Validation and Verification of Safety-Critical Systems in Avionics

Mounia Elqortobi

A Thesis

In the department of

Concordia Institute for Information Systems and Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy in Information Systems Engineering (CIISE)

at Concordia University

Montreal, Quebec, Canada

June 2023

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mounia Elqortobi

Entitled: Validation and verification of safety-critical systems in avionics

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

Doctor of Philosophy in information systems engineering (CIISE)

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

Signed by the final examining committee:

____Chair
Dr. Onur Kuzgunkaya

____External Examiner
Dr. Hanifa Boucheneb (Polytechnique Montreal)

____ Examiner
Dr. Juergen Rilling (CSE)

____ Examiner
Dr. Nizar Bouguila

____ Examiner
Dr. Amr Youssef

____Thesis Supervisor
Dr. Jamal Bentahar

Approved by

____

Dr. Zachary Patterson, Graduate Program Director

June 5th, 2023

____

Dr Mourad Debbabi., Dean of Faculty

Abstract

Validation and Verification of Safety-Critical Systems in Avionics

Mounia Elqortobi, PhD
Concordia University, 2023

This research addresses the issues of safety-critical systems verification and validation. Safety-critical systems such as avionics systems are complex embedded systems. They are composed of several hardware and software components whose integration requires verification and testing in compliance with the Radio Technical Commission for Aeronautics standards and their supplements (RTCA DO-178C). Avionics software requires certification before its deployment into an aircraft system, and testing is mandatory for certification. Until now, the avionics industry has relied on expensive manual testing. The industry is searching for better (quicker and less costly) solutions.

This research investigates formal verification and automatic test case generation approaches to enhance the quality of avionics software systems, ensure their conformity to the standard, and to provide artifacts that support their certification.

The contributions of this thesis are in model-based automatic test case generations approaches that satisfy MC/DC criterion, and bidirectional requirement traceability between low-level requirements (LLRs) and test cases.

In the first contribution, we integrate model-based verification of properties and automatic test case generation in a single framework. The system is modeled as an extended finite state machine model (EFSM) that supports both the verification of properties and automatic test case generation. The EFSM models the control and dataflow aspects of the system. For verification, we model the system and some properties and ensure that properties are correctly propagated to the implementation via mandatory testing. For testing, we extended an existing test case generation approach with MC/DC criterion to satisfy RTCA DO-178C requirements. Both local test cases for each component and global test cases for their integration are generated. The second contribution is a model checking-based approach for automatic test case generation. In the third contribution, we developed an EFSM-based approach that uses constraints solving to handle test case feasibility and addresses bidirectional requirements traceability between LLRs and test cases. Traceability elements are determined at a low-level of granularity, and then identified, linked to their source artifact, created, stored, and retrieved for several purposes. Requirements' traceability has been extensively studied but not at the proposed low-level of granularity.

Acknowledgments

I would like to thank my supervisor Dr. Jamal Bentahar for supporting me through this long process, and through the highs and lows of my progress, and Drs. Warda El-Khouly and M. El-Menshawy for their collaboration and useful comments. I would like to also thank my colleagues Amine Rahj and Amine Laraj and all the members of our research team including the representatives of our industrial partners.

I would like to thank all members of my examining committee for their time and dedication.

I especially want to thank my family for all the emotional support, late nights, weekends, and for allowing me to take the time in the past years to try and fulfill, to the best of my abilities, this academic goal I have set for myself.

This work started in 2016 and was partly funded in part by NSERC DG and NSERC/CRD CMC CS Canada and Project CRIAQ AVIO 604, CRDPJ 463076-14.

Dedication

I dedicate this work to my family, especially my husband and my little girls!

Table of Contents

List of Figures

List of Tables

Initials and Abbreviations

FSM	Finite state machine
EFSM	Extended finite state machine
CEFSM	Communicating extended finite state machine
LLR	Low lever requirement
HRL	High level requirement
LGS	Landing gear system
UML	Unified modeling language
MC/DC	Modified condition / decision coverage
CFG	Control flow graph
DFG	Data flow graph
IUT	Implementation under test
FIFO	First in first out
CSP	Constraint satisfaction problem
UIO	Unique input output
ATP	Automated theorem proving
MAS	Multi-agent system
CTL	Computation tree logic
CTLCC	Computation tree logic for conditional commitments

Chapter 1

1. Introduction

In this research, we address the issue of software quality assurance for avionics software systems and its compliance with RTCA DO178C [1]. Software quality assurance (SQA) is an important domain in the industrial environment. SQA is an umbrella of several techniques that includes software validation and verification. As Boehm outlined [2], the validation activity addresses the challenge "Are we building the right product", i.e., the software should do exactly what the user requires. The verification activity addresses the challenge "Are we building the product right", i.e., the software should conform to its specification. As software testing cannot demonstrate the absence of errors for most systems, the goal of testing is to maximize error detection and minimize the development cost.

Software formal verification is a very active research domain. Its main challenges are the modeling and specification of systems' properties, state explosion, and the lack of tool support that can handle real systems. Verification is used at the design level on more abstract system models. When a property has been checked, the assumption is that it is propagated properly at the implementation level. This assumption is not always valid. In the avionics industry, the software product needs to be certified according to RTCA standards [1]. Until now, testing has been the only means to validate avionics software systems. While testing is indispensable for all software development, it has been carried out independently from verification activity. Testing is labor-intensive and expensive. It can account for more than 50% of the total development cost. The Avionics industry is looking for ways to reduce the cost of testing and improve the effectiveness of tests by automating the entire testing process. In the last decade, we have seen many test automation tools and a rapid growth in their use. The availability of the tools is a result of decades of research in code-based testing and testing based on models. Despite this availability of tools, there are categories of complex systems that do not benefit from the current level of automation. There is a clear need for automatic testing tools that comply with avionics systems standards. In addition, any tool that is used in the development cycle needs to be qualified according to RTCA standards [1].

The testing process has many activities. The most challenging one is test case generation. This activity requires an adequate coverage criterion that can show the efficiency of the derived tests to discover errors and satisfy coverage criteria. For model-based testing, there is a need for models that can express the nature of systems such as avionics systems.

In the following subsection, we introduce the industrial research context, research motivations, problem statement and research objectives.

1.1. Context of research and motivations

This research work is part of an industrial project entitled CRIAQ/NSERC/ CMC & CS Canada AVIO 604 “Specification and Verification of Design Models for Certifiable Avionics Software” that started in 2016. As depicted in Figure 1, this project addresses the creation of design models, model-based testing, and verification techniques for avionics software certification. The project objectives are to: (1) specify, develop and verify software design models and more specifically create new UML profiles and Simulink design standards that suit avionics systems descriptions; (2) enable verification techniques and design by contract in the context of Model Driven Development; (3) develop low level requirements (LLR) testing techniques in conformity with avionics software standards DO-178C and DO-331 [1]; and (4) enable low level requirements-based automatic test generation. This research focuses on the 2nd, 3rd and 4th objectives.

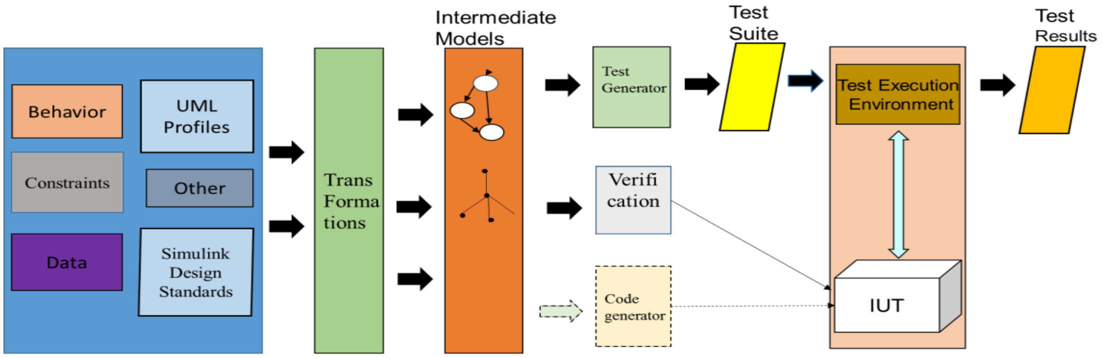


Figure 1: CRIAC NSERC CMC CS Canada/ AVIO 604 research project

1.2. Problem statement and research questions

It is imperative for safety-critical systems, such as avionics software systems, to ensure the validity of a system's requirements. A set of rigorous standards are in place, such as the RTCA DO-178C for avionics systems [1], as part of enforcing such requirements. Testing is the only means for the avionics industry to demonstrate compliance to the standard. There is a need for more research to improve the verification and validation of avionics systems. The focus now is on new approaches, processes and algorithms that can improve their quality.

The main point this proposed research addresses is how to automatically ensure the quality of safety-critical avionics systems and their compliance to the RTCA standards [1]. To break this down, how to verify and test them automatically, knowing that avionics software is an embedded complex software, real-time safety-critical, composed of multiple communicating components and subsystems, and has both discrete and continuous inputs? In this research, we address only a subset of these aspects. A subset of the research questions (RQs) addressed in this research is given below.

RQ #1: How to assess the compliance of avionics software systems to RTCA DO 178 C and its supplements [1]?

RQ #2: How to model systems and properties, verify systems' properties at the design level and ensure that the validated properties are correctly propagated and maintained in the downstream development cycle?

RQ #3: How to automatically generate test sequences from various models that satisfy a combination of required coverage criteria such as Modified Condition/Decision Coverage (MC/DC)?

RQ #4: How to generate test cases to cover systems requirements (Low Level Requirements (LLRs) and High-Level Requirements (HLRs)?

RQ #5: How to ensure bidirectional traceability?

RQ #6: How to perform test results analysis?

1.3. Research contributions

The actual costs of avionics software validation and compliance with standards are extremely high. Avionics industrial players are looking at the automation of testing and the use of formal verifications as possible ways to reduce costs, save time, increase safety, and achieve compliance with the standards. Several research projects are being funded in major countries where the avionics industry exists. Most of the projects involve academia, where research on formal models is conducted.

The first contribution of this research is an approach that integrates formal verification and testing. The verification will help at the design level. It requires a system model and the specification of the properties to be verified. The testing entails the automatic generation of test cases from low level requirements (LLRs), developing models, applying model transformations to handle different levels of abstractions and uses, and satisfying coverage criteria that are mandatory in the RTCA DO 178C standards [1]. As testing is mandatory for avionics software systems certification, it will be used to ensure that the verified properties are properly propagated in the development cycle. The proposed approach should also facilitate the conformity to DO-178C. We propose incremental approaches to address the different research questions. The second contribution is an approach that addresses multilevel test case generation using multi-agent systems technology and model checking. The approach can express the communication aspects that are not well captured in existing proposals. The third contribution is related to requirements coverage, test results analysis and traceability. The following is a list of the contributions and their objectives to answer the listed research questions.

First contribution: Develop a general approach that integrates model-based verification and testing.

Existing research works do not integrate verification and testing. They are considered as two different tasks and are not naturally linked in the entire development cycle. We developed an integrated approach where a system's properties are validated via testing to ensure that the verified properties hold in the implementation. In addition, we perform testing even where verification is difficult to achieve, as in the coverage of MC/DC.

> Objective #1: Develop a quality assurance approach that encompasses verification and testing and uses them in a complementary manner. *RQ #1*
>
> Objective #2: Develop a general model that can be utilized for the verification of properties and can be transformed for testing purposes. *RQ #1* and *#2*

Objective #3: Model, specify and verify some avionics properties using and extending existing verification languages and tools. *RQ #2*

Objective #4: Modify algorithms for automatic test generation with a combination of coverage criteria that comply with RTCA DO 178 C. *RQ #1, #2, #3, and #4*

Second contribution: Develop an approach for multi-level test case generation using model checking and multi-agent systems.

Objective #1: Develop an approach for multi-level testing that integrates formal verification, multi-agent systems and test case generation. *RQ #1*

Objective #2: Model avionics systems as multi-agent systems. *RQ #1 and #2*

Objective #3: Verify the properties using EISPL+ that extends ISPL+, the input language of the symbolic model checker MCMAS+. *RQ #2*

Objective #4: Generate test cases, use MCMAS+ model checker for intelligent systems to automatically generate counterexamples (i.e., traces showing the violation of given properties) and witness traces (i.e., traces that show how given properties are satisfied) interpreted into test cases that achieve different coverage criteria (e.g., state coverage, transition coverage, and path coverage). *RQ #1, #2, #3, and #4.*

Third contribution: develop an approach that generates test cases for MC/DC using constraints solving and that supports bidirectional traceability between low level requirements and test cases.

Objective #1: Handle path executability in test case generation that satisfies MC/DC. RQ #3, RQ #4

Objective #2: Address bidirectional traceability *at low level of granularity by allowing identification, creation, storage, and retrieval of traceability artifacts. RQ #5*

Objective #4: Develop an approach for test results analysis that complements the coverage analysis. *RQ #6*

1.4. Thesis organization

The thesis is composed of 5 chapters. Chapter 1 is the introduction and context of the thesis. Chapter 2 presents background information and a literature review. As part of the background information chapter 2 includes an overview of avionics standards, a literature review of test generation approaches, model-based testing, test data generation techniques, verification techniques and their relation to test case generation, requirements-based testing, and test results analysis. Chapter 3 presents the first contribution that addresses the integration of model-based verification and test generation of software avionics systems. Chapter 4 presents the second contribution, an approach for test case generation using model checking. Chapter 5 presents the third contribution, an approach to test case generation using constraints solving and that supports traceability between LLRs and test cases at a low level of granularity. This approach creates the necessary records to prepare bidirectional traceability and to produce traceability artifacts. The following activities are supported: traceability elements determination, identification, creation, storage, and retrieval during the test case generation process. Finally, we end this thesis with a conclusion and avenues for future work.

Chapter 2

2. Background information and literature review

Here we introduce the background information and the state-of-the-art pertaining to these research topics. As the research is in the context of the avionics industry, the main standard RTCA DO 178 B & C and their supplements are briefly presented in Section 2.1, followed by formal models, test coverage and conformance relations in Section 2.2. Sections 2.3, 2.4, and 2.5 are a review of existing work related to testing methods based on models such as Extended Finite State Machine (EFSM), Communicating EFSM (CEFSM), formal verification and test case generation.

2.1 Avionics standards RTCA DO-178 B & C

The avionics industry has developed a set of standards to prevent catastrophic events related to their systems. The RTCA DO-178B standard provides guidance for producing software for airborne systems that performs its function with a level of confidence in safety that complies with airworthiness requirements [1, 3]. Indeed, the RTCA DO-178B defines objectives for software life cycle processes, activities, and data with a strong emphasis on verifying the satisfaction of High- and Low-Level Requirements (HLRs, LLRs). Moreover, the complexity of airborne software has dramatically increased to the point that current validation techniques based only on testing are no longer adequate [3, 4]. These technical advances have led the avionics industry to introduce alternative means of validation and verification within a revised version of their standard, the TRCA DO-178C and its supplements DO-331, DO-332 and DO-333 [1]. The DO-178C standard [1] includes a supplement on formal methods called DO-333, where a formal method is defined as "a formal model combined with a formal analysis.". Conventionally, a model is formal when it has unambiguously and mathematically defined syntax and semantics. This allows automated and exhaustive verification of requirements using formal analysis techniques. Specifically, DO-333 provides three categories of formal analysis techniques: deductive methods such as theorem proving, model checking, and abstract interpretation. Today, formal methods are used in a wide range of application domains including hardware, railway, and aeronautics. RTCA DO-178C and its supplements have been successfully applied to the production of software systems at Dassault-

Aviation and Airbus [5]. Formal verification such as model checking and theorem proving are mainly used to verify properties at the design level. However, the verified properties may not be correctly propagated to the implementation of the final product, making it essential to fully test these properties. Figure 2 shows the testing procedure in RTCA DO- 178C.

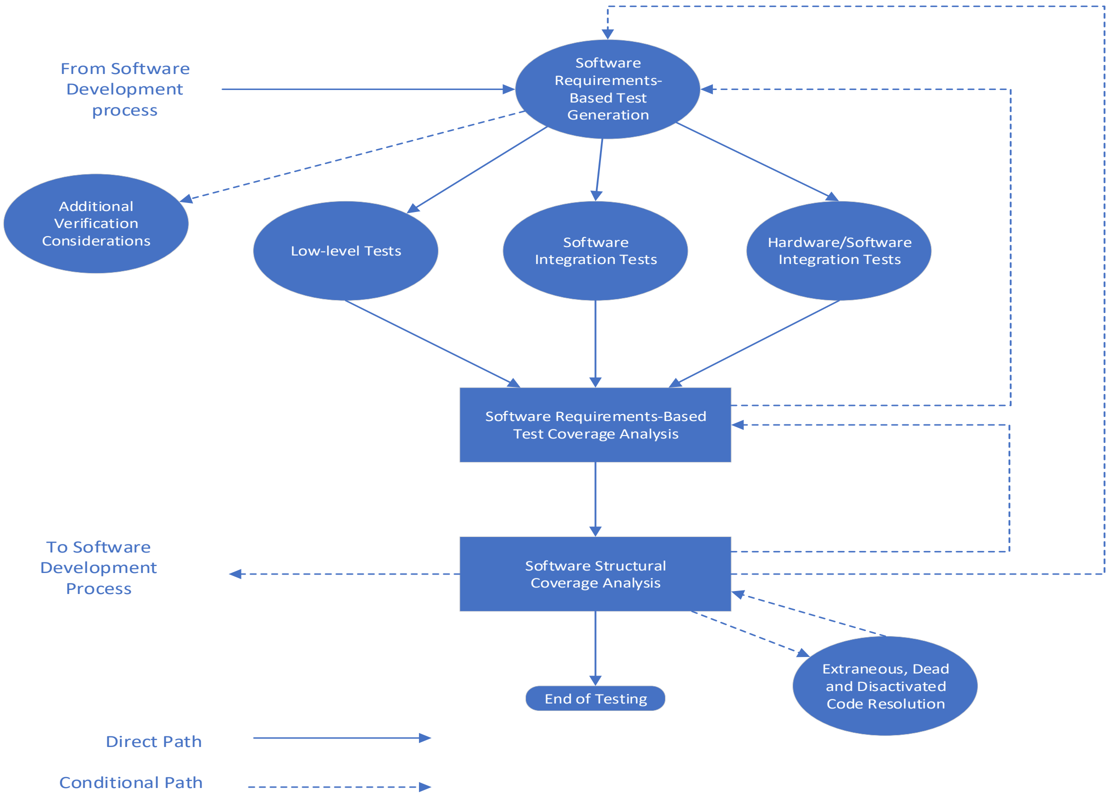

From Software Development process
Software Requirements-Based Test Generation
Additional Verification Considerations
Low-level Tests
Software Integration Tests
Hardware/Software Integration Tests
Software Requirements-Based Test Coverage Analysis
To Software Development Process
Software Structural Coverage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

Figure 2: Testing in DO-178C [1]

Our work is related to the last five objectives of RTCA DO 178C that are listed below, where * means that the objectives are partially met.

1. The executable code complies with the high-level requirements. *
2. The executable code complies with the specifications (low-level requirements). *
3. Test coverage of the high-level requirements is achieved. *
4. Test coverage of the specifications (low-level requirements) is achieved.
5. Test coverage of the executable code is achieved. *

The Software Levels defined by ARP4754, 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 examine 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 below.

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

B- Hazardous - Failure has a large negative impact on safety, on 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." [ref]

Figure 3 illustrates the tracing between certification artifacts, as required by the RTCA DO-178C standard. The red traces are required only for Level A, the purple traces are required for Levels A, B, and C, and the green traces are for Levels A, B, C, and D. No traces are required for Level E. Our work is related to level A.

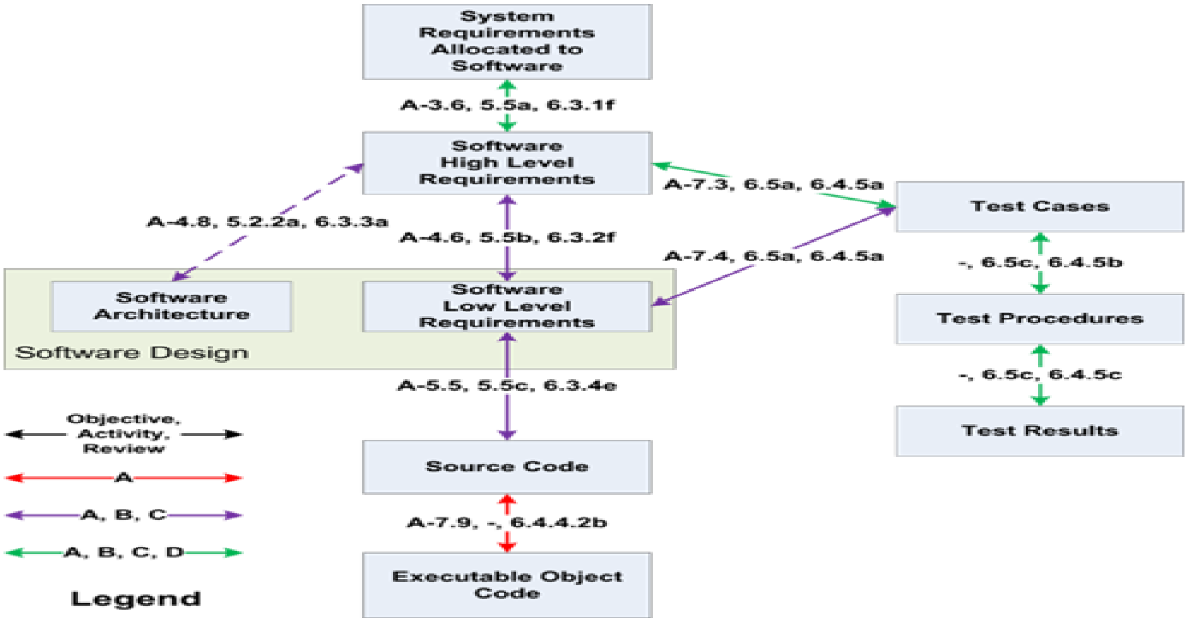


Figure 3: RTCA DO-178C required traceability [1]

2.2 Model-based testing

There are several models that we can select based on the aspects of the real system that we need to express.

The four basic aspects are control, which is often modeled by a Finite State Machine (FSM) or an input output automaton, the data aspect, often modeled by an Extended Finite State Machine (EFSM), the communication aspect, modeled by a Communicating EFSMs (CEFSM), and the time aspect, that is modeled by Timed FSM. Modeling one aspect is not sufficient to test systems as they exhibit several aspects at the same time. More complex models are needed to express the combination of aspects, such as CEFSM that combines the communication between components that are modeled as EFSMs, called CEFSMs. Similarly, there are systems composed of Timed EFSMs. The more aspects we compose, the more complex the model is, but the model becomes more realistic while still being abstract.

In this research, the target models are extended finite state machines (EFSMs) and communicating EFSMs (CEFSMs), as they express the control, data and the communication between components. In model-based testing, we try to establish a conformance relationship between a specification and its "implementations" [6, 7, 8, 9]. Models are extracted from the specification and used for test cases derivation purposes. More precisely, we establish a relationship between the model specification and an assumed abstract model of the implementation being tested. 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 all the announced assumptions. The model for testing the control aspect is limited; it cannot express data, communication, 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. We define an EFSM, a CEFSM, and a global system below.

Definition 1. An EFSM is formally represented as an 8-tuple < S, $s_{0,}$ I, O, T, A, δ, 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: $\delta: S \times A \rightarrow S$, and
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 2n-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; and
- F_i is a First In First Out (FIFO) list for C_i, i=1..n.

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 sets 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 its 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 2n-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 $\delta_i(S, \alpha)$ is defined; and
- *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 $m'_j = m_j\ \forall\ (j \neq i)$; and
- if *input* ≠ Ø and *output* = *b*, then $m'_i = W$ and $m'_k = m_k^b$.

Definition 5. A test case is composed as <preamble, target, postamble, 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 postamble is the sequence of transition that starts at the ending state of the target and ends at the final state; this may 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.

2.2.1 Test coverage criteria

The notion of coverage is important in test case generation. It characterizes the quality of a test case and test suites. 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 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 [10, 11], are not sufficient to ensure that a tested software satisfying a coverage criterion is error-free. The only way to ensure that software is error-free via testing is to perform exhaustive testing, which is often very expensive or impossible due to its 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 requirements coverage and MC/DC [12, 13]. Figure 4 shows the Rapps & Weyuker original structural coverage criteria augmented with the representation of the MC/DC branch [10, 12, 13]. Next, we focus on definition-use paths (du-paths) and MC/DC criterion. These two criteria are not comparable. They can be used separately or integrated in a test sequence generation algorithm to enhance test case quality.

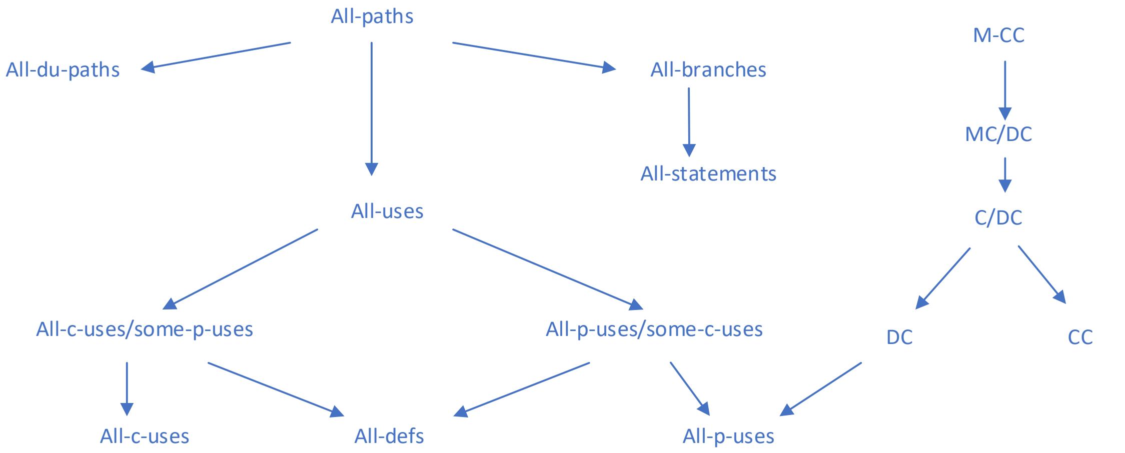


Figure 4: Coverage Criteria [10]

A du-paths criterion is a dataflow coverage criterion that links the definitions 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, such as an assignment and read input that modifies the value of the variable. A usage of a variable includes all operations that read the value of a variable without modification, such as computation use (C-use), 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 that is less strong but manageable than all-paths.

The objective of a 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 critical systems have decisions that need testing. Some of the specified decisions are complex and require specific techniques such the decomposition of expression to address them. An MC/DC criterion is stronger than condition and decision criteria. The satisfaction of the MC/DC criterion is required by the RTCA DO-178C standards of avionics systems. For more details, a tutorial that introduces the MC/DC criterion and summarizes the issues and challenges of its testing is available [13]. A decision is a Boolean expression 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 of 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 the Multiple-Condition Coverage criterion and the Condition/Decision Coverage criterion that have a lower number of test cases [1, 12, 13, 14. 15, 16]. MC/DC has been 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 MC/DC exist and all of them are dedicated to code testing [12, 13, 14. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. The existing techniques are based on one of the following four categories: Binary Trees [19], truth table for each Boolean expression [13, 15, 18, 24], n-cube graph [28], and constraints solving [27, 28]. Each of the proposed techniques, if not combined with another technique 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 to the number of variables. There are very few approaches to MC/DC test case generation for system specifications that consist of complex decisions. Most of them assume that each decision is independent. T. SU et al. [27] presented three different approaches, essentially based on binary trees and constraint solvers, to generate MC/DC test cases for decisions. Most of the proposed techniques do not address multiple decisions and cycles in test case generation. Very few semi-automatic test case generation tools exist, and all suffer from the above challenges.

In the following we review some relevant EFSM-based test generation techniques.

2.2.2 EFSM-based test generation approaches

Testing based on the EFSM has been studied extensively. The EFSM is a model that extends the FSM model with variables and predicates that appear within condition statements and statements. Test sequence generation is more complex in the EFSM and faces a state explosion problem that leads to incomplete coverage [8, 9, 30]. The data selection that is required is often undecidable and path feasibility/executability is not cost efficient.

The first comprehensive survey was published by Bourhfir et al. in 1997 [30]. In 2014, Yang et al. published a more recent and exhaustive survey entitled “EFSM-based Test Case Generation: Sequences, Data and Oracle” [9]. However, it does not address the testing of communication and time aspects. In 1997, Bourhfir et al. proposed an EFSM-based test sequence generation method that generates executable test sequences [31, 32]. 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 postamble to each du-path [33]. The executability of each path is verified in the fourth step. They used cycle analysis, symbolic execution, and Constraints Satisfaction Problem (CSP) techniques to solve the path executability problem. In the fifth and last step, 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 [37] criterion and multiple UIOs with Wp as the state identification method. More techniques are also used, such as search techniques, and handling executability issues exist [49, 50, 51, 52, 53, 54]. Table 1 below presents a summary and some comparison criteria of the relevant EFSM-based methods. A more recent and complete list can be found in the following surveys published by Yang et al. and Dssouli et al. [8, 9].

Table 1: EFSM-based test sequence generation approaches

(- means not given or not addressed)

Authors and Date	**Model Transformation**	**Coverage Criteria**	**Signature**	**Data Selection**	**Path Executability Technique**
H. Ural 1991, 1993 [34, 35]	EFSM to Control graph and Data Flowgraph	IO-df-chain	-	-	-
R.E.Miller 1992 [38]	EFSM to FSM (both Control flow Data flow)	all-def-obs paths	UIO	-	Change the value of influencing variable, backtracking. Problem of efficiency
C. Bourhfir 1997, 2001 [31, 32]	EFSM to Data Flowgraph	IO-df chains	M-UIO Wp	Random	Cycle analysis, Symbolic execution, CLP-BNR technique
R. M. Heirons 2002 [36]	SDL-EFSM to NF-EFSM to EEFSM/PEFSM	all-uses	-	-	Path splitting, state decomposition, predicate decomposition. Simplex algorithm
W.E. Wong 2008, 2009 [37]	EFSM	all-nodes, all-edges Hot spots	-	Symb. execution	Conflict detection Possibility to use CSP

2.2.3 CEFSM-based test generation approaches

There are many challenges to meet in concurrent systems testing [40, 41] Concurrent communicating machines have a distinct set of features, "such as communication, synchronization, and non-determinism" [42], features that amplify the testing complexity [43]. In addition, the body of knowledge available for testing parallel communicating systems is not as mature as it is for sequential systems. The semantics of concurrent communicating processes needs to be defined in terms of interleaving actions or true concurrency, and the communication mode must be defined in terms of synchronous communication with the possibility of blocking, or asynchronous communication, a non-blocking mechanism. Parallel communicating machines are known for their non-deterministic behavior and their race condition. Non-determinism is observed when we execute the same process with the same sequence of inputs and observe different behaviors, which means that different paths are being executed. In addition, the composition of deterministic processes may lead to non-deterministic behavior. Most of the techniques in testing concurrent systems use the composition of control graphs obtained by the transformation of the Communicating Finite State Machine (CFSM) or the Communicating Extended Finite State Machine (CEFSM). This composition is obtained by cross-product with interleaving semantics, which may lead to a state explosion problem. In fact, it is the same as the Reachability Graph (RG) that is used for reachability analysis and model checking.

The approaches for testing the CEFSM can be classified into three types: 1) the composition of all single modules to obtain one composite model and the application of existing EFSM test sequence generation methods; 2) the generation of test sequences for a single EFSM, known as local test sequences, and then generating global test sequences by the composition of local test sequences to cover the set of communicating transitions; and 3) the transformation of each EFSM into control and data flow graphs and the use of flow graph-based criteria to generate test sequences. In the following sections we review the case of a CEFSM that exchanges messages through channels (asynchronous communication mode) [42, 43 44, 45, 46, 47, 48]. Synchronous communication can be simulated by a buffer of size 1. The literature review relevant to this research and related to test generation techniques based on the CEFSM is summarized in Table 2.

Table 2: CFSM/CEFSM-based test sequences generation approaches

(- means not given or not addressed)

Research work	**Model transformation**	**Coverage Criteria**	**Signature**	**Test generation method**
Luo et al. [42]	CNFSM to NFSM	Fault coverage multiple faults	Wp	Wp
Lee et al. [43]	FSM to CFSM	all-transitions	-	Guided random walk. Observers.
Hierons [43]	CFSM	fault model	UIO	Heuristics

Research work	Model transformation	Coverage Criteria	Signatur e	Test generation method
				VRP
Bourhfir et al. [44]	CEFSM to partial reachability graph	all def-uses all-transitions	UIO Wp	Guided test case generation - uses communication transitions between modules
Li and Wong [45]	CEFSM to flow diagram	branch coverage transitions predicates	-	Incremental test case generation based on coverage
Wong and Lei [46]	CFSM to reachability graph	all-nodes all-edges	-	Hot-spot prioritization Topological sort
Yao et al. [47]	PaP EFSM to reachability graph)	def-use paths	Wp	Hot-spot prioritization Topological sort

2.2.4 Test data selection techniques

To test a system, both variables and parameters need values that must be selected from their domain definitions in a combinatorial manner, called the input domain. Selected data has the important role of stimulating the path and revealing any errors. Selected data should simultaneously satisfy all the predicates along the path for their feasibility (executability). Infeasible paths remain an important problem in test case generation that requires effective techniques [49, 50]. The difficulty is that the input domain that combines all the variables and parameter domains is too large or infinite to consider complete coverage. It is known that test data generation is an undecidable problem [51]. Several techniques have been explored for test data selection. One of these 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 [52]. Test data generation and selection techniques can be grouped in the following categories: symbolic execution [30, 31, 32, 53, 54, 55], random, mutation, linear regression to narrow intervals, and search-based techniques [56, 57, 58, 59, 60].

2.3 Formal verification and test case generation

Formal verification is a collection of algorithmic techniques that use static analysis based on mathematical transformations. It is defined as the process of formally establishing that a design satisfies a given specification or set of properties representing requirements. There are two categories of formal methods for formal verification: theorem proving and model checking. Theorem proving, also known as Automated Theorem Proving (ATP) or automated deduction, is part of the automated reasoning field and mathematical logic. It deals with proving mathematical theorems by computer programs. ATP is composed of procedures that can be used to check whether a given formula *F,* called the "goal", is a logical consequence of a set of formulas *N* known as the "theory" [61]. Model checking is another technique [62, 63, 64] that builds a finite state transition system and exhaustively explores it for property violations. If the model checker analyzes all reachable states and detects no violations, then the property holds. However, if the model checker finds a reachable state that violates the property, it returns a counterexample which is a sequence of reachable states beginning in a valid initial state and ending with the property violation. The model checker is used as an oracle to compute the expected outputs and the counterexamples it generates are also used as test sequences. Some of model checker approaches consider avionics systems using the landing gear system (LGS)) [65]; other approaches are more general. Some the approaches focus only on modeling and verification and others include the modeling, verification, and testing of critical avionics systems.

The following shows table 3 which summarizes the approaches that address modeling, verification, and testing, and table 4 which summarizes approaches that focus on the modeling and verification of avionics systems.

Table 3: Approaches focusing on modeling, verification, and testing.

Works	**Model**	**Verification tool**	**Logic/ language**	**Aspects**	**Approach**	**Issues**
Arcaini et al. (2014a) [66]	ASM Abstract State Model	AsmetaSMV	CTL	Communication & commitment not expressed	Transformation of ASM to NuSMV for verification	Too abstract -No communication
Dhaussy and Teodorov (2014) [67]		OBP Observer-based-Prover	Fiacre Properties in CDL	Output modeled as global variables	- Context -Problem is decomposed into small problems that are verified independently	The correctness of local properties does not infer the correctness of the whole system
Berthomieu et al. (2014) [68]	TTS Timed Transition System	Selt in Tina	Fiacre/ LTL/ TTS	Implicit communication between components	Pilot = a set of shared Boolean variables	-No explicit communication TTL and TTS -- -No accountability

Table 4: Approaches focusing on modeling and verification avionics systems (LGS)

Authors	**Model**	**Verification tools**	**Logic/ language**	**Test case generator**	**Approach**	**Issues**
Hansen et al., 2014 [69]	Specification in B method Refinment	ProB model checker BMotionStudio for visualisation	Event based B LTL	The ProB animator	Generate valid traces for regression testing	Only liveness properties LTL - No communication - No accountability
Arcaini et al. (2014b) [70]	ASM Abstract State Model	AsmetaSMV (transformation of ASM spec to NuSMV input)	CTL	Communication & commitment not expressed	Links between implementation (java) and its specification ASM, test the code against ASM model	- Too abstract CTL -No communication -No accountability

In the following chapter, we present our first contribution, an approach that integrates model-based verification and testing in the same framework. This work has been published as a conference paper [85] and its extension as a journal paper [71].

Chapter 3

3. Model-based verification and testing approach for safety-critical systems

In this chapter we address the issue of safety-critical software verification and testing, as they are key requirements for achieving RTCA DO-178C regulatory compliance for airborne systems. Formal verification and testing are often considered two different activities within the airborne standards, and they belong to two different levels in the avionics software development cycle. We propose an approach that integrates model-based verification and testing within a single framework. We address the verification and testing of parallel communicating agents based on formal models. In this work, properties are extracted from requirements and formally verified at the design level, while the verified properties are propagated to the implementation level during the development cycle and checked via testing. This approach is composed of six steps:

I. Modeling behaviors and specifying properties for formal verification at the design stage;

II. Performing verification using and extending existing tools;

III. Transforming the verification model to a testing model using model transformation refinement;

IV. Automatic test generation for testing individual agents (components) in their context (conformity);

V. Automatic test generation for the integration of all agents based on CEFSM; and finally,

VI. Checking that the verified properties are held at the abstract model representing the implementation level via testing.

For test case generation, we combined MC/DC, DU-path and UIO coverage criteria [10, 17]. The combination of these criteria enhances the efficiency of test cases. The results of formal verification and testing can be used as evidence for avionics software certification. The approach begins with formally modeling the avionics system from the given informal requirement specifications, producing an FSM-like model as described in Figure 5. This work was published in 2020 [71].

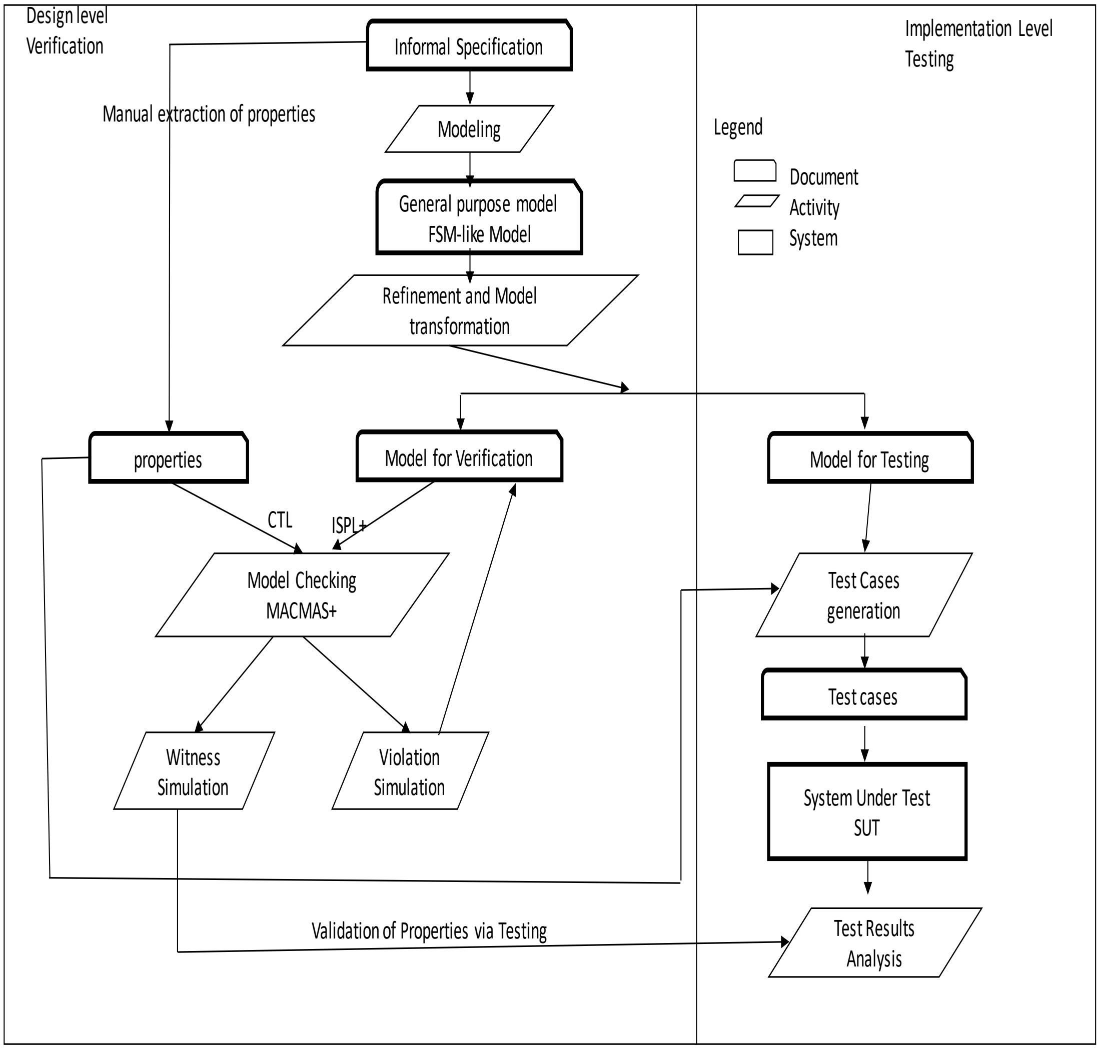


Figure 5: Overview of our approach [71, 85]

We assume that a correct informal specification exists. Next, the obtained model is refined and encoded using ISPL+ (an extended version of the input language of the symbolic model checker MCMAS+ introduced in [72]) to verify agent-based intelligent systems. We extract and express the system requirements in the form of temporal properties using Computation Tree Logic (CTL)

[73, 74]. MCMAS+ automatically checks whether the model satisfies the intended properties and graphically produces witness examples or counterexamples [63, 75]. The produced witness examples prove the satisfaction of properties, while the produced counterexamples guide designers to detect and repair design errors in the formal system model. In the validation/implementation level, our approach automatically transforms the formal models into a partial product of the Communicating Extended Finite State Machine (CEFSM) that uses our proposed algorithms to automatically generate abstract test cases. These algorithms and tools address the conformity of the implementation under test to Low-Level Requirements (LLRs). After assigning values to the required data sets, the generated test cases are transformed into concrete ones with respect to the expressed properties. The concrete test cases will be applied to the implementation under test. The Modified Condition/Decision Coverage (MC/DC) criterion is integrated into the test generation algorithm to satisfy the requirements of the RTCA DO-178C [1, 76, 77]. Finally, our approach analyzes the obtained test results and compares them with the produced witness-examples to validate our properties via testing.

The main limitation of this methodology is the computational complexity of the test case generation algorithm, which is NP hard. The methodology offers forward traceability by construction, but it does not address the issue of backward traceability. There is a need to extend it with graph exploration and test results analysis.

3.1 Modelling case study: Landing gear system (LGS)

Our case study, a landing gear system for an aircraft, was proposed by Frédéric Boniol and Virginie Wiels in [65] as a representative scenario for complex industrial needs. The case study is very rich, as it is not restricted to software and includes complex system modeling. The landing system is responsible for maneuvering the landing gears and attached doors. It consists of three landing packages situated in the front, right, and left part of the aircraft. Each landing package includes a door, a landing gear, and related hydraulic cylinders. A door can be opened or closed, while the gear can be retracted, extended, or maneuvered. The landing system is controlled by a software package and can be in two modes: normal or emergency. In outgoing and retraction situations, the normal mode is the default mode. The emergency mode is deployed to handle failure situations. This work only considers the outgoing sequence and its normal and emergency modes. The architecture of the system consists of three parts (see Figure 6): 1) a pilot part; 2) a mechanical part that incorporates the mechanical devices and three landing packages; and 3) a digital part that includes the control unit software. Regarding the pilot part, a pilot has a button switch at her/ his disposal with two positions: UP or DOWN. When the button switch goes from UP to DOWN, the outgoing sequence is initialized. The pilot has three lights in the cockpit that reflect the current status of the gears and doors. These lights are as follows:

- Green light indicates that "gears are locked down";
- Orange light indicates that "gears are maneuvering"; and
- Red light indicates a "landing gear system failure".

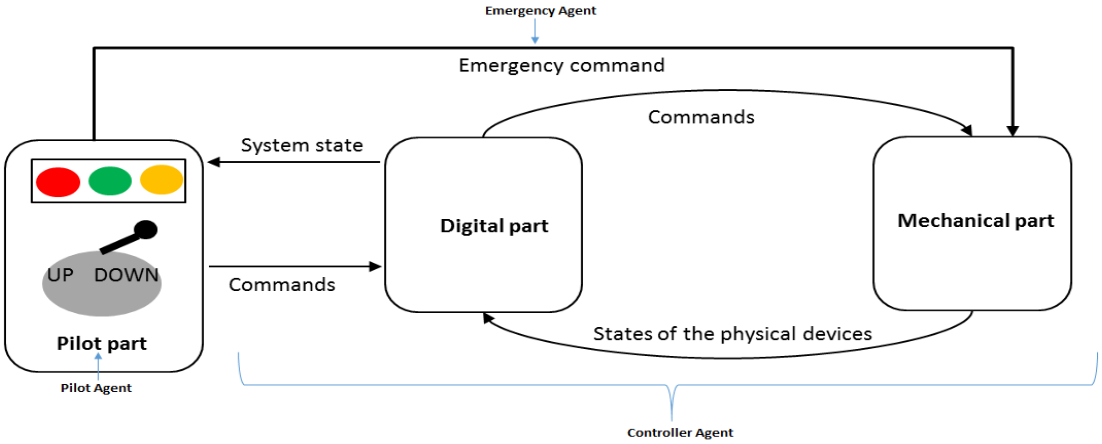


Figure 6: General architecture of the landing gear system

Before initializing the outgoing sequence, all the landing gears are locked in their up position and all the lights are off. In case of failure (i.e., the red light is on), the pilot manually pulls the mechanical handle to deploy the emergency hydraulic system. The expected consequence of this deployment is to lock the gears in the down location. When all gears are successfully extended and all accompanying sensors are valid, the green light must be lit. Regarding the mechanical part, the motion of landing gears and doors is performed by a set of hydraulic cylinders such that the cylinder position basically corresponds to the door or landing gear location. For example, when a door is open, the corresponding cylinder is extended. The hydraulic power of these cylinders is supplied by a set of electro-valves. The digital part sends an electrical order to activate each electro-valve. Notably, the three doors (and their gears) are controlled in parallel by the same electro-valve. The digital part plays an intermediate role between the pilot part and the mechanical part. Specifically, the software embedded in the digital part is responsible for controlling the gears

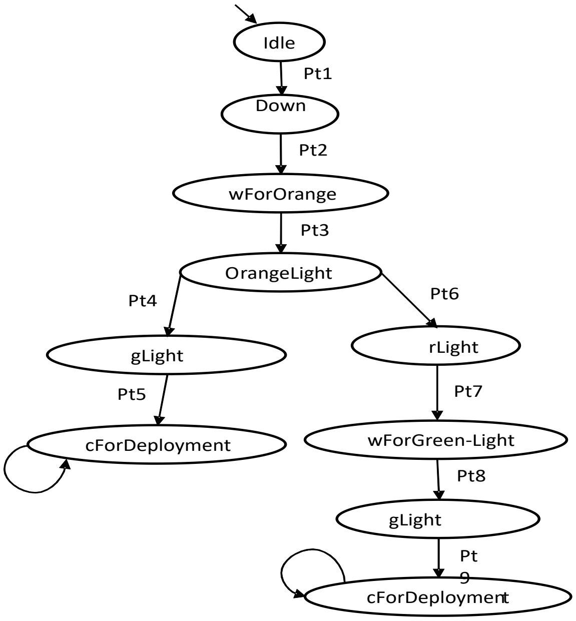
and doors, detecting anomalies, and informing the pilot about the status of the system through a set of lights. It also generates commands directed to the hydraulic system to open or close the doors and extend or retract the gears with respect to the values of employed sensors, and it captures the pilot orders.
In the following we focus on modelling the case study.

3.1.1 Modelling the landing gear system (***Step I***)

In this section, we show how our model M can formally model a landing gear system. In our modeling, we consider the normal and emergency modes of the landing gear system without going into low-level details regarding the mechanical devices of sensors and electro-valves. To realize our model, we introduce three agent machine models: the pilot agent machine model M_p models the behavior of the pilot part, the control unit agent machine model M_c models the behavior of the digital part, and the emergency agent machine model M_e models the behavior of the emergency system. Instead of adding another agent machine to model the behavior of the hydraulic cylinders, we depend on the status of the doors and gears to directly represent the status of the employed cylinders. This works because the description above states that the doors' cylinders are extended when the doors are open, and a similar relation holds between gears and their cylinders.

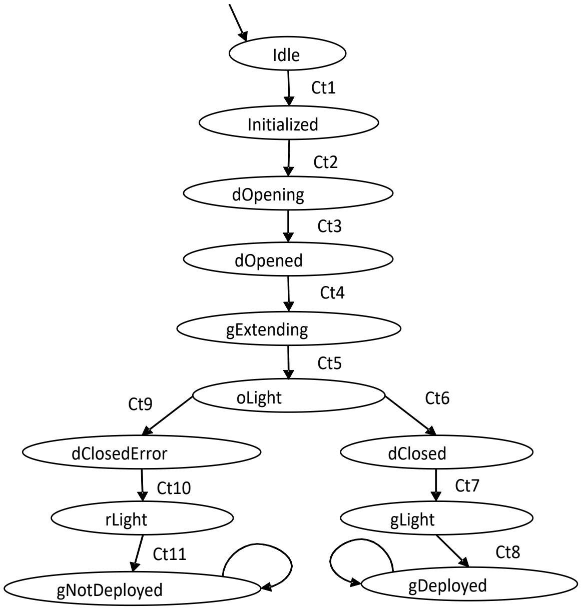
In the published case study paper, there are two types of requirements, and the authors classify them as strong and weak [65]. The weak requirements did not consider deadline constraints/time constraints. Although we selected the weak requirements, the time constraints are abstractly represented in our model where each transition takes one-time unit as in all standard abstracted temporal models.

Figures 7, 8, and 9 show the EFSM models of the pilot, control unit, and emergency agent machines, respectively. In each figure, we introduce the input and output of each transition in a tabular form where the symbols "?" and "!" refer to receiving and sending actions. The output of a transition can be directly assigned by shared and unshared variables when there is no explicit output action. Given that, it is easy to define the Boolean predicate of each transition using the conjunction operator between its input and its output.



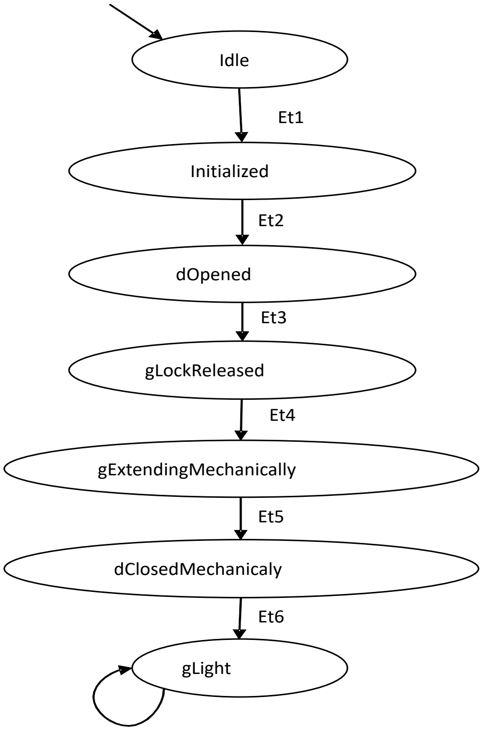
Label	Input, output & predicate
Pt1	? Landing-Specs(speed, distance) ! Press-Down-Button
Pt2	? Press-Down-Button-Ack ! Wait-For-Orange-Light
Pt3	? Orange-Light-On ! Orange-Light-On-Ack
Pt4	? Green-Light-On ! Green-Light-On-Ack
Pt5	? Confirm-Gear-Deployment ! Deployment-Status:=Success
Pt5	? Red-Light-On ! Red-Light-On-Ack
Pt7	? Confirm-Gear-Deployment-Error ! Initialize-Emergency-System
Pt8	? Green-Light-OnM_e ! Green-Light-OnM_e-Ack
Pt9	? Confirm-Gear-DeploymentM_e ! Deployment-Status:=Success

Figure 7: Pilot Agent Machine model, Mp [71, 85]



Label	Input, Output & Predicate
Ct1	? Press-Down-Button ! Press-Down-Button-Ack
Ct2	? Process-Received-Command ! Open-Gear-Doors
Ct3	? Open-Gear-Doors-Ack ! Outgoing-Gears
Ct4	? Outgoing-Gears-Ack ! Orange-Light-On
Ct5	? Orange-Light-On-Ack ! Close-Gear-Doors
Ct6	? Close-Gear-Doors-Ack ! Doors-Close-Success
Ct7	? Gears-Extended ! Green-Light-On
Ct8	? Green-Light-On-Ack ! ControlUnit-Disconnected
Ct9	? Close-Gear-Doors-Error ! Doors-Close-Error
Ct10	? Gears-NotExtended ! RedLight:=On
Ct11	? Red-Light-On-Ack ! ControlUnit-Disconnedted

Figure 8: Controller Agent Machine model, Mc [71, 85]



Label	Input, Output & Predicate
Et1	? Initialize-Emergency-System ! Open-Gear-Doors
Et2	? Open-Gear-Doors-Ack ! Outgoing-Gears
Et3	? Outgoing-Gears-Ack ! Verify-Gears-Position
Et4	? Verify-Gears-Position-Ack ! Lock-Doors-Mechanically
Et5	? Lock-Doors-Mechanically-Ack ! Green-Light-On
Et6	? Green-Light-On-Ack ! Gear-Status-Extended

Figure 9: Emergency Agent Machine model, Me [71, 85]

3.2 Verification of LGS properties (***Step II***)

To perform the verification, we introduce the MCMAS+ tool. We consider that each EFSM is an agent and CEFSM is a multi-agent system. This is a symbolic model checker that extends MCMAS, a model checker for Multi-Agent Systems (MAS) that uses Ordered Binary Decision Diagrams (OBDD) [63, 78]. MCMAS takes two inputs: a model description for the system to be verified and a set of properties specified by different logics, such as CTL and CTLC [79, 80]. The inputs of MACMAS are formatted by the ISPL language which is used to describe the communicating MAS to be checked and to encode the desired specifications. ISPL+ is a dedicated programming language for interpreted systems that formalize MASs (Fagin & Halpern, 1994) [81]. MCMAS+ automatically evaluates the truth value of the encoded specifications and produces counterexamples that can be analyzed graphically for false specifications. MCMAS+ can also provide witness executions for the satisfied specifications and graphical interactive simulations.

For clarity, we introduce the syntax of CTL that is given by the following grammar rules:

$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid EX\phi \mid EG\phi \mid E(\phi\ U\ \phi)$ where:

1) $p \in Ap$ (a set of atomic propositions) is an atomic proposition and E is the existential quantifier on paths;
2) X, G, and U are temporal operators standing for "next", "globally", and "until", respectively; and
3) The Boolean operators $\neg$ and $\vee$ are defined and used in the usual way.

To validate our model M (a composition of M_p, M_c, and M_e) we need to perform the review and tracing activities. As a first validation activity, we must review the model with the wide range of features implemented in the MCMAS+ graphical user interface [80]. This graphical interface highlights syntax errors, automatically displays content, and assists and supports text marking and formatting. After fixing all the highlighted errors, we have a clear and error-free encoding model. Tracing the activity allows us to track the behavior of the encoded model. The MCMAS+ tool offers an Explicit Interactive Mode. This tool starts with the initial state and offers all the transitions available at this state, and also offers the possibility to choose the transitions. After we select one of these transitions, the tool moves to the reachable state connected with the initial state by this transition and then displays the available transitions at the new state. This step allows us to evaluate whether the model is progressing as we intended. If an error is detected, we return to our encoding and update it. This process continues until we reach the end state. Then, we start again from the initial state and select another transition. Our graphical interface supports a new feature, which displays the whole model. By completing these two activities, we ensure that our encoding model exactly captures the intended behavior of the landing gear system. In fact, these two activities are key to ensuring that the model is correct; otherwise, errors in the design model could jeopardize the entire activity of the design formal verification using a model checking technique.

3.2.1 Model checking

According to the model checking technique, we must formally: 1) model the system underlying the verification process; and 2) express the requirements. The correctness of these requirements has been proven on the modeled system using MCMAS+. We have just shown how we complete the first activity.

For the second activity, we used the Computation Tree Logic (CTL) [73] supported by the MCMAS+ model checker tool [82] to express the following requirements:

$\phi_1 = AG(PressedDown) \rightarrow AF(GearsExtended \wedge DoorsClosed)$

$\phi_2 = \mathrm{EG(E(PressedDown\ U\ PressedDown\ \wedge\ GearsExtended}\ \wedge DoorsClosed))$

$\phi_3 = AF\ \neg\mathrm{E}(\neg\ \mathrm{PressedDown\ U\ (GearsExtended}\ \wedge\ DoorsClosed))$

$\phi_4 = A\mathrm{G}\ \neg(\mathrm{PressedDown}\ \wedge\ AG\ (\neg GreenLight))$

$\phi_5 = AF(GreenLight)$

In the case study [65], a set of requirements is presented with respect to the normal mode. The requirement called R11bis states that "when the command line is working (normal mode), if the landing gear command handle has been pushed DOWN and stays DOWN, then eventually the gears will be locked down and the doors will be seen closed". We expressed this requirement in the three different CTL formulae ϕ_1, ϕ_2 andϕ_3.

The first formula (ϕ_1) can be read as follows: along all computation paths through all states, when the button is pressed down, then along all computation paths in the future, the gears will be extended, and the doors will be closed. The second formula (ϕ_2) can be read as follows: there exists a computation path such that in all its states the gears will be not extended, and the doors will be not closed until the button is pressed down. The third formula (ϕ_3) can be read as follows: along all computation paths in the future, the gears will be not extended, and the doors will be not closed if the button has never been pressed down before. The CTL formula ϕ_4 expresses the safety requirement, which plays an important role in avoiding a bad situation. This bad situation in the fourth formula can be read as follows: the button has been pressed down and along all paths, the green light is never lit. The last CTL formula ϕ_5 expresses the liveness requirement and can be read as follows: along all computation paths, the green light can be eventually lit. The quantifier ranging over all computation paths ("A") enables us to check the status of both normal and emergency modes. For example, the liveness formula allows us to check the status of the good thing ('green light') that will happen eventually in each mode. All these formulas are evaluated to true on the model M using MCMAS+. Therefore, our design model is error-free, and it is strong, as it achieves the safety and liveness requirements required in both modes. We can also report some statistical results, such as that the execution time of verifying these formulas is 0.298 seconds and the memory consumed is 6 Megabytes.

3.3 Model-based test generation approach

The goal is to generate, starting from a model, a set of test cases for the verified properties, apply them to the implementation under test and then analyze the test results to get a verdict (pass, fail or inconclusive). The main objective here is to demonstrate that the verified properties are properly propagated from the design level to the implementation level, and that they hold true within the Implementation Under Test (IUT). This demonstration requires model transformation, local and global test sequence generation, testing and test results' analysis. The approach both verifies the properties at the design level and demonstrates their validity at the implementation level using global test sequences, allowing the satisfaction of RTCA DO 178C by generating local test sequences with the required coverage criteria. In addition, we extend the set of Du-paths to include additional paths to satisfy MC/DC criterion.

3.3.1 Model transformation (***Step III***)

Using our test case generation techniques with well-defined coverage criterion such as the MC/DC [76, 77], we transform the verification model into a testing model. The notion of shared variables used in our verification model can be transformed into input parameters in the EFSM model. The interaction mode considered here is message passing. The discussion about whether to use one model or two distinct models can take place. The solution for avoiding the use of a single model is to manually extract one model for verification and one model for testing. In this case, two different quality assurance groups should be involved, and the two models should cover the same set of requirements to satisfy the need for independence between verification and testing activities. The model transformation can show the relationships between two models, such as inclusion or equivalence.

Model-to-graph transformations is also used in our approach. We transform an EFSM to control and monitor data flow graphs. The transformation can be done using graph rewriting techniques [83, 84] using grammar for each of them.

3.3.2 Local test cases generation (***Step IV***)

To generate global test sequences for a system that is composed of several communicating agents, we start by generating test cases that are local to each EFSM agent. There are several techniques to automatically generate test cases at the module level, but any selected technique needs to take into account MC/DC criterion that is mandatory for avionics software certification according to RTCA DO 178C [1]. In this research, we developed two different techniques that generate local test cases, the first extends Bourhfir et al. work with MC/DC procedures (first approach) [71, 85]; the second uses constraints solving (third approach) [86, 87].

Test Generation Algorithm Satisfying MC/DC Criterion

The proposed test generation algorithm generates feasible test sequences covering MC/DC and Du-paths criteria. To integrate MC/DC criterion to Bourhfir et Al. technique [32], we need to pin-point in the algorithm the parts necessary to identify all Du-paths. For each element in the preamble list, we add a set of possibilities to satisfy the MC/DC criterion. This binary set will represent the possibilities for each information item that influences a decision [83, 84].

As the proposed approach extends the Bourhfir test generation technique, we start by presenting the high-level view of the algorithms utilized.

Boughfir et al. approach starts with an SDL specification in Normal Form, generates data flow, identifies all nodes, generates preambles and postambles, generates paths for all definitions' use paths, makes paths executable using inputs and symbolic execution, and then it generates test cases for all Dupath criteria. It also uses state identification methods known as UIOs (unique input output methods).

The algorithms shown in appendix A are extracted from Bourhfir et al.: A test case generation approach for conformance testing of SDL systems. Computer Communications, vol.24, no.3-4, pp.319–333, 2001 [32].

Extension of the approach to cover MC/DC criterion

There are several ways to approach this issue, for example: 1) create a standalone procedure executed at the end that will have access to all the paths generated initially, and generate additional ones to satisfy the MC/DC criterion; 2) integrate the MC/DC while the paths are being generated to cover all feasible paths and identify non-executable paths related to a decision; or 3) analyze the non-executable paths (fail paths) and choose the ones satisfying the MC/DC criterion, using a hybrid approach based on approach number 2. The third approach is the one we selected. The algorithm that generates local test sequences is outlined below:

(1) Transform the EFSM to graphs (control and data flow graph G) using graph rewriting.

(2) Expand the graph by an expansion mechanism (if needed), use state decomposition and graph splitting to handle MC/DC criterion.

(3) Identify all nodes with predicates.

(4) Select input values for each input parameter that can affect the control flow.

(5) Generate executable Du-paths according to data flow graph G and remove redundant paths. Append the state identification sequence and postamble (return to the initial state) to each Du-path to form a complete test path.

(6) Check test path executability; if non-executable, use cycle analysis to make it executable, discard if non-executable. This is done during the generation of a path.

(7) Verify if there are uncovered transitions, add test paths to cover them.

Handling the MC/DC criterion in the EFSM Test Generation algorithm is explained in the following five steps.

1. Define a variable of binary values called a vMCDC using its truth table. This variable will take the values that will help satisfy the coverage criterion and will be used solely for MC/DC criterion satisfaction for test case generation.
2. In the test generation algorithm, add all possible values for the identified input parameters that satisfy the MC/DC criterion and that are not already covered by the algorithm in its original state.
3. Call a procedure that will analyze the discarded paths to ensure that they would not be involved in any MC/DC.
4. Use the vMCDC to analyze non-executable paths.

Algorithm EFTG (EFSM Test Generation)

1. Read an EFSM specification.
2. Generate the dataflow graph G from the EFSM specification.

3. Choose a value for each input parameter influencing the control flow, augment the scope to consider the possible values for MC/DC.
4. If the path is still non-executable, perform the Analyze-discarded-path (P) procedure.

Create the procedure Analyze-discarded-path (P). This procedure will use the binary variable vMCDC and evaluate the information of path P to determine if it should be removed or not.

Analyze-discarded-path (P)

1. Define a binary values (truth) table with accepted values for green-orange-red states.
2. For each variable, in every transition in the discarded path, compare the values with the binary table (truth table) for green-orange-red-gear.
3. If the values conform to the table, discard the non-executable path.
4. If they do not conform, add this path to the MC/DC list of conformances (use the same logic for executable paths and flag them for MC/DC satisfiability).

In the procedure executable-Du-path-generation, we add another loop to consider the vMCDC to identify the paths between transitions.

Procedure Executable-DU-Path-Generation (flowgraph G)

1. Incorporate the MC/DC variables from the vMCDC variables.
2. Generate all possible paths (call to Find-All-Paths (T,U, vMCDC) for each variable that has an A-use in T, and each transition U that has a P-use or a C-use.

We replace the procedure handle-executability to not discard non-executable paths and call it procedure handle-executability-MCDC. If a path is non-executable, it will not be removed but identified. This is rather complicated, as the algorithm is sound in making sure that all non-executable paths are confirmed twice as non-executable and are then discarded. Another possibility is to add a condition that allows us to identify from which variable a path has been defined. If it was from a vMCDC variable, then we will not remove the non-executable path. Satisfying MC/DC criterion will result in adding several non-executable paths corresponding to fail paths in a decision node. This step is needed to ensure that erroneous paths are handled correctly, which will control both the satisfaction of the properties and the alternatives triggered by glitches or possible malfunctions. All MC/DC related paths are identified for coverage computation.

Local test cases are generated for each component of the system and are modeled by an EFSM. To test the interaction between components, global test cases that will be used for testing the integration of systems components must be generated. This is known as the communication aspect, and it is often modeled as a CEFSM.

3.4 Global test sequences generation (***Step V***)

There are several approaches to generate global test sequences, some use a complete cross product of the interacting models representing components and generate global test cases, others use a partial product and are based on the work done by Bourhfir et al and Cavalli et al [30, 31, 32, 88. 89, 90].

In our work, since the system is composed of parallel components that communicate with each other, we propose a test generation technique for parallel communicating agents. The generation of test sequences starts with the verification model. We first model each agent in its context and then create a list of transitions for the communication between a pair of agents. We use a transition-marking algorithm that marks every transition involved in communication as an EFSM, along with its context. This technique generates local test sequences for each agent. Next, we compose the obtained EFSMs to build a global system M that is in fact a Communicating Extended Finite State Machine (CEFSM) (Figure 9). In this case study and for the sake of readability, the EFSMs are only a partial representation of a landing gear system (LGS).

Figure 10 describes the test generation process. To generate global test sequences, we first derive local test sequences for each EFSM representing an agent. Second, we obtain the communication graph from all EFSMs (see Figure 11). Third, guided by the communication graph, we obtain the global system, or the CEFSM. Finally, from the local test sequences and the CEFSM, we generate the global test sequences. The following sections detail the different activities of the test generation for the case study.

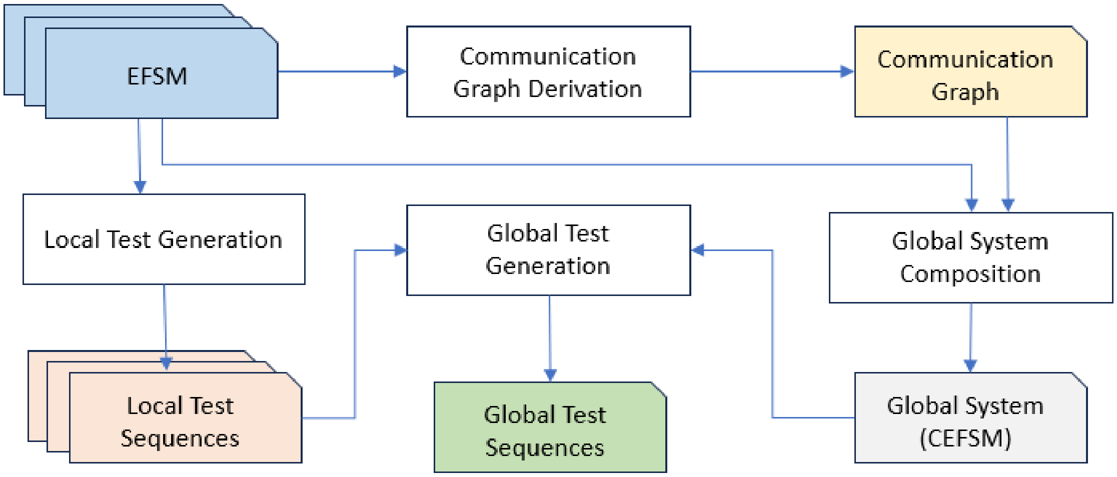


Figure 10: Test Generation Process

Activity 1: Communication Graph Derivation

To generate global test sequences for a global system M composed of several agents, we need to abstract an EFSM agent into an abstract state and identify the communication transitions and their parameters that are used for communication. The communication graph is an abstract graph that represents the interaction between the different EFSMs. For our case study, it is assumed that the communication between the machines M_p, M_c and M_e is two-way. Figure 11visualizes the communication graph with the representation of each agent model by an abstract state.

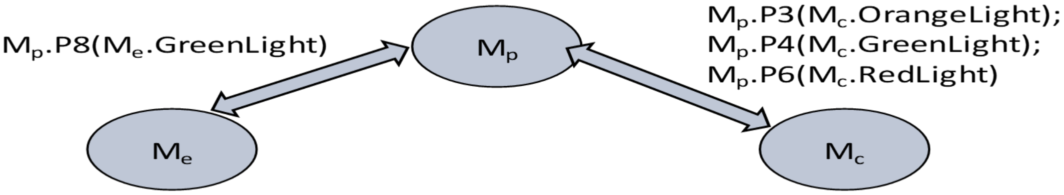

M_p.P8(M_e.GreenLight)
M_p
M_p.P3(M_c.OrangeLight);
M_p.P4(M_c.GreenLight);
M_p.P6(M_c.RedLight)
M_e
M_c

Figure 11: Communication graph representation

Activity 2: Global Model with Communication Points

Using the EFSMs (Figures 7, 8, and 9) and guided by the communication graph (Figure 11), we obtain (by partial composition) the global system model with its communication points (Figure 12). Figure 12 represents the composite system model M with its communication points, labels and transitions, and the input and output lists. We can see that the M_c and M_p agents start at the same time. It is in fact a parallel communication system. The transitions representing the communication among agents are shown in orange, green, and red to represent the landing gear system lights of the same color.

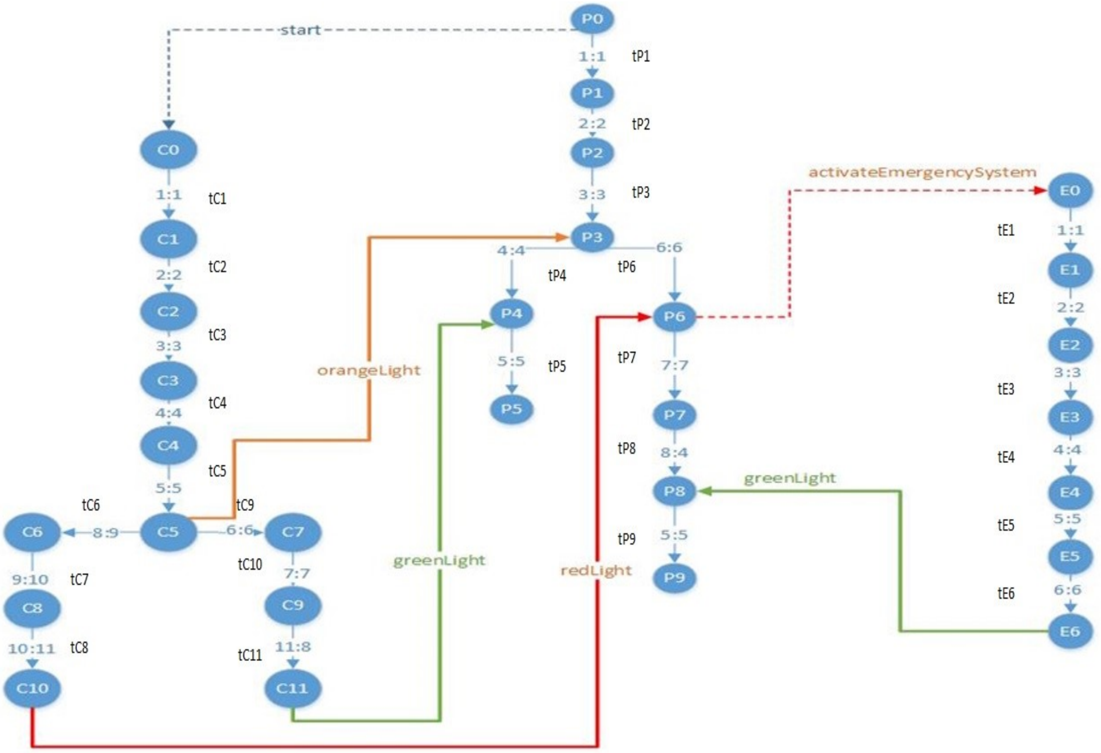


Figure 12: System model M: Composed of Mp, Mc, and Me with communication points.

Activity 3: Global Test cases generation

In this section, we briefly describe the test generation algorithm and its application to the case study. Three possible approaches can be applied. The first approach is to perform a cross product of all ESFMs agents and apply the local test case generation approaches. This is equivalent to a brute force type of algorithm and will incur a state explosion problem. The second approach that minimizes the state explosion problem constructs a partial product that is guided by the interaction points between agents. The third approach starts with the set of local test cases and a list of interaction points (communication/synchronization points). It takes each interaction point, composes all paths in backwards and forwards searches that contain this interaction point, and checks their feasibility and the satisfaction of MC/DC criterion. This approach also has a partial product of communicating agents for which local test cases have been generated.

We adopted the third approach to generate global test sequences. We need the final model with all the aforementioned information, the local test sequences, as well as the communication graph to guide global test cases. The algorithm given in [32, 77, 57], called the generation of def-use executables paths, defines four different variable usages: assignment-use (A-usage), input-use (I-usage), computational-use (C-usage), and predicate-use (P-usage). These variable usages enable the links between the test sequences of each machine and help check the test sequences' executability. The algorithm provides a full set of executable and non-executable test sequences that will go through all the transitions in the system under test. We generate the paths linking two states from different machines by marking them as communication or synchronization points.

Global Test Sequences for LGS

To generate global test sequences, we need to identify the communication variables. In the case of a landing gear system, the variables are:

{Start, activateEmergencySystem, OrangeLight(on,off), GreenLight(on,off), RedLight(on,off) }

These variables indicate the possible communication between the agents. For example, if activeEmergencySystem is on, it means that the RedLight variable is also on. This is the only time the emergency system will be called upon. To identify the communication points, the input and output list is defined for each transition. The related input and output lists, as well as the predicates, are described in Figures 7, 8, and 9. They are used as inputs for the algorithm to generate global test sequences. In general, a test case is composed of the following elements: <preamble, target, postamble, veridict>. The preamble and the postamble elements can be empty. The preamble is the sequence of transitions used to reach the target transition for testing as given in Table 5; it also shows examples of the application of the algorithm using the landing gear system (LGS) case study. It identifies the different usage lists enabling the identification of executable test sequences. Table 6 shows an example of executable test sequences to reach specific transitions in the system model. The chosen transitions represent a case of parallelism.

Table 5: Example of usage lists and preamble for specific transitions of the LGS

Trans.	A-usage	I-usage	P-usage
Pt2	OrangeLight	-	-
Pt4	-	GreenLight Ct11	OreenLight on
Ct11	-	GreenLight on; OrangeLight off	GreenLight on
Trans	**Preamble**		
Pt2	Pt1		
Pt4	Pt1, Pt2, Pt3, [Ct11]		
Ct11	Ct1, Ct2, Ct3, Ct4, Ct5, Ct9, Ct10		

Table 6: Executable test sequences of the LGS

Transition	Executable test sequences
Pt5	Pt1, Pt2, Pt3, Ct1, Ct2, Ct3, Ct4, Ct5, Ct9, Ct10, Pt4, Pt5

Table 7 presents an example of non-executable test sequences. These are non-executable because they need a preamble execution from another agent to reach the desired transition and render the sequence executable. Table 8 shows the parallelism in the executable test sequences required to make the transitions shown in Table 6 executable.

Table 7: Non-executable test sequence of the LGS

Transition	Non-executable test sequences
Pt5	Pt1, Pt2, Pt3, Pt4, Pt5

Table 8: Parallelism shown for executable test sequences Pt5 of the LGS

	Executable test sequences – Pt5		
M_p	Pt1, Pt2, Pt3		P4, Pt5
M_c	Ct1, Ct2, Ct3, Ct4, Ct5	Ct9, Ct10	

In the following subsections, we show how to verify the different properties obtained from the validation phase.

3.5 Verification of properties (***Step VI***)

Table 9 shows specific executable test sequences for a selection of witness properties for liveness. Due to a limitation in all model checker tools in terms of generating witness examples and counterexamples that include the universal operator "A", we used other formulas that achieve the same requirement and allow MCMAS+ to generate witness examples. The executable test sequences are given by the input and output information, as well as by the transitions for which that input and output information proved the witness-example to be true. The executable test sequences represent the transition in which the witness example holds. Hence, these are all the possible transitions forming a path needed to render a test sequence executable, up to the mentioned transition. For example, EF GreenLight holds true when a sequence executes up to transition Pt5 (refer to Table 5 for the complete executable test sequence).

Table 9: Executable test sequences for witness-examples for liveness properties

Witnessexamples for liveness properties	**Executable test sequences**
EF GreenLight	Sequences leading to transitions: M_p: Pt4 – Pt5 – Pt8 M_c: Ct10 – Ct11 M_e: Et5 – Et6
EF (RedLight && EF GreenLight)	Sequences leading to transitions: M_p: Pt8 – Pt9 M_c: none M_e: Et5 – Et6
EF (PressedDown && EF GreenLight)	Sequences leading to transitions: M_p: Pt4 – Pt5 – Pt8 – Pt9 M_c: Ct10 – Ct11 M_e: Et5 – Et6

Several properties are defined in Table 10 to verify whether the algorithm validates the properties. The executable test sequence shown in Table 6 were analyzed with regards to those properties.

Both executable test sequences for transitions Pt5 and Pt9 verify all the properties identified so far. Table 10 confirms that all the global test sequences generated render the defined properties true. According to the algorithm used in [32], none of the executable test sequences validate the given properties. However, those that represent the full paths in the global system do validate them, being the paths generated for transitions Pt5 and Pt9. This implies that through that algorithm, only a subset of test sequences can validate the different properties, but not necessarily all of them.

Table 10: LGS properties validated with the executable test sequences.

CTL	**Status**
$\phi_1 = AG(PressedDown) \rightarrow AF(\text{GearsExtended} \wedge DoorsClosed)$	*True*
$\phi_2 = EG(E(PressedDown\ U\ PressedDown \wedge GearsExtended \wedge\ DoorsClosed))$	*True*
$\phi_3 = AF\ \neg E(\neg PressedDown\ U\ (\ GearsExtended \wedge\ DoorsClosed))$	*True*
$\phi_4 = AG\ \ \neg\ (PressedDown \wedge\ AG(\neg GreenLight))$	*True*
$\phi_5 = AF(GreenLight)$	*True*

3.6 Coverage criterion: MC/DC

To comply with the avionics standard DO-178C, the proposed test generation algorithm needs to satisfy the modified condition/decision coverage (MC/DC) criterion. This will ensure that all possible conditions are tested. Therefore, we use a graph expansion mechanism to handle this type of coverage. MC/DC is applied using binary values, and every condition will have a value of true or false. It is probable that some MC/DC test cases are not feasible within the system [77]. This means that some test cases' execution will fail [47, 77]. In other words, all the outcomes of every decision, as well as the conditions within those decisions, should be executed at least once. By doing so, all paths regarding possible values taken by the system under test will be executed. For example, in the global system, a single decision must be made at P3 to move further to P4 or P6 as follows:

```
If (OrangeLight is on and GreenLight is off and RedLight is off )
   Return light status (RedLight or GreenLight on) from the controller;
EndIf;
```

To satisfy the MC/DC criterion, we need to visualize a path as binary decisions and conditions. The algorithm will analyze a path with all possible conditions as binary as follows:

Decision → go to controller
Conditions
→ if (OrangeLight is on/off)
→ if (GreenLight is on/off)
→ if (RedLight is on/off)

There are three conditions to consider within this decision: whether the OrangeLight is on, the GreenLight is off, and the RedLight is off and so on. This translates to the following possibilities shown in Table 11. This table is also known as truth table, and as the MC/DC table. MC/DC tables are often provided by engineers developing the system and are part of the specifications for manual testing.

Table 11: Possible binary values and possible outputs

OrangeLight	**GreenLight**	**RedLight**	**Output**
True	False	False	Go to controller
False	True	False	Error
False	False	True	Error
True	True	False	Error
True	False	True	Error
False	True	True	Error
True	True	True	Error
False	False	False	Idle

There is a value in executing test sequences from the MC/DC criterion that result in an error, as it ensures that a test sequence will fail. As such, we also cover the possibility of faulty signals being sent to the pilot, the controller, and the emergency agent. The errors are the result of a status or a state that is not naturally feasible by the system. To generate test sequences for the MC/DC criterion [76, 77], we need to identify a way to consider the binary sequence and condense it into one single segment. This will enable the generation of MC/DC test sequences using model checking. For example, we could add information to the input and output values for transition Pt3 by adding the different possibilities covered through the MC/DC criterion and use that information to generate the required test sequences. Table 12 shows examples of test cases generated for

MC/DC conformity. Some test cases for a specific transition are not executable and will either end in an error or be idle.

Table 12: Examples of MC/DC compliant test cases

Transition	**Test case**	**Status**
tP5	tP1, tP2, tP3, tP4 --	Error green light is on
tP5	tP1, tP2, tP3, tP4 --	Error red light is on
tP9	tP1, tP2, tP3, tP4, tP6, tP7, tP8, tP9	

The generation of test cases that satisfy the MC/DC criterion will also generate several unfeasible paths. For certain type of software, it is important to ensure that erroneous paths are handled correctly, which will control both the conformity of the properties and the alternatives triggered by glitches or possible malfunctions.

3.7 Conclusion

The proposed approach integrates the verification of properties at the design level and the testing phase, based on a model with required coverage criteria such as the MC/DC. When verification is used at the design level, it is not enough to ensure the propagation of these properties in the final product. Given that the avionics industry requires testing for their certification process, the approach complies with this requirement and enhances the certification artifacts with test generation via model checking. The approach is applied to the LGS. All test generation techniques must confront the state explosion problem and test efficiency in terms of coverage and error detection power.

In the next chapter we introduce a collaborative work lead by El Khouly et al. that explores test case generation based on model checking. While this work does not address MC/DC coverage criteria and requirements traceability, it does show the limitations of verification and its complementarity to test case generation.

Chapter 4

4. Model checking-based test case generation

In this chapter, to facilitate the comparison between approaches, we present a collaborative work that is a continuation of the previous chapter where we addressed testing and verification. It does not consider the MC/DC criterion. Here we propose another approach for test case generation based on model checking [91]. The model checker is used not only as a prover of desirable properties that we expect to be true, but also to find counterexamples and witness traces for certain properties that we expect to be false. This work follows a test-driven development, an approach that provides iterative and incremental system construction. Such an approach offers the following advantages: it addresses multi-level testing (unit, integration, system, regression) as well as requirements coverage, where every requirement has at least one test case associated with it. It helps also debug and alleviate the oracle problem, as counterexamples help to identify the location of the violation on the model. For unit testing, local test cases for agents are generated to satisfy one of the following criteria: states, transitions, protocol or policies' coverage. For integration testing, test case generation handles the interaction of agents with each other under the assumption that each agent has passed unit testing. The interactions can be modeled either by a message passing system [80] via sending and receiving messages among agents, or by social commitments and their fulfillment when agents depend upon each other. Agents' dependencies are often the source causes for the integrations. In [79], an agent called a *debtor* is committed to another agent called a *creditor* to perform a task called a consequence when a certain condition called the antecedent is satisfied. Notably, the *creditor* depends on the *debtor* to perform some tasks. Once the consequence of a commitment is achieved, the commitment is fulfilled. Obviously, we test reachable commitment and fulfillment properties to make sure that each pair of agents work together correctly, thanks to the reachability properties, which cover and test paths from initial states to target states [93]. For system (or society) testing, the whole system is tested after integrating all the tested components to capture errors that cannot be attributed to individual agents, or to the interaction among agents. Particularly, the properties are that the intended system must be safe, live, and reachable is a subset of all path coverage [94].

Multi-level testing enables us to better focus on the specific problems that might occur at each level. Therefore, our approach has two main phases. The first phase focuses on unit testing and has the following steps:

1) Define coverage criteria.
2) Create "never-claims" (also called trap properties) for coverage criteria [92].
3) Check the correctness of never-claims in the encoded multi-agent models by model checking to generate counterexamples/witness traces, i.e., test sequences that cover never-claims.
4) Interpret and map counterexamples/witness traces into concrete test cases.

The second phase has the same steps as above but focuses on integration testing and system testing by generating test cases with respect to a given set of requirement properties. These requirement properties include safety properties and are used as test metrics that can assess the quality of the system under test.

In these two phases, the MAS is modeled using our extended interpreted systems formalism. This formalism supports communication among components, input variable actions, guarded predicates, and post-conditions of performed actions. The formalized models of agents are then encoded using the extended ISPL+ (EISPL+). The original ISPL+ is the input language of the symbolic model checker MCMAS+ introduced in [72]. The motivation behind the extension of this language is to make it compatible with our extended formalism of interpreted systems. Requirement properties are extracted from the specifications and expressed using an expressive language, namely computation tree logic for conditional commitments (CTLCC) [72]. CTLCC extends CTL [73] in an elegant way that makes it able to express not only temporal properties, but also communication requirements in the form of public commitments under given conditions. The limitations of this proposed technique are the traditional limitations related to verification techniques, namely state explosion, memory utilisation and algorithm complexity.

4.1 Modelling

In this section, we use an LGS case study that is modeled and used for the first approach. Figures 3.3, 3.4, and 3.5 show the formal models of the pilot, control unit, and emergency agent machines, respectively. In each figure, we introduce the input and output of each transition in a tabular form where the symbols “?” and “!” refer to the process of receiving and sending a message, respectively. Moreover, the output of a transition can be directly assigned by the shared and unshared variables when there is no explicit output action. Thus, it is easy to define the Boolean predicate of each transition using the conjunction operator between the input and the output.

4.1.1 Extended interpreted systems

The formalism of interpreted systems [80] provides a very popular mainstream framework for modeling and reasoning about MASs. This framework models locally autonomous, intelligent, and heterogeneous agents that interoperate within a global system by sending and receiving messages [63, 72, 73, 77, 94]). This framework is suitable for modeling behaviors of intelligent components cooperating to construct an autonomous and critical avionics system. The formalism has been extended by El Kholy et al. (2014, 2015) [75, 76] with sets of shared and unshared variables to account for interactions that occur during the execution of MASs. However, the post-conditions that result after firing transitions are not supported in the current version of this formalism. These post-conditions are required to capture the effects of executing particular actions. In avionics systems, these post-conditions are of great importance to being able to reason about the actions performed at each moment. We divide local actions into input and output variable actions and define guarded predicate conditions, which make the pre-conditions needed to fire transitions. This work extends the version of interpreted systems introduced in [72, 74] with context domain and assignment actions that define the post-conditions of firing transitions. This extended version of interpreted systems defines the set of interacting agent machines. The model of each interacting agent machine is given in Definition 1.

Definition1. The model of an interacting agent machine i is a 10-tuple:

$M_i = (L_i, D_i, Act_i, \mu SV_i, SV_i, P_i, \iota_i, O_i, V_i, T_i)$ where

1. L_i is a countable set of local states. Each local state l_i privately represents the whole information about the system that the agent has at a given moment.
2. *D_i is a context domain.*

3. *$Act_i \subseteq iAct_i \cup oAct_i$ is a countable set of local input actions $iAct_i$ and local output actions $oAct_i$ available to the agent i such that for the same agent machine $iAct_i \cap oAct_i = \emptyset$. Act_i accounts for the temporal evolution of the system.*

4. *μSV_i is a countable set of unshared context variables. Each variable from μSV_i has one value from the domain D_i.*

5. *SV_i is accountable set of shared context variables that the agent machine i shares with another agent. Similar to μSV_i, each variable in SV_i has one value from the domain D_i. Two different variables from the same set (μSV_i or SV_i) could have the same value, but $\mu SV_i \cap SV_i = \emptyset$, i.e., the two sets are disjoint in terms of their variables.*

6. *P_i is a local protocol function P_i: $L_i \rightarrow 2^{Acti}$, which represents the decision-making procedure of i and produces the set of enabled actions that might be performed by i in a given local state.*

7. *ι_i is a set of initial states such that $\iota_i \subseteq L_i$.*

8. *O_i is a countable set of local assignment outputs. An assignment output has the form $xji = v_{ij}$ where $xji \in SV_i \cup \mu SV_i$ and $v_{ij} \in D_i$ for $j \in N$ (N is the set of positive natural numbers).*

9. *V_i: $Ap_i \rightarrow 2L_i$ is a local valuation function defining what local atomic propositions hold from the set Ap_i in the local states of i.*

10. *T_i is a countable set of transitions among local states. Each transition is a 5*-tuple $t = (l_{start}, l_{end}, input, output, pre)$ where:
    - $l_{start} \in L_i$ is the starting state of the transition *t*.
    - $l_{end} \in L_i$ is the ending state of the transition *t*.
    - *input* $\in iAct_i$ is the local input of the transition *t*.
    - *output* $\in O_i \cup oAct_i$ is the local output of the transition *t*.
    - pre is a predicate that must be true to fire the transition t. It has three parameters: unshared and shared variables in μSV_i and SV_i, and the input action.

Conventionally, a MAS consists of two or more agents. In our approach, the model of the MAS is composed of n agent machine models $M = (M_1, M_2, ..., M_i, ..., M_n)$ such that each agent machine model has a unique identifier in the whole system and works in parallel with other agent machine models. In this modeling, a global state represents the instantaneous configuration of all agent machine models in the system at a certain moment.

Definition 2. A global state $s \in$ S is a tuple $s = (l_1, ..., l_n)$ where each element $l_i \in L_i$ represents the local state of an agent machine i. The set of all global states S $\subseteq L_1 \times ... \times L_n$ is a subset of the Cartesian product of all local states for all agent machines. All local transitions are combined $T_1 \times ... \times T_n$ to define the global transition T.

Definition3. For two global states s and s', a global transition is defined by T(s,a,o, p) = s' where a is a joint action $a \in ACT = Act_1 \times ... \times Act_n$ (one for each agent machine), o is a joint assignment output, and p is a predicate defined using 3 arguments:
1) a joint input action in $\in I_n = iAct_1 \times ... \times iAct_n$, 2) a joint unshared variable $v \in \mu SV = \mu SV_1 \times ... \times \mu SV_n$, and 3) a joint shared variable $u \in SV = SV_1 \times ... \times SV_n$.

Definition4. The model of the MAS $M = (M_1, M_2, ..., M_i, ..., M_n)$ is unwound into a set of computational paths (traces) in which each path π = s_0, a_0, o_0, p_0, s_1, a_1, o_1, p_1,... is an infinite sequence of states that increase simultaneously over time in which a, o, and p are defined in Definition 3, $s_i \in S$, and $T(s_i, a_i, o_i, p_i) = s_{i+1}$ for each ≥ 0. π(k) is the k-th state of the path π.

Let $l_i(s)$ represent the local state of agent machine i in the global state s and let the value of a variable x in the set SV_i at $l_i(s)$ be denoted by $l_i^x(s)$.

Definition 5. A synchronous communication channel between two agent machines i and j can be established if and only if there exists exactly one shared variable, i.e., $|SV_i \cap SV_j| = 1$.

Two agent machines i and j can communicate if they share a communication channel that is represented by a shared variable between them.

Definition 6. When j has an unknown value for x at s and depends on i to get access to this value, then j establishes the communication channel using x with i so that the value of x becomes available to read. The reading process is completed at s'; so
$l_i^x(s) = l_i^x(s') = l_j^x(s')$.

Definition 7. For two agent machines (i, j), $\sim_{i\to j} \subseteq S \times S$ is a social accessibility relation defined by $s \sim_{i\to j} s'$ if and only if the following underlying conditions hold:

1) $l_i(s) = l_i(s')$,

2) $(s, s') \in T$,

3) $\forall\ x \in SV_i \cap SV_j$ such that $SV_i \cap SV_j \neq \emptyset$ we have $l_i^x(s) = l_j^x(s')$, and

4) $\forall$ y $\in \mu SV_j$, we have $l_j^y(s) = l_j^y(s')$.

The intuition of the accessibility relation between two global states s and $s'(s \sim_{i\to j} s')$ in Definition 7 means that:

1. the local states of the debtor agent machine i are indistinguishable in s and s' ($l_i(s) = l_i(s')$), i.e., i still persists on its commitment.

2. the accessible state s' has a transition relation with the current state or the commitment state s' ($(s, s') \in T$), i.e., i can reach its accessible states.

3. there exists one shared variable x and its value for the debtor agent machine I and the creditor agent machine j in s' is the same $l_i^x(s) = l_j^x(s')$). i.e., j has been read the required information through the established communication channel; and

4. the value of all unshared variables y for the creditor agent machine j is unchanged in s and s' ($l_j^y(s) = l_j^y(s')$), i.e., j does not gain any new information with the communication of i through its unshared variables. These variables are needed to establish communication between j and other agent machines, except i. The accessibility relation is used to define the semantics of the conditional commitment operator as shown in the following subsection.

4.1.2 Syntax and semantics of CTLCC

The CTLCC logical language directly and intuitively supports communication by exchanging commitments among interacting agents [72]. Because CTLCC is a logic for communicative autonomous MASs, it is suitable for avionics systems where components are autonomous and communicating. As we mentioned before, we adopt CTLCC to express temporal properties.

Definition 8. The syntax of CTLCC is given inductively by the following:

$\varphi ::= p | \neg \varphi \mid \varphi \vee^{\varphi} \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi \;\; U \varphi) | CC(i, j, \varphi, \varphi) | Fu(i, CC(i, j, \varphi, \varphi))$ where

- $p \in \mathcal{PV}$ is an atomic proposition. ¬ and ∨ are the usual Boolean operators.
- *E* is the existential quantifier on paths.
- *X*, *G* and *U* are linear CTL path modal operators standing for “next-time”, “globally”, and “until” respectively
- *i* and *j* are two interacting agent machines. *CC* and *Fu* stand for conditional commitment and fulfillment operators.

The temporal operators *EX*, *EG*, and *EU* pertain to the existence of paths through which specific conditions are recognized successfully. For instance, *EG p* is read as “there exists a path such that *p* holds globally along the path”. Other operators that can be abbreviated from the connectives and operators as usual. For instance, $\top \triangleq (p \vee \neg p)$, $\varphi \rightarrow \psi$, $\neg\varphi \vee \psi$, $EF\,\phi$, $E(\top \text{ U } \phi)$, $\text{AX}\,\phi \triangleq \neg EX\neg\phi$, and $AG\phi \triangleq \neg EF\neg\varphi$ where *A*,→ and *F* refer to the universal quantifier on paths, the implication operator and the eventually operator, respectively.

The CC operator formally models the communication between two agent machines by capturing the intuition that communicating agent machines should share some variables representing the communication channel, and the receiver’s local state is impacted by the local state of the sender. The formula $\text{CC}(i, j,\psi,\phi)$ is read as “agent machine *i* strongly commits towards agent machine *j* to consequently perform the task ϕ once the antecedent ψ holds.” The antecedent ψ and the consequence ϕ in the commitment operator can be any arbitrary CTLCC formula. The formula $\text{Fu}(i,\text{CC}(i, j,\psi,\phi))$ is read as “the strong conditional commitment $\text{CC}(\text{i, j},\psi,\phi)$ is fulfilled by agent machine i.” The semantics of CTL formulae (the basic fragment of CTLCC) is defined as usual semantics (see, for example, Clarke et al., 1999 [73]). The semantics of CC and Fu are given in Definition 9 from (El Kholy et al., 2014 [72]).

Definition 9. Given the model M of a MAS, the satisfaction of the CC (respectively Fu) formula in a state *s* denoted by (M,s)|= CC(...) (respectively (M,s)|= Fu(...)) is defined as follows −(M,s)|= CC(i, j,ψ,ϕ) iff ∃s0∈S s.t. s∼i→j s0 and (M,s0)|= ψ, and ∀s0 ∈S s.t. s∼i→j s0 and (M,s0)|= ψ, we have (M,s0)|= ϕ −(M,s)|= Fu(i,CC(i, j,ψ,ϕ)) iff ∃s0 ∈S s.t. s0 ∼i→j s and (M,s0)|= CC(i, j,ψ,ϕ) and (M,s)|= ψ ∧ ¬CC(i, j,ψ,ϕ).

From Definition 9, the formula $CC(i, j,\psi,\phi)$ is satisfied in the model M at s iff there exists a state s0 satisfying the antecedent ψ and accessible from s by $\sim i\rightarrow j$ and the consequence ϕ holds in every state satisfying ψ and accessible from s. Intuitively, the antecedent ψ and the consequent ϕ of commitments should be achieved at least in one state accessible from s. The state formula $Fu(i,CC(i, j,\psi,\phi))$ is satisfied in the model M at s iff s satisfies the antecedent ψ and the negation of the strong commitment $CC(i, j,\psi,\phi)$ and there exists a state s0 holding the commitment and s is "seen" from this state via$\sim i\rightarrow j$.

4.2 Model-based unit testing

In this section, we show how to generate test cases for modeled agents (units): M_p, M_c, and M_e. Such test cases are produced from counterexamples automatically generated by model checking M_p, M_c, and M_e against properties that satisfy certain coverage criteria. To meet this aim, we start with encoding M_p, M_c, and M_e by making use of the extended input language EISPL+ of MCMAS+. We adopt MCMAS+ because its EISPL+ supports the semantics of the extended version of interpreted systems with shared and unshared variables and assignment outputs as post-conditions. Moreover, MCMAS+ alleviates the state explosion problem, thanks to the ordered binary decision diagrams (OBDDs) that consume less memory than explicit representations.

4.2.1 Encoding

Prior to encoding individual modeled agent machines (M_p, M_c, and M_e) in our EISPL+ language, we present the main sections of this language. Indeed, these sections are automated by the implemented graphical user interface plugged in the Eclipse platform. An EISPL+ program has the following sections:

1. Agents' declarations to define a list of EISPL+ agents with four sub-sections according to the following syntax Agent <agentID> <agent body> end Agent where <agentID> is an EISPL+ agent identifier and <agent body>contains: 1) a set of local states; 2) a set of shared and unshared variables; 3) a set of local actions; 4) local protocol; and 5) evolution transition function.

2. Evaluation function is defined as follows: Evaluation <proposition> if <condition on states> end Evaluation where <proposition> is an EISPL+ proposition and <condition on states> is a truth condition that defines a set of local states for the atomic proposition.

3. Initial states to define the set of initial state conditions as follows: InitStates <condition on states> end InitStates.

4. List of formulae that must be verified is defined using the following syntax: Formulae <formulae list> end Formulae.

To encode M_p, M_c, and M_e with the help of Figures 7, 8, and 9 the designers must fill up the above sections. We also extend the evolution transition function in the EISPL+ program with the predicates that control the firing of transitions and with the possibility to define the post-conditions of firing transitions.

4.2.2 Review and tracing

Prior to model checking the encoded models (M_p, M_c, and M_e) by MCMAS+, we need to perform the review and tracing activities. We review the models with the wide range of features implemented in the graphical user interface of MCMAS+ [74]. The graphical interface highlights syntax errors and supports text marking and formatting. By fixing all the highlighted errors, we have a clear and error-free encoding models. By the tracing activity allows us to track the behavior of the encoded models using the possibility released in the MCMAS+ tool called Explicit Interactive Mode. This tool starts with the initial state and offers all the transitions available at this state and gives the possibility to choose particular ones. After we select one of these transitions, the tool moves to the reachable state connected with the initial state by this transition and displays the available transitions at the new state. By carrying out this step, we will be able to evaluate whether the models progress as we intended or not. In the case of detecting an error, we return to our encoding and update it. This process continues until we reach the end state. We then start again from the initial state and select another transition and continue as we did previously. Our graphical interface supports a new feature that displays the modeled agent machines. By completing these two activities, we ensure that our encoding models capture exactly the intended behavior of the corresponding parts in the landing gear system. In fact, these two activities are of extreme importance to ensure the model is correct; otherwise, errors in the design model jeopardize the entire activity of the formal verification using a model checking technique. However, it may not be feasible for the review and tracing activities to explore (trace) all the possible transitions when state space is particularly large. This is a limitation of any validation activity, because unlike verification that aims at being complete, this validation is by definition incomplete.

4.2.3 Coverage criteria and trap properties

The test purpose specifies the desired features of a test case. The test purposes can be systematically defined according to certain coverage criteria. For example, a coverage criterion could specify the final state (or goal) of the test case or a sequence of states that must be traversed. A full coverage criterion is accomplished when all elements specified by that criterion are covered by at least one test case. Given that, each test purpose is specified in the CTLCC logic and transformed to a never-claim (also called trap properties) by negation [92]. This asserts that the test purpose never becomes true. By model checking the never-claim on a formal model, it generates a counterexample when the test predicate is feasible (i.e., when the never-claim is not satisfied). The counterexample demonstrates how the never-claim is violated, and hence shows how the original test purpose is fulfilled.

State coverage

To create a test suite that covers goal states of the modeled agent machines, a never-claim property for every local atomic proposition holding in these states is needed, claiming that the value (atomic proposition) is not held: $AG\neg(ap_k)$ where $0 < k \leq n$ and n is the number of atomic propositions. These atomic propositions are labelled goal states using the valuation function V_i. Table 13 shows some examples of state coverage in the form of never-claim properties. From the table, some goal states are PressedDown, DoorsOpening, RedLight, and HandlePulled. A counterexample to each never-claim property is any path that includes a goal state. These never-claim properties for state coverage are safety properties.

Table 13: Some examples of state coverage in Mp, Mc, and Me

	φ1	***φ2***	***φ3***
$\boldsymbol{M_p}$	*AG¬PressedDown*	*AG¬OrangeLight*	*AG¬GreenLight*
$\boldsymbol{M_c}$	*AG¬DoorsOpening*	*AG¬OrangeLight*	*AG¬GearExtended*
$\boldsymbol{M_e}$	*AG¬RedLight*	*AG¬HandlePulled*	*AG¬DoorsClosedMechanically*

Transition coverage

The transition coverage criterion is defined with respect to the local transition relation (e.g., T_p in the pilot agent machine model). A never-claim property representing a transition coverage states that in all paths globally when the pre-state condition α and the guard condition γ are true, the post-state condition β might not be satisfied in the next state: $AG(\alpha \wedge \gamma \rightarrow AX\neg\beta)$. The pre-state condition α is an atomic proposition in $iAct_\mathrm{i}{}^{\mathrm{Api}}$ where $iAct_\mathrm{i}{}^{\mathrm{Api}}$ is the set of atomic propositions resulting from mapping local input actions in the set $iAct_i$. The guard condition γ is a predicate resulting from $\mu SV_i \cup SV_i \cup iAct_i$. The post-state condition β is an atomic proposition in $O_i^{Api} \cup oAct_i^{Api}$ where $oAct_i^{Api}$ is the set of atomic propositions resulting from mapping local output actions in the set $oAct_i$ and $O_i^{Api}i$ is the set of atomic propositions resulting from mapping local assignment outputs in the set O_i. Table 14 shows some examples of transition coverage in the form of never-claim properties. From the table, the constant always true is used in lieu of a predicate, which is evaluated to be true. Notice that counterexamples to these never-claim properties end with the transition from α to β. If this is not observed, then an additional sequence is necessary.

Protocol coverage

For each protocol (plan or strategy), there exists a never-claim property stating that in all paths globally when the pre-state condition α and the guard condition γ are true, the post-state condition β might not be satisfied in the next state:

$$\mathrm{AG}\ (\alpha \wedge \bigwedge_{i=1}^{n}(\gamma i \ \rightarrow AX\neg\beta_i))\ [94]$$

where n is the number of permissible actions at the current state. Again, the post-state expression β is negated to force creation of suitable counterexamples for the case when the pre-state and guard conditions hold. For example, in M_c, when the orange light is acknowledged, there are two possibilities: 1) the gear does not close the door and then the door close error appears; or 2) the gear close is acknowledged and then the door is closed successfully.

Formally, $\varphi 1 = AG\ (\mathit{OrangeLightOnAck} \land \mathit{CloseGearDoorsAck} \rightarrow AX\neg \mathit{DoorsCloseSuccess})$, and $\varphi 2 = AG\ (\mathit{OrangeLightOnAck} \land \mathit{CloseGearDoorsError} \rightarrow AX\neg \mathit{DoorsClosedError})$.

Table 14: Some examples of transition coverage in Mp, Mc, and Me

M_p	$\varphi 1 = AG(\mathit{PressDownAck} \land \mathit{true} \rightarrow AX\neg \mathit{WaitForOrangeLight})$ $\varphi 2 = AG(\mathit{ConfirmGearDeploymentMe} \land \mathit{true} \rightarrow AX\neg \mathit{DeploymentSuccess})$
M_c	$\varphi 1 = AG(\mathit{OpenGearDoorsAck} \land \mathit{true} \rightarrow AX\neg \mathit{OrangeLightOn})$ $\varphi 2 = AG(\mathit{CloseGearDoorsAck} \land \mathit{true} \rightarrow AX\neg \mathit{DoorsCloseSuccess})$
M_e	$\varphi 1 = AG(\mathit{InitializeEmergencySystem} \land \mathit{true} \rightarrow AX\neg \mathit{OpenGearDoors})$ $\varphi 2 = AG(\mathit{GearLightOnAck} \land \mathit{true} \rightarrow AX\neg \mathit{GearExtended})$

4.2.4 Test case generation

Model checking is a three-step process: 1) modeling the system underlying the verification process; 2) expressing the requirements; and 3) running the verification of the model against the expressed requirements. So far, we have completed the first two activities. For the third activity, we use MCMAS+ to verify the correctness of all our never-claim properties on the modeled and encoded agent machines and produce the corresponding counterexamples.

Each modeled agent machine is a reactive, labelled transition modeled system, i.e., it reacts to inputs by producing the corresponding output values. Local test cases can be generated directly from the computation paths representing the generated counterexamples. According to Definition 4, test cases or counterexamples are sequences of states interleaved with actions, assignment outputs, and predicates. Therefore, it is necessary to discriminate between paths with or without loopback when mapping a path to a test case because test cases are always finite. When a path does not contain a loopback state, the path and test case are identical. On the other hand, when the path does contain a loopback, then the lasso-shaped sequences need to be unfolded. Our direct solution is to use a truncation strategy. Technically, because we adopt a white box testing technique applied to a model, the complete internal state is known. Therefore, a test case can be terminated whenever the same state has been visited twice at the same location in the loop.

4.3 Model-based integration testing

Because critical avionics systems, including the landing gear systems have normal and emergency modes, then we can define two pairs of modeled agent machines. The first pair includes M_p and M_c, while the second pair includes M_p and M_e. In the following, we show how to generate test cases by model checking the models of these pairs against never-claim properties, which satisfy certain coverage criteria. Following the approach, we have to start with encoding these pairs using EISPL+.

4.3.1 Encoding

Encoding the pairs of modeled agent machines means considering the interactions between M_p and M_c in the first pair and the interactions between M_p and M_e in the second pair. Indeed, these interactions are encoded using joint actions, joint assignment outputs, predicates, and commitments and their fulfillment (see Definition 3). Moreover, the designers need to:

1. synchronize the interactions between modeled agent machines to decide the transitions that must be fired in parallel; and
2. identify the dependency between modeled agent machines to define shared and unshared variables and consequently commitments and their fulfillment.

For example, since the first transition between the local states p0 and p1 in the pilot EISPL+ program should work in parallel with the first transition in the control unit agent machine model, the two local actions "*Action=I_LandingParameters_O_PressDownButton*" and "*ControlUnit.Action=I_PressDownButton_O_PressDownButtonAck*" should be joined and performed simultaneously.

Moreover, the control unit agent machine model can read the value of the shared variable Button from the pilot agent machine model using the established channel. We also extended the evolution transition function in the EISPL+ program with the predicates that control the firing of transitions and with the possibility to define the post-conditions of firing transitions. For instance, the predicate of the first transition is defined using the connective operator 'and' and its postcondition is defined by changing the value of the shared variable Button from Up to Down. The pilot agent machine model needs to read the values of the orange, green, and red lights from the control unit agent machine model, we then encoded three shared variables that take two values On and Off. Finally, the refined and synchronized modeled agent machine pairs are reviewed and traced.

4.3.2 Coverage of social communication properties

All the above coverage approaches focus on structural coverage criteria based on a behavioral model of the system under test. It is often desirable to generate test cases with regards to a given set of temporal properties. The authors in (Engels et al., 1997 [92]) proposed an approach to use requirement properties as test purposes. However, it is not always potential to create suitable counterexamples by directly negating requirement properties, because the negation of a safety property might produce a counterexample, which encompasses merely one state (i.e., initial state). The authors in (Callahan et al., 1996 [97]) suggested to use an equivalence partitioning of the computation tree in which each requirement property has two kinds of computation paths that can be distinguished in the computation tree: 1) those paths through which the property is fulfilled; and 2) those paths through which the property is violated. Our full coverage can precisely be created by collecting properties and their negations using the conjunction operator over disjoint partitions in the computation tree. For clarity purposes, assume we have two requirement properties φ_1 and φ_2. Then, there are four possible disjoint partitions for these two properties: $\varphi_1 \land \varphi_2$, $\neg\varphi_1 \land \varphi_2$, $\varphi_1 \land \neg\varphi_2$, and $\neg\varphi_1 \land \neg\varphi_2$. Each one of these combinations is a coverage property. Because our focus here is on establishing and fulfilling the interactions between modeled agent machines and their pairs, our coverage properties include essentially the social commitment and fulfillment operators. These properties are self-descriptive reachability properties and can be seen as path coverage as well.

For pair (M_p, M_c),

- $\varphi 1$ = *EF CC*(*ControlU*, *Pilot*, *PressedDown*, *EX ProcessReceivedCommands*)
- $\varphi 2$ = *EF Fu* (*ControlU*, *CC*(*ControlU*, *Pilot*, *PressedDown*, *EX ProcessReceivedCommands*))
- $\varphi 3$ = *EF CC*(*ControlU*, *Pilo*t, *GearsExtended ∧ DoorsClosed*, *AF GreenLightOn*)
- $\varphi 4$ = *EFFu* (ControlU,CC(ControlU, Pilot, GearsExtended ∧ DoorsClosed, AF GreenLightOn))

For pair (M_p, M_e),

- *$\varphi 5$ = EF CC (Emergency, Pilot, RedLightOn, AF ExtendGearsMe ∧ CloseDoorsMe)*
- *$\varphi 6$ = EF Fu (Emergency, CC (Emergency, Pilot,RedLightOn ,AF ExtendGearsMe ∧ CloseDoorsMe))*
- *$\varphi 7$ = EF CC (Emergency, Pilot,GearsExtended ∧ DoorsClosed, AFGreenLightOn)*

- *φ8 = EF Fu (Emergency, CC(ControlU, Pilot,GearsExtended ∧ DoorsClosed, AF GreenLightOn))*
- *φ9 = EF CC (Pilot, Emergency, GreenLightOn, AF ConfirmDeployment)*
- *φ10 = EF Fu (Pilot, CC(Pilot, Emergency, GreenLightOn, AF ConfirmDeployment))*

4.3.3 Test case generation

We use MCMAS+ to verify the correctness of our coverage properties and extract global test cases from generated witness traces instead of counterexamples. In fact, the two ideas of generating test cases from counterexamples and witness traces are complementary because the negation of the adopted properties is sufficient to move from one to the other.

4.4 Model-based system testing

Having guaranteed that both modeled agent units and modeled agent components are error-free design, in the following, we show how to generate test cases for the whole modeled MAS: M_p, M_c, and M_e. we follow the same steps, through the EISPL+ encoding activity, we refined and synchronized the interactions among the three modeled agent machines (M_p, M_c, and M_e) and then reviewed and traced the whole model to remove any design errors. The review and tracing activities are illustrated in El-Khouly et al. [91]. It is worth noting that the whole model is generated by the MCMAS+ tool8 and reachable states are computed using three algorithms introduced by Lomuscio et al. [78].

4.4.1 Temporal properties' coverage

As mentioned above, we use temporal requirement properties as test purposes. To achieve this aim, we express the following properties using CTLCC:

- φ1 = AG(PressedDown→ AF(GearsExtended ∧ DoorsClosed))
- φ2 = EGEF(E(PressedDown U (PressedDown ∧ GearsExtended ∧ DoorsClosed)))
- φ3 = AF¬E(¬PressedDown U (GearsExtended ∧ DoorsClosed))
- φ4 = AG¬(PressedDown ∧ AG(¬GreenLight))
- φ5 = EF(PressedDown ∧ EF GreenLight)
- φ6 = AF(GreenLight)
- φ7 = EF(RedLight ∧ EF GreenLight)

In [65], a set of requirements is presented with respect to the normal mode. The requirement, called R11bis, states that "when the command line is working (normal mode), if the landing gear command handle has been pushed down and stays down, then eventually the gears will be locked down and the doors will be seen closed" [65]. We expressed this requirement in the three formulae φ1, φ2 and φ3. Formula φ1 states that whenever the command is pushed down, eventually the gear will be extended, and the doors will be closed. Formula φ2 expresses that there is a computation where the command stays pressed down until the gear gets extended and the doors get closed. Formula φ3 is read as follows: along all computation paths in the future, the gears will not be extended, and the doors will not be closed if the button has never been pressed down before. Formula φ4 expresses the safety requirement, which plays an important role in avoiding the bad situation saying that the button has been pressed down and along all the paths, the green light is never lit. Formula φ5 is another safety property expressing that the button will eventually be pressed down and then the green light will be lit. Formula φ6 expresses the liveness requirement, saying that along all the computation paths, the green light can eventually be lit. Formula φ7 is another liveness property expressing that the green light will eventually follow a possible red light. According to our approach, we have 27 possible combinations of coverage properties. Moreover, the quantifier ranging over all the computation paths ("A") enables us to check the status of both normal and emergency modes. For example, the liveness formula φ6 allows us to check the status of the good event ('green light') that will happen eventually in each mode.

4.4.2 Test case generation

The defined 50 requirements were evaluated to be true on the model M using MCMAS+ in 0.167 seconds using 6.61 megabytes of RAM (see [91]). Figures 13 and 14 display the witness traces of φ5 and φ7, respectively. Next, we proceed to create test cases from generated witness traces. For example, the global test case generated from the witness trace of property φ5 in Figure 13 is as follows:

Table 15: Global test case from property φ5's witness trace

Start State	***Transition Label***	**End State**
S0	*I-LandingParameters-O-PressDownButton,I-PressDownButton-O-PressDownButtonAck,null*	*S1*
S1	*I-PressDownButtonAck-O-WaitForOrangeLight,I-ProcessReceivedCommand-O-OpenGearDoors,null*	*S2*
S2	*null,I-OpenGearDoorsAck-O-OutgoingGears,null*	*S3*
S3	*I-OrangeLightOn-O-OrangeLightOnAck,I-OutgoingGearsAck-O-OrangeLightOn,nul*	*S4*
S4	*null,I-OrangeLightOnAck-O-CloseGearDoors,null*	*S5*
S5	*null,I-CloseGearDoorsAck-O-DoorsCloseSuccess,null*	*S6*
S6	*I-GreenLightOn-O-GreenLightOnAck,GearsExtended-O-GreenLightOn,null*	*S7*

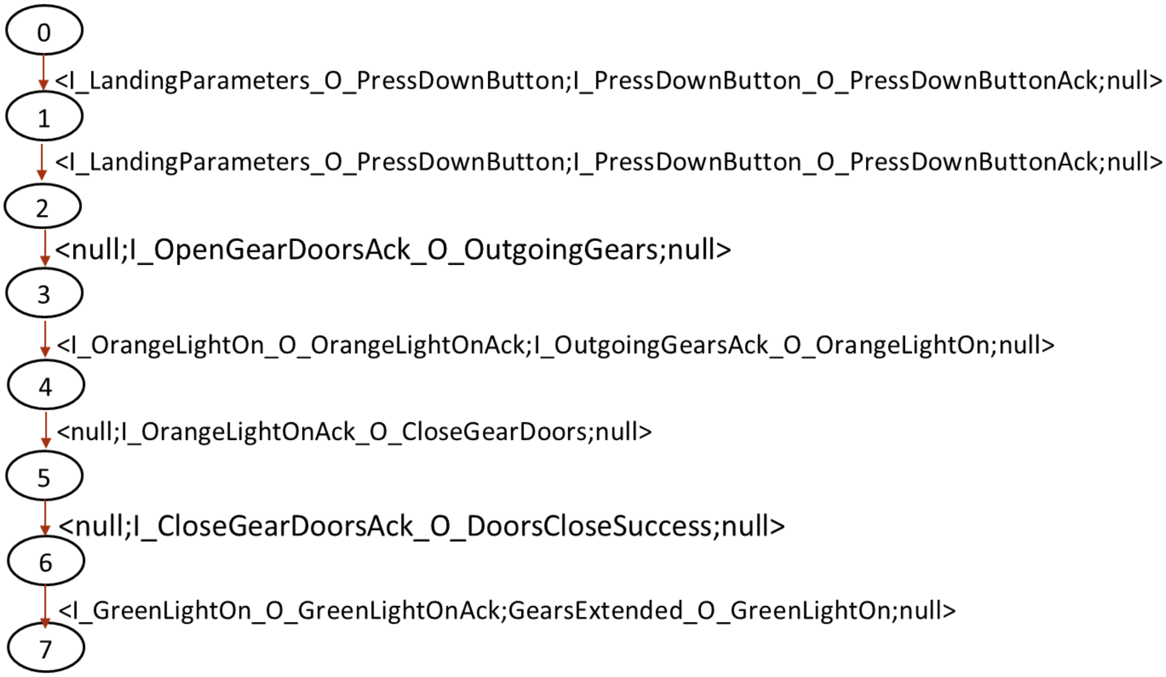


Figure 13: The witness trace of property φ5

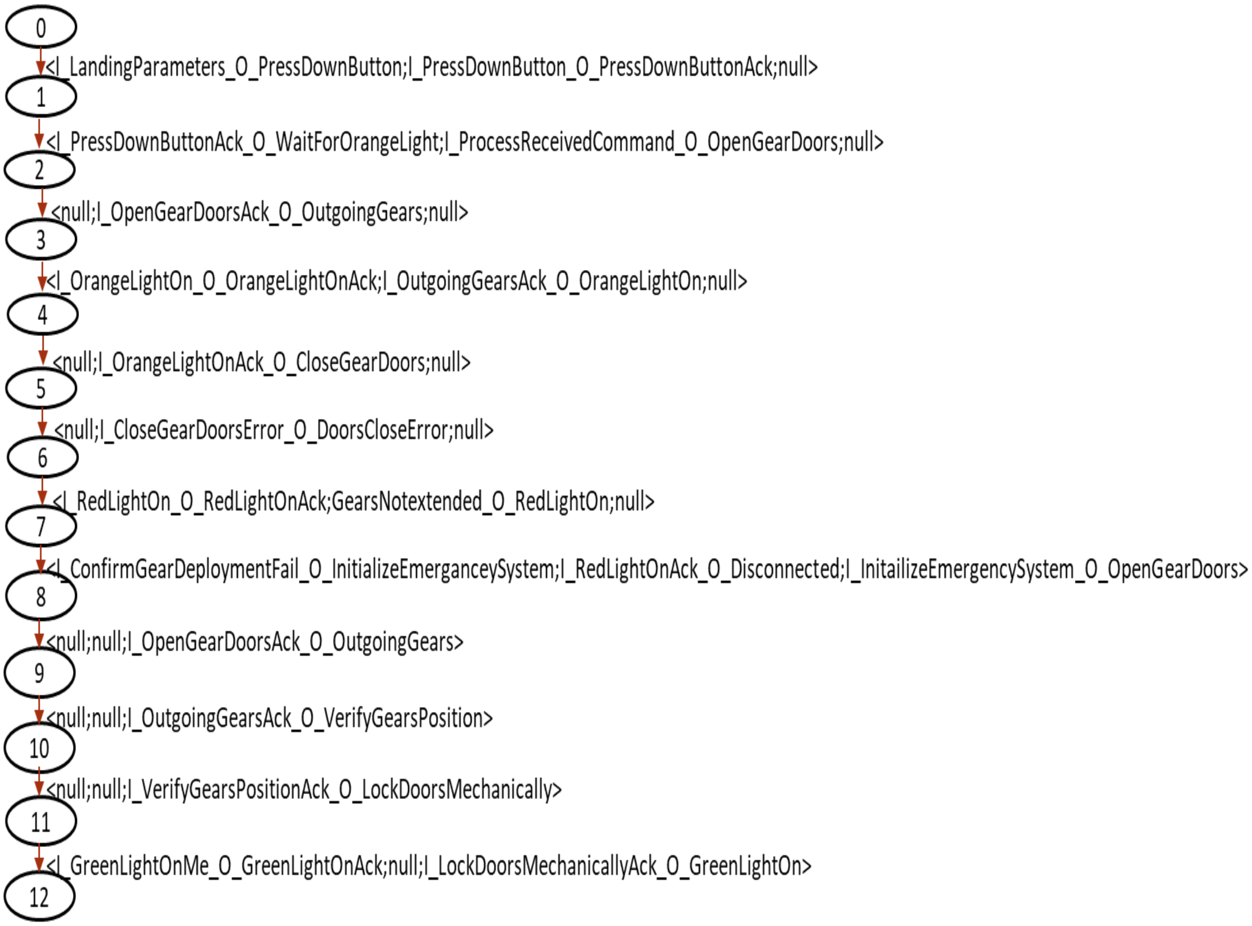

0
<I_LandingParameters_O_PressDownButton;I_PressDownButton_O_PressDownButtonAck;null>
1
<I_PressDownButtonAck_O_WaitForOrangeLight;I_ProcessReceivedCommand_O_OpenGearDoors;null>
2
<null;I_OpenGearDoorsAck_O_OutgoingGears;null>
3
<I_OrangeLightOn_O_OrangeLightOnAck;I_OutgoingGearsAck_O_OrangeLightOn;null>
4
<null;I_OrangeLightOnAck_O_CloseGearDoors;null>
5
<null;I_CloseGearDoorsError_O_DoorsCloseError;null>
6
<I_RedLightOn_O_RedLightOnAck;GearsNotextended_O_RedLightOn;null>
7
<I_ConfirmGearDeploymentFail_O_InitializeEmergenceySystem;I_RedLightOnAck_O_Disconnected;I_InitailizeEmergencySystem_O_OpenGearDoors>
8
<null;null;I_OpenGearDoorsAck_O_OutgoingGears>
9
<null;null;I_OutgoingGearsAck_O_VerifyGearsPosition>
10
<null;null;I_VerifyGearsPositionAck_O_LockDoorsMechanically>
11
<I_GreenLightOnMe_O_GreenLightOnAck;null;I_LockDoorsMechanicallyAck_O_GreenLightOn>
12

Figure 14: The witness trace of property φ7 [91]

4.5 Conclusion and discussion

The main contribution of this work lies in proposing a novel formal approach that effectively addresses the open challenging issues of modeling, verifying, and testing critical avionics systems. We applied the proposed approach to the landing gear system as a real, typical, and complex case study. We modeled each component in the system as an intelligent agent and introduced a new formalism for intelligent systems called extended interpreted systems, which supports autonomy, communication, input and output actions, predicate conditions, and post-conditions. We also used the formal logic of conditional commitments and their fulfillment to model social communication and its content among intelligent interacting agents while capturing their accountability. The MCMAS+ symbolic model checker was used to run the verification of the landing gear system model, encoded in EISPL+, the extended input language compatible with our extended formalism, against coverage criteria and temporal properties expressed in CTLCC. A new testing approach was introduced, it follows a test-driven development approach and performs unit testing, component testing, and system testing in each increment, and directly uses MCMAS+ to automatically generate counterexamples and witness traces to satisfy new coverage criteria and create concrete test suites.

The performed experiments showed the efficiency and scalability of the developed approach (see [91]). The reported results show that up to 35 agent machines having a large state space (1.4418e+06 states) can be verified in 117.854 seconds with 84.892 megabytes of memory. The time and space complexities of the developed approach have polynomial trend functions of order 4, which are computed experimentally. To determine the response time and the required resources, we computed analytically the expected time and memory values of up to 500 agent machines. These values are 458917.742 minutes and 49566.428kilobytes of RAM. We compared our approach with the transformation-based approach that makes use of NuSMV. The results showed that our approaches scale-up better than the transformation approach. According to our positive results, the developed approach can complement and enrich the testing activities usually employed in avionics system settings effectively. It also plays a key role in detecting design errors and cease their propagation to implementation. Therefore, designers can start with formal verification to remove design errors and ensure that designed systems meet certain requirements, and then embark on generating test suites by mapping counterexamples and witness traces of these requirements. Although the paper presents novel contributions to both theoretical and practical aspects of verifying and testing intelligent and autonomous avionics systems, this work still has some limitations that require further investigation. These limitations are mainly related to the consideration of two aspects of particular significance in critical and intelligent avionics systems. The first aspect is about expressing, verifying, and testing properties with timing constraints. This type of properties is needed to express requirements with deadlines and explicit time that branching temporal logics cannot express. The second aspect is the modeling of the uncertainty and verification and testing of probabilistic behaviors. Reasoning about non-deterministic choices and computing the probabilities of specific properties such as deadlock, liveness, and safety and of specific actions and events are highly desired in the context of critical systems. As future work, we plan to investigate the extension of our approach to model check and test the broad class of intelligent cyber physical systems where different complex physical and software components are deeply intertwined. We will also consider timing constraints to express quantitative coverage criteria with deadlines and real-time properties. We also plan to consider the probabilistic aspect and uncertainty of the system by making use of probabilistic interpreted systems and Markov Decision Processes (MDP) as well as probabilistic model checking so that the stochastic properties of the system and their coverage can be supported. Exploiting deep and reinforcement learning to learn the dynamic behavior of the components so we can better adapt the verification and testing strategies to the emerging behaviors of the autonomous components is another challenging line of future research. These extensions will contribute to solving demanding theoretical and practical problems and will have a direct implication of developing secure and safe cyber physical systems.

The proposed approaches, MBV&T and MCBT, are complementary, as they address different concerns in testing and verification. The first approach has to answer the question of avionics software certification using automatic test case generation with an MC/DC criterion. It also must address issues related to the propagation of verified properties to the final product and keep such testing as the mandatory activity for software certification. The second approach explores model checking and MAS to generate test case that addresses the communication and the commitment between agents. It generates test cases for different levels of testing such as unit, integration and system testing. The coverage criteria are state, transition, protocols, and paths. A comparison between the two approaches is summarized in Table 16.

Table 16: Comparison between the first and the second approaches

Criteria	Integrated verification and testing approach	Model checking-based test case generation approach
Objectives	Integration of verification and testing techniques ✓ Verify Properties at design level and ✓ Testing based on LLR ✓ Test the implementation for properties' propagation. ✓ Forward Traceability	Generation of test cases based on Model Checking and multi agent systems. Traceability Test Results Analysis

Criteria	**Integrated verification and testing approach**	**Model checking-based test case generation approach**
Levels	Conformance testing (Black Box Testing) Low level requirements verification and testing ✓ Unit testing ✓ Module testing ✓ System testing	Properties/ requirements-based test case generation, fulfillment Black Box Testing ✓ Unit testing ✓ Integration testing ✓ System testing
Agent Model	EFSM	EFSM
System model	Verification: reachability tree ✓ Testing: partial cross product, guided by communication points between Parallel CEFSMs Agents (Multi-agent systems). ✓ Uses symbolic execution. ✓ Truth table for MCD	Verification: Cross product (reachability tree) Multi Agent Systems
Coverage	Verification: requirements as properties Testing: all Du-paths, MC/DC, UIO (including state and transition) Conformity to RTCA DO 178C.	States, Transitions, Protocol, temporal properties (paths)
Assumptions	Testing: deterministic, coverage-based	Levels of abstraction
Limitations	Infinite input domain, non-exhaustive testing, state explosion	State explosion Limited coverage

Chapter 5

5. Granular traceability between low level requirements and test cases

This chapter presents a model-based test generation using constraints solving that supports granular bidirectional traceability between low level requirements (LLRs) and test cases. Requirements' traceability is mandatory in developing safety-critical systems as prescribed by safety guidelines, such as the RTCA DO178C, and is vital for avionics. Testing is also mandatory for requirements' validation to ensure the safety and quality of software products. Requirements' traceability along the development cycle is a must as illustrated in Figure 3 [1]. Requirements traceability is used to gather traceability artifacts that are used for avionics software systems certification. We present the state-of-the-art of requirements' traceability and the motivations for this work next, followed by forward and backward traceability and test generation.

5.1 Requirements' traceability, state-of-the-art and motivations

Researchers have extensively studied requirements' traceability [98, 99, 100, 101, 102, 103, 105, 111, 112, 113, 114]. Standardization organizations have established satisfiability criteria in several application domains. The IEEE standard 830-1984 [108] states that: "A software requirements' specification is traceable if (i) the origin of each of its requirements is clear and if (ii) it facilitates the referencing of each requirement in future development or enhancement documentation.". Gotel and Finkelstein [98] express the concept in a more complete way: "Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases.)". Pinheiro [99] wrote "Requirements traceability refers to the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements.". Tools have been developed for the first part of the development cycle. Surveys of existing tools that deal with the first part of requirements engineering can be found in [105, 113]. Some research papers address bidirectional traceability between requirements and use cases [101, 111, 112, 114, 115]. A few techniques exist; either manual techniques such as requirements matrices [104], or techniques based on specification languages' transformation [114]. To our knowledge, no existing work offers the low-level granularity that supports bidirectional traceability and coverage assessment on models' feature elements.

Requirements traceability is an established research domain. The focus has been primarily on the first part of requirements engineering, which covers requirements' derivation from natural languages and semi-formal specification languages. Requirements engineering automation needs the development of meta models and models to specify traceability artifacts to be used for certification. High-level languages such as SysML, UML, AADL can be used for automatic model extraction if their compilers and interpreters are qualified.

Despite significant research efforts, requirement traceability remains a challenging problem in avionics software systems development. Requirements' coverage is required and conformity to the RTCA DO178C standard [1] is mandatory as described in Figure 3. Avionics software needs certification before its integration into an airplane system. The certification process requires the creation of certification artifacts and their validation. All certification artifacts are produced during the development life cycle. Testing is mandatory and should satisfy the Modified Condition and Decision Coverage criterion (MC/DC). There is a need for bidirectional traceability between requirements levels and test cases.

Up to now, the development of avionics systems has been heavy on human intervention, which explains its high costs in terms of development and time. Engineers develop system specification, extract high-level requirements (HLRs), refine them into LLRs, design, implement, and develop tests in accordance with the RTCA DO178C standard [1]. Tools that can support automation of the development process require qualification before their integration and use within the development cycle. Avionics industries are looking at model-based development as a viable way of developing their systems, enhancing their safety and quality, and reducing their costs and development times.

Testing based on models has witnessed extensive research as shown in these surveys [7, 8, 9, and 30]. However, the traceability related to the last part of the safety critical software development process that contains test derivation to validate implementation code and assess the backward requirements traceability chain from test cases to requirements has received less attention. The creation of traceability elements and certification artifacts to assess requirements' coverage at low levels of granularity is not yet well studied. Test generation methods exist, but they do not include the creation of traceability elements and coverage data that support bidirectional requirements' traceability and the retrieval of traceability data.

We present a test case generation process that supports requirements traceability, and we propose a granular requirements traceability approach that extends traceability between system specifications and HLRs, between HLRs and LLRs, and between LLRs and test cases by creating the necessary traceability elements on an EFSM model and its corresponding graphs (DFG, CFG) during test case generation. Test case generation must satisfy the MC/DC criterion, as mandated by the RTC DO178C [1]. This approach offers forward traceability by construction and supports backward traceability using graph exploration algorithms to collect traceability elements and artifacts. To create traceability elements and artifacts we use identification, links (ID), links, coverage elements, graph labeling techniques, complex data structure, repository, and databases. An overview of the MC/DC-TG-RT-Tool's architecture is depicted in Figure 21 [86, 87, 117].

Our approach to requirements traceability is focused on traceability element determination, identification, creation, storage, and retrieval at a low-level of granularity. The granularity addressed is related to coverage elements for test case generation that handles test coverage criteria such as the MC/DC criterion that is mandatory to satisfy the RTCA DO178C standard [1] in addition to the Du-path criterion.

The traceability relationship between HLRs, LLRs and test cases has to be established and validated as given in Figure 3. The type of relations that exist between those artifacts can be bijection, one to many, or many to many. Bijection relation is the easiest, but it is not the most common. The other relations require the management of lists of source artifacts and lists to target artifacts. Often a qualification is given as a relation, such as dependency relations between requirements. The extension of traceability relations to coverage elements during the creation of traceability data must support and propagate all modes of traceability and all types of relations.

In the following we describe the requirements traceability approach and how it is supported by our proposed test case generation with constraints solving.

5.2 Traceability approach supported by test case generation

Each requirements traceability approach should have the following: trace element definition, identification, creation, storage, retrieval, utilization, and maintenance.

The proposed requirements traceability approach supports traceability links from the origin artifacts to requirement specifications given as UML profile documents that represents High-level Requirements that is the target artifact. Direct transformation of specification languages can be used, down to Low level requirements (LLRs) from which we perform model extraction to obtain an EFSM behavior model. We use model-to-model transformation to obtain control flow graphs (CFGs) and data flow graphs (DFGs) from the EFSM model, we determine coverage elements using graph features, use labeling, IDs, links to graphs and original artifacts, and we produce records along with the generation of test cases (paths) for future uses.

In this work, the focus is on the latest leg of the requirements traceability chain. We address the low granularity of requirements traceability from HLRs as the origin artifact to test cases as the target artifact, and backwards from the target to the source. We create traceability elements using labels, IDs, links, locations on graphs, and the recording and storage of traceability artifacts of requirements and test cases during their generation. We achieve forward traceability from LLRs to test cases by construction. However, backward traceability requires the use of graph exploration algorithms to collect traceability artifacts and validate the traceability relationship with the test cases and the LLRs. The other usage of traceability collection is coverage assessment of the MC/DC criterion. Figure 15 depicts the entire traceability approach.

More specifically, the proposed traceability approach starts with high level requirements (HLRs) that are specified with UML. Abstract models such as EFSMs and CEFSMs are extracted from the UML specification. These are models for low-level requirements (LLRs) that can be used for automatic test case generation. In this work, we assume that HLRs and LLRs have been specified and verified with industrial partners and research collaborators. The aim is to answer the traceability question using specified requirements as EFSMs or CEFSMs and the defined traceability elements to ensure bidirectional requirements traceability between LLRs and test cases.

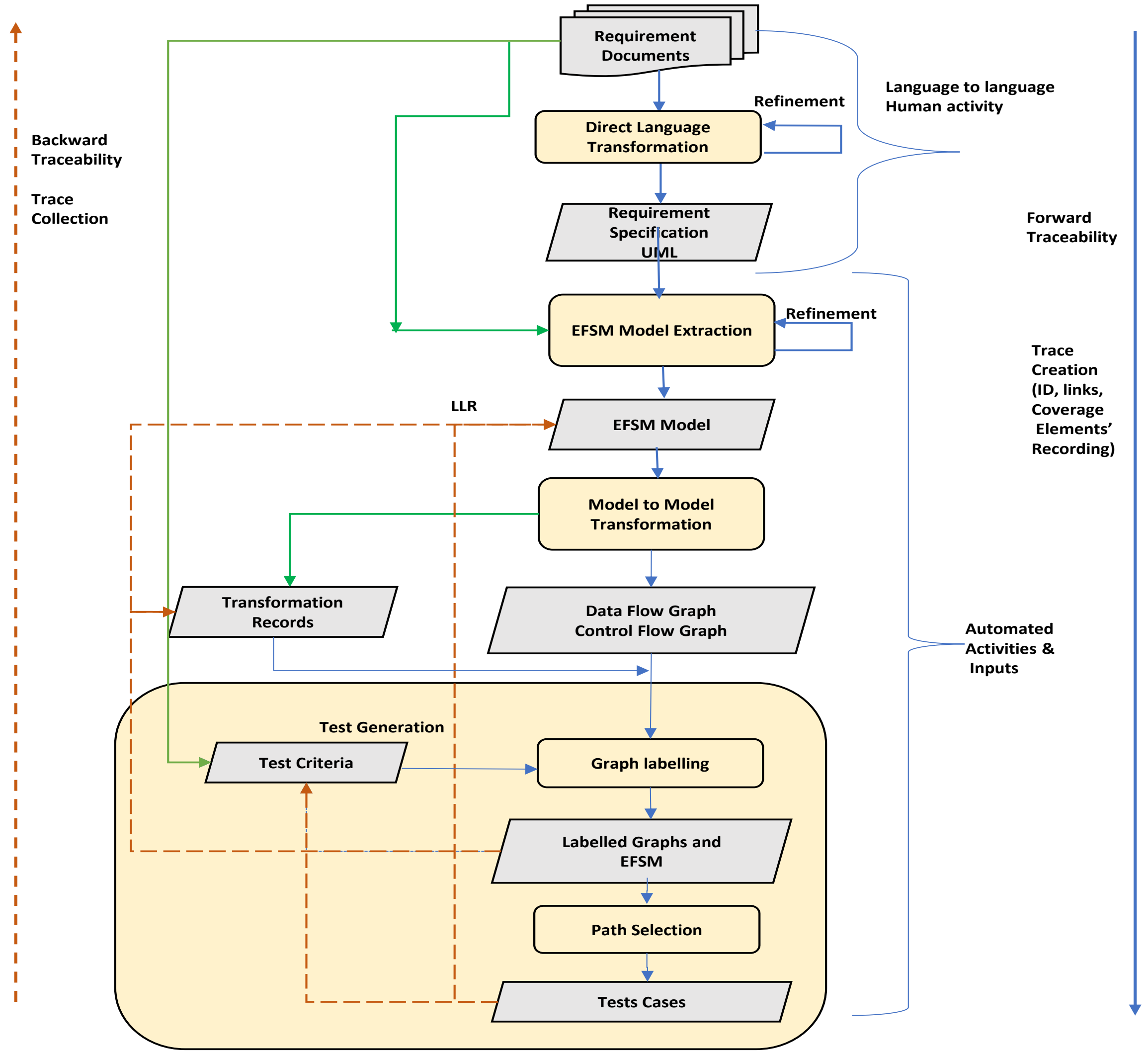


Figure 15: Test case generation that supports traceability, approach overview.

In the following we present how forward and backward traceability are viewed in this approach.

5.2.1 Forward traceability

In this approach, forward traceability from LLR to their related test cases is obtained by construction, as in the case of test-case generation t is guided by test coverage criteria. In our case, the test case generation uses coverage MC/DC and du path criteria and selects paths using constraints solving and exploration algorithms. MC/DC requirements are satisfied and validated during test case generation such that all MC/DC requirements are met by construction. Figure 15 shows forward traceability downstream the test case generation approach (blue vertical arrow). Traceability elements are created and stored in each activity of test generation process.

5.2.2 Backward traceability

To support backward traceability from test cases to LLR, the approach uses graph features as a definition for traceability elements and creates traceability elements along with test case generation. Coverage elements are determined for source and target artifacts as shown by the blue vertical arrows on Figure 15. The EFSM is transformed into DFGs and DFCs using graph rewriting, and transformation records are created and stored. These records locate each traceability element (decision node, edges, predicates, variables, truth tables, and all coverage elements in data structures) using a traceability repository and databases. Each source artifact (HLR) has an ID. Each target artifact (LLR) carries the links of its origin artifact(s) and has an ID; each extracted EFSM carries the ID(s) of its source LLR(s) and has an ID. Each EFSM is transformed to CFGs and DFGs that inherit its ID, and the EFSM, CFGs and DFGs are labeled. Nodes, edges, predicates, and variables are located on their corresponding graphs and their source EFSM and the chain of their source ancestors (LLRs and HLRs) are noted. The existing relationships between LLRs and test cases can be one-to one (bijection), meaning that each LLR is covered by one test case, one-to-many, which means that one LLR is covered by many test cases (which is often the case), or many-to-many. In addition, two or more LLRs may interact with each other and share test cases that we represent by a global test case.

Requirements’ identifiers are also used to retrieve paths during test case generation. Each test case that is generated has the ID(s) of the LLR(s) they are derived from. Backward traceability is then used to check that a test case or a set of test cases cover the LLR(s). This backward traceability can be done using graph exploration algorithms to recover traceability data from traceability repositories and data bases. The documentation that is produced constitutes traceability artifacts that can be used for avionics software certification purposes.

Next, we present an overview of the test case generation approach that offers traceability element creation and forward traceability by construction. Our approach offers an automatic and detailed traceability of coverage elements that distinguishes it from existing works.

5.3 Test case generation approach that supports requirements traceability

This third approach generates local test cases for component testing. It uses an EFSM model like the one used for the first approach. The main coverage criteria are MC/DC and Du-path. We assume that a truth table associated with each decision is given as an input and will be used for test case generation. To satisfy the MC/DC criterion in the code we need to satisfy 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.

In this approach, the satisfaction of MC/DC applies to a model that is assumed to be close to the abstract implementation model, and the predicates are the same as those used in the implementation. In addition, the decisions are decomposed into a simple form.

The proposed traceability approach is supported by model-based test case generation. It starts with the generation of local test cases based on an EFSM specification representing an LLR. The test case generation uses graph rewriting as a means for model-to-model transformation to obtain a Data Flow Graph (DFG) and a control flow graph (CFG) of the EFSM. In the presence of interacting LLRs, we assume that each LLR is modeled as an EFSM, and their interactions are modeled by the composition of EFSMs (CEFSM) from which we extract a communication graph that shows their relationships. The communication graph guides the generation of global test cases to reflect the interaction between LLRs. If a test case contains a node that belongs to a communication graph, it means that 2 LLRs are in a relationship. In our case, we use a test case generation algorithm that generates test cases that cover LLRs and satisfies the MC/DC criterion that is mandatory by RTCA 178C. The details of test case generation can be found in [87, 116, 117].

The limitation of this test case generation approach is mainly related to the complexity of its algorithms. This approach generates local test cases, and the communication aspect is not addressed.

Figure 16 shows a high-level overview of the test case generation approach. The idea is to build on our previous test case generation technique [71, 86, 87, 117], it shows where we defined traceability elements at a low level of granularity. We have added traceability element creation, transformation records, identification, links, a retrieval process, and utilization for traceability and coverage analysis. The maintenance for this level of granularity requires redoing all the processes if a request for change modifies the EFSM specification.

5.4 Traceability element creation

Traceability between requirements and test cases requires traceability element creation during the test case generation process. Traceability element creation is designed with Forward Traceability from LLRs to test cases and should show the coverage of LLRs by construction. Furthermore, the creation of traceability elements should also create traceability records to explore the labeled graphs and collect traceability elements for assessing Backward Traceability. The main issue is to determine the traceability elements required for traceability creation processes. In our case, we need first, the IDs of all the LLRs that propagate links to their corresponding HLRs and the complete chain of ancestors' artifacts. Second, we need all the model's traceability features (edges, nodes, predicates, and variables, and coverage elements per type of graph (CFG, DFG) that are obtained by model-to-model transformation and labeling processes. During the test generation process additional traceability elements are recoded to link each model's traceability elements to their LLRs, such as executable and non executable paths.

To show traceability elements creation within the process of test case generation, we need to describe the test case generation process, the creation operations, and what we obtain from the process. Figure 16 shows where traceability elements are created in green.

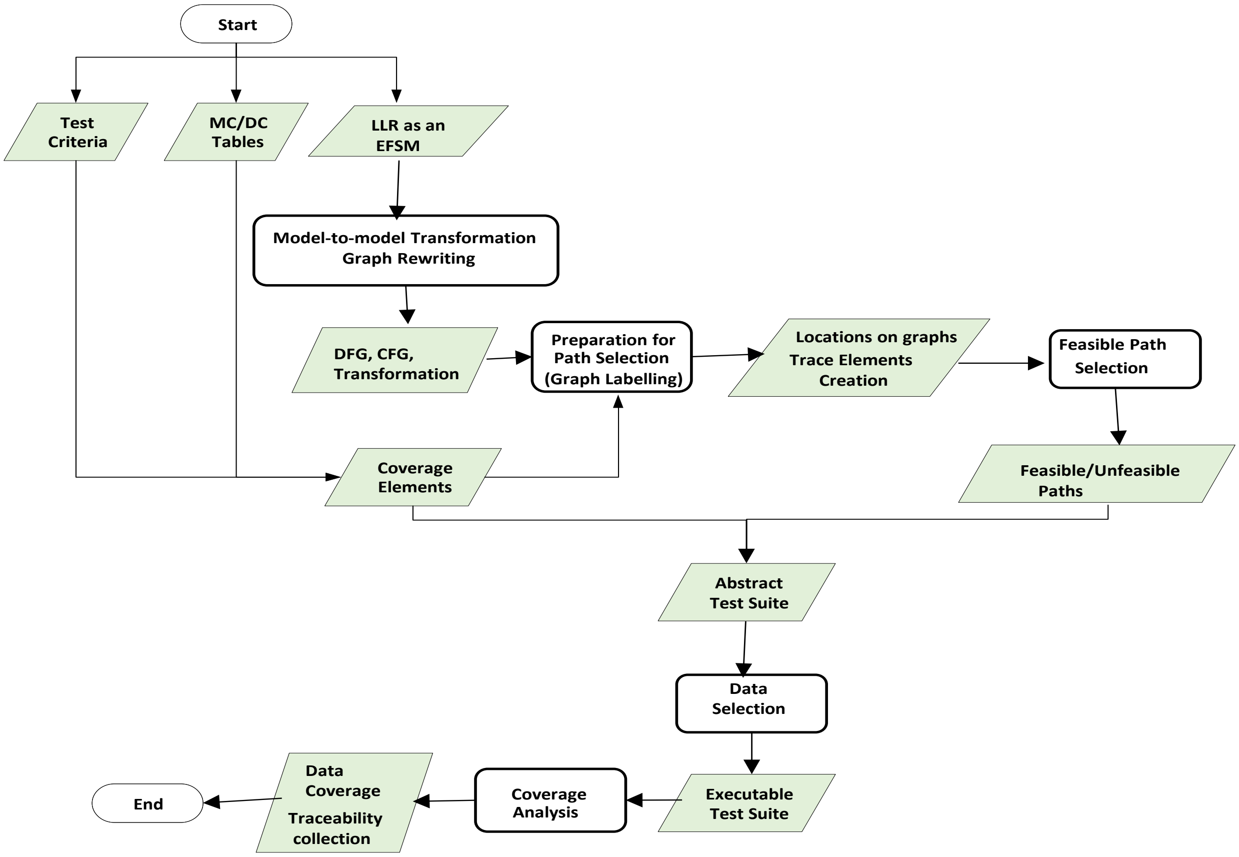

Start
Test Criteria
MC/DC Tables
LLR as an EFSM
Model-to-model Transformation Graph Rewriting
DFG, CFG, Transformation
Preparation for Path Selection (Graph Labelling)
Locations on graphs Trace Elements Creation
Feasible Path Selection
Coverage Elements
Feasible/Unfeasible Paths
Abstract Test Suite
Data Selection
Executable Test Suite
Coverage Analysis
Data Coverage Traceability collection
End

Figure 16: Traceability elements' creation and collection in EFSM-based test generation.

5.5 Test generation steps

Here we overview the key steps of the approach depicted in Figure16.

Automatic model-to-model transformation is utilized to obtain both control and data flow graphs using a graph rewriting technique (EFSM → (CFG, DFG graphs)). The EFSM is linked to its source artifact. All elements of the EFSM (nodes, edges, predicates, variables,…) are identified and records are created and stored in the database.

Step 1: Use of graph rewriting to achieve model to model transformation

Graph rewriting is a technique that helps create a new graph from an original graph using an algorithm. It is like the translation between languages using grammar. The algebraic approach to graph rewriting is a rigorous approach based on category theory as defined in Rozenberg [93]. There are several sub-approaches; the one used in our approach is known as the single-pushout (SPO) approach. In the literature, graph grammar is used as a synonym for a graph rewriting system. The definitions of the following are needed to formally transform models: Grammar, Rule Graph, State Graph, Match, Rule Morphism, Rules and Rule Application. Figure 17 provides an overview of the graph rewriting approach.

In our case, the grammar has been defined for the source and target models, as given in Table 17. For more details, please refer to our papers [71, 86] and to Amine Rahj's thesis [117]. In this model transformation, both the source and target models are identified and linked. Traceability elements at the level of granularity of the graph including those related to nodes, edges, predicates, and variables are identified and records are then created as per Figure 17.

Table 17: 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 Bloc	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

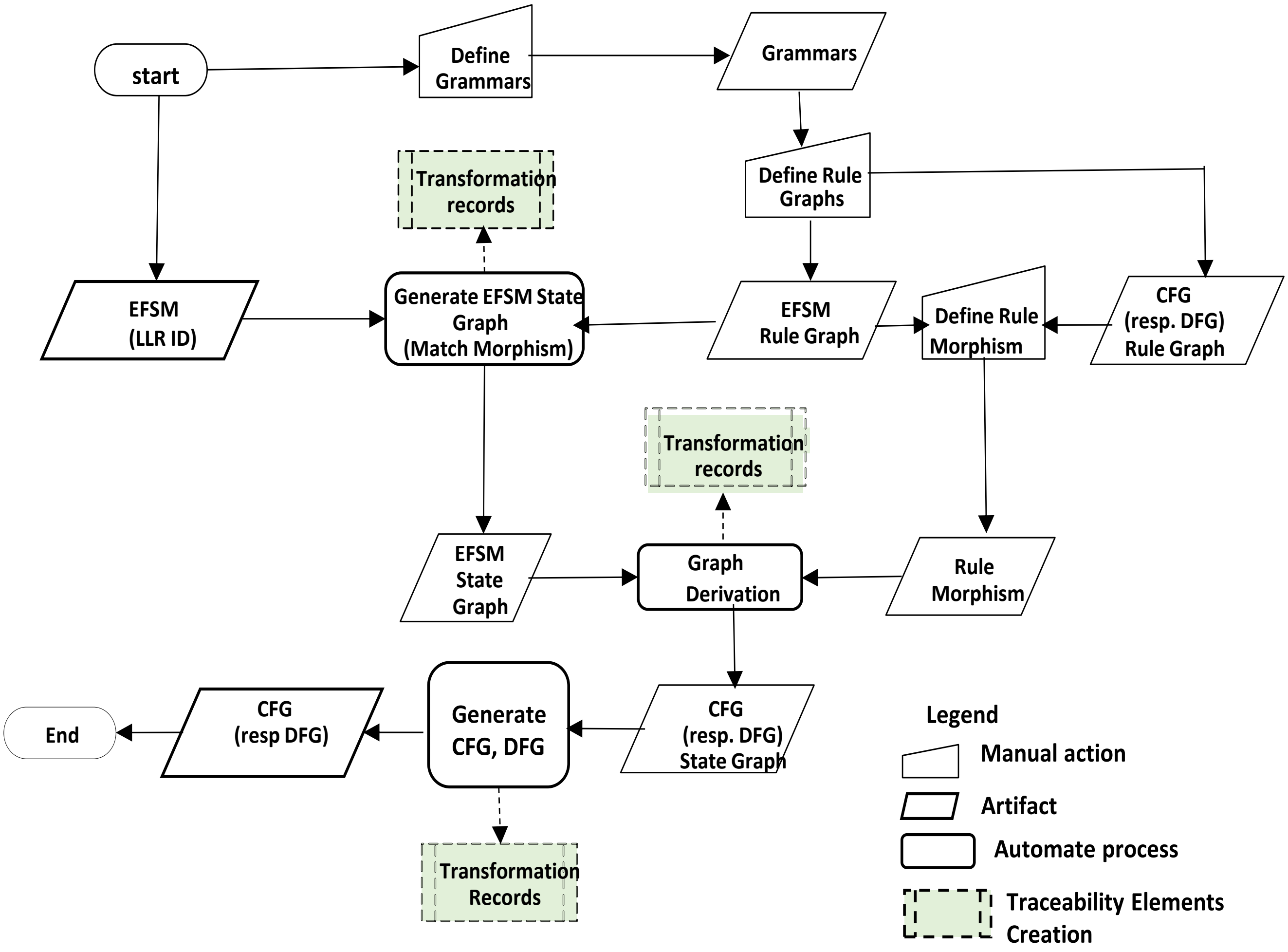


Figure 17: Traceability elements and records creation during graph rewriting

Step 2: Preparation for Path labeling

Figure 18 below shows the steps of graph labelling. There are four types of information that we want to pinpoint on the graphs' elements. The MC/DC Tables (or Decisions) are affected by the graph elements, the Rows, the Conditions, and the values of the Conditions. The final label depends upon each graph element, as shown in Figure 18. 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 (with one predicate). Then for each table, we label the outgoing branches from the decision points with the row ID that matches the Decision outcome of that row. We also label the predicate edges from the DFG by means of transformation records. Fnally, for each condition we move to labeling the definition-usage (def-use) for all variables affecting that Condition on the DFG.

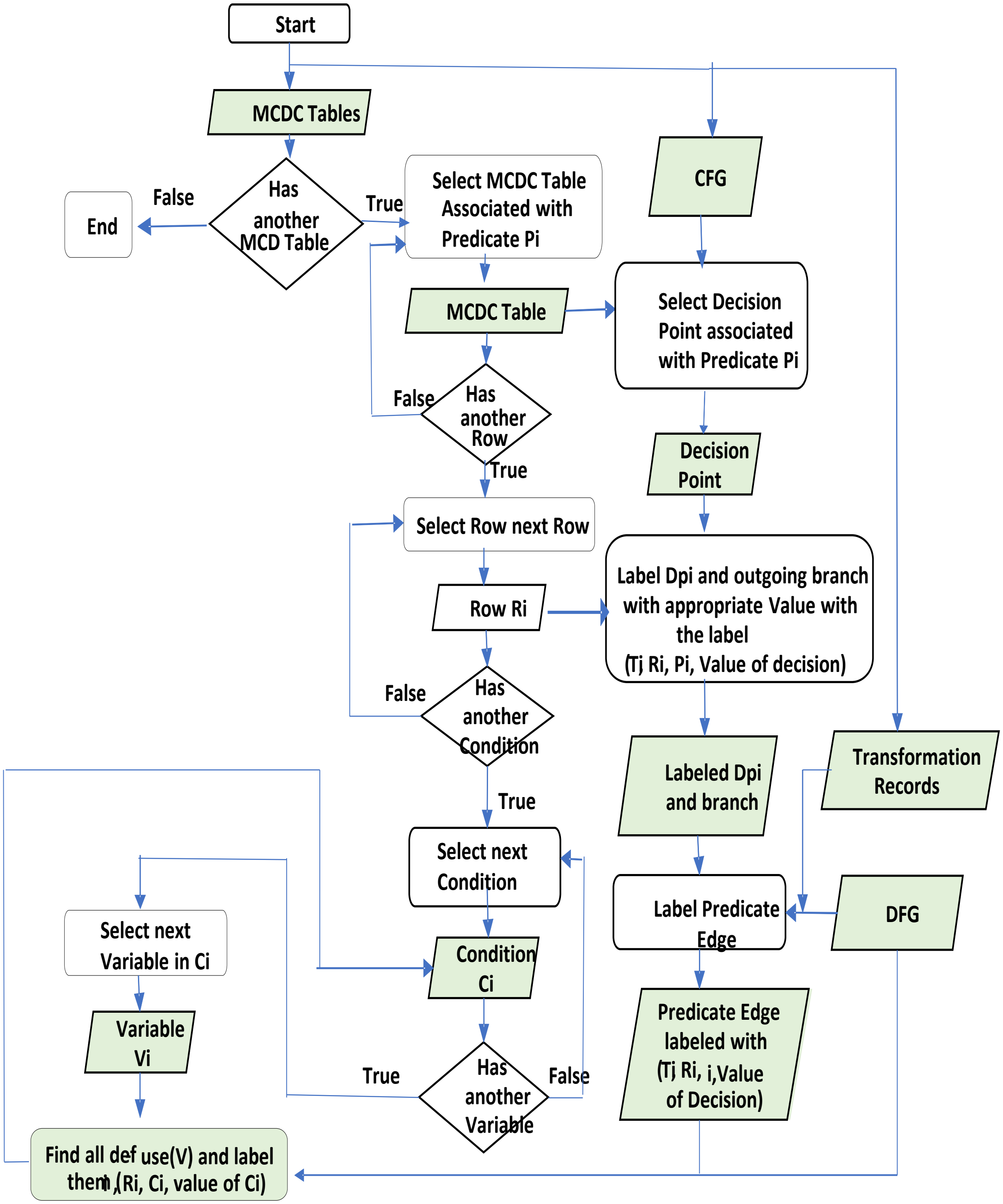

Start
MCDC Tables
Has another MCD Table
False
End
True
Select MCDC Table Associated with Predicate Pi
CFG
MCDC Table
Select Decision Point associated with Predicate Pi
False
Has another Row
True
Decision Point
Select Row next Row
Row Ri
Label Dpi and outgoing branch with appropriate Value with the label
(Tj Ri, Pi, Value of decision)
False
Has another Condition
True
Labeled Dpi and branch
Transformation Records
Select next Condition
Label Predicate Edge
DFG
Select next Variable in Ci
Condition Ci
Variable Vi
True
Has another Variable
False
Predicate Edge labeled with (Tj Ri, i,Value of Decision)
Find all def use(V) and label them (Ri, Ci, value of Ci)

Figure 18: Traceability element creation during graph labeling (green color)

Step 3: Path Selection

The aim of this step is to select paths that have the potential to produce executable test cases and to decide on their feasibility [118]. In our tool, we use jSMTLIB for parsing SMTLIB expressions and use the solvers with a test generation tool [119]. The following brief description shows how this approach selection of feasible paths is addressed. Search algorithm A* and a multi-objective search algorithm are used for finding the "shortest" path. "Shortest" is expressed in terms of feasibility and the uses of the involved variables. The algorithm A* is used between the "nearest" def-use and the p-use. We also use a multi-objective search algorithm based on Yano et al. [120]. 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. An extract of the modified algorithm given in Amine Rahj's thesis [119] for traceability element creation is given below.

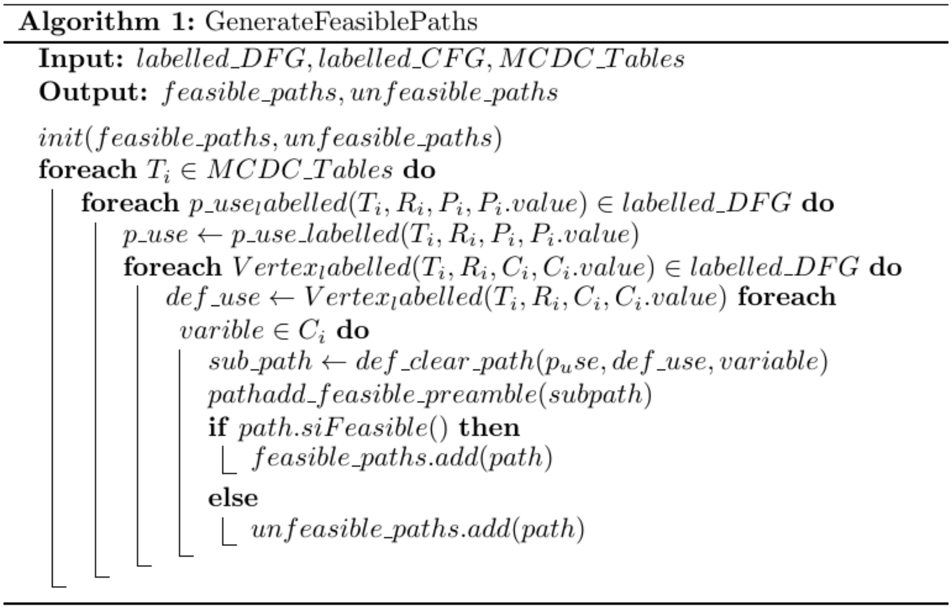
```
Precondition: MC/DC tables, labeled DFG
Labels applied during the previous steps
<T, R, C, Value of C> for def-uses
<T, R, P, Value of P> for p-uses
Where:
T: table
R: Row of MC/DC table
C: Condition
P: Predicate/ decision
For each table T in the set of MC/DC table
  For each (Row) R in T
    Find 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, and Create traceability elements.
```

(*) min-def-use(C) in the nearest of def-uses of the variables involved in C in term of "Approximation Level".

(**) 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 the 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) and Create traceability elements.



```
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
            $unfeasible\_paths.add(path)$
```

All the obtained executable and non-executable paths are recorded and linked to their original artifacts.

Automatic test results analysis

Testing activity is a detection mechanism. To detect faults, the generated test cases (inputs) are applied to a system under test, and the obtained test results must be analyzed to determine if the system under test passes or fails the test cases. Test results analysis requires the comparison between the outputs of the system implementation that are known as the actual outputs and the outputs expected to detect faults. The result of the test results analyzer is called a verdict that is in the set of (pass, fail, inconclusive). The verdict is assigned by a judge or an oracle [117], often a human that extracts the expected outputs from software artifacts and compares them with the actual outputs. An automatic oracle has some challenges that are known as test oracle problems [121, 122]. The major issues with the design and development of an oracle are related to controllability and observability in black-box testing, which are in turn related to the degree of software testability.

In this research we suggest the following simplified test results analysis approach, depicted in Figure 20.

The assumptions for this work are that the EFSM specification exists, and it is deterministic. It will be used to generate a specific test case from the inputs obtained from the trace. The inputs are given to the tester, which will find the corresponding expected outputs and its associated verdict. Then the comparison between the expected and observed outputs can be carried out as depicted in Figures 19 and 20. The trace is a test result. It can also be obtained during the normal operations of a system.

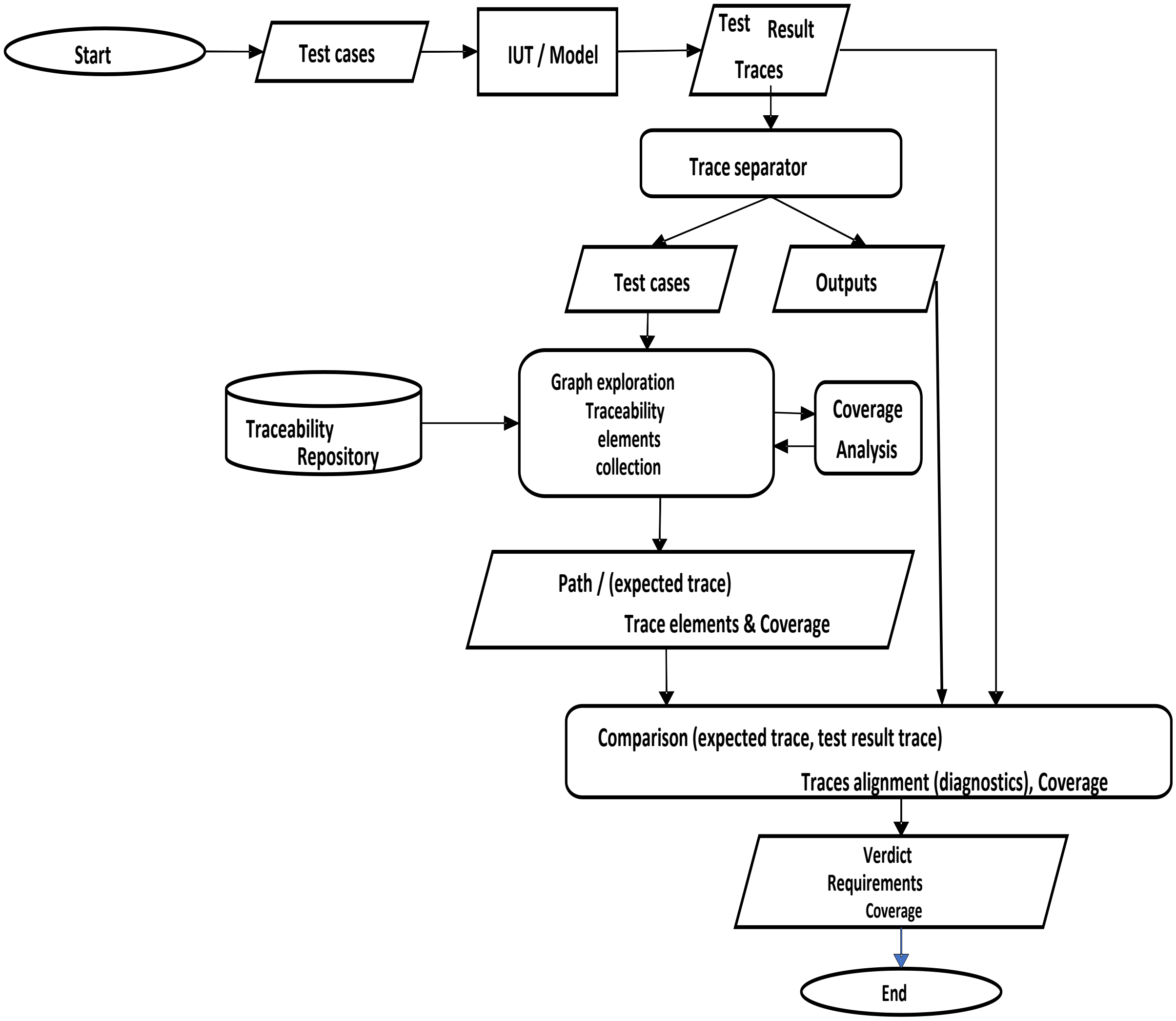


Figure 19: Backward traceability, coverage, and test results analysis.

In this approach, a test results analyzer takes two inputs and delivers an output as the verdict (pass or fail). The first input is an EFSM specification, parsed to obtain an internal form that is suitable for the test case generator (in our case it is in the form of graphs). The second input is a trace that will be separated (by projections) in a sequence of inputs and the observed outputs. The tester uses the internal form and the sequence of inputs obtained from the trace and generates the expected outputs for the input sequence. The role of the comparator is to determine if the observed outputs and expected outputs are equivalents. The comparator is very simple for deterministic EFSMs. In the case of non-deterministic EFSMs, the comparison must be made within the set of valid expected outputs.

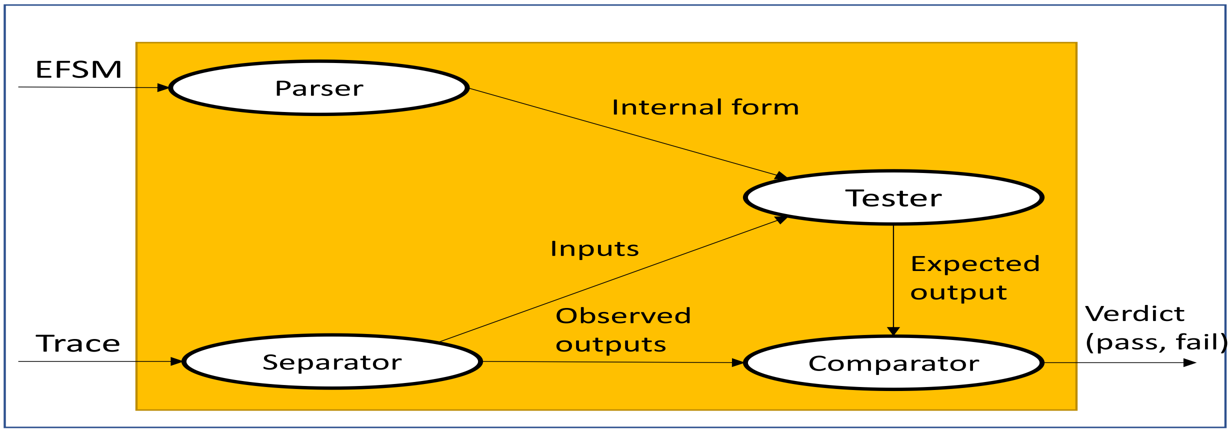


Figure 20: Simple form of Test Results Analyzer

Step 4: Data selection

This step *helps* to obtain feasible paths. For sake of space, this step it not addressed in this paper. Details can be found in [7, 8, 9].

Step 5: Coverage analysis

Coverage analysis is important in test case generation and in backward traceability. Examples of its use include: (1) requirements coverage validation from a given test case that was derived manually, when the covered LLR values are desired; (2) for trace coverage analysis obtained from a run of test cases on an implementation; and (3) for trace failure analysis/diagnostics as given in Figures 19 and 20.

5.6 Architectural view

A test case generation and requirements traceability tool (MC/DC-TG-RT) has been designed and partially implemented. It is a multi-layered architecture with clear design concepts and communication between layers and within each layer. The main module in this approach is the

Test Generation Module. It implements the main routines of the approach. It is supplemented with two auxiliary modules: a Data Module, and a Graph Operations Module. Figure 21 depicts the architecture of MC/DC-TG-RT Tool. The basic architecture design related to test generation can be found in Amine Rahj's master's thesis [117]. This architecture is extended by requirements' traceability capabilities.

In this section, we justify the technical decisions as we outline the function and information 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-purpose graph operations. Its goal is to ensure the maintainability and reconfigurability of the algorithms. The data module retrieves user inputs, constructs and 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-TG-RT tool in read-only mode.

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, such records will also simplify Steps 2 to 5. Recording the direct references to graph morphism images will help to bypass search algorithms using object properties.

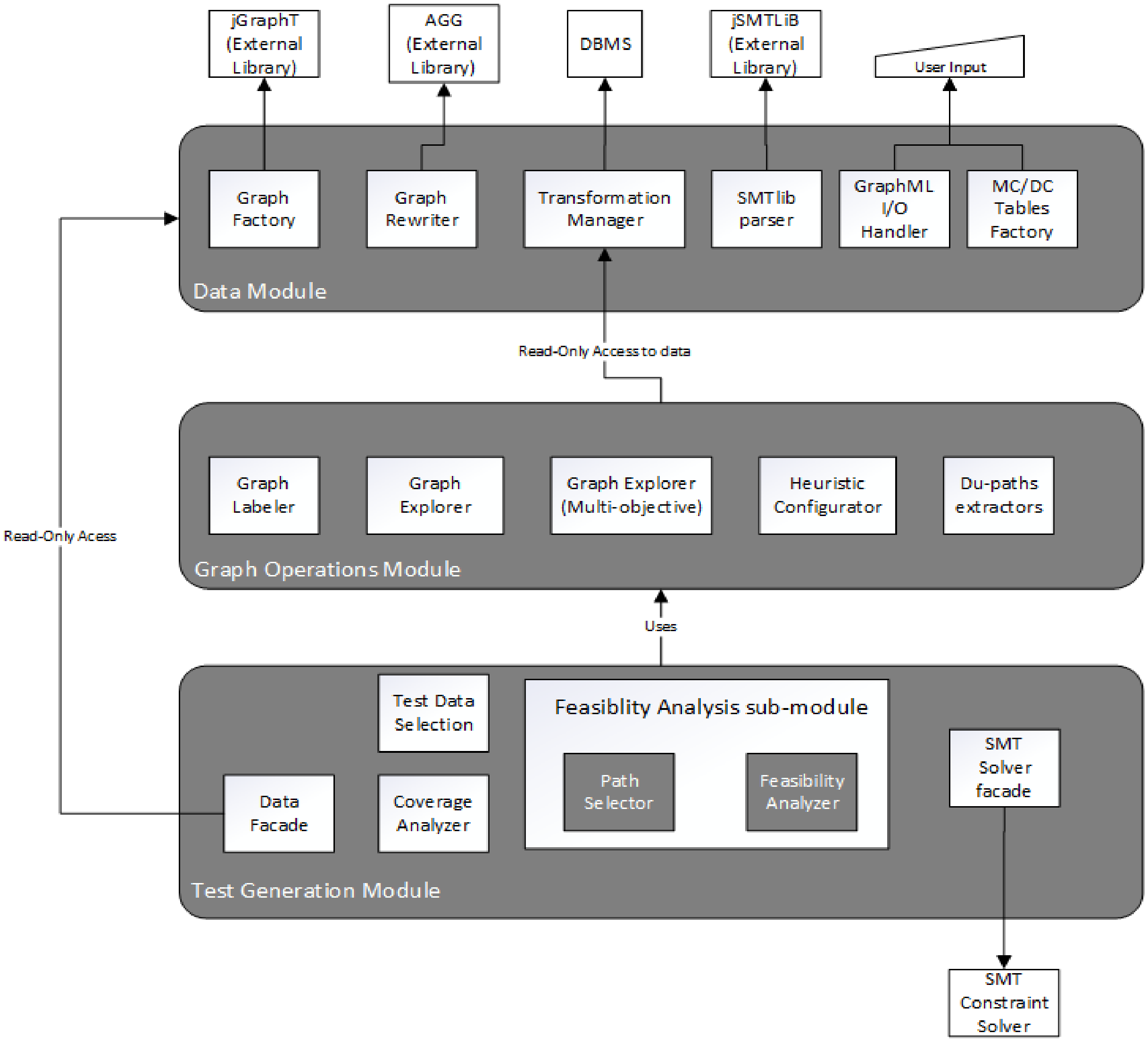


Figure 21: MC/DC-TG-RT Tool architecture [86, 87,117].

This figure is shared between Amine Rahj's work and Mounia Elqortobi's work.

5.7 Conclusion and discussion

In this chapter, we presented a test case generation that supports bidirectional traceability between low-level requirements and test cases. The proposed approach extends requirements' traceability to a very low level of granularity that is still challenging in the field due to the number of traceability elements required to identify, create, store, and recover for analysis purposes. The coverage elements are defined on an extended finite state machine model and its corresponding graphs, obtained by model-to-model transformation using graph rewiring, including their nodes, edges, predicates, and variables that may influence path execution. The creation of traceability elements is performed during test case generation that should satisfy the MC/DC criterion. Forward requirements traceability is obtained by construction. As mentioned before, satisfying the MC/DC criterion is mandatory in avionics industries. The test case generation method uses a constraints satisfaction technique as well as graph exploration algorithms to retrieve traceability elements during backward traceability or trace analysis for diagnostic purposes after failure. To our knowledge, our contribution represents the first technique that reaches this level of granularity in establishing traceable bidirectional relationships between LLRs and test cases in model-based development. The traceability related to the code is beyond the scope of this paper. In future work we intend to: (i) finalize the implementation of MC/DC-TG-RT Tool;(ii) apply the approach to a sizable example with a chain of HLRs, LLRs, Models and graphs, and test cases; and (iii) address non-functional requirements' traceability.

This proposed approach uses a test case generation based on constraints solving. It is an alternate solution to local test case generation that addresses the problem of path feasibility using constraints solving. It has also the capacity of bidirectional requirements traceability using identification, graph labeling and links. Table 18 presents a comparison between the proposed test case generation approaches 1 and 3.

Table 18: Comparison of contributions 1 and 3

Criteria	**Model-based verification and testing**	**Test case generation using constraints solving**
Objectives	Integration of verification and testing techniques • Verify properties at design level and • Test the implementation for properties propagation. • Generate test cases for MC/DC.	Model-based test case generation for one module using MC/DC tables for test case generation, • Constraints solving for path feasibility. • Bidirectional Requirements Traceability.

Criteria	**Model-based verification and testing**	**Test case generation using constraints solving**
Levels	Conformance testing (black box testing). Low level requirements verification and testing. • Unit testing (local test sequences) • Module testing (integration testing) • System testing (global test sequences)	Conformance testing black box testing. • Unit testing • Module testing • System testing (can be extended)
System model	Verification: cross product (reachability tree) Testing*:* partial cross product, guided by communication points Parallel Communicating EFSMs (Agents) Multi agent systems	Component model EFSM Model-to-model transformation (control flow graph and data flow graph) using graph rewriting. MC/DC tables as input, new path selection process, and path feasibility using constraint solving. Requirements Traceability Elements • Creation, Recording, Retrieval • Search Algorithms

Chapter 6

6. Conclusion and future work

The main contributions of this work lie in proposing, in an incremental manner, several approaches that address open issues of modeling, verifying, and testing critical avionics systems. We applied two of the proposed approaches to the landing gear system as a typical and complex case study, modelling each component in the system as an EFSM and their integration as an CEFSM.

The first two proposed approaches are complementary, as they address different concerns in testing and verification. The first approach, which integrates verification and testing, has to answer the question of avionics software certification using automatic test case generation with the MC/DC criterion. It also must address issues related to the propagation of verified properties to the final product and keep the testing process as the mandatory acitivity for software certification. To our knowledge, it was the first approach that integrates verification and testing for the propagation of properties.

In the second approach, we explored model checking to generate a test case that addresses the communication and the commitment between agents. The approach generates test cases for different levels of testing such as unit, integration and system testing. The criteria covered are states, transitions, protocols, and paths. but Dataflow-related coverage criteria are not addressed, nor is the MC/DC criterion. The first approach uses a partial cross product when dealing with CEFSMs, while the second approach uses a complete cross product.

The third approach is comparable to the first in terms of objectives. They both use EFSM models, and generate test cases using MC/DC and Du-path coverage criteria. However they differ in the technique that helps generate test cases. Instead, the third approach uses constraints-solving that addresses path feasibility. In developing the third approach we looked for a type of test case generation that can support traceability and allow the validation of MC/DC. This approach uses a formal model-to-model transformation based on graph rewriting that supports requirements' creation, storage and recovery, which are the foundation for establishing reationships between requirements (levels) and test cases. A comparison between the three approaches is summarized in Table 19 below.

Table 19: Comparison between the 3 contributions

Criteria	**Integration of verification and testing techniques**	**Model checking-based test case generation**	**Test case generation using constraints solving / traceability**
Objectives	Integration of verification and testing techniques Verify properties at design level and Test the implementation for properties' propagation.	Generation of test cases based on model checking and multi agent systems.	Model-based test case generation for one module using MC/DC tables for test case generation. Constraints solving for path feasibility. **Bidirectional Requirements Traceability.**
Levels	Conformance testing (Black Box Testing) ✓ Low level requirements verification and testing ✓ Unit testing ✓ Module testing ✓ System testing	Properties/ requirements-based test case generation, fulfillment Black Box Testing ✓ Unit testing ✓ integration testing ✓ System testing	Component model EFSM Model to Model transformation (control flow graph and data flow graph using graph rewriting. MC/DC tables as input, new path selection process Path feasibility using **constraint solving.** **Requirements Traceability Elements** Creation, Recording, Retrieval, Search Algorithms
Agent Model	EFSM	EFSM	EFSM
System model	Verification: reachability tree Testing: partial cross product, guided by communication points between Parallel CEFSMs Agents	Cross product (reachability tree) Multi Agent Systems	EFSM

Criteria	**Integration of verification and testing techniques**	**Model checking-based test case generation**	**Test case generation using constraints solving / traceability**
Coverage	Verification: requirements Testing: all du-paths, MC/DC (including state and transition)	States, Transitions, Protocol, temporal properties (paths)	all Du-paths, MC/DC criteria (including state and transition)
Assumptions	Testing: deterministic, coverage based	Levels of abstraction	Testing: deterministic, coverage based
Limitations	Infinite input domain, non-exhaustive testing, state explosion. Executability issues	State explosion Limited coverage	Constraints solving related issues

For future work, the implementation of the tool should be completed such that a sizable example can be modeled, its properties verified, and local and global test cases generated.

In a more theoretical direction, avionics systems face several challenges in their modeling and test case generation. They have several aspects; in addition to the control and dataflow that are modeled as an EFSM, the time aspect is critical and requires more complex modeling and test generation techniques. Avionics systems have a wide range of inputs such as input data from various actuators that an EFSM model alone cannot express. The inputs may be discrete or continuous, the latter is still a challenge for modelling and testing. They also have a high volume of outputs that require better analysis techniques. Both data input selection and output analysis constitute real challenges and more research and innovation are needed to address them properly. Researchers are looking at creating new hybrid models that can express the diversity of data inputs. Test case generation based on these new models needs to be developed and implemented. The oracle problem (for trace analysis) requires the use of recent advances in data mining and artificial intelligence for analyzing, correlating outputs, and searching in artifacts, such as requirement specifications, logs, and test architectures. Developing more efficient test generation algorithms will greatly help to advance work in this field.

Bibliography

[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-33, 2011.

[2] Boehm, B. W. (1979):Software engineering. Proceedings of the 4th International Conference on Software Engineering, Munich, Germany, September 1979, pages 11–21, 1979.

[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] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, B. Monate: Testing or Formal Verification DO-178C Alternatives and Industrial Experience. Journal of IEEE Software, Volume 30, Issue 3, pp. 50-57, 2013

[6] D. Lee, M. Yannakakis: Principles and methods of testing finite state machines, a survey. Proc. IEEE 84 (8) 1090–1123, 1996.

[7] 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.

[8] 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, pp. 95-155, 2017.

[9] 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) 633–667. (© World Scientific), 2015.

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

[11] T. Nahhal, Th. Dang: Test Coverage for Continuous and Hybrid Systems. CAV 2007: 449-462

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

[13] J. J. Chilenski: An investigation of three forms of the modified condition decision coverage (MC/DC) criterion. DTIC Document, Tech, no. April. 2001.

[14] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, L. K. Rierson: A Practical Tutorial on Modified Condition / Decision Coverage. no. May. 2001. Microsoft Word - cover 210876.doc (nasa.gov), May 1st 2023.

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

[16] J. R. Chang, 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.

[17] S. A. Vilkomir, J. P. Bowen: From MC/DC to RC/DC: Formalization and Analysis of Control-Flow Testing Criteria, Formal Methods and Testing, 10.1007/978-3-540-78917-8_8, (240-270), 2008.

[18] R. Bloem, K. Greimel, R. Koenighofer, and F. Roeck: Model-Based MC/DC 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] J. Wang, B. Zhang, Y. Chen: Test case set generation method on MC/DC based on binary tree. vol. 8783, no. Icmv 2012, p. 87831P, Mar. 2013.

[20] Z. Awedikian, K. Ayari, 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, M. Staats: Observable modified condition/decision coverage. 2013 35th International Conference on Software Engineering (ICSE), 102-111, 2013.

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

[23] 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.

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

[25] T. Kitamura, Q. Maissonneuve, E.-H. Choi, C. Artho, A. Gargantini: Optimal Test Suite Generation for Modified Condition Decision Coverage Using SAT Solving, Developments in Language Theory. 10.1007/978-3-319-99130-6_9, (123-138), 2018.

[26] R. N. Kacker, D. R. Kuhn, E. Wong: MC/DC-Star: An Open-source MC/DC Measurement Tool. Proceedings of 11-th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, -1, [online], https://doi.org/10.1109/DSA.2018.00027, 2018, (Accessed May 1, 2023).

[27] 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.

[28] H. Sartaj, M.Z. Iqbal, A.A.A. Jilani, M.U. Khan: 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, 2019.

[29] J. Jaffar, S. Godboley, R. Maghareh: Optimal MC/DC test case generation. ICSE '19: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings May 2019 Pages 288–289https://doi.org/10.1109/ICSE-Companion.2019.00118, 2019.

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

[31] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico: 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, pp. 75–90, 1997.

[32] C. Bourhfir, E. Aboulhamid, R. Dssouli, N. Rico : A test case generation approach for conformance testing of SDL systems:Computer Communications, vol.24, no.3-4, pp.319–333, 2001.

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

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

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

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

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

[38] R. E. Miller, S. Paul: Generating Conformance Test Sequences for Combined Control and Data Flow of Communication Protocols. Proc. of International Symposium on Protocol Specification, Testing and Verification, XII, pp. 13-27, 1992.

[39] K. El-Fakih, A. Simao, N. Jadoon, J. C. Maldonado: An assessment of extended finite state machine test selection criteria. Journal of Systems and Software, vol. 123, pp. 106–118, 2017.

[40] T. Hoare: Communicating Sequential Processes. Prentice-Hall, 1985.

[41] R. Milner: Communication and Concurrency. Prentice Hall, 1989.

[42] G. L. Luo, G. V. Bochmann, A. Petrenko: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-method. IEEE Transactions on Software Engineering, 20(2), 149–161, 1994.

[43] R. M. Hierons: Testing from semi-independent communicating finite state machines with a slow environment. Software Engineering - IEE Proceedings 144 (5-6), 291-295, 1997.

[44] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico: A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. IWTCS, 1998, pp. 275–290. P. Coward, Symbolic execution and testing, Inf. Softw. Technol. 33 (1) 53–64, 1991.

[45] J. Li , 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, 181–185, 2002.

[46] W. E. Wong, 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, 803-822, 2008.

[47] 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), 1-9, 2013.

[48] D. Lee, K. Sabnani, D. Kristol, S. Paul: Conformance Testing of Protocols Specified as Communicating Finite State Machines - A Guided Random Walk Based Approach. IEEE Transactions on Communications, 44 (5) 631 – 640, 1996.

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

[50] A.Y. Duale, M.Ü. Uyar : Generation of Feasible Test Sequences for EFSM Models. Ural, H., Probert, R.L., v. Bochmann, G. (eds) Testing of Communicating Systems. IFIP Advances in Information and Communication Technology, vol 48. Springer, Boston, MA, 2000. https://doi.org/10.1007/978-0-387-35516-0_6

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

[52] B. Beizer: Software testing techniques. (2. ed.). Van Nostrand Reinhold, ISBN 978-0-442-20672-7, 1990.

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

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

[55] J. Zhang, C. Xu, X. Wang: 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, pp. 242–250, 2004.

[56] A. S. Kalaji, R. M. Hierons, S. Swift: An integrated search-based approach for automatic testing from extended finite state machine (efsm) models. Information and Software Technology, vol. 53, no. 12, pp. 1297– 1318, 20

[57] M. Rajagopal, R. Sivasakthivel, K. Loganathan, L. E. Sarris: An Automated Path-Focused Test Case Generation with Dynamic Parameterization Using Adaptive Genetic Algorithm (AGA) for Structural Program Testing. Information, 10.3390/info14030166, 14, 3, (166), 2023.

[58] R. Sheikh, M. Imran Babar, R. Butt, A. Abdelmaboud, T. Abdalla Elfadil Eisa: An Optimized Test Case Minimization Technique Using Genetic Algorithm for Regression Testing. Computers, Materials & Continua, 10.32604/cmc.2023.028625, 74, 3, (6789-6806), 2023.

[59] A. Arrieta, P. Valle, J. A. Agirre, G. Sagardui: Some Seeds Are Strong: Seeding Strategies for Search-based Test Case Selection. ACM Transactions on Software Engineering and Methodology, 10.1145/3532182, 32, 1, (1-47), 2023.

[60] N. Neelofar, K.Smith-Miles, M. Andrés Muñoz, A. Aleti: Instance Space Analysis of Search-Based Software Testing. IEEE Transactions on Software Engineering, 10.1109/TSE.2022.3228334, 49, 4, (2642-2660), 2023.

[61] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, B. Monate : Testing or formal verification: DO-178C alternatives and industrial experience. IEEE Software, 30(3):50–57, 2013.

[62] E. M. Clarke, O. Grumberg, D. Peleg, (1999): Model Checking. The MIT Press. Crispin, L. and Gregory, J. (2008). Agile Testing, A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional, first edition.

[63] A. Lomuscio, H. Qu, F. Raimondi : MCMAS: A model checker for the verification of multiagent systems. Bouajjani, A. and Maler, O., editors, CAV, volume 5643 of LNCS, pages 682—688. Springer, 2009.

[64] P. Arcaini, A. Gargantini, E. Riccobene : AsmetaSMV: A way to link high-level ASM models to low-level nusmv specifications. Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, pages 61–74, 2010.

[65] F. Boniol, V. Wiels: The landing gear system case study. 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

[66] P. Arcaini, A. Gargantini, and E. Riccobene: Modeling and analyzing using ASMs: The landing gear system case study. In Boniol, F., Wiels, V., Ameur, Y. A., andSchewe,K.-D.,editors,ABZ:TheLandingGearCaseStudy,volume433,pages 36–51, 2014.

[67] P. Dhaussy, C. Teodorov: Context-aware verification of a landing gear system. Boniol, F., Wiels, V., Ameur, Y. A., and Schewe, K.-D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 52–65, 2014.

[68] B. Berthomieu, S. D. Zilio,, L. Fronc: Model-checking real-time properties of an aircraft landing gear system using Fiacre. In Boniol, F., Wiels, V., Ameur, Y.A., and Schewe, K.-D., editors, ABZ: The LandingGearCaseStudy, volume 433, pages 110–125, 2014.

[69] D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, M. Leuschel : Validation of the ABZ landing gear system using ProB. In Boniol, F., Wiels, V., Ameur, Y. A., and Schewe, K.-D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 66–79, 2014.

[70] P. Arcaini, A. Gargantini, and E. Riccobene : Offline model-based testing and runtime monitoring of the sensor voting module. In Boniol, F., Wiels, V., Ameur, Y.A.,andSchewe,K.-D.,editors,ABZ:TheLandingGearCaseStudy,volume433, pages 95–109, 2014b.

[71] M. Elqortobi, W. El-Khouly, A. Rahj, J. Bentahar, R. Dssouli: Verification and testing of safety-critical airborne systems: A model-based methodology. Comput. Sci. Inf. Syst. 17(1): 271-292, 2020.

[72] W. El-Kholy, J. Bentahar, M. El-Menshawy, H. Qu, R. Dssouli: Conditional commitments: Reasoning and model checking. ACM Trans. on Soft. Eng. and Metho. 24(2), 9:1–9:49, 2014

[73] E. Clarke, O. Grumberg, D. Peled: Model Checking. The MIT Press, Massachusetts, 1999

[74] W. El-Kholy, M. El-Menshawy, J. Bentahar, H. Qu, R. Dssouli: Formal specification and automatic verification of conditional commitments. IEEE Intelligent Systems 30(2), 36–44, 2015

[75] A. Lomuscio, H. Qu, F. Raimondi: MCMAS: A model checker for the verification of multiagent systems. J. International Journal on Software Tools for Technology Transfer, Vol.19, N°1, 2017.

[76] C. Ackermann: MC/DC in a nutshell, Fraunhofer CESE, Maryland USA, 2006.

[77] A. Prestschner: Compositional generation of MC/DC integration test suites, Elsevier Science B.V, 2003

[78] A. Lomuscio, H.Qu, F. Raimondi : MCMAS: An open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer, 19:9–30, 2015.

[79] W. El Kholy, J. Bentahar, M. El-Menshawy, H. Qu, R. Dssouli: Conditionalcommitments: Reasoningandmodelchecking. ACMTrans.onSoft.Eng.and Metho., 24:9:1–9:49, 2014.

[80] W. El Kholy, M. El-Menshawy, J. Bentahar, H. Qu, R. Dssouli: Formal specification and automatic verification of conditional commitments. IEEE Intelligent Systems, 30:36–44, 2015.

[81] R. Fagin, J. Y. Halpern, Y. Moses, M. Vardi: Reasoning about Knowledge. The MIT Press, 1995.

[82] H. Huan, W.-T. Tsai, R. Paul, Y. Chen: Automated model checking and testing for composite web services. In Proceedings of the 8th IEEE International Symposium on Object oriented Real-time Distributed Computing, pages 300–307. IEEE Computer Society, 2005.

[83] G. Rozenberg: Handbook of Graph Grammars and Computing by Graph Transformations, Volumes 1-3, World Scientific Publishing, ISBN 9810228848, 1997.

[84] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini : 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, 1997.

[85] M. Elqortobi, W. El-Khouly, J. Bentahar, R. Dssouli : Model-Based Verification and Testing Methodology for Safety-Critical Airborne Systems. In: Abdelwahed E. et al. (eds) New Trends in Model and Data Engineering. MEDI 2018. Communications in Computer and Information Science, vol 929. Springer, pp:.63-74, 2018.

[86] 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, Proceedings of ACM 2020, ISBN 978-1-4503-7733-1, 2020.

[87] 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.

[88] X. Yin, Y. Jiangyuan, Z. Wang, X. Shi, J. Wu: Modeling and Testing of Network Protocols with Parallel State Machines, IEICE Transactions on Information and Systems E98. D(12) :2091-2104, 2015.

[89] C. Besse, A. Cavalli, M. Kim, F. Zadi: Automated generation of interoperability tests. Testing of Communicating Systems XIV, 169-184, 2002.

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

[91] W. El-Kholy, M. El Menshawy, J. Bentahar, M. Elqortobi, A. Laarej, R. Dssouli : Model Checking Intelligent Avionics Systems for Test Cases Generation using Multi-Agent Systems in Expert Syst. Appl 156: 113458, 2020.

[92] A. Engels, L. Feijs, S. Mauw :Test generation for intelligent networks using model checking. In Brinksma, E., editor, Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, pages 384–398. Springer-Verlag, 1997.

[93] A. S. Bataineh, J. Bentahar, M. El-Menshawy, R. Dssouli :Specifying and verifying contract-driven service compositions using commitments and model checking. Expert Syst. Appl., 74:151–184. Beck, K. (2003). Test Driven Development: By Example. Addison-Wesley Professional, 2017.

[94] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, H. Ural: Data flow testing as model checking. In Proceedings of the 25 thInternational Conference on Software Engineering, pages 232–242. IEEE Computer Society, 2003.

[95] N. Drawel, J. Bentahar, H. Qu: Computationally grounded quantitative trust with time. In Proceedings of the19th International Conferenceon Autonomous Agents and MultiAgent Systems, AAMAS, Auckland, New Zealand, May 9–13, pages 1837–1839 (in press), 2020a.

[96] N. Drawel, H. Qu, J. Bentahar, E. Shakshuki: Specification and automatic verification of trust-based multi-agent systems. Future Gener. Comput. Syst., 107:1047, 2020b.

[97] J. Callahan, F. Schneider, S. Easterbrook : Automated software testing using modelchecking. In In Proceedings of SPIN Workshop. Also WVU Technical Report NASAIVV- 96-022, 1996.

[98] O.C.Z. Gotel, A.C.W. Finkelstein: An analysis of the requirements traceability problem. Proceedings of ICRE94, 1st International Conference on Requirements Engineering. 1994; Colorado Springs, Co; IEEE CS Press, 1994.

[99] F.A.C. Pinheiro and J.A. Goguen: An object-oriented tool for tracing requirements. IEEE Software; 13(2):52-64, 1996.

[100] O. Gotel et al.: Traceability Fundamentals, Software and Systems Traceability, 2012.

[101] R. Matthias, and M. Hubner: Traceability-driven model refinement for test case generation, Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the. IEEE, 2005.

[102] H. Schwarz, J. Ebert, A. Winter:Graph-based traceability: a comprehensive approach - Software & Systems Modeling, Springer, 2010.

[103] R.F. Paige: Traceability in model-driven safety critical software engineering, 6th ECMFA Traceability Workshop, 2010.

[104] N. Mustafa, Y. Labiche and D. Towey: Traceability in Systems Engineering: An Avionics Case Study, 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 2018, pp. 818-823, doi: 10.1109/COMPSAC.2018.10345, 2018.

[105] N. Mustafa, Y. Labiche: The Need for Traceability in Heterogeneous Systems: A systematic literature review. IEEE COMPSAC (1) 2017: 305-310, 2017.

[106] B. Ramesh, M. Jarke: Toward Reference Models for Requirements Traceability. IEEE Trans. Softw. Eng.. 27(1): p. 58-93, 2011.

[107] J. Hassine, J. Rilling, J. Hewitt: Change Impact Analysis for Requirement Evolution using Use Case Maps. IWPSE 2005: 81-90

[108] IEEE Computer Society. ANSI/IEEE Standard 830-1984, 1984.

[109] S. S. Al-Qahtani, E. E. Eghan, J. Rilling: Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach. ICST 2017: 80-91

[110] S. S. Al-Qahtani, J. Rilling: An Ontology-Based Approach to Automate Tagging of Software Artifacts. ESEM 2017: 169-174

[111] M. Unterkalmsteiner, R. Feldt, T. Gorschek : A taxonomy for requirements engineering and software test alignment, ACM Transactions on Software Engineering and Methodology (TOSEM) 23.2 : 16, 2014.

[112] N. Kesserwan, R. Dssouli, J. Bentahar: From use case maps to executable test procedures: a scenario-based approach. Softw. Syst. Model 18, 1543–1570 (2019). https://doi.org/10.1007/s10270-017-0620-y, 2019.

[113] H. Tufail, M. F. Masood, B. Zeb, M. W. Anwar: A systematic review of requirement traceability techniques and tools, 2nd International Conference on System Reliability and Safety (ICSRS), 2017.

[114] N. Kesserwan1, J. Al-Jaroodi: Model-driven Framework for Requirement Traceability, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 12, No. 2, 2021.

[115] W. Krzysztof, L. Ahlberg, J. Persson: On the delicate balance between RE and Testing: Experiences from a large company. Requirements Engineering and Testing (RET), 2014 IEEE 1st International Workshop on. IEEE, 2014.

[116] M. Elqortobi, A. Rahj, J. Bentahar: Granular Traceability between Requirements and Test Cases for Safety Critical Software Systems, Submitted to MobiWis'2023, to be held in Marrakech, Morocco, 14-16, 2023.

[117] A. Rahj: EFSM-based Test Suite Generation for MC/DC Compliant Systems: Tool Design, Master of Applied Science Thesis, Concordia University,2023.

[118] A.S. Kalaji, R.M. Hierons, S.Swift : 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, 2009.

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

[120] T.Yano, E. Martins, F. L. de Sousa : 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, 164-173, 2011.

[121] E. J. Weyuker: The Oracle Assumption of Program Testing, in Proceedings of the 13th International Conference on System Sciences (ICSS), Honolulu, HI, pp. 44-49, January 1980.

[122] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo *:* The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering. 41 (5). 507–525. *doi*:*10.1109/TSE.2014.2372785*, November 2014.

Appendices

Appendix A: Algorithm for test case generation

The following are algorithms extracted from Bourhfir et al.: A test case generation approach for conformance testing of SDL systems. Computer Communications, vol.24, no.3-4, pp.319–333, 2001 [32].

“The EFTG algorithm

```
 Algorithm EFTG (Extended Fsm Test Generation)
   Begin
          Read an EFSM specification
          Generate the dataflow graph G form the EFSM specification
          Choose a value for each input parameter influencing the control flow
          Executable-Du-Path-Generation (G)
          Remove the paths that are included in others
          Add state identification to each executable Du-path
          Add a postamble to each Du-path to form a complete path
          For each complete path
                 Re-check its executability
                 If the path is not executable
                        Try to make it executable
                 EndIf
                 If the path is still not executable Discard it
                 EndIf
           EndFor
          For each uncovered transition T
                 Add a path which covers it (for control flow testing)
          EndFor
          For each executable path
                 Generate its input/output sequence using symbolic evaluation
          EndFor
End;
```

```
Procedure Executable-Du-Path-Generation(flowgraph G)
 Begin
          Generate the shortest executable preamble for each transition
          For each transition T in G
                 For each variable v which has an A-Use in T
                        For each transition U which has a P-Use or a C-Use of v
                               Find-All-Paths(T,U)
                        EndFor
                 EndFor
        EndFor
End.
```

```
Procedure Find-all paths (T1, T2, var)
Begin
          If a preamble, a postamble or a cycle is to be generated
             Preamble:=T1
          Else
             Preamble:= the shortest executable preamble from the first transition to T1
```

```
            EndIf
            Generate-All-Paths(T1,T2,first-transition, var, preamble)
End;
```

The following algorithm is used to find all executable preambles and all executable Du-paths between transition T1 and transition T2 with respect to the variable var defined in T1.

Procedure Generate-All-Paths(T1, T2, T, var, Preamble)

```
Begin
      If (T is an immediate successor of T1) (e.g. t3 is an immediate successor of t2)
            If (T=T2 or (T follows T1 and T2 follows T in G)) (e.g., t4 follows t2)
                If we are building a new path
                          Previous:= the last generated Du-path (without its preamble)
                      If (T1 is present in the previous path)
                         Common:= the sequence of transitions in the previous path before T1
                      EndIf
               EndIf
               If we are building a new path
                         Add Preamble to Path, Add var in the list of test purposes for Path
             EndIf
            If Common is not empty
                 Add Common to Path
            EndIf
            If (T = T2)
                  Add T to Path, Make-Executable(Path)
            Else
                 If T is not present in Path (but may be present in Preamble) and T does not have an A-use of var
                    Add T to Path Generate-All-Paths (T, T2, first-transition, var, Preamble)
                 EndIf
        EndIf
      EndIf
 EndIf
T:= next transition in the graph
 If (T is not Null)
          Generate-All-Paths(T1, T2, T, var, Preamble)
 Else
        If (Path is not empty)
              If (the last transition in Path is not an immediate precedent of T2)
                        Take off the last transition in Path
              Else
                        If (Path is or will be identical to another path after adding T2)
                                 Discard Path
                       EndIf
             EndIf
        EndIf
   EndIf
End.
```

Procedure Handle_Executability(path P)

```
Begin
      Cycle:= not null
            Process(P)
      If P is still not executable Remove it
      EndIf
```

```
End;
```

Procedure Process(path P)

```
Begin
  T:= first transition in path P
  While (T is not null)
      If (T is not executable)
          Cycle:= Extract-Cycle(P,T)
   EndIf
        If (Cycle is not empty)
        Trial:=0
        While T is not executable and Trial<Max_trial Do
                     Let Precedent be the transition before T in the path P
                    Insert Cycle in the path P after Precedent
                    Interpret and evaluate the path P starting at the first transition of Cycle to see if the
                    predicates are satisfied or not
                    Trial:= Trial+1
        EndWhile
     Else Exit
    EndIf
       T:= next transition in P
   EndWhile
     End.”
```