Computer Science and Information Systems 17(1):271–292 https://doi.org/10.2298/CSIS190430040E

Verification and Testing of Safety-Critical Airborne Systems: a Model-based Methodology

Mounia Elqortobi[1], Warda El-Khouly[1], Amine Rahj[1], Jamal Bentahar[1] and Rachida Dssouli[1]

[1] Concordia University, Quebec, Canada
m_elqort@ mail.concordia.ca,
warda_elkholy@yahoo.com
{amine.rahj, jamal.bentahar, rachida.dssouli}@ concordia.ca

Abstract. In this paper, we address the issues of safety-critical software verification and testing that are key requirements for achieving DO-178C and DO-331 regulatory compliance for airborne systems. Formal verification and testing are considered two different activities within airborne standards and they belong to two different levels in the avionics software development cycle. The objective is to integrate model-based verification and model-based testing within a single framework and to capture the benefits of their cross-fertilization. This is achieved by proposing a new methodology for 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 and checked via testing. The contributions of this paper are a methodology that integrates verification and testing, formal verification of some safety critical software properties, and a testing method for Modified Condition/Decision Coverage (MC/DC). The results of formal verification and testing can be used as evidence for avionics software certification.

Keywords: Model-based Verification, Model Checking, Communication Graph, Methodology, Model-based Testing, Partial Reachability Graph, MC/DC (Modified Condition/Decision Coverage).

Sponsored by NSERC/CRD CMC CS Canada. Project CRIAQ AVIO 604, CRDPJ 463076-14.

1. Introduction

Developing safety-critical software requires rigorous processes. To prevent catastrophic events, the avionics industry has introduced a rigorous certification process, described in the RTCA [1, 2] standard. The DO-178C standard [1] includes a supplement on formal methods called DO-333. In the DO-333 standard, a formal method is defined as “a formal model combined with a formal analysis”. The DO-178C and its supplement have been successfully applied into the production of software systems at Dassault-Aviation and Airbus [3]. The motivation of this work is to increase software dependability by integrating formal verification techniques with testing and to capture the benefits of their cross-fertilization. In addition, formal verification and test results can be used as evidence for certification. Although model-based testing [5, 6] and verification activities [3, 4, 5] are natural approaches to the certification of avionics software, the integrated model-based engineering approach is not yet well studied in the literature, and several challenges still need to be addressed [4, 7, 12, 14].

We propose a model-driven approach that encompasses two main levels: verification/design and validation/implementation. As shown in figure 1, in the first level, we adopt model checking, a formal and fully automatic technique for model-based verification. It is a natural choice for a rigorous verification of avionics systems against desirable properties, including safety and liveness. In the second level, we transform the finite state machine (FSM) verification model [9, 10, 11] into an Extended Finite State Machine (EFSM) testing model that is an FSM-like model extended with variables [16]. We generate both local test cases for each EFSM component modeled as agent in its context of communication, and global test cases for a Communicating EFSM (CEFSM) model. The CEFSM is a composition of EFSMs. The test generation method satisfies the Modified Condition/Decision Coverage (MC/DC) criterion, all Definition-Use (DU)-paths, and ensure that the verified properties hold in the implementation. The selection of coverage criteria is based on the satisfaction of DO 178C for MC/DC and on the use of middle ground structural coverage for all DU-paths. Using a better structural coverage criterion, such as all-paths, is often impractical.

Model–based verification and model-based testing are still very active research domains [5, 6, 14, 16, 23, 24, 25]. They are considered as two distinct research areas and supported by different research communities. EFSM-based testing and Communicating EFSM have been extensively studied [18, 19, 20, 21, 23, 25]. A more recent research area is test generation based on model checking, with several publications [17, 22, 26, 27, 28, 29] discussing that topic. The principle is basically to generate counterexamples or witness traces that can be used to derive test cases. The major problems in all the published work are related to performance, the notion of test coverage or test efficiency, non-determinism, and the abstraction level of test cases, derived from counterexamples and witness traces, that need more refinement to be accurate and be utilized to test implementations [17, 29]. To the authors' best knowledge, there is no work on the methodologies that link testing and verification in the same framework.

The rest of the paper is organized as follows. In Section 2, we present an overview of the proposed approach and our case study about a landing gear system [11]. This is followed by a summary of our model-based verification system, formal modeling of our case study, and our experimental results verifying the correctness of the modeled landing gear system against desirable properties including safety and liveness in Section 3. We then present our model-based testing and show how to automatically generate test cases in Section 4. In Section 5, we cover the details of MC/DC criterion and how to integrate non executable paths. We offer our discussion, conclusions, and identify future work in Section 6.

2. The Proposed Methodology and Case Study

2.1. The Proposed Methodology

We introduce the proposed verification and testing framework in this sub-section. The methodology begins with formally modeling the safety-critical airborne system from the given informal requirement specifications, producing an FSM-like model as described in Figure 1. We assume that a correct informal specification exists. Next, it proceeds to refine and encode the obtained model using ISPL+ (an extended version of the input language of the symbolic model checker MCMAS+ introduced in [9]) to verify agent-based intelligent systems.

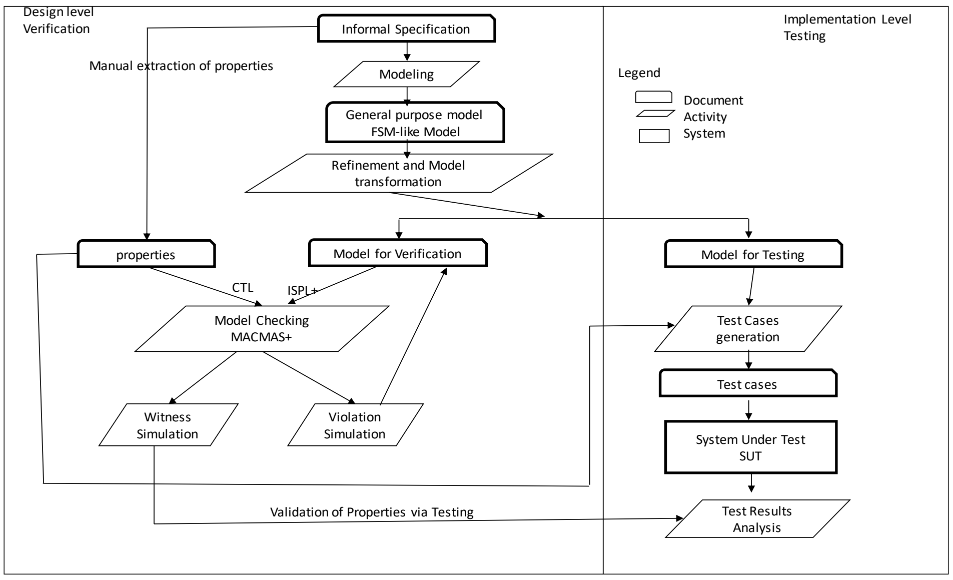


Fig 1. Overview of our Approach

Parallel with this step, our approach extracts and expresses the system requirements in the form of temporal properties using Computation Tree Logic (CTL) [8]. MCMAS+ automatically checks whether the model satisfies the intended properties and graphically produces witness-examples or counter-examples [12, 13]. The produced witness-examples prove the satisfaction of properties while the produced counter-examples 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 reduced Communicating Extended Finite State Machine (CEFSM) that uses our developed algorithms and tools to automatically generate abstract test cases. These algorithms and tools address the conformity of the implementation under test to Low-Level Requirements (LLR), instead of to high-level requirements as in existing automated test generation techniques; thereby allowing them to be more applicable and efficient for the satisfaction of avionics standards. 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 are then 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 DO-178C [1, 30, 31]. Finally, our approach analyzes the obtained test results and compares them with the produced witness-examples to validate our properties via testing.

2.2. Case study: Landing Gear System

Our case study, a landing gear system for an aircraft, was proposed by Frédéric Boniol and Virginie Wiels in [11] 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 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 open or closed, while the gear can be retracted, extended, or maneuvered. The landing system can be 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. 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 2): 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:

- One green light, indicates that “gears are locked down”;
- One orange light, indicates that “gears are maneuvering”; and
- One red light, indicates a “landing gear system failure”.

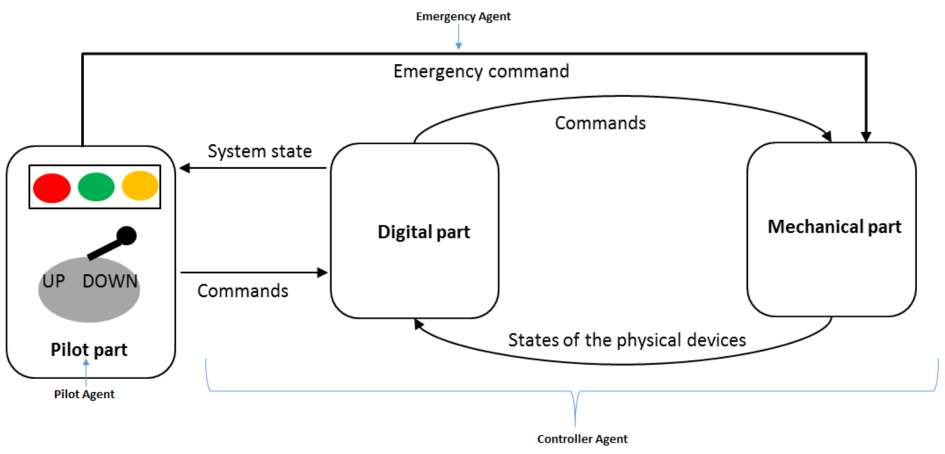


Fig 2. General Architecture of the Landing Gear System

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 is in charge of sending 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 captures the pilot orders.

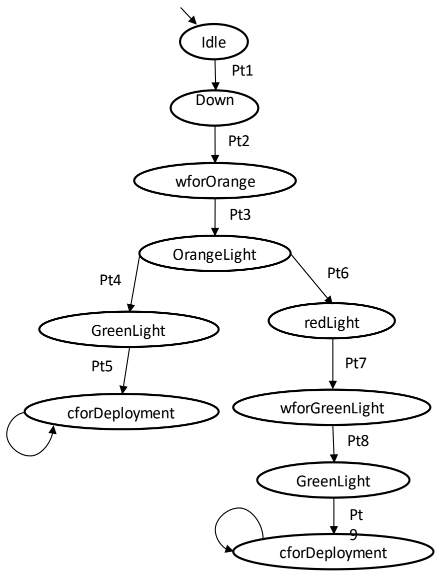
3. Model-based Verification

3.1. Modeling the Landing Gear System

In this section, we show how our model M can formally model the landing gear system. In our modeling, we specifically 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 achieve this aim, we introduce three agent machine models: M_p for pilot, M_c for control unit, and M_e for emergency. The pilot agent machine model M_p models the behavior of the pilot part and the control unit agent machine model M_c models the behavior of the digital part. 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 doors and gears to directly represent the status of the employed cylinders. This is 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.

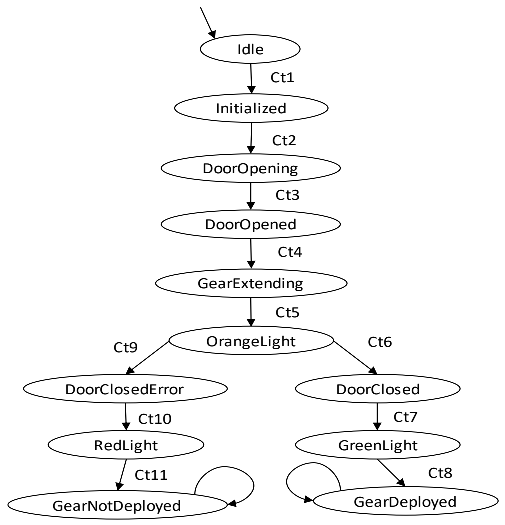
In the published case study paper, there are two types of requirements and the authors classify them as strong and weak. The weak requirements that did not consider deadline constraints/time constraints. Although we selected the weak requirements, the time constraints are abstractly represented in our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

Figures 3, 4, and 5 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 the process of receiving and sending an action. The output of a transition can be directly assigned by the 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.



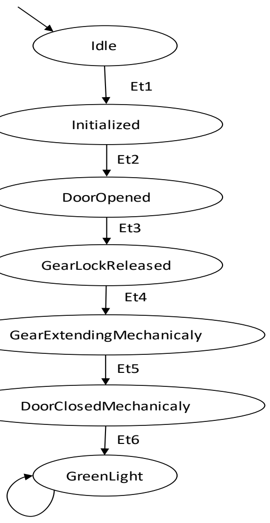
Label	Input, output & predicate
Pt1	? LandingSpecs(speed, distance) ! PressDownButton
Pt2	? PressDownButtonAck ! wforOangeLight
Pt3	? OrangeLightOn ! OrangeLightOnAck
Pt4	? GreenLightOn ! GreenLightOnAck
Pt5	? ConfirmGearDeployment ! DeploymentStatusSccess
Pt5	? RedLightOn ! RedLightOnAck
Pt7	? ConfirmGearDeploymentError ! InitializeEmergencySystem
Pt8	? GreenLightOnM_e ! GreenLightOnM_eAck
Pt9	? ConfirmGearDeploymentM_e ! DeploymentStatusSuccess

Fig 3. Pilot Agent Machine model, M_p



Label	Input, Output & Predicate
Ct1	? PressDownButton ! PressDownButtonAck
Ct2	? ProcessReceivedCommand ! OpenGearDoors
Ct3	? OpenGearDoorsAck ! OutgoingGears
Ct4	? OutgoingGearsAck ! OrangeLightOn
Ct5	? OrangeLightOnAck ! CloseGearDoors
Ct6	? CloseGearDoorsAck ! DoorsCloseSuccess
Ct7	? GearsExtended ! GreenLightOn
Ct8	? GreenLightOnAck ! ControlUnitDisconnected
Ct9	? CloseGearDoorsError ! DoorsCloseError
Ct10	? GearsNotExtended ! RedLightOn
Ct11	? RedLightOnAck ! ControlUnitDisconnedted

Fig 4. Controller Agent Machine model, M_c



Label	Input, Output & Predicate
Et1	? InitializeEmergencySystem ! OpenGearDoors
Et2	? OpenGearDoorsAck ! OutgoingGears
Et3	? OutgoingGearsAck ! VerifyGearsPosition
Et4	? VerifyGearsPositionAck ! LockDoorsMechanicaly
Et5	? LockDoorsMecanicalyAck ! GreenLightOn
Et6	? GreenLightOnAck ! GearStatusExtended

Fig 5. Emergency Agent Machine model, M_e

3.2. Validating

To perform the verification, we introduce the MCMAS tool. This is a symbolic model checker that extends MCMAS, a model checker for Multi-Agent Systems (MAS) that uses Ordered Binary Decision Diagrams (OBDD) [12, 13]. 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 [9, 10]. The inputs of MACMAS are formatted by the ISPL language which is used to describe the communicating MAS to be checked and encode the desired specifications. The ISPL+ is a dedicated programming language for interpreted systems that formalize MASs (Fagin & Halpern, 1994). 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) $p \in Ap$ (the 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.
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 [10]. This graphical interface specifically 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 gives 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.

3.3. 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) [8] supported by the MCMAS+ model checker tool [12] to express the following requirements:

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

In [11], 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 formulaeϕ_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.

4. Model-based Test Generation Approach

The goal is to generate, starting from the verification model, a set of test cases for the verified properties, apply them to the implementation under test and to then analyze the test results. The main idea 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 DO 178C by generating local test sequences with the required coverage criteria. In addition, we extend the set of paths to include additional paths to satisfy MC/DC.

4.1. Model Transformation

In model checking, a simple FSM is often used. Testing can use richer modeling techniques such as Extended Finite State Machines (EFSM). To use our test case generation techniques and tools with well-defined coverage criterion such as MC/DC [30, 31], 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 to use one model or two 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 independency between verification and testing activities. The model transformation can show the equivalence between two models in this case.

4.2. Global Test Sequence Generation

Several approaches to generate global test sequences are based on or are otherwise similar to the work of Bourhfir and Cavalli [18, 19, 20, 23, 32, and 33]. 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 the 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 Machines (CEFSMs) (see Figure 8).

4.3. Test Generation Process for the Case Study

In this case study and for the sake of readability, the EFSMs are only a partial representation of a landing gear system.

Following the DO-178C standards, the satisfaction of the MC/DC criterion is mandatory, and it is used as a criterion in this paper for test sequence generation. The MC/DC is a widely used and known coverage criterion in software avionics [30, 31].

The figure 6 describes the test generation process. To generate global test sequences, we first derive the local test sequences for each EFSM. Second, we obtain the communication graph from all EFSMs (see figure 7). 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 will detail the different steps of the test generation for the case study.

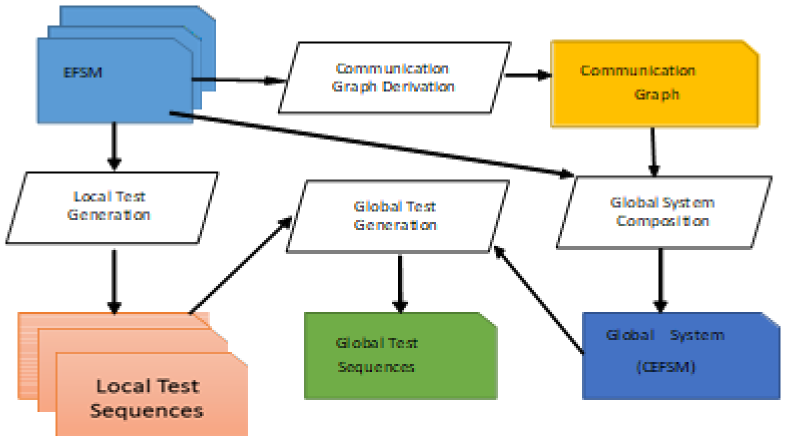


Fig 6. Test Generation Process

Communication Graph. To generate global test sequences for a global system 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 represents the interaction between the different EFSMs (see Figure 7). For our case study, it is assumed that the communication between the machines M_p, M_c and M_e is two-way. Figure 7 visualizes the communication graph with the representation of each machine by an abstract state.

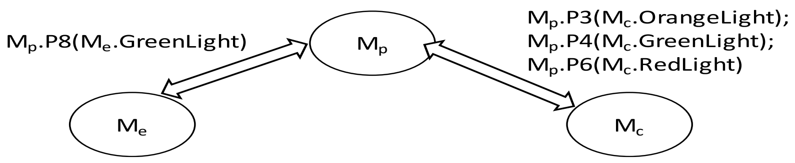


Fig 7. Communication graph representation

Global Model with Communication Points. Using the EFSMs (Figures 3, 4, and 5) and guided by the communication graph (Figure 7), we obtain (by composition) the global system model with its communication points (Figure 8). Figure 8 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 communicating 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. Similarly, a computation graph is also a composition of its constituents.

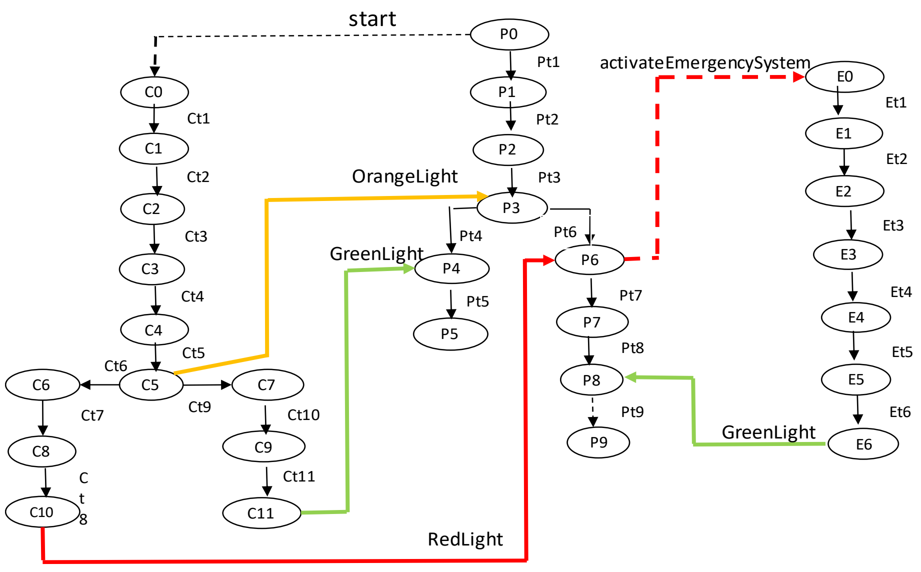


Fig 6. System model M: Composed of M_p, M_c, and M_e with communication points

Algorithm. In this section, we briefly describe the test generation algorithm and its application to the case study. More specifically, we extend our algorithm to generate the test sequences that satisfy the MC/DC criterion. To generate executable test sequences, we need the final model with all the aforementioned information, the local test sequences, as well as the communication graph. The algorithm given in [18 and 31], called the generation of def-use executables, 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 possible transitions existing in the system under test. We generate the paths linking two states from different machines by marking them as communication or synchronization points.

The Generated Test Sequences. To generate the test sequences, we first need to identify the communication variables. In the case of 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 for each transition is defined. The related input and output lists, as well as the predicates, are described in Figures 3, 4, and 5. They are used as inputs for the algorithm to generate the global test sequence. In general, a test case is composed of the following elements: <preamble, target, postamble>. Preamble and

Postamble might be empty. The preamble is the sequence of transitions used to reach the target transition for testing as given in Table 1.

Table 1 shows examples of the application of the algorithm using the landing gear case study. It identifies the different usage lists enabling the identification of executable test sequences. Table 2 shows an example of executable test sequences to reach specific transitions in the system model. The chosen transitions represent a case of parallelism.

Table 1. Example of usage lists and preamble for specific transitions of the landing gear system

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 2. Executable test sequences of the landing gear system

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

Table 3 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 4 shows the parallelism in the executable test sequences required to make the transitions shown in Table 3 executable.

Table 3. Non-executable test sequence of the landing gear system

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

Table 4. Parallelism shown for executable test sequences Pt5 of the landing gear system

	Executable test sequences – Pt5		
Mp	Pt1, Pt2, Pt3		P4, **Pt5**
Mc	Ct1, Ct2, Ct3, Ct4, Ct5	Ct9, Ct10	

In the following sections, we will verify the different properties obtained from the validation phase.

4.4. Witness Properties' Verification

Table 5 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 counter-examples 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 4 for the complete executable test sequence).

Table 5. Executable test sequences for witness-examples for liveness properties

Witness-example 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

4.5. Properties' Verification

Several properties are defined in Table 6 to verify whether the used algorithm validates the properties. The two executable test sequences shown in Table 2 were analyzed with regards to those properties. Both executable test sequences for transitions Pt5 and Pt9 verify all the properties identified so far. Table 6 confirms that all the global test sequences generated render the defined properties true.

According the algorithm used in [18], 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 set of test sequences can validate the different properties, and not necessarily all of them.

Table 6. LGS properties validated with the executable test sequences

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

5. MC/DC

In order 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.

5.1. Handling MC/DC criterion

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 [31]. This means that some test cases' execution will fail.

The following requirements should be satisfied in MC/DC-based testing. For all decisions, at least once: 1) all possible outcomes are covered; 2) all possible outcomes for all conditions are covered; and 3) all conditions impacting the decision's outcomes are covered [30, 31].

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)
```

Another representation would be that the executable paths consist of all green transitions, and the non-executable ones are all red. The additional non-executable paths that will ultimately generate errors are partially red. For example, a path consisting of the values orange on / green off / red off will be part of a feasible path. However, going to the next state is impossible if within a path the values are orange on/green off/red on. Those additional paths exist to satisfy the MC/DC criterion. The objective is to render all the paths by considering the binary possibilities for each condition found in a decision, based on whether the orange light is on or off. However, the two other lights should be taken into consideration for conformity.

There are three conditions to consider within this decision: whether the OrangeLight is on, the GreenLight is off, and the RedLight is off. This translates to the following possibilities shown in Table 7, in which true and false are on and off, respectively:

Table 7. 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 MC/DC criterion that result in an error, as it ensures that a test sequence will fail. As such, we also cover the possible 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 MC/DC criterion [30, 31], 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 in the input and output values for transition Pt3 by adding the different possibilities covered through MC/DC criterion and use that information to generate the required test sequences.

5.2. Test Generation Algorithm Satisfying MC/DC Criterion

The proposed test generation algorithm generates feasible test sequences. To satisfy the MC/DC criterion, the test generation algorithm must be modified and all of the decision branches need to be tested. For each binary decision, two paths will be generated for each simple condition involved in that decision. To integrate this coverage criterion, 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 binary set of possibilities to satisfy the MC/DC criterion. This binary set will represent the possibilities for each information influencing a decision [30, 31].

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 created initially, and generate additional ones to satisfy MC/DC criterion; 2) integrate MC/DC coverage while the paths are being generated to cover all feasible and non-executable paths related to a decision; or 3) analyze the non-executable 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 sketched as follows:

(1) Transform the EFSM to data flow graph G using graph rewriting.
(2) Expand the graph by an expansion mechanism; use state decomposition and graph splitting to handle MC/DC coverage criterion
(3) Select input values for each input parameter that can affect the control flow.
(4) Generate executable DU-paths according to data flow graph G and remove redundant paths. Append the state identification sequence and post amble (return to the initial state) to each DU-path to form a complete test path.
(5) 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.
(6) Verify if there are uncovered transitions, add test paths to cover them.

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

Step 1: Define a second variable of binary values called vMCDC. This variable will take the values that will conform to the coverage's criterion. This variable will be used solely for MC/DC criterion satisfaction for test case generation.

Step 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. Next, call a procedure that will analyze the discarded paths to ensure that they would not be involved in any MC/DC. Step (4) is used to analyze non-executable paths.

Algorithm EFTG (Extended FSM 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, conduct the Analyze-discarded-path(P) procedure.

Step 3: Create 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 binary values 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 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).

Step 4: In the procedure executable-DU-path-generation, we add another loop to take into consideration vMCDC to identify the paths between transitions.

```
Procedure Executable-DU-Path-Generation (flowgraph G)
(1)  Take in 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.
```

Step 5: we replace the procedure handle-executability in order to not discard non-executable paths and call it procedure handle-executability-MCDC. If a path is non executable, it will not be removed. 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. 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.

6. Discussion and conclusion

Business case studies play a fundamental role in the progress and development of formal methods and help prospective users and designers demonstrate the application of different formal methods to model, verify, and test concrete, complex systems. In addition, they help to compare different formal techniques in terms of performance and ease of use. Relevant proposals have been put forward to model and formally analyze the landing gear system, a complex real-life case study published in [11]. Specifically, these proposals have suggested:

- Formal modeling methods including the Event-B methods [34, 35, 36, 37, 42], abstract state machines (ASM) [38, 43], and the Fiacre formal language [39, 40];
- Verification techniques including a proof theory [34, 35, 37, 43] and model checking [35, 39, 41, 40, 43]; and
- A test case generation technique [41], a run-time monitoring approach [40, 41], and a simulation technique [44].

These proposals provide only a partial solution with a unique objective, either modeling, verification, or testing. Moreover, these proposals do not consider all the industrial requirements of the case study. For example, a formal verification using model checking is used mainly to verify properties at the design level; the verified properties may not be propagated to the implementation stage. Therefore, testing of these properties is still needed. Although model-based testing and verification activities, as shown in DO-178C and DO-333, are natural approaches to the certification of avionics software, combining formal verification and testing in a single framework is still in its infancy and needs further investigation.

The proposed methodology and its application show that the integration of software quality assurance activities is needed to achieve software certification in the airborne industry. There are more challenges to overcome to be able to automate all the activities of the methodology. The avionics software has several types of inputs such as data from various actuators, and high volume of outputs. Both data input selection and trace analysis constitute real challenges and need more research and innovation to address them properly. Some hybrid modeling of the diversity of data input is needed. The oracle problem needs more data mining and intelligence for analyzing and for correlating outputs and searches in artifacts, such as requirement specifications, logs and test architectures. More efficient algorithms will also advance work in this field.

Acknowledgement. We would like to thank the reviewers and Dr. M. El Menshawy for their useful comments. This work was funded in part by NSERC/CRD CMC CS Canada and Project CRIAQ AVIO 604, CRDPJ 463076-14.

References

[1] http://www.rtca.org. RTCA/DO-178C (2011) "Software Considerations in Airborne Systems and Equipment Certification", December 13, DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A, DO-333 Formal Methods Supplement to DO-178C and DO-278A

[2] Zoughbi, G., Briand, L., Labiche, Y.: 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

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

[4] John 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] Peleska, J., Siegel, M.: Test Automation of Safety-Critical Reactive Systems. South African Computer Jounal 19, pp. 53-77. (1997):

[6] Jan Peleska (2013): Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. MBT 2013: 3-28

[7] Gotzhein, R., Khendek, F.: Compositional Testing of Communication Systems. LNCS 3964, pp. 227 – 244 (2006) Gotzhein, R., Khendek, F.: Compositional Testing of Communication Systems. LNCS 3964, pp. 227 – 244 (2006)

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

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

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

[11] Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D. (eds.) Proceeding of 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z.

Communications in Computer and Information Science, vol. 433, pp. 1–18. Springer (2014)

[12] Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multiagent systems. CAV. LNCS, vol. 5643, pp. 682–688. Springer (2009)

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

[14] Berrada, I., Castanet R., Felix P.: Testing Communicating Systems: a Model, a Methodology, and a Tool. LNCS 3502, pp. 111–128 (2005)

[15] Kalaji, A.S., Hierons, R.M., and Swift, S.: 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.

[16] Clarke, E., Wing, J.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, Vol. 28, No. 4 (1996)

[17] Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test., Verif. Reliab. 19(3): 215-261 (2009)

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

[19] Bourhfir, C., Aboulhamid, E., Dssouli, R., Rico, N.: Automatic executable test case generation for extended finite state machine protocols. IWTCS (1997)

[20] Bourhfir, C., Aboulhamid, E., Dssouli, R., Rico, N.: A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols. IWTCS (1998)

[21] Aichernig B. K., Delgado, C. D.: From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems. LNCS 3922, pp. 324–338 (2006)

[22] Paul, E. Black.: Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In Proc. of the 19th Digital Avionics Systems Conference, pages 1.B.3–1–1.B.3–6 vol.1, 2000.

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

[24] Utting, M., Pretschner, A., Legeard, B.: A taxonomy on model-based testing. University of Waikato, Hamilton, New Zealand (2006)

[25] Bochman, G. V., Khendek, F.: Test Selection Based on Finite State Models. IEEE Transactions on Software Engineering (1991)

[26] Clarke, E., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In Verification: Theory and Practice, volume 2772 of Lecture Notes in Computer Science, pages 208–224, 2004.

[27] Clarke, E., Grumberg, O., McMillan, K. L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In Proceedings of the 32st Conference on Design Automation (DAC), pages 427–432. ACM Press, 1995

[28] Jéron, T., Morel, M.: Test generation derived from model-checking. In CAV ’99: Proceedings of the 11th International Conference on Computer Aided Verification, pages 108–121, London, UK, 1999. Springer-Verlag. ISBN 3-540-66202-2

[29] Fraser, G., Wotawa, F., Ammann, P.: Issues in using model checkers for test case generation. Journal of Systems and Software 82(9): 1403-1418 (2009)

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

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

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

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

[34] Su, W., Abrial, J-R. Aircraft landing gear system: Approaches with Event-B to the modeling of an industrial system. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D. editors, ABZ: The Landing Gear Case Study, volume 433, pages 19–35, 2014.

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

[36] Mammar, A., Laleau, R.: Modeling a landing gear system in Event-B. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 80–94, 2014.

[37] Méry M., Kumar Singh, N.: Modeling an aircraft landing system in Event-B. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 154–159, 2014.

[38] Kossak, F., Landing gear system: An ASM-based solution for the ABZ case study. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 142–147, 2014.

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

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

[41] Arcaini, P., Gargantini, A., Riccobene, E.: Offline model-based testing and runtime monitoring of the sensor voting module. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 95–109, 2014.

[42] Banach, R.: The landing gear case study in hybrid Event-B. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, ABZ: The Landing Gear Case Study, volume 433, pages 126–141, 2014.

[43] Arcaini, P., Gargantini, A., Riccobene, E.: Modeling and analyzing using ASMs: The landing gear system case study. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 36–51, 2014.

[44] Savicks, V., Butler, B., Colley, J.: Co-simulation environment for Rodin: Landing gear case study. In Boniol, F., Virginie Wiels, Ameur, Y.A., Schewe, K.D., editors, ABZ: The Landing Gear Case Study, volume 433, pages 148–153, 2014.

Mounia Elqortobi is a Ph.D. degree student in Information Systems Engineering, Concordia University. She received her M. Eng. in QSE and bachelor's degree in CSE from Concordia University (2015, 2010).

Warda EI-Kholy received her PhD degree in Information Systems Engineering from Concordia University, she is a lecturer in Menofia University, Egypt.

Amine Rahj is a master's degree student in Quality Systems Engineering, Concordia University. He received his bachelor's degree (2015) from INPT, Morocco.

Jamal Bentahar is a Full Professor with Concordia Institute for Information Systems Engineering, Concordia University, Canada. His research interests include services computing, applied game theory, computational logics, model checking, multiagent systems, and software engineering.

Rachida Dssouli is Full professor and founding Director of Concordia Institute for Information Systems Engineering (CIISE), Concordia University. Her research area is in Testing and Verification.

Received: April 30, 2019; Accepted: September 17, 2019