```

(*) min-def-use(C) in the nearest of def-uses of the variables involved in C in term of “Approximation Level” [52].

(**) If there is a c-use w.r.t. that particular variable, we ignore it for the MC/DC approach.

The def-clear path is constructed using a standard A* algorithm with feasibility as heuristic: H(t) = +1 if the transition is feasible, H(t) =+ 100 if not.

Link the other def-use(C) with min-def-use(C)

```
Algorithm 1: GenerateFeasiblePaths
Input: $labelled\_DFG, labelled\_CFG, MCDC\_Tables$
Output: $feasible\_paths, unfeasible\_paths$

$init(feasible\_paths, unfeasible\_paths)$
foreach $T_i \in MCDC\_Tables$ do
    foreach $p\_use_labelled(T_i, R_i, P_i, P_i.value) \in labelled\_DFG$ do
        $p\_use \leftarrow p\_use\_labelled(T_i, R_i, P_i, P_i.value)$
        foreach $Vertex_labelled(T_i, R_i, C_i, C_i.value) \in labelled\_DFG$ do
            $def\_use \leftarrow Vertex_labelled(T_i, R_i, C_i, C_i.value)$ foreach
             $varible \in C_i$ do
                $sub\_path \leftarrow def\_clear\_path(p_use, def\_use, variable)$
                $pathadd\_feasible\_preamble(subpath)$
                if $path.siFeasible()$ then
                    $feasible\_paths.add(path)$
                else
                    $unfeasible\_paths.add(path)$
```

### 3.3.6 Step 4: Data Selection

Once the feasible/unfeasible paths are selected and associated to the coverage element, they form an Abstract Test Case. If the path is feasible, it means there exists a succession of variable assignments that ensures each transition along the path is satisfied. If the path is unfeasible, it means no such succession exist. However, in the context of MC/DC that eventuality has to be tested as well. The goal of this step to find the variable assignments needed in case of feasible paths. The constraint solver works on the deciding satisfiability of the system of equations along the path. This has already been done in the previous step as it is the basis for selecting the path. A Path is feasible if the system of equations derived from it is satisfiable. Data selection is simply finding and selecting one of the solutions.

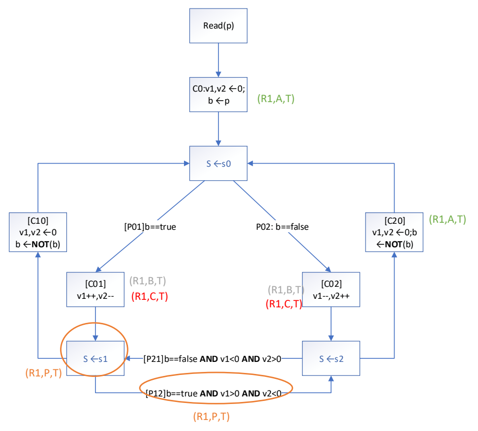
We use a temporal logic to express the succession of the predicates [47, 49]. For unfeasible paths, no “selection” is needed if test should fail no matter what the input is. However, we should make sure that the Decision and Conditions take the proper valuations as is specified in the coverage element (or MC/DC table row). We truncate the path up to the transition with Decision. The path should be feasible up to that point and thus the data selected is mainly to satisfy MC/DC, which bring us to the final step. Figure 3.11 (a, b, and c) shows the application of Step 3 (path selection) and Step 4 (data selection) on the example of Figure 3.7.

### 3.3.7 Step 5: Coverage Analysis

After the generation of test sequences, test criteria coverage needs to be analyzed. This step is important for test sequences that aims at satisfying the MC/DC criterion. In the case of MC/DC, there is also the need to analyze the satisfaction of its requirements. The goal of this step is twofold: verifying the requirement coverage and the MC/DC (or structural coverage if the generation has been parameterized with additional coverage criteria). This is a preliminary step that precedes the execution of the test cases. We run a symbolic execution along the path to calculate two metrics:

- Requirement Coverage = $\frac{Number\ of\ Concrete\ Test\ Cases\ generated\ with\ a\ "Pass"\ Verdict}{Number\ of\ coverage\ Elements}$
- MC/DC Coverage = $\frac{Number\ of\ feasible\ and\ unfeasible\ Concrete\ Test\ Case\ generated}{Number\ of\ coverage\ Elements}$

This step is performed independently from the other steps. One of its goals is to verify that the test suite indeed achieves 100% MC/DC coverage, and the other is to ensure forward requirement traceability (by construction) and backward traceability between the test cases and the LLR (by labelling).



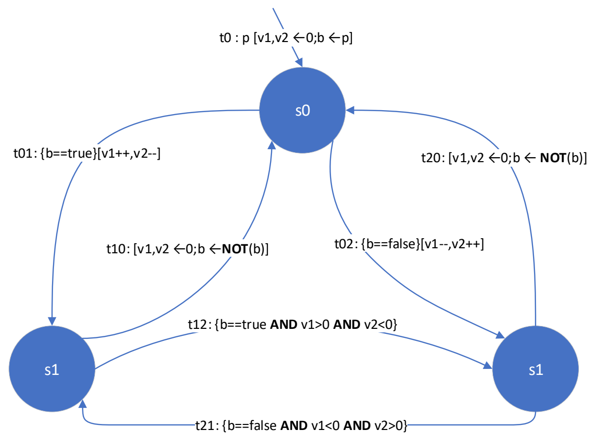
- min-def-use(A==T) = 3
- min-def-use(B=T) = 1, min-def-use(C=T) = 1 [C01]

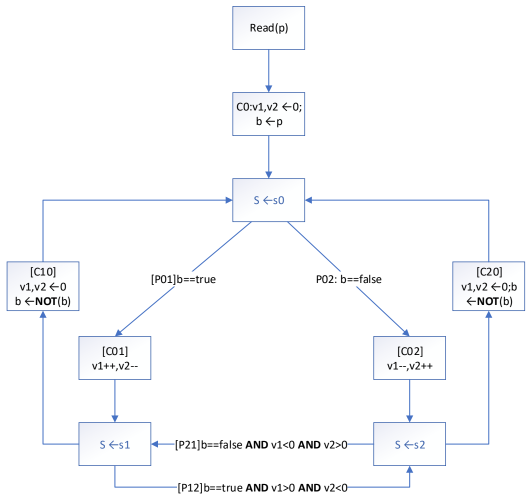
- We form a def- clear-path* between [C01] and and [P12]

- Vertex [C01] is selected.
- We link either [C20] or [C0] to a [C01] (both are feasible depending on the value of "p" which will be selected at step 4)

- Add feasible preamble and feasible postamble w.r.t. all the variables starting with the one(s) that had min-def-use

a


t0 : p [v1,v2 ←0;b ←p]
s0
t01: {b==true}[v1++,v2--]
t20: [v1,v2 ←0;b ← NOT(b)]
t10: [v1,v2 ←0;b ←NOT(b)]
t02: {b==false}[v1--,v2++]
t12: {b==true AND v1>0 AND v2<0}
s1
s1
t21: {b==false AND v1<0 AND v2>0}

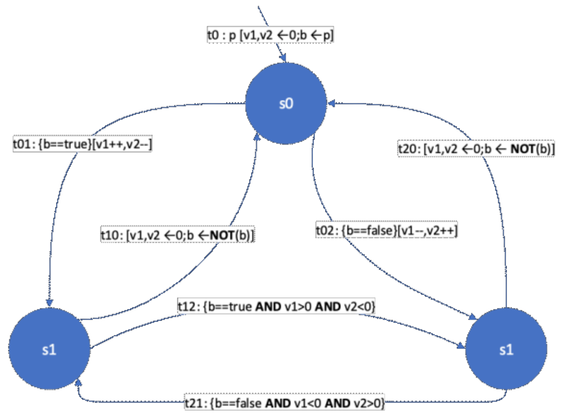


Selected Feasible Path (after adding feasible preambles and postambles) is:
t0-t01-t12-t20-t0

b

- Step 4: Selecting Test Data
  - Finding a solution for the system of equation
  - Symbolic execution may be used to decides on values for the parameters

- Example
  - Read(p)
  - Then: v1, v2 <- 0; b <- p
  - Then: b==true
  - Then: v1--, v2++
  - Then: P21


t0 : p [v1,v2 ←0;b ←p]
s0
t01: {b==true}[v1++,v2--]
t20: [v1,v2 ←0;b ← NOT(b)]
t10: [v1,v2 ←0;b ←NOT(b)]
t02: {b==false}[v1--,v2++]
t12: {b==true AND v1>0 AND v2<0}
s1
s1
t21: {b==false AND v1<0 AND v2>0}

c

Figure 3.11: Application of Steps 3 and 4 from graph rewriting of Figure 3.7

## 3.4 Architectural View

The main module in our approach is the Test Generation Module. It implements the main routines of our approach. We supplement it with two auxiliary modules: a Data Module, and a Graph Operations Module. Figure 3.12 outlines the high-level view of the architecture MC/DC-TGT. In this section, we justify the technical decisions as we outline the function and information's exchange for each module. In general, we preferred Java open source libraries whenever possible. The tool is designed so that those libraries could be substituted for others as long as they serve the same theoretical functions (e.g. graph rewriting using attributed grammar).

The Graph Operations module is dedicated to frequently used, general purposed, graph operations. Its goal is to ensure maintainability and reconfigurability of the algorithms. The data module retrieves user input, constructs, manages data, and provides proxies to external libraries involved in creating and transforming the different graphs. The Data Module is open to the rest of the MC/DC-TGT in read-only mode. In the upcoming sections, we detail each of the submodules, their functions, the information and data they exchange and/or modify, as well as when they are used.

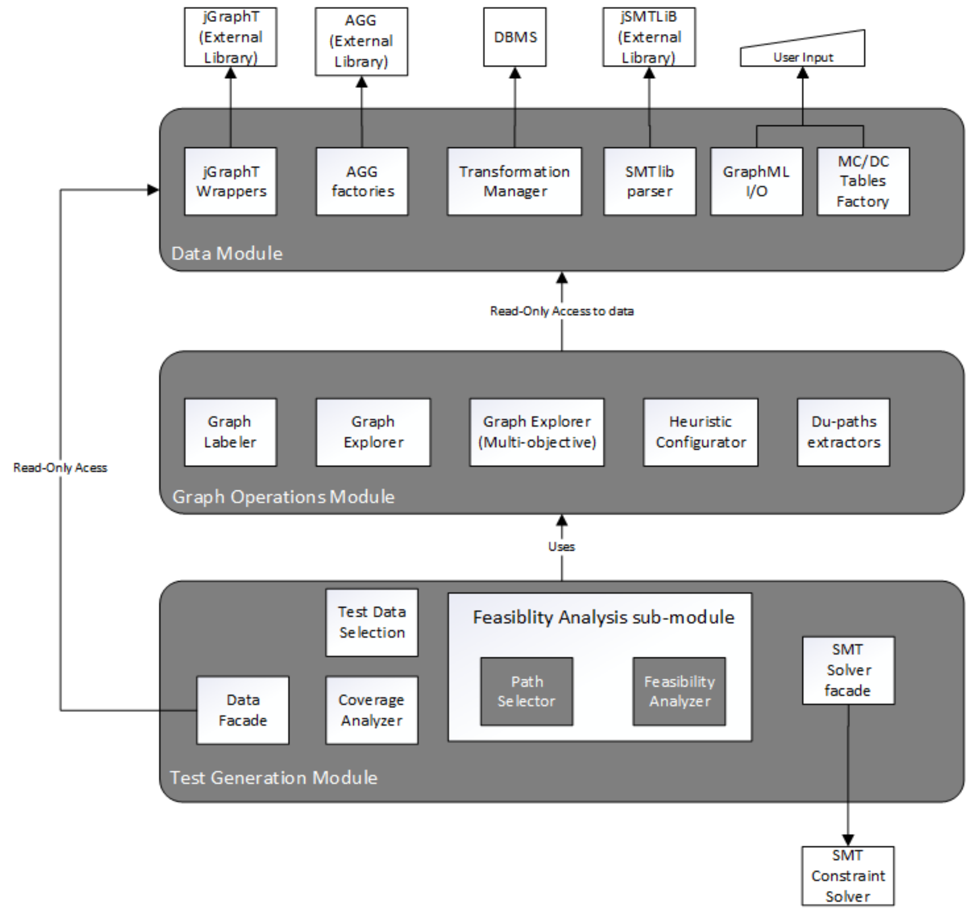


Figure 3.12: EFSM test generator architecture

### 3.4.1 Test Generation Module

The test generation module implements major steps of the methodology such as preparation of path selection, path selection, test data selection and coverage analysis, as described in process view. The three main submodules are treated as chain of responsibility pattern. The Data Facade submodule is a proxy to the Data module and provides a read-only access to all the data structures. The Path selection submodule has two sub-submodules of its own that collaborate in selecting the feasible paths. It takes a collection of MC/DC tables and labeled EFSM, DFG, and CFG as issued from Step 2 (preparation for path selection) and produces a collection of feasible and unfeasible paths. The path selector uses the graph explorer, multi-objective graph explorer, and du-path extractors to select candidate paths that satisfy the MC/DC Row. Concurrently, the feasibility analyzer module provides feasibility verdicts with the help of an SMT solver. Calls to the SMT solver are done using the SMT solver facade.

The test data selector is responsible for Step 4 (test data selection) also uses the SMT solver since data selection is akin to choosing a solution to the SMT equation across the selected feasible path. The Coverage Analysis module is responsible for generating coverage data according to the formulas from Section 3.3.7. It needs access to the labeled graphs and the test suite.

### 3.4.2 Data Module

The data module is composed of proxies to external libraries and tools. It is composed of AGG factories, jGraphT wrappers, a Transformation Manager, a GraphML I/O handler, a SMTlib parser, and MC/DC tables' factory. The input data for the MC/DC-TGT tool are the test criteria given as a simple parameter, MC/DC Tables and EFSM. The other two inputs are not as simple to handle. The first module is the MC/DC factory, which parses user input into instances of the class. The second Module is a GraphML I/O parser. We chose the GraphML [50] format for its simplicity and versatility as it can express both the pseudo-graph structure (EFSM) and graph structures (DFG, CFG). The "output" part of the module is necessary for persistence and logging purposes.

The graph factory module is responsible for creating the graph objects. It proxies the jGraphT library [52] and creates objects compliant to the graph grammars detailed in graph rewriting. The obtained objects are as described in the Data View section. While EFSM is received through user input, DFG and CFG are generated by the Graph rewriter module which implements the process via the AGG library [53]. As graph rewriting is being carried, records are made to a database through the Transformation Manager.

The last module is another proxy to an external library: jSMTlib [54], a parsing library for SMTlib expressions. Guard elements are created using this library. It is also used for generating SMTlib scripts that are fed into the SMT constraint solver in feasible path selection and test data selection.

### 3.4.3 Graph Operation Module

This module separates the graph-related operations from the test generation itself. Each submodule is implemented as a strategy pattern, enabling for a freedom of varying the subroutines used on the graphs. The test generation module only needs access graph operations sub-module from the data module. The graph operation module is dedicated to exploring various graph with search algorithms and path extractors. It is composed of Graph Labeler, Graph Explorer with simple algorithm, Graph Explorer with multi-objective search algorithm, Heuristic Configurator, and Du-path extractor.

## 3.5 Data View

Due to the large amount of data manipulated in this approach, and the increasing complexity of information therein after each step, it is necessary to persist relevant data. We need to keep track of how data was generated, as the information used in the approach is spread between the different graphs and the MC/DC tables. To that end we create a "transformation records" data structure, which will be generated as the graph rewriting step takes place. Each transformation records entry is a result of a rule application. The transformation records simplify the graph-related operations as well. In addition to describing each of the data structures in this section, we will explain how traceability, in terms of DO-178C, is assured. The data will be represented by UML class diagrams, except for the transformation records, which is a database table.

### 3.5.1 MC/DC Tables

The MC/DC truth tables are the "central" data structures for many control decisions in numerous algorithms of our approach. An erroneous representation compromises the reliability of our tool. It is recommended to opt for a simple and lightweight structure to avoid errors and overhead associated with traversal and search in traditional table structures. In the context of this research, MC/DC tables are provided externally. Their validation is outside the scope of our research, but we allow for extending the structure to include validating them internally. Figure 3.13 shows the class diagram of the MCDC/data structure. An MC/DC table is associated with a

Decision. Each row is a tuple R =< P, C >, where P Boolean value which represents the outcome the Decision takes when each Condition is given a value. C is the vector of those values. The table has N + 1 rows, where N = #(C).

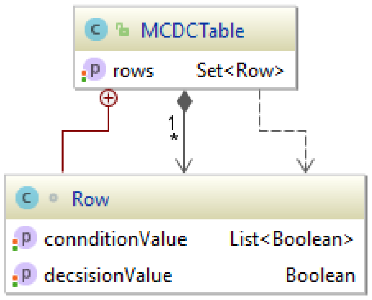


Figure 3.13: MC/DC table class diagram

### 3.5.2 State Graphs and Rule Graphs

The relationship between rule graphs and their state graphs is homologous to the one between classes and their instances in OOP. The proper description for state graphs would be object diagrams. However, we will narrow this subsection to describing the classes of (Figures 3.14 to 3.16), i.e. we will only cover the rule graphs. Having used attributed grammar to express the type graphs, describing them in object form is a simple one-to-one transformation. Each type is a class, each subtype is a subclass, and each attribute is a property/field (depending on the desired encapsulation).

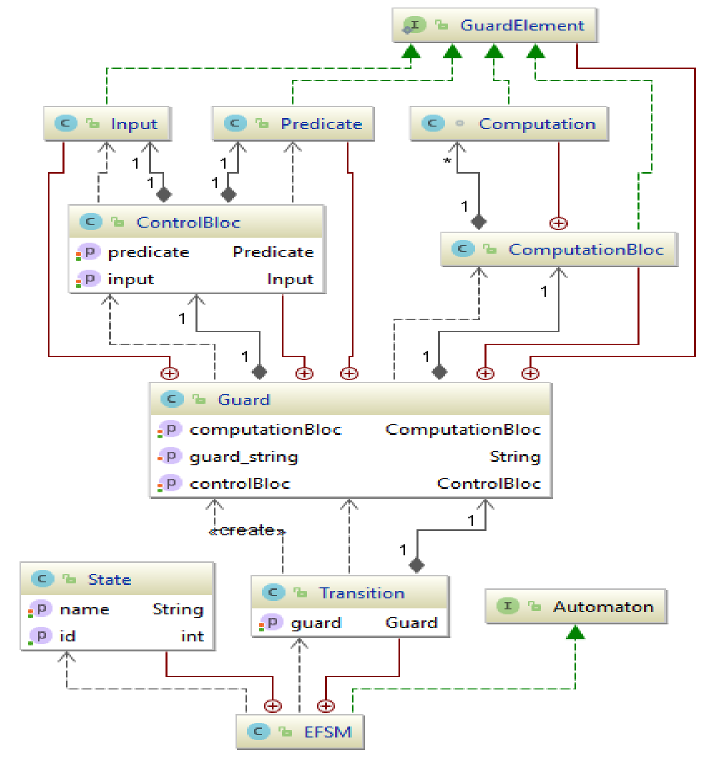


Figure 3.14: Guard

These classes use the JGraphT library [51] which offers a flexible way to manipulate graph structures. It is especially convenient for us as it offers routines adapted to pseudo-graphs such as the EFSM structure. The design of the classes is in conformance with the rule graphs provided in graph rewriting. We opted for nested classes for constructs such as states in an EFSM, to restrict the existence of such object outside of an already defined EFSM.

The guard construct is an exception to this design choice, as the class proved too complex to be implemented as a nested class. Furthermore, direct references to guards are preferred as we need to be able to manipulate the guards without added overhead of lookup through the parent EFSM (see Figure 3.14). Another contributing factor to this decision is that the MC/DC tables are associated with predicates - an element of the guard; and as mentioned previously, MC/DC tables in the context of this research are externally defined. Such data need to be instantiated before the construction and validation of the EFSM.

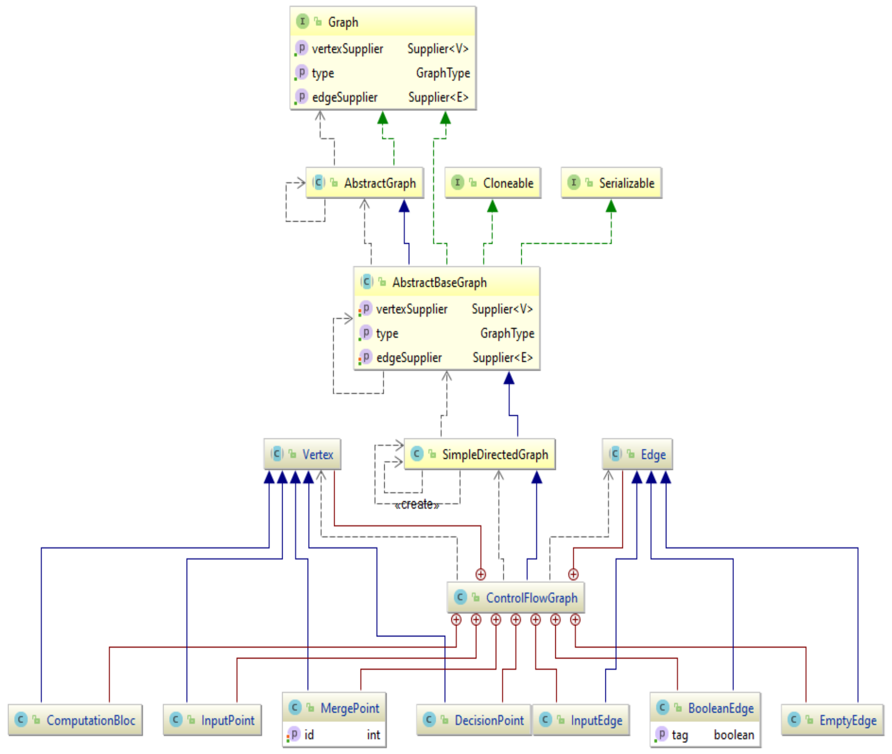


Figure 3.15: Control flow graph

### 3.5.3 Transformation Records

The complexity of the data structures and algorithms involved in this approach means that it would be useful to keep a detailed record of the results of the rule applications from the graph rewriting. Aside from being mandatory for V&V, they also simplify the Steps 2 to 5. Recording the direct references to graph morphism images will help us bypass search algorithms using object properties.

We use a static database as the transformation is done only once per graph type. Any change to the EFSM requires rerunning the graph rewriting from scratch, which means dynamic data bases have no use in our case. Once the graph rewriting is completed, access to the transformation records database is read-only. And since graph rewriting is a finite process, we favor consistency and availability of data above scalability. As such, a relational database is the recommended option.

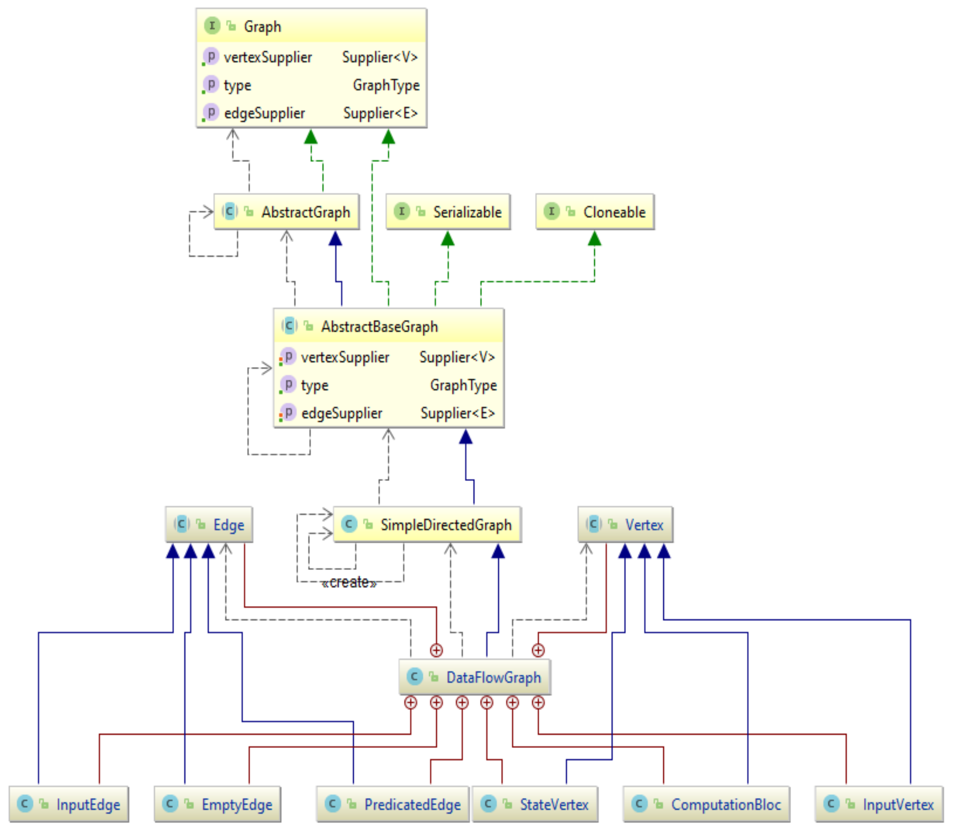


Figure 3.16: Data flow graph

The key entries are constructed using references to the EFSM constituents used in the rule morphism. Each entry corresponds to a rule application. The columns are populated with the references to resulting elements from the target graph. We use the same database for both the DFG and CFG transformation, which will inherently give a link between the two graphs, even though the transformations are done separately. Querying a transition from the EFSM returns both a Data Flow Subgraph and a Control Flow Subgraph as per preparation for path selection, thus bypassing an algorithmic search on the graphs. Using the same database will put those subgraphs on the same row. As such, moving between the DFG and the CFG is more efficient. This “direct” link between the DFG and the CFG reduces the complexity of representing the MC/DC tables on the data structures and adheres to the design choice. In addition to the functional contribution, these transformation records allow us to ensure traceability as per the DO-178C standard.

## 3.6 Conclusion

In this chapter, we addressed the tool design of automatic test sequence generation based on model for the satisfaction of MC/DC and du-path criteria. MC/DC is a requirement for avionics software certification according to DO-178C for level A. We followed the IEEE standard to describe the 4 views: the context view, the process view, the architecture view and the data view. The design of the proposed test generation tool is being developed for educational purposes. The developed techniques use constraints solving, handle cycles and uses multi-objectives search algorithms. Several existing tools have been integrated to solve some aspects of the problem. There is room for exploring an integrated set of coverage criteria and techniques to lower the risk of state explosion problem and enhance the efficiency of the generated tests cases. We also addressed the issue of validation in terms of compliance with DO-178C requirements and traceability as per the RTCA standard.

# Chapter 5

# Discussion and Conclusion

Quality assurance is the way to ensure the quality software that conforms to its specification and the required standards. Testing is the preferred activity in avionics industry and is mandatory by the standards [1, 57, 58]. In fact, the avionic software needs to be thoroughly tested before deployment. Exhaustive testing is not always feasible. Until now, the avionics industry is generating test cases manually. This technique requires significant resources in term of cost and manpower. Model-based testing is now explored by avionics industry to achieve automation of test case generation. This work is part of this exploration work.

In this thesis, we addressed the tool design of automatic test sequence generation based on model for the satisfaction of MC/DC and du-path criteria. MC/DC is a requirement for avionics software certification according to DO-178C for level A. The design of the proposed test generation tool is being developed for the need of avionics industry and for educational purposes. The proposed approach uses constraints solving, handles cycles and uses multi-objectives search algorithms. Several existing tools have been integrated to solve some aspects of the problem. There is room for exploring an integrated set of coverage criteria and techniques to lower the risk of state explosion problem and enhance the efficiency of the generated tests cases. We also addressed the issue of validation in terms of compliance with DO-178C requirements and both ways traceability as per the RTCA standard. This work has been published in conference and Journal paper [59,60].

The proposed methodology in this thesis can be used to test complex avionics software. Possible extensions of this methodology for future work includes:

- ✓ More validation, coverage assessment and metrics.
- ✓ Integration of more sophisticated model checking algorithms with the testing activity to check intelligent and autonomous components; and
- ✓ Integration of test and verification results analysis in a unified framework.

# References

[1] http://www.rtca.org. RTCA/DO-178C (2011) "Software Considerations in Airborne Systems and Equipment Certification", December 13, DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, DO-331

[2] *ARP4754A* Guidelines for Development of Civil Aircraft and Systems. SAE International.. *S–18 (2010).*

[3] G. Zoughbi, L. Briand, Y. Labiche, "Modeling safety and airworthiness (RTCA DO-178B) information: conceptual model and UML profile", Journal of Software & Systems Modeling, Volume 10, Issue 3, pp. 337-367, 2011

[4] J. Rushby, "New Challenges in Certification for Aircraft Software", Proceedings of the 9th ACM International Conference on Embedded Software, pp. 211-218, 2011, www.csl.sri.com/users/rushby/papers/emsoft11.pdf

[5] S. Rapps, E. Weyuker, Selecting software test data using data flow information, IEEE Trans. Softw. Eng. (1985) 367–375.

[6] R. Dssouli, K. Saleh, El M. Aboulhamid, A. En-Nouaary, C. Bourhfir: Test development for communication protocols: towards automation. Computer Networks 31(17): 1835-1872 (1999)

[7] C. Bourhfir, R. Dssouli, E.M. Aboulhamid, in: Automatic Test Generation for EFSM Based Systems, Technical Report IRO 1043, University of Montreal, 1996.

[8] R. Yang, Z. Chen, Z. Zhang, B. Xu, EFSM-based test case generation: sequence, data, and oracle, Int. J. Softw. Eng. Knowl. Eng. 25 (4) (2015) 633–667. (© World Scientific).

[9] R. Dssouli, A. Khoumsi, M. Elqortobi, J. Bentahar, Testing the control-flow, data-flow and time aspects of communication systems: a survey, Book Chapter in Advances in Testing Communication Systems, Atif Memon, Ed. 1, v.17, ISBN 978-0-12-812228-0, Elsevier, 2017 pp. 95-155.

[10] T. Su, C. Zhang, Y. Yan, L. Fan, G. Pu, Y. Liu, et al., Towards Efficient Data-flow Test Data Generation, 2019, [online] Available: http://arxiv.org/abs/1803.10431.

[11] J. J. Chilenski and S. P. Miller, Applicability of modified condition/decision coverage to software testing, Software Eng. J. 9 (1994), no. 5, 193-200. https://doi.org/10.1049/sej.1994.0025.

[12] T. Ayav, Prioritizing MCDC test cases by spectral analysis of boolean functions. Softw Test Verif Reliab. 2017;e1641. https://doi.org/10.1002/stvr.1641

[13] Ackermann, C.: MCDC in a nutshell, Fraunhofer CESE, Maryland USA, 2006

[14] Prestschner, A.: Compositional generation of MCDC integration test suites, Elsevier Science B.V, 2003

[15] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, A Practical Tutorial on Modified Condition / Decision Coverage, no. May. 2001.

[16] J. Wang, B. Zhang, and Y. Chen, "Test case set generation method on MC/DC based on binary tree," vol. 8783, no. Icmv 2012, p. 87831P, Mar. 2013.

[17] J. R. Chang and C. Y. Huang, "A study of enhanced MC/DC coverage criterion for software testing," Proc. - (a) (b) 79 Int. Comput. Softw. Appl. Conf., vol. 1, no. Compsac, pp. 457–464, 2007.

[18] R. Bloem, K. Greimel, R. Koenighofer, and F. Roeck, "Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall," VALID 2013, Fifth Int. Conf. Adv. Syst. Test. Valid. Lifecycle, no. c, pp. 1–6, 2013. 2013

[19] A. Haque, I. Khalil, and K. Z. Zamli, "An Automated Tool for MC / DC Test Data Generation," 2014 IEEE Symp. Comput. Informatics, Kota Kinabalu, Sabah, Malaysia, 2014.

[20] Z. Awedikian, K. Ayari, and G. Antoniol, "MC/DC Automatic Test Input Data Generation," in Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, 2009, pp. 1657–1664. 2009

[21] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," 2013 35th International Conference on Software Engineering (ICSE), 102-111, 2013

[22] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," DTIC Document, Tech, no. April. 2001

[23] P. Ammann, J. Offutt, and H. H. H. Huang, "Coverage criteria for logical expressions," 14th Int. Symp. Softw. Reliab. Eng. 2003. ISSRE 2003., 2003.

[24] M. A. Salem, K. I. Eder, "Novel MC/DC Coverage Test Sets Generation Algorithm, and MC/DC Design Fault Detection Strength Insights", 2015 16th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2015.

[25] Jun-Ru, C., & Chin-Yu, H. 2007. "A Study of Enhanced MC/DC Coverage Criterion for Software Testing." In Proceeding of 31st Annual IEEE International Computer Software and Applications Conference, Beijing, 2007, pp-457-464.

[26] Sartaj, H., Iqbal, M.Z., Jilani, A.A.A., Khan, M.U. (2019). A Search-Based Approach to Generate MC/DC Test Data for OCL Constraints. In: Nejati, S., Gay, G. (eds) Search-Based Software Engineering. SSBSE 2019. Lecture Notes in Computer Science(), vol 11664. Springer, Cham. https://doi.org/10.1007/978-3-030-27455-9_8

[27] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, in: Automatic executable test case generation for extended finite state machine protocols, Proceedings of the the 10th International IFIP Workshop on Testing of Communicating Systems, Jeju Island, Korea, 1997, pp. 75–90.

[28] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, in: A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols, IWTCS, 1998, pp. 275–290.

[29] C. Bourhfir, E. Abdoulhamid, F. Khendek, R. Dssouli, Test cases selection from SDL specifications, Comput. Netw. 35 (6) (2001) 693–708.

[30] Mounia Elqortobi, Warda El-Khouly, Amine Rahj, Jamal Bentahar, Rachida Dssouli: Verification and testing of safety-critical airborne systems: A model-based methodology. Comput. Sci. Inf. Syst. 17(1): 271-292 (2020)

[31] H. Ural, B. Yang, A test sequence selection method for protocol testing, IEEE Trans. Commun. 39 (4) (1991) 514–523.

[32] H. Ural, A.W. Williams, in: Test generation by exposing control and data dependencies within system specifications in SDL, FORTE 1993, 1993, pp. 335–350.

[33] X. Li, T. Higashino, M. Higuchi, K. Taniguchi, Automatic generation of extended UIO sequences for communication protocols in an EFSM model, in: in: 7th International Workshop on Protocol Test Systems, Tokyo, Japan, November, 1994, pp. 225–240.

[34] R.M. Hierons, T.H. Kim, H. Ural, in: Expanding an extended finite state machine to aid testability, Proceedings of the 26th Annual International Computer Software and Applications, Oxford, UK, 2002, pp. 334–339.

[35] W.E. Wong, A. Restrepo, Y. Qi, in: An EFSM-based test generation for validation of SDL specifications, Proceedings of the 3rd International Workshop on Automation of Software Test, Leipzig, Germany, 2008, pp. 25–32.

[36] J. Yao, Z. Wang, X. Yin , X. Shi , J. Wu, "Reachability Graph Based Hierarchical Test Generation for Network Protocols Modeled as Parallel Finite State Machines", 22nd International Conference on Computer Communication and Networks (ICCCN), (2013) 1-9.

[37] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verif. Reliab. 14 (2) (2004) 105–156.

[38] P. Coward, Symbolic execution and testing, Inf. Softw. Technol. 33 (1) (1991) 53–64.

[39] L.A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. Softw. Eng. 2 (3) (1976) 215–222.

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

[41] J. Zhang, C. Xu, X. Wang, in: Path-oriented test data generation using symbolic execution and constraint solving techniques, Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, 2004, pp. 242–250.

[42] S. Fujiwara, G. V. Bochmann, F. Khendec, M. Amalou, A. Ghedamsi, "Test Selection Based on Finite State Models", IEEE Transaction on Software Engineering, June (1991) 17 (6) 591-603.

[43] J. Li and W. Wong, "Automatic test generation from communicating extended finite state machine (CEFSM)-based models," in Proceedings of 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC (2002), 181–185.

[44] W. E. Wong, and Y. Lei, "Reachability Graph-Based Test Sequence Generation For Concurrent Programs", Int. J. Soft. Eng. Knowl. Eng. International Journal of Software Engineering and Knowledge Engineering 18.06 (2008) 803-822.

[45] Rozenberg, Grzegorz (1997), Handbook of Graph Grammars and Computing by Graph Transformations, Volumes 1-3, World Scientific Publishing, ISBN 9810228848.

[46] Ehrig, H.; R. Heckel; M. Korff; M. Löwe; L. Ribeiro; A. Wagner; A. Corradini (1997). "Chapter 4. Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach". In Grzegorz Rozenberg (ed.). Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific. pp. 247-312. CiteSeerX 10.1.1.72.1644. ISBN 978-981-238-472-0

[47] Barrett, C., Stump, A., & Tinelli, C. (2010, July). The smt-lib standard: Version 2.0. In Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England) (Vol. 13, p. 14).

[48] Bersani, M. M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., & San Pietro, P. (2010, September). Bounded reachability for temporal logic over constraint systems. In 2010 17th International Symposium on Temporal Representation and Reasoning (pp. 43-50). IEEE.

[49] Finkbeiner, B., & Schewe, S. (2007, November). SMT-based synthesis of distributed systems. In Proceedings of the second workshop on Automated formal methods (pp. 69-76).

[50] Brandes, U., Eiglsperger, M., Lerner, J., & Pich, C. (2013). Graph markup language (GraphML) (pp. 517-541).

[51] Michail, D., Kinable, J., Naveh, B., & Sichi, J. V. (2020). JGraphT—A Java Library for Graph Data Structures and Algorithms. ACM Transactions on Mathematical Software (TOMS), 46(2), 1-29.

[52] Yano, T., Martins, E., and de Sousa, F. L., (2011): MOST: A Multi-objective Search-Based testing from EFSM, in Proc. 4th International Conference on Software Testing, Verification and Validation Workshops, IEEE Computer Society, Berlin, Germany, (2011) 164-173.

[53] Löwe, M., & Beyer, M. (1993, June). AGG—an implementation of algebraic graph rewriting. In International Conference on Rewriting Techniques and Applications (pp. 451-456). Springer, Berlin, Heidelberg.

[54] Cok, D. R. (2011, April). jSMTLIB: Tutorial, validation and adapter tools for SMT-LIBv2. In NASA Formal Methods Symposium (pp. 480-486). Springer, Berlin, Heidelberg.

[55] Kalaji, A.S., Hierons, R.M., and Swift, S., (2009): Generating feasible transition paths for testing from an extended finite state machine (EFSM)", International Conference on Software Testing Verification and Validation, ICST, pp.230–239.

[56] D. Hedley, M.A. Hennell, in: The causes and effects of infeasible paths in computer programs, Proceedings of the 8th International Conference on Software Engineering, London, UK, 1985, pp. 259–266.

[57] Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D. (eds.) Proceeding of 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. Communications in Computer and Information Science, vol. 433, pp. 1–18. Springer (2014)

[58] C. Efkemann and J. Peleska, Model-based testing for the second generation of integrated modular avionics, in Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW ’11, Washington, DC, USA, 2011, IEEE Computer Society, pp. 55–62.

[59] A. Rahj, M. Elqortobi, J. Bentahar, R. Dssouli, Test Generation Tool Design For Modified Condition/ decision Coverage: Model Based Approach, International Journal of Computer Science and Applications, ©Technomathematics Research Foundation Vol. 18, No. 1, pp. 1 – 25, 2021.

[60] M. Elqortobi, A. Rahj, J. Bentahar, R. Dssouli, Test Generation Tool for Modified Condition/Decision Coverage: Model Based Testing. SITA 2020: 38:1-38:6