Hindawi
Modelling and Simulation in Engineering
Volume 2018, Article ID 3847843, 14 pages
https://doi.org/10.1155/2018/3847843

Research Article

Model-Based Testing Applied to Software Components of Satellite Simulators

Paulo Diego Barbosa da Silva,[1] Ana Maria Ambrosio,[2] and Emilia Villani[1]

[1]*Aeronautics Institute of Technology (ITA), São José dos Campos, Brazil*
[2]*National Institute for Space Research, São José dos Campos, Brazil*

Correspondence should be addressed to Emilia Villani; evillani@ita.br

Received 25 August 2018; Accepted 8 November 2018; Published 12 December 2018

Academic Editor: Azah Mohamed

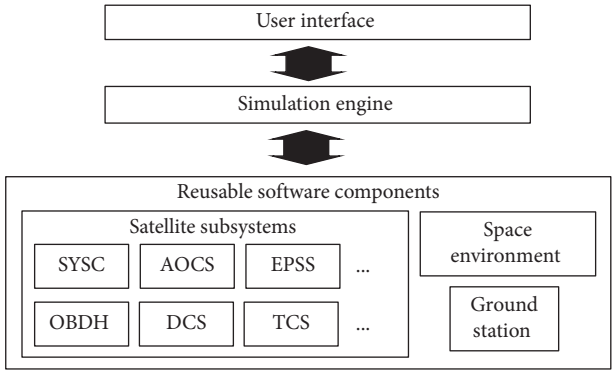
Operational simulators have a fundamental role in space programs. During a satellite operation, these simulators are essential for validating critical manoeuvres, testing new on-board software versions, and supporting the diagnosis of anomalies. With the purpose of reusing the operational simulators, the Brazilian National Institute for Space Research has proposed a new standard for the specification of the components that must be integrated in their in-house developed simulators. The new standard describes the behaviour of satellite subsystems using cause-effect tables that relate telecommands, electrical switches, equipment working states, energy consumption, telemetries, and operating modes of the subsystem. Using this new standard as input, this work proposes an approach that merges model-based testing and model checking to verify the correct implementation of new components in the satellite simulator. The verification approach consists of extracting state machines from the cause-effect tables and used it to automatically derive a test case suite. In order to validate the proposal, we applied it to three different satellite subsystems and assessed the results obtained from the test campaigns. In all the three cases, the proposed approach identified errors in the simulator components that were not initially detected by the traditional testing approach used at the Brazilian National Institute for Space Research.

1. Introduction

The use of simulators permeates all the phases of a satellite lifecycle, from the mission analysis to the satellite disposal. In this work, we approach the problem of verifying the software components that compose satellite operational simulators, which are used after the satellite launch for training new operators, validating operational procedures, testing new versions of on-board software, and supporting the diagnosis of anomalies [1, 2].

Due to the high cost and effort needed to develop an operational simulator, the National Institute for Space Research (INPE) in Brazil has been investing in developing satellite operational simulators that can be reused from one mission to another, with minimum customization effort. For this purpose, the simulator is organized in a modular architecture, where the simulation engine, known as the simulator core, is separated from the software components related to the satellite subsystems, ground station, and space environment. This architecture is illustrated in Figure 1 and is adopted in the simulator of the CBERS (China-Brazil Earth Resources Satellites).

Concomitant to the introduction of the simulator modular architecture, a new specification standard is adopted at INPE for describing the simulator components that are associated with the satellite subsystems. This standard uses cause-effect tables that relate telecommands, electrical switches, equipment working states, energy consumption, telemetries, and operational modes of the satellite subsystem. Generally, cause-effect tables, also known as decision tables or input-output tables, relate input conditions and/or events to output states and/or events. The choice of cause-effect tables as specification language emerged as a joint proposal of the Systems Engineering and



Satellite subsystems:
AOCS - attitude and orbit control subsystem
DCS - data collection subsystem
EPSS - electrical power supply subsystem
OBDH - on-board data handling
SYSC - system circuit subsystem
TCS - thermal control subsystem

Figure 1: Architecture of satellite simulators at INPE.

Software Engineering teams. It has been successfully used as a common language among different specialists that are involved in the development of satellites at INPE, such as mechanical, electrical, and control engineers.

The use of cause-effect tables as specification language has the advantage of being extremely easy to understand. It does not require knowledge of specific modelling languages and can be shared among all the participants of a space project, including in-house specialists, subcontractors, partners, and stakeholders. On the contrary, the adoption of cause-effect tables as specification language induces the testing engineer to limit the test cases to the conditions explicitly described on the tables' lines. Combined sequences of commands are often neglected and errors in the simulator implementation may pass undetected. These errors can lead to the execution of unsafe sequences of commands in the satellite and may result in the satellite damage or even in the complete loss of a mission. This problem has been observed at INPE, where the verification of the satellite simulator components has been performed in a manual and ad hoc way and is strongly dependent of the tester experience.

In order to overcome this limitation, this work proposes and evaluates a model-based testing approach derived from CoFI (conformance and fault injection) testing methodology [3]. The approach receives as input the satellite component specification written according to the INPE standard. A step-by-step method is proposed to translate the cause-effect tables in finite state machines, which are then submitted to formal verification in the UPPAAL model checking tool. Finally, the state machines are used to generate a set of test cases that are then applied to the simulator component.

In order to evaluate the proposed approach, we applied it to three different software components of the SIMCBERS, the simulator of the CBERS (China-Brazil Earth Resources Satellite) satellite family. During the model checking activity, the generation of the test case suite and its application to the simulator are automated with the help of computational tools; we observe that the translation method is performed manually, in order to provide a first experimental evaluation of the approach. Tools to support the complete process automation are currently under development.

The main contribution of our work is on merging existing verification methods and adapting them to the specificities of an application domain (satellite simulators) in order to make its use feasible in the industry. It takes into account the limitations imposed by the specification language (cause-effect tables) adopted in that domain, which was selected by consensus among specialists from different disciplines that are involved in the development of satellites (thermal, electrical, orbital controlling, etc.).

The remaining of the paper is organized as following: Section 2 briefly presents the two verification techniques that are the base of the proposed approach. Then, Section 3 describes the steps of our approach, while Section 4 analyses the results of its application to three components of the SIMCBERS simulator. Section 5 discusses published works and compares them to our proposal and results. Finally, Section 6 draws some conclusions.

2. The CoFI Methodology and Model Checking in UPPAAL

The CoFI model-based testing methodology combines conformity test validation with fault injection. It guides the construction of a set of Mealy state machines [4] representing the component behaviour, where the transitions are associated with a pair (input and output). The state machines are used for the automatic generation of test cases, following a "black box" approach (i.e., only the controllable inputs and the observable outputs are used). The test case generation is automatically performed by the ConDado tool [5]. All test cases start and finish in the initial state. The test case specification contains the sequence of inputs, as well as the expected outputs (the test oracle). Figure 2 presents an example of Mealy state machine and two corresponding test cases generated by the ConDado tool.

The algorithm used by ConDado starts with a reachability tree of the FSM, containing all the FSM transitions. Then, for each branch of the tree, it complements the branch with paths that take the machine to the final state (in the example, the final state is coincidently the initial state). Considering the FSM of Figures 2 and 3 shown in black, the initial reachability tree is obtained from the FSM. The transitions and nodes in grey illustrate the path added to return to initial state S1. As a result, ConData generated 4 test cases.

The CoFI [3] methodology guides the tester to depict the system behaviour into different classes: normal behaviour, specified exceptions, sneak paths, and fault tolerance. Each class is modelled as a different set of state machines, maintaining the model size at an understandable level. In this work, we are particularly concerned with the sneak paths. It describes what happens in the system when known inputs are received at unexpected moments. Previous works [6, 7] have shown that the test cases derived from the sneak paths models are the ones that most contribute to

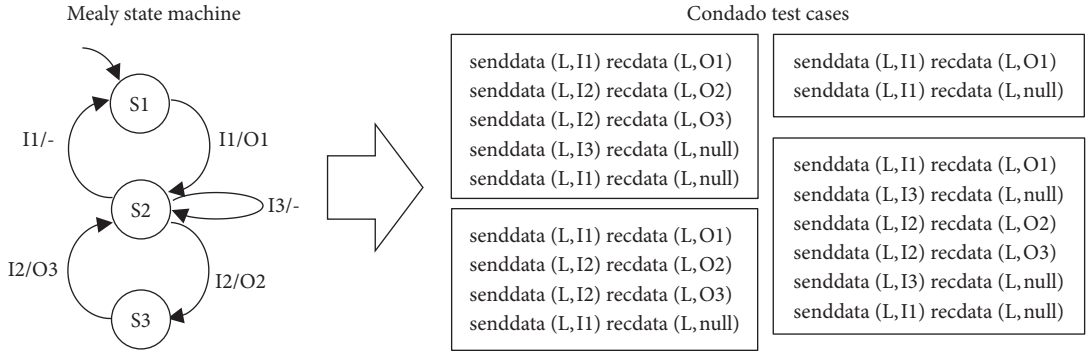


Figure 2: Example of the Mealy state machine and test cases from the ConDado tool.

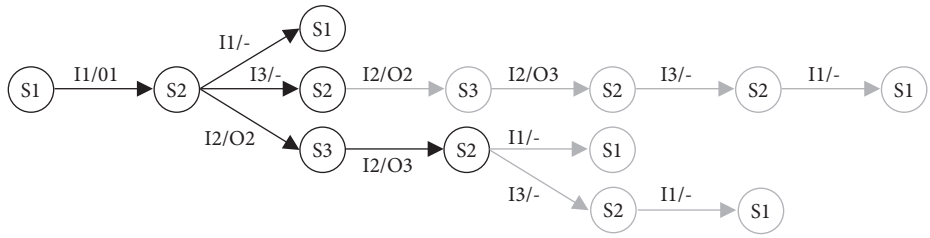


Figure 3: ConDado algorithm for test case generation.

identification of errors in the system under testing. The main reason is that the sneak paths are not explicitly described in the specification and, therefore, is usually not considered in the system implementation.

The second verification technique considered in our proposal is model checking, which is based on the model state space exploration. Given a state transition model and a property, the model checking algorithm explores exhaustively the model state space in order to determine whether it satisfies the checked property or not. In our work, we use UPPAAL to first simulate and then formally verify the state machines obtained manually from the cause-effect tables of the specification document. UPPAAL provides a modelling, simulation, and verification framework for real time systems [8]. In UPPAAL, the system is modelled as a network of timed automata extended with clocks and data types. The timed automata communicate between them through channels and shared variables. For verification purposes, the system requirements must be specified as model properties defined in CTL (computational tree logic), which are then submitted to the model checking engine.

Although the case studies presented in this paper do not explicitly include time intervals, we have chosen UPPAAL due to the possibility of extending our approach to real-time systems. Moreover, UPPAAL has a user-friendly interface and we can easily establish a direct correspondence between the CoFI and the UPPAAL models, as shown in [9] which explores the automatic translation of a FSM into a UPPAAL automaton.

3. The Verification Approach

The proposed approach receives as input the specification of the software component that simulates a satellite subsystem and aims at verifying the corresponding implementation after its integration into the satellite simulator.

The approach is organized in four main steps as illustrated in Figure 4. Step 1 consists of extracting a Mealy state machine from the specification of the satellite subsystem. For this purpose, a manual extraction method is proposed. In the following, Step 2 aims at complementing the state machine with the sneak paths' transitions, which represent the behaviour of the system when receiving unexpected inputs. In Step 3, the Mealy state machine is converted to an automaton and verified in UPPAAL model checking tool. Eventual errors imply in the correction of the specification, the Mealy state machine, and/or the UPPAAL model. Then, in Step 4, the Mealy state machine is used to automatically generate abstract test cases. Finally, in Step 5, the abstract test cases are automatically transformed into executable test cases and applied to the simulator. The results are analysed, and the nonconforming outputs provide feedbacks for correcting the specification, the state machines, the UPPAAL model, or the simulator implementation.

Following, we first present the standard adopted at INPE to describe the behaviour of the satellite subsystems, as implemented in the simulator components. We then detail the steps of the method, using as an example the data collecting subsystem (DCS) of the CBERS.

3.1. The Satellite Subsystem Specification. The specification of the satellite subsystem is the input document for the verification process. It is composed of the following sections [10]:

(i) Physical view: it describes the physical components of the satellite subsystem as a block diagram, including redundancy. An example is presented in

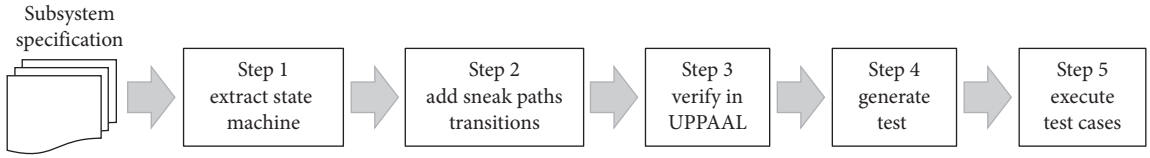


Figure 4: Steps of the verification approach.

Figure 5 and illustrates part of the physical architecture of the DCS subsystem. It is composed of the following equipment: two redundant transponders (Eq-A1 and Eq-A2), one transmitter box with internal redundancy (Eq-B), one diplexer box with internal redundancy (Eq-C), and two antennas (Eq-D and Eq-E).

(ii) Logical view: it presents the logical connection among telecommands (identified as TCnn), electrical switches (identified as SWnn), and telemetries (identified as TMnnn) of the satellite subsystem. The telecommands are commands sent from ground station, whereas telemetries are pieces of information about the satellite state sent to the ground station. The logical view is complementary to the physical architecture and shows how logical command lines are energized or not according to the received telecommands and the corresponding state of the switches. It also shows how the telemetries are modified as a consequence of the switches' states. Figure 6 illustrates part of the circuit that controls the power distribution to the Diplexer, as an example. The switch SW01 is closed by TC01 and opened by TC02. The telemetry TM006 provides the status of its output.

(iii) Operating modes and working states: it defines the operating modes of the subsystem (identified as OMnn), the operating modes of the subsystem equipment (identified as EM_xx_ii), and the working state of the equipment's internal components (identified as WK_xx_ii). The subsystem operating mode is a consequence of its equipment operating modes, which is a consequence of the working state of its components. In the case of the DCS subsystem, Table 1 presents two of the DCS operating modes, Table 2 exemplifies the possible operating modes of Eq-A1, and Table 3 brings the working state of Eq-A1 components (transponder 1 and oscillator 1).

(iv) Power consumption view: it describes the power distribution (PW) according to the operating modes of the equipment. Table 4 brings an example for the DCS subsystem.

(v) Subsystem behaviour: it contains the cause-effect tables that specify how telecommands, external parameters, and switches affect telemetries, component working states, equipment operating modes, and the subsystem operating mode. The following cause-effect tables are specified:

(a) TC ⟶ SW: how telecommands affect switches (example in Table 5).
(b) SWs ⟶ WK: how switches affect component working states (example in Table 6).
(c) WKs ⟶ TMs: how component working states affect telemetries (example in Table 7, observing that it is not possible to have both transponders on simultaneously, imposed by the subsystem design constraint).
(d) WKs ⟶ EM: how components' working states affect equipment operating mode (example in Table 8).
(e) EMs ⟶ OM: how equipment operating modes affect the subsystem operating mode (example in Table 9).

3.2. Step 1: Extract the State Machines. The first step of the verification process consists of building a set of finite state machines based on the specification of the subsystem. This is a manual activity and, in order to guide it, we propose an extraction guide composed of 9 activities, as illustrated in Figure 7.

We describe each activity in the following:

(i) *Activity 1*: identify the operating modes (OM) of the subsystem.

(ii) *Activity 2*: associate each operating mode (OM) to a state of the Mealy state machine.

(iii) *Question 1*: does the system have redundant equipment/channels? If yes, go to Activity 3, if not go to Activity 4. The term "channel" indicates a set of equipment that provides a certain functionality and may have redundancy. In the case of the DCS example, the subsystem has two channels, as illustrated in Figure 5.

(iv) *Activity 3*: identify the redundant options for each subsystem operating mode. The redundant options depend on the system architecture and allowed combination of equipment. They are specified in the physical view of the subsystem. Copy the states of those operating modes in the state machine, according to the number of redundant options.

(v) *Activity 4*: identify the equipment operating modes (EMs), components working states (WKs), and switches (SWs) configuration corresponding to each state of the state machine. We observe that this information is added to the model in order to make easier the next activities. The state label is not a formal component of the FSM and does not impact in the test case generation.

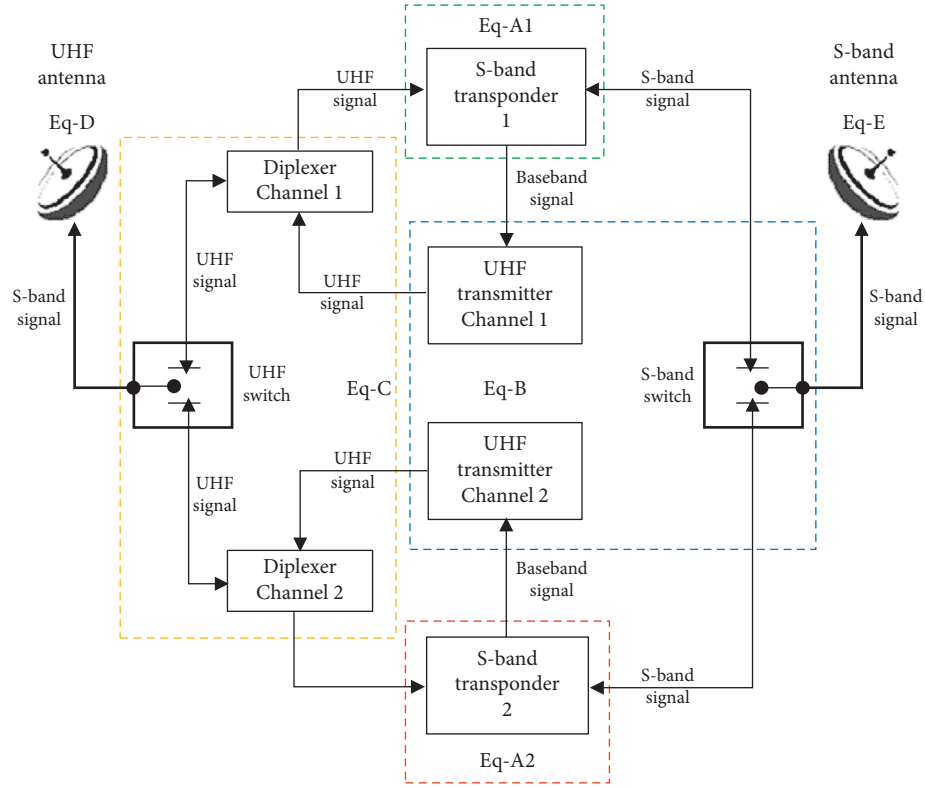


Figure 5: Physical architecture—DCS example.

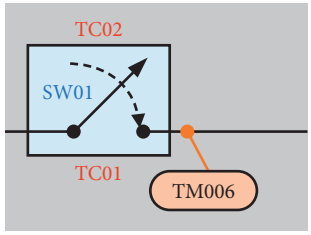


Figure 6: Part of the logical view—DCS example.

Table 1: Satellite subsystem operating modes—DCS example.

Operating modes of DCS	Value
S-band and UHF band on	OM01
S-band and UHF band off	OM02
S-band on and UHF band off	OM03
Stand-by, only oscillators are turned on	OM04

Table 2: Equipment operating modes—DCS example.

Operating modes of Eq-A1	Values
Operating	EM_A1_01
Stand-by	EM_A1_02
No oscillator	EM_A1_03
Powered off	EM_A1_04

(vi) *Activity 5*: identify the possible transitions among subsystem operating modes (OM). Add the corresponding transitions to the state machine.

(vii) *Activity 6*: identify the telecommand (TCs) that leads the system from one state to another.

(viii) *Activity 7*: associate the input/outputs of each transition. The input is the telecommand (TC) identified in Activity 6, and the outputs are the equipment operating modes (EMs), the components working states (WKs), and the switches (SWs) configurations identified in Activity 5.

(ix) *Activity 8*: identify the initial position of the switches (SWs) and define the corresponding state as the initial state of the Mealy state machine.

Figure 8 presents the Mealy state machine for the DCS subsystem, which was obtained using the described extraction guide. The bold circle indicates the initial and final state. We observe that WK, SW, EM, and OM variables are omitted in order to maintain the model readability.

3.3. Step 2: Add Sneak Paths' Transitions. The second step is inherited from the CoFI methodology. It consists of complementing the Mealy state machine with abnormal or unexpected behaviour. For this purpose, we built the sneak paths' table. It defines the expected output when the subsystem receives an unexpected input, i.e., any TC not yet considered. Table 10 presents a partial view of the sneak paths table, with the new transitions for states OM02_CH1 and OM04_CH1. For these two states, it resulted in adding 3 new faulty states to the Mealy machine (OMf01-CH1, OMf02-CH1, and OMf07-CH1) and 12 new unexpected transitions. These new states and transitions are usually not

Table 3: Component working states—DCS example.

Equipment	Component working state	Description	Values
Eq-A1	WK_A1_01	Power state of transponder 1	On Off
	WK_A1_02	Power state of oscillator 1	On Off

Table 4: Equipment power distribution—DCS example.

Equipment	Equipment operating mode	Consumption (PW)	
		PW02	PW03
Eq-A1	EM_A1_01	3.96	1.1
	EM_A1_02	0	1.1
	EM_A1_03	3.96	0
	EM_A1_04	0	0

Table 5: Example of TC ⟶ SW cause-effect table.

Conditions	Effects
Telecommand	SW01
TC01	On
TC02	Off

Table 6: Example of SWs ⟶ WK cause-effect table.

Conditions		Effects		
SW04	SW02	WK_A1_02	WK_A2_02	WK_B_03
On	CH1	On	Off	On
	CH2	Off	On	
Off	CH1 CH2	Off	Off	Off

covered when the tests are based on the system specification and are an important contribution of this work.

The model with the sneak paths' transitions is presented in Figure 9. We observe that WK, SW, EM, and OM variables are omitted in order to maintain the model readability.

3.4. Step 3: Verify in UPPAAL. The third step consists of converting the Mealy state machine to an automaton and verifying it in UPPAAL. The following rules are used in the conversion:

(i) A Boolean variable is created in UPPAAL for each TM, SW, WK, and OMs.

(ii) A channel is created for each TC.

(iii) An automaton is built with the same states and transitions of the Mealy machine.

(iv) A second automaton is created, which is capable of sending TCs in any sequence, as illustrated in Figure 10.

(v) The input of a Mealy machine transition is modelled as an UPPAAL channel marked with "?," which corresponds to a synchronous transition firing initiated by the TC generator. The output of each transition of the Mealy machine is converted to an action in UPPAAL that updates the telemetries values (TMs) and other variables (), as illustrated in Figure 11.

Figure 12 illustrates the resulting model in UPPAAL. As for Figure 10, WK, EM, and OM variables are omitted in order to maintain the model readability. The initial state is OM2_CH2 and is marked with a double circle. It is first verified using simulation. Then, formal verification is carried out by submitting properties to check the consistency of the variable values in each state, the reachability of all the states, and the absence of deadlock. Examples of properties verified using model checking are presented in Table 11.

The verification process has detected 25 errors, corresponding to incomplete or wrong transition outputs. One example is if the statement TM005 = 1 is missing from the output of a transition, then the corresponding property that check its value in OM01_CH2 is false. In order to correct the error, UPPAAL allows the user to request a diagnostic trace that will show a sequence of transition firings that makes the property false. By analysing it, we can find the error and add the missing statement to the corresponding transition. The errors and inconsistencies detected in UPPAAL must be corrected in both the UPPAAL model and the Mealy state machine.

3.5. Step 4: Generate Test Cases. Once that the Mealy machine has been corrected, it is used to generate test cases with the ConDado tool. In the case of the DCS subsystem, ConDado generated 444 test cases. However, the ConDado tool provides abstract test cases, which must then be converted into executable test cases. This activity is strongly dependent of the testing environment.

In this work, a dedicated tool was developed to perform this transformation for the case of SIMCBERS simulator. It has two main functions: (1) translate the test cases into a proper execution format in the SIMCBERS operational simulator and (2) compare the simulator outputs with the related test oracles automatically generated by the ConDado tool.

3.6. Step 5: Execute Test Case. In Step 5, we apply the concrete test cases to the satellite simulator and register the outputs. We then compare the outputs to the test case oracle, provided by ConDado. The result of the comparison can be "success," indicating conformity between the outputs produced by the simulator and the outputs of the test oracle, or "failure," indicating a nonconformance or failure. Finally, we analyse discrepancies to identify whether the problem is in the specification, the state machine models, or the simulator.

In the case of the DCS subsystem, the execution of the 444 test cases in the CBERS simulator resulted in 23 nonconformances, which were caused by 9 errors. Examples of identified errors are provided in Table 12.

Table 7: Example of WKs ⟶ TMs cause-effect table.

Conditions		Effects				
WK_A1_01	WK_A2_01	TM001	TM002	TM006	TM009	TM011
On	On	N/A	N/A	N/A	N/A	N/A
	Off	21, 5	0	On	On	Off
Off	On	0	21, 5	On	Off	On
	Off	0	0	Off	Off	Off

Table 8: Example of WKs ⟶ EM cause-effect table.

Conditions		Effects
WK_A1_01	WK_A1_02	Eq_A1
On	On	EM_A1_01
	Off	EM_A1_03
Off	On	EM_A1_02
	Off	EM_A1_04

Table 9: Example of EMs ⟶ OM cause-effect table.

Conditions				Effects
EM_A1	EM_A2	EM_B	EM_C	OM
Operating	Powered off	Operating at Channel 1	Operating at Channel 1	OM01
		Stand-by at Channel 1		OM03
Powered off	Operating	Operating at Channel 2	Operating at Channel 2	OM01
		Stand-by at Channel 2		OM03
Stand-by	Powered off	Stand-by at Channel 1	Stand-by at Channel 1	OM04
Powered off		Powered off	Powered off	OM02
Powered off	Stand-by	Stand-by at Channel 2	Stand-by at Channel 2	OM04
	Powered off	Powered off	Powered off	OM02

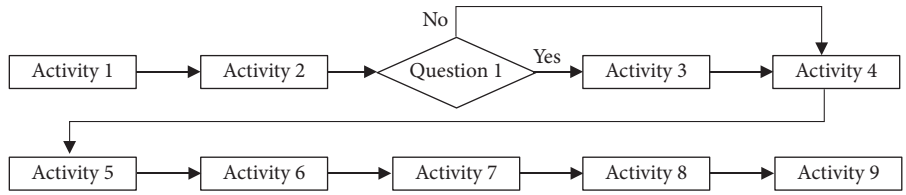


Figure 7: Extraction guide—manual method.

4. Case Studies and Results

In order to evaluate the contribution of the proposed approach, we applied it not only to the DCS but also to two additional subsystems from the SIMCBERS, the simulator of the CBERS [11]. In all the three cases, the subsystems implementation in the simulator had already been submitted to the usual test campaigns adopted at INPE for the SIMCBERS modules, which are composed of a set of tests defined based on the experience of the test engineer, with the purpose of covering the subsystem specification.

In total, the following CBERS subsystems were tested:

(i) DCS (data collection subsystem): as introduced early, it corresponds to the subsystem responsible for collecting data from platforms scattered on the ground and retransmitting the data to ground stations.

(ii) OBDH (on-board data handling subsystem): it corresponds to the satellite on-board computer system.

(iii) SYSC (system circuit subsystem): it corresponds to the circuit responsible for providing electric power to the payloads.

Table 13 summarizes the results obtained in the three cases. It presents the number of test cases, test failures, and implementation errors detected using the proposed approach. In all the cases, the test campaign identified implementation errors that were not initially detected by the SIMCBERS current testing approach. It is interesting to see that the DCS and SYSC subsystem presents similar results, with a relatively low percentage of test case failure. Comparatively, the OBDH had a very high percentage error of test failure—this is due to

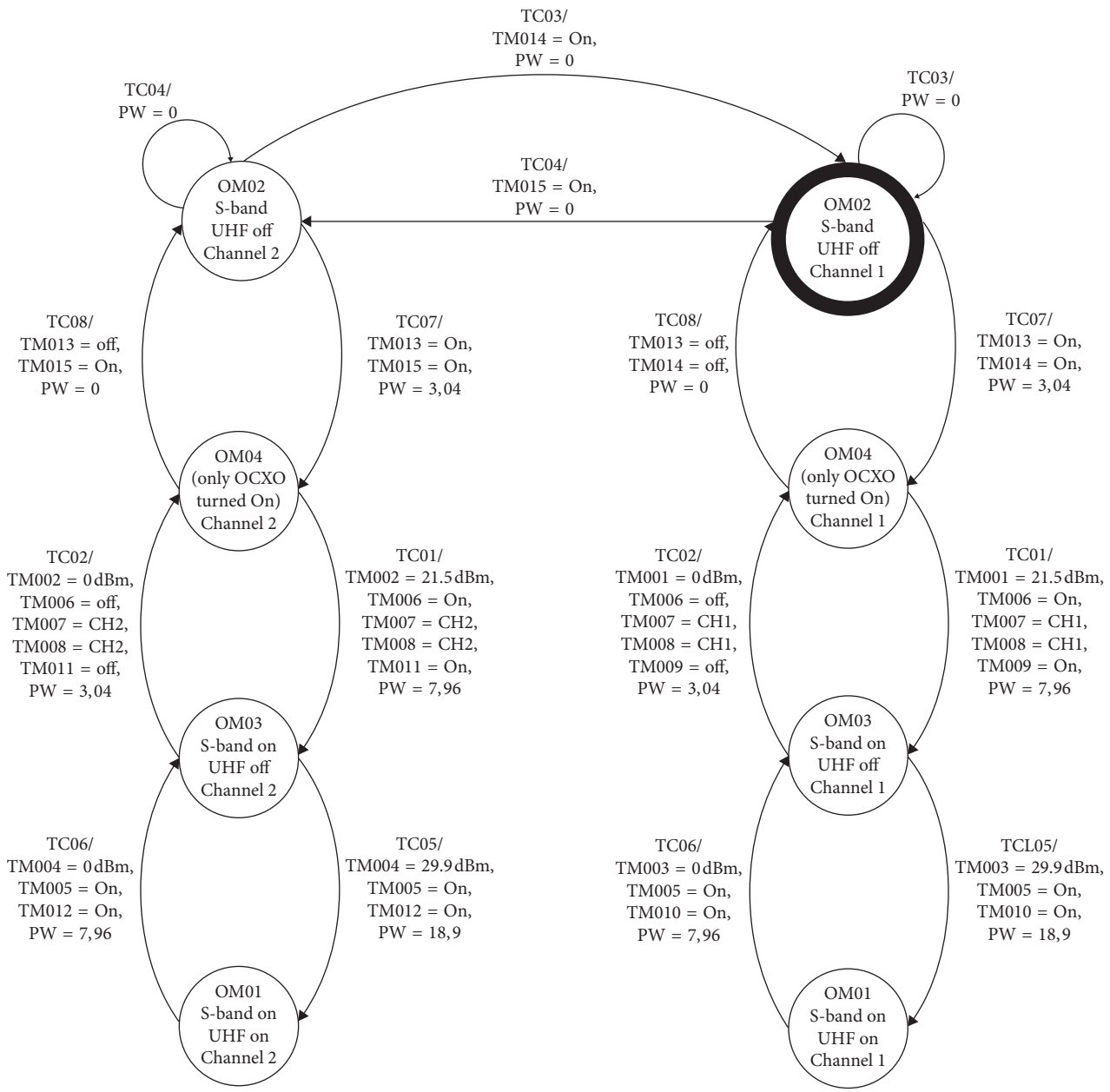


Figure 8: Mealy state machine for the DCS subsystem.

Table 10: Sneak paths table–DCS example.

	TC01	TC02	TC03	TC04	TC05	TC06	TC07	TC08
OM02_CH1	OMf01-CH1	OM02-CH1	OM02-CH1	OM02-CH2	OMf02-CH1	OM02-CH1	OM04-CH1	OM02-CH1
OM04_CH1	OM03-CH1	OM04-CH1	OM04-CH1	OM04-CH2	OMf07-CH1	OM04-CH1	OM04-CH1	OM02-CH1

a missing variable in the subsystem implementation, which results in the systematic missing of information in the test case outputs. Another interesting observation is the variation in the ratio between test failures and implementation errors. A single test failure can identify multiple errors, or a single error can lead to failure in multiple test cases. This is because the error is usually related to a transition. Consequently, all the test cases that execute the erroneous transition will be affected and fail. Similarly, if a single test case activates two erroneous transitions, then both will affect the test case output.

We observe that the amount of errors indicated in Table 13 does not include those errors previously detected by the UPPAAL verification process, once they were corrected before the test case generation.

Additionally, we compare the results obtained with our approach (Approach A) to those obtained using the original

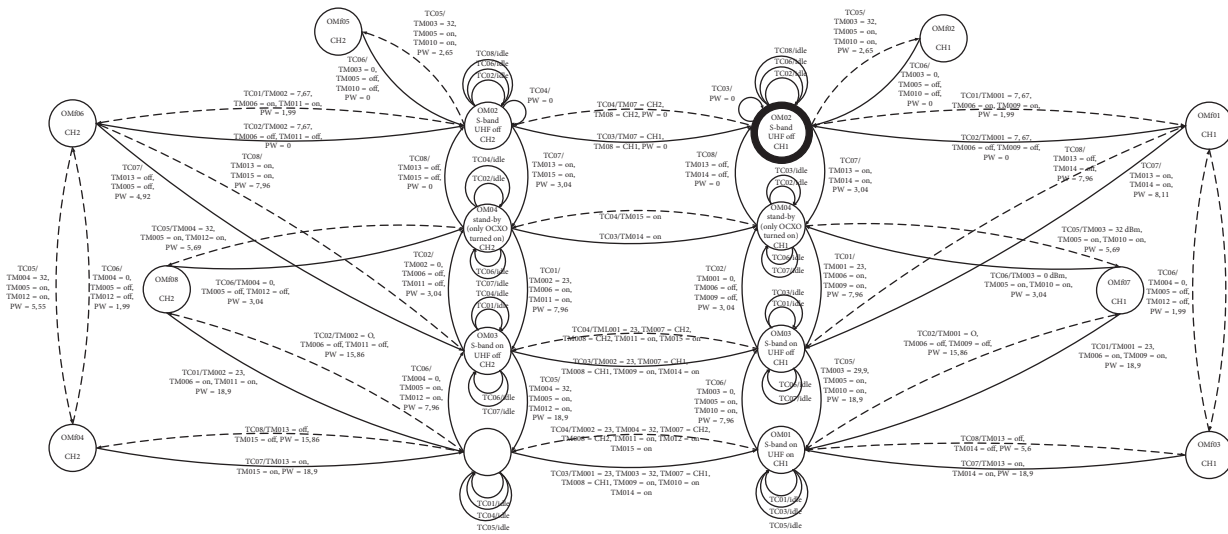


Figure 9: Mealy state machine with sneak paths—DCS example.

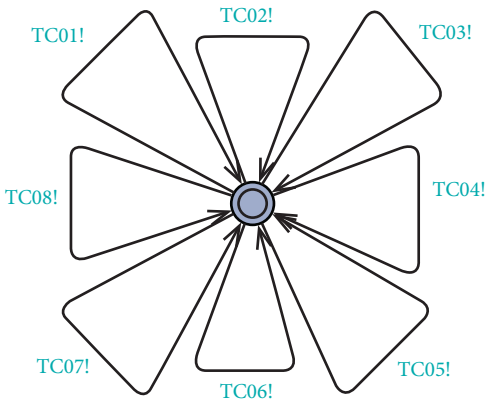


Figure 10: TC generator in UPPAAL.

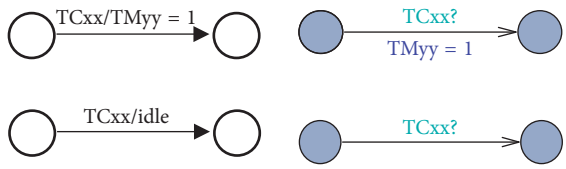


Figure 11: Equivalence between UPPAAL automaton and mealy machine.

CoFI methodology (Approach B), as proposed in [3]. The test cases from both approaches were applied to the satellite simulator. The results are presented in Table 14 and shows that, in all the 3 case studies, the approach proposed in this work (Approach A) resulted in a large amount of "failures," and therefore, in a large number of errors detected.

What differs in Approach A compared with Approach B is the use of UPPAAL. While the FSM of Approach A had been verified by UPPAAL, those of Approach B had not. The FSM of Approach B was incomplete, with missing information in the input/output events of the transitions. As a result, some of the errors were not detected in Approach B. We observe that although the FSM of Approaches A and B was different, the differences were all in the input and output events associated with the transitions. Consequently, the number of test cases is the same for both approaches because the paths generated by the ConDado tool were the same. The number of paths is modified only when we either change the source or final state of a transition, or add or remove transitions from the FSM. Complementing Tables 14 and 15 presents the modelling errors identified during the model checking verification in UPPAAL.

The results of both approaches are analysed using the chi-squared test in order to verify the following hypothesis H_0 or reject it (which means to verify the hypothesis H_1).

> H_0: the number of failures found during the test campaign does *not depend* on the approach.
>
> H_1: the number of failures found during the test campaign *depends* on approach.

We tabulate the observed and expected frequency of each approach in order to verify the hypothesis. Table 16 shows the expected frequencies, calculated according to equation (1) [12] for each case study:

$$E_{ij} = \frac{\left(X_{i+} * X_{j+}\right)}{X_{++}}, \tag{1}$$

where E_{ij} is the expected frequency, X_{i+} is the sum of row i (success or failure), X_{j+} is the sum of column j (Approach A or B), and X_{++}: sum of all rows and columns.

Given the expected frequencies, the next step is to calculate the expected Q_{ij} for each cell ij, according to the following equation:

$$Q_{ij} = \frac{\left(O_{ij} - E_{ij}\right)}{E_{ij}}, \tag{2}$$

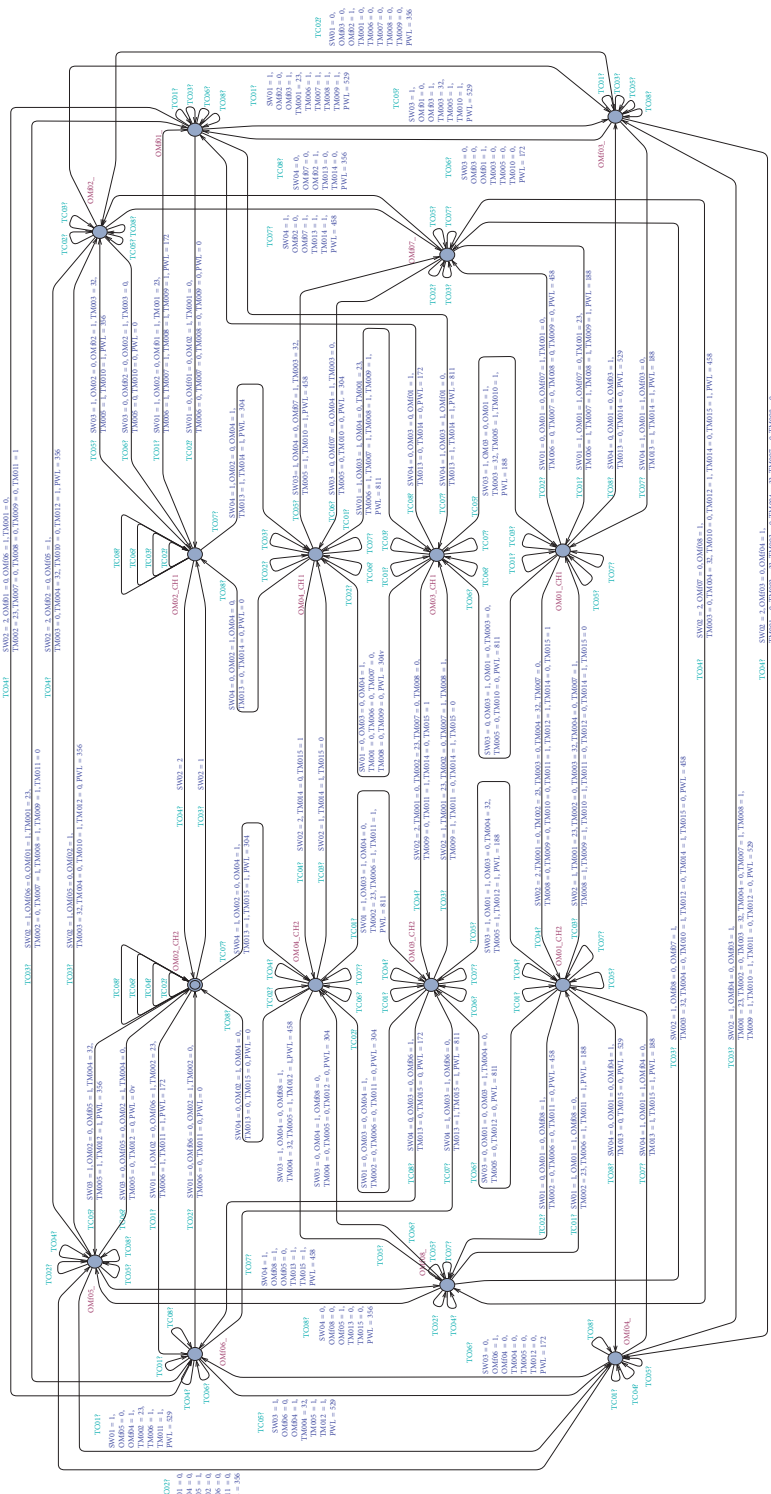


Figure 12: UPPAAL automaton of the subsystem.

where E_{ij} is the expected frequency (from Table 16), O_{ij+} is the observed frequency (from Table 14), and Q_{ij} is the chi-squared value for cell ij.

The resulting $Q_{\text{calculated}}$ associated with each case study is the sum of all Q_{ij} of that case study. The results are in Table 17 for each subsystem. In order to test the hypothesis, the $Q_{\text{calculated}}$ is compared to the Q_{critical}, which is equal to 3.84 for a degree of confidence of 95%.

In the case of the DCS and the OBDH, $Q_{\text{calculated}} > Q_{\text{critical}}$. This indicates that the hypothesis H_0 is strongly rejected, confirming that the proposed approach contributes to the identification of more failures. For the SYSC, $Q_{\text{calculated}}$

Table 11: Examples of properties for the DCS subsystem.

Properties	Description
A[] not deadlock	The system never enters in deadlock
A[] SYSTEM.OM01_CH1 imply SW02 == 1	Whenever the system is operating on Channel 1, then the switch SW02 is configured to Channel 1
A[] SYSTEM.OMf5_CH2 imply WK_B_01 == 1	Whenever the system is in the fault mode OMf5, the working state of equipment B is 1
A[] SYSTEM.OM01_CH2 imply TM_005 == 1	Whenever the system is operating on Channel 2, telemetry TM005 is equal to 1

Table 12: Example of errors detected for the DCS subsystem.

Variable	Variable description	Error description
WK_C_01	Electrical status of the diplexer	The variable is not defined when the diplexer enters in the stand-by mode
TM007	UHF switch status, which indicates the channel the transmitter is working on	The variable is not updated according to the cause-effect table of the DCS specification
TM008	S-band switch status, which indicates the channel the transmitter is working on	The variable is not updated according to the cause-effect table of the DCS specification
OMf4	Fault operation mode where the oscillator and transmitter are on	The simulator does not include faulty state; the variables assume wrong values

Table 13: Results for DCS, OBDH, and SYSC.

	DCS	OBDH	SYSC
Number of test cases	444	39	420
Test case failures	23	22	16
% of test case failure	5.1	56	3.8
Errors detected in test campaign	9	79	16
Ratio of implementation error per test failures	0.39	3.59	1

Table 14: Experiment results.

	Case study 1—DCS		Case study 2—OBDH		Case study 3—SYSC	
	Approach A	Approach B	Approach A	Approach B	Approach A	Approach B
Success	421	438	17	30	404	410
Failure	23	6	22	9	16	10
Total	444	444	39	39	420	420

Table 15: Modelling errors identified in UPPAAL.

	DCS	OBDH	SYSC
Errors detected in UPPAAL	25	17	27

Table 16: Expected frequencies.

	Case study 1—DCS		Case study 2—OBDH		Case study 3—SYSC	
	Approach A	Approach B	Approach A	Approach B	Approach A	Approach B
Success	429.5	429.5	23.5	23.5	407	407
Failure	14.5	14.5	15.5	15.5	13	13

Table 17: Chi-squared test results.

	Case study 1—DCS		Case study 2—OBDH		Case study 3—SYSC	
	Approach A	Approach B	Approach A	Approach B	Approach A	Approach B
Success	0.17	0.17	1.80	1.80	0.02	0.02
Failure	4.98	4.98	2.73	2.73	0.69	0.69
$Q_{calculated}$	10.3		9.05		1.43	

$< Q_{critical}$, which indicates that, in this case, the proposed approach did not result in a significant improvement in failure identification. One of the factors that can explain the difference among the subsystems is the low complexity of SYSC subsystem when comparing with the OBDH and DCS.

5. Related Work

This section discusses published works related to two main topics that are combined in our approach: cause-effect tables, also called decision tables, and model-based testing. Regarding cause-effect tables, we present a few examples, from the early 80s to recent years, which illustrate the use of cause-effect tables for specification, implementation, and verification purposes. They also show that cause-effect tables are widely used and easily understood by people from different backgrounds. Concerning model-based testing, we focus on works that discuss their applicability to specific industrial domains, such as the case of our approach.

Harrison [13] advocated the use of cause-effect tables for both specification and implementation of systems. Comparing to our work, he proposed an inverse approach, which consists of deriving the cause-effect tables from state machines. The tables were used as an efficient way of implementing the corresponding software. Traverso [14] proposed the use of cause-effect tables to obtain the specification of a system and, then, exhaustively verify the system. He introduced a method to obtain a complete table with all possible inputs and the expected outputs and derive test cases directly from the table. The approach was applied to a case study from the railway transportation domain. Cheng et al. [15] uses I/O tables to represent results obtained from a FMEA analysis. The table contained both normal and failure behaviours and were used as an alternative way to automatically obtain fault trees and deriving tests.

In a more recent work, Huysmans et al. [16] analysed the comprehensibility of different modelling languages (cause-effect table, decision true and propositional rules, and oblique rules). The authors performed an experiment to access how the end-user perceives each language. It measures the accuracy, response time, and answer confidence when the user is required to perform a different takes using the models. It shows that cause-effect tables perform significantly better on all three criteria, in terms of ease of use. The results obtained by Huysmans et al. corroborate the choice made by INPE specialists of using cause-effect tables for describing the satellite subsystems.

Regarding model-based testing, a number of works in the literature describes efforts performed in the direction of integrating model-based testing techniques in the industry.

The Automated Generation and Execution of Test Suites in Distributed Component-based Software (AGEDIS) was a three-year project funded by the European Union [17] to investigate the automation of software testing activity. Five case studies were performed in three different companies: IBM, Telecom (France), and Intrasoft. The project results indicated that the creation of tools to execute and record the huge number of test cases was the main challenge when introducing model-based testing in the industry. Relating to our work, we developed a dedicated tool for transforming the abstract test case into executable test cases, assisting the test case execution, and the result analysis, overcoming the difficulties associated with the large amount of test cases.

Pretschner et al. [18] investigated whether model-based testing approaches are feasible in terms of quality and cost, based on a case study from the automotive industry. In terms of quality, model-based test cases were compared to manually generated test cases. The conclusion was that model-based tests did not identify more implementation failures than manual test cases. On the contrary, the authors observed that the tests generated from models resulted in a significant increase in the detection of defects in the requirements. Among the limitations of the work was the fact that the results of the nonconforming test cases were manually inspected, due to the lack of automatic tools. Another observation of Pretschner et al. was that the testers considered some of the test cases generated automatically from the model as "nonsense" because at the beginning of the test case the system was supposed to be in an unusual condition. In order to avoid this problem, the tool used in our work implements the switch cover method, which guarantees that all the generated test cases depart from the initial state of the state machine, thus avoiding test cases that do not make sense.

Regarding specifically the use of CoFI model-based testing approach, Mattiello-Francisco et al. [19] and Villani et al. [20] described lessons learned from the porting of the CoFI methodology from the space to the automotive industry. One of the challenges pointed out by both works was the need to identify an adequate level of refinement of the models, so that they are not so complex, resulting in impracticable number of test cases, nor too simplified, resulting in a superficial verification of the system. Another limitation was the difficulty of interpreting the specification, since the specification was written in natural language, giving room for ambiguities. In our work, the issue related to the level of detail of the models was overcome by the fact that the system specification unambiguously lists the inputs and outputs to be considered in the modelling. The ambiguity of the natural language specification was circumvented by the

use of cause-effect tables, while the difficulty in modelling was reduced by means of the proposed method for translating the cause-effect tables into state machines.

Finally, both Anjos et al. [7] and Pontes et al. [6] combined the use of the CoFI methodology with UPPAAL model checking. However, unlike our work, the techniques were applied independently, without any reuse of models. In our work, model checking is used to verify the models used by CoFI, contributing to the correctness of the model used in the automatic test case generation.

6. Conclusion

This work proposed an approach to systematize the generation of test cases for components of satellite simulators that follows a new specification standard elaborated by the Brazilian National Institute for Space Research (INPE). The specification is based on cause-effect tables to define cause-effect relations among telecommands, telemetries, switches, and equipment states. The choice of using cause-effect tables was derived from the need of sharing the specification among people from different fields and backgrounds.

Although they are very easily understood, the cause-effect tables have the disadvantage of inducing testing engineers to limit the test case suite only to the most obvious sequences of the telecommands, jeopardizing the verification process. In order to overcome this limitation, this work proposes to derive state machines from the cause-effect tables and then use the state machine models to automatically generate test cases that cover different combination of events. The process of creating state machines from the cause-effect tables are currently performed manually and is supported by the CoFI methodology and UPPAAL model checker.

The CoFI methodology contributes to the modelling process by incorporating new states and transitions associated with unexpected behaviour that results from receiving telecommands at unexpected events. They assure the robustness of the satellite simulator. The UPPAAL model checker contributes to verifying the consistency of the models that will be used to generate test cases, avoiding unnecessary rework due to modelling errors.

Once the state machines corresponding to the satellite components have been verified, the ConDado tool is used to generate abstract test cases. A dedicated tool has been developed to translate the test cases into concrete test cases and apply it to the satellite simulator.

In order to validate the proposed approach, we applied it to three subsystems of SIMCBERS, the simulator of the China-Brazil Earth Resources Satellites, developed at INPE. All the three subsystems had already been verified using traditional testing methods in practice at INPE. The results showed that, in all the three cases, our approach was able to identify additional errors, contributing to the robustness of the simulator. We can, therefore, affirm that our approach complements the methods presently in use at INPE, by identifying problems and errors in the simulator implementation that had not been detected before.

Additionally, a comparison with the original CoFI methodology has also been performed in order to determine the contribution of the UPPAAL verification. For two subsystems, the increase in the number of test case failures due to model checking was considered statistically significant. In one case, it was not significant. We believe that the difference in the results is due to the complexity of the subsystems. When modelling a subsystem with redundant equipment, a large number of telecommands, switches, and operation modes, the modeller is more prone to introduce errors.

Summarizing the contributions of the proposed method, the introduction of a modelling step, which converts the cause-effect tables to a state machine, highlights the dynamic behaviour of the system emerging from the combination of multiple cause-effect tables. While a single cause-effect table shows the effect of a single event on the system state, the state machine explicitly shows the effect of a sequence of events. The use of model-based testing approach, such as CoFI, assures that the test cases cover more than the obvious sequences of commands related to normal behaviour of the system. Comparing to the methods currently in use at INPE, our method assures that the coverture of the set of test cases reduces the dependence on the operator experience.

Data Availability

The following data supported the findings of this study: satellite subsystems' specifications, UPPAAL models, FSM, test cases, and test case results. The DCS specification, the UPPAAL models, and the FSM are available from the corresponding author upon request. The availability of the remaining data is not authorized by the National Institute of Space Research (INPE) due to security reasons, as it is related to operating satellites.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to acknowledge SIMCBERS development team at INPE, CAPES, and CNPq (Process 306408/2014-7).

References

[1] ECSS-European Cooperation dor Space Standardization, "Space engineering: system modelling and simulation," Technical Report ECSS-E-TM-10-21A, ESA-ESTEC, Noordwijk, Netherlands, 2010.

[2] J. Eickhoff, "Simulating spacecraft systems," in *Springer Series in Aerospace Technology*, Springer Science & Business Media, Berlin, Germany, 2009.

[3] A. M. Ambrosio, E. Martins, N. L. Vijaykumar, and S. V. Carvalho, "A conformance testing process for space applications software services," *Journal of Aerospace Computing, Information, and Communication (JACIC)*, vol. 3, no. 4, pp. 146–158, 2006.

[4] T. Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, Pearson, London, UK, 3rd edition, 2005.

[5] E. Martins, S. B. Sabião, and A. M. Ambrosio, "ConData: a tool for automating specification-based test case generation for communication systems," *Software Quality Journal*, vol. 8, no. 4, pp. 303–320, 1999.

[6] R. P. Pontes, P. C. Véras, A. M. Ambrosio, and E. Villani, "Contributions of model checking and CoFI methodology to the development of space embedded software," *Empirical Software Engineering*, vol. 19, no. 1, pp. 39–68, 2012.

[7] J. M. S. Anjos, G. K. Coracini, and E. Villani, "A proposal and verification of a software architecture based on LabVIEW for a multifunctional robotic end-effector," *Advances in Engineering Software*, vol. 55, pp. 32–44, 2013.

[8] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, vol. 3185, Bertinoro, Italy, September 2004.

[9] E. Villani, R. P. Pontes, G. K. Coracini, and A. M. Ambrosio, "Integrating model checking and model based testing for industrial software development," *Computers in Industry*, vol. 104, pp. 88–102, 2018.

[10] J. Tominaga, C. Cerqueira, J. Kono, and A. Ambrosio, "Specifying satellite behavior for an operational simulator," in *Proceedings of SESP 2012: Simulation and EGSE facilities for Space Programmes*, ESTEC, Noordwijk, Netherlands, September 2012.

[11] P. D. B. Silva, "Sistematização do teste de componentes de simuladores de satélites," Master thesis, Aeronautics Institute of Technology, São José dos Campos, Brazil, 2017.

[12] A. Agresti, *An Introduction to Categorical Data Analysis*, John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2007.

[13] P. G. Harrison, "Efficient table-driven implementation of the finite state machine," *Journal of System and Software*, vol. 2, no. 3, pp. 201–211, 1983.

[14] A. M. Traverso, "A tool for specification analysis: complete decision tables," *IFAC Proceedings Volumes*, vol. 18, no. 12, pp. 53–56, 1985.

[15] Y. L. Cheng, H. C. Wei, and J. Yuan, "On establishment of I/0 tables in automation of a fault tree synthesis," *Reliability Engineering & System Safety*, vol. 40, no. 3, pp. 311–318, 1993.

[16] J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen, and B. Baesens, "An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models," *Decision Support Systems*, vol. 51, no. 1, pp. 141–154, 2011.

[17] I. Craggs, M. Sardis, and T. Heuillard, "Agedis case studies: model-based testing in industry," in *Proceedings of 1st European Conference on Model Driven Software Engineering*, Nuremberg, Germany, December 2003.

[18] A. Pretschner, W. Prenninger, S. Wagner et al., "One evaluation of model-based testing and its automation," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 392–401, St. Louis, MO, USA, May 2005.

[19] M. F. Mattiello-Francisco, E. Villani, E. Martins, T. Dutra, B. Coelho, and A. M. Ambrosio, "An experience on the technology transfer of CoFI methodology to automotive domain," in *Proceedings of 6th Latin-American Symposium on Dependable Computing (LADC)*, vol. 1, pp. 1–4, Rio de Janeiro, Brazil, April 2013.

[20] E. Villani, A. M. Ambrosio, T. A. Dutra, M. V. O. Araujo, and E. Martins, "From academia to industry: challenges and lessons learned from a model-based-testing experience," in *Proceedings of 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, vol. 1, pp. 27–30, Rio de Janeiro, Brazil, June 2015.