Model-based Avionic Systems Testing for the Airbus Family

Jan Peleska
Department of Mathematics and Computer Science
University of Bremen
Bremen, Germany
peleska@uni-bremen.de

***Abstract*—This paper is about practical verification of Airbus avionic systems during type certification, with special focus on automated testing. The material is based on test and verification services performed for Airbus by a spinoff company of the University of Bremen, as well as on consultancy services delivered by our research group to Airbus and its suppliers. In the context of model-based systems engineering, the test automation approach is currently shifting from manual test procedure programming to model-based testing (MBT), where test cases are automatically identified in models describing the application behavior, allowing for automated test data calculation and test procedure generation. We describe the situations where today's MBT technology is already adequate to increase the effectiveness of automated testing in industry. In addition, we describe some open challenges arising from practical avionic systems testing, where satisfactory solutions still require some research effort.**

***Index Terms*—model-based testing, test automation, model-based systems engineering, avionic systems**

I. Introduction

A. Motivation

HW/SW integration testing of avionic systems has been automated at the beginning of this century in the sense that test procedures were programmed and could be executed automatically and in real-time. Apart from the motivation to reduce the effort involved in manual testing, this automation step was unavoidable: new generations of avionic systems made manual interactions with the system under test (SUT) during test executions more and more unreliable, if not infeasible. This was due to the growing speed and the growing number of interfaces to be stimulated and monitored during test executions.

During recent years, the methodology of *model-based systems engineering (MBSE)* and its supporting tools has been observed by aircraft manufacturers like Airbus with growing interest, and

- the potential of automated generation of target system code, as well as
- the possible efficiency increase to be gained from *model-based testing (MBT)*

have been investigated. In some areas, these possibilities have already been integrated into the development and verification processes, and we expect the MBSE paradigm to be adopted more widely in the avionic systems domain within the next years.

In this paper, we report about MBT campaigns recently performed for Airbus by a company specialized on embedded systems verification[1] and supported by our research group at the University of Bremen. We summarize the technology used for MBT and describe the main benefits to be gained from applying MBT today, as well as some open challenges for future improvements of the underlying methodology.

When using the term MBT in the context of this paper, this is understood in the following sense. A *test model* is developed specifying the expected behavior of the SUT, as far as observable on the interfaces of a *hardware-in-the-loop (HiL)* testing environment. The tests are black-box, so the internal model structure will just represent a functional decomposition of the applications to be tested and not necessarily reflect the internal SUT design. The test model is used to automatically identify test cases, calculate concrete test data, and generate test procedures running the test cases against the SUT. This includes the generation of *test oracles* checking the SUT responses observed against the expected behavior encoded in the model.

B. Overview

In Section II, an overview of the verification activities to be performed for avionic systems for the purpose of type certification is given. A case study derived from a real-world test model is presented in Section III; this will be used in the subsequent sections to illustrate various aspects of MBT for HW/SW-integration testing of avionic systems. In Section IV, we describe how MBT is performed in practice, with focus on HW/SW integration testing. Open challenges for MBT in the avionic domain are described in Section V, with some research-directed suggestions how these problems might be solved in the future. Section VI presents a conclusion. We refer to related work throughout the text, where appropriate.

II. Verification Process for Avionic Systems

The verification process for avionic systems has been standardized in RTCA DO-178B and its successor DO-178C [1],

The work presented in this contribution has been partially funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) in the context of project STEVE, grant application 20Y1301L.

[1]Verified Systems International, http://www.verified.de

978-1-5386-3728-9/18/$31.00 ©2018 IEEE

[2]. The main difference between these versions consists in the explicit consideration of MBSE and Formal Methods in the successor standard.

One of these standards' main objectives is to ensure different verification perspectives that are applied to the artifacts of all stages of the development process. Verification is considered as a separate process of the system life cycle, distinguished from the processes project planning, development, quality assurance, configuration management, and certification liaison. For the highest criticality levels A and B, verification activities have to be performed with independence; in particular, software developers are not allowed to verify their own software.

As verification techniques, reviews, analyses (with varying degrees of formality), and testing are applicable according to the standards. Software requirements are typically reviewed. The standard distinguishes between *high-level requirements* directly related to the applicable system requirements and *low-level requirements* arising during the software design process. Low-level requirements can represent refinements of high-level requirements; alternatively, they can be derived from design decisions or restrictions imposed by the target hardware.

An interesting aspect of these avionic standards is that they do not require full sets of tests on all levels, from unit tests via software integration tests and HW/SW integration tests to system tests. Instead, the standard specifies a set of *completeness criteria* for the testing activities. If completeness can be achieved by means of, say, HW/SW integration testing, no further unit tests are required to gain certification credit. The DO-178C standard specifies the following criteria for the test activities to be sufficient.

1) All high-level requirements have been covered by tests.
2) All low-level requirements have been covered by tests.
3) Robustness tests have been executed for all requirements (high-level and low-level).
4) Tests have been performed on the target hardware to show that the executable code is compatible with the target hardware.
5) Tests or analyses have been performed for all elements of the system parameterization.
6) Test procedures have been verified (typically by review) that they implement the allocated test cases in a correct way; this includes the checks of the expected results typically performed automatically by the test procedure.
7) The collection of all tests (performed on all levels) achieves the code coverage required for the SUT's criticality level; for example, MC/DC source code coverage is required for level A systems.
8) The collection of tests covers the call structure of the software and access structure of software components to global resources, such as global data. The verification of this variant of structural coverage is called *data and control coupling analysis* [3].
9) For systems of criticality level A, it has to be ensured that the collection of all tests also covers object code that cannot be traced back to the C-code, because the compiler introduces additional branching statements, function calls or assignments in the object code. The process for verifying this object code coverage is called *source code to object code traceability analysis* [4].

A noteworthy consequence of these criteria distinguishes the RTCA DO-178B,C from standards in, for example, the railway domain: the avionic standards do not postulate the investigation of *test strength*[2] during the verification process; it suffices just to meet the above completeness criteria.

Another consequence of the completeness criteria consists in the fact that the avionic standards do not really need additional rules for model-based testing: if model-generated test suites fulfill the correctness and coverage criteria listed above, the tests are considered as adequate, regardless whether they have been manually written or automatically generated from a model.

The addendum RTCA DO-331 on model-based development and verification emphasizes that model simulation cannot replace tests of the target system regarding criteria 2, 3, 4, and 9 [5, Section MB.6.8.2]. The addendum discusses at length, under which circumstances some other parts of the testing process might be performed in model simulations (this is also called *model-in-the-loop (MiL) testing*. For example, it is conceded that errors like incorrect loop operations or incorrect logic decisions can be detected in MiL tests of the design model, if the latter uses the same source code as is used for compiling the target object code and is compiled with the same options as used for the target. The obligations to justify the replacement of on-target tests by MiL tests may require considerable effort. Therefore, the efficiency gained by MiL tests in comparison to tests on the target may be undone by this extra justification effort. As a consequence, we advocate MBT for the purpose of automatically generating test that can be executed on the target, and the option of MiL testing will not be considered in the remainder of this paper. This assessment of the importance of on-target testing is also supported by the supplement RTCA DO-331 which states that even the comprehensive use of formal methods during verification of documents and code cannot replace testing to ensure the compatibility between executable object code and target computer [6, Section FM.6.0].

A crucial aspect influencing the application of MBT is the requirement that all tools automating aspects of the development or verification process must be *qualified*, see [2, Section 12.2] and the sub-standard [7]. In the context of automated MBT, tools typically automate the check of the criteria 1, 2, 6, 7, and perhaps even 5, 8, and 9. As a consequence, tool verification evidence needs to be produced, showing that the tools perform these activities without errors [8].

III. Case Study – Fasten Seatbelt Sign Control

To illustrate various aspects of MBT for avionic systems, a case study concerning the control of fasten seatbelt (FSB)

[2]the ability of a test suite to uncover failures in the SUT

signs is presented in this section. The study has been derived from the real FSB control function as used in today's aircrafts, but it has been reduced with respect to the input and output interfaces to be handled, and the control logic has been simplified to facilitate the presentation of various MBT features.

A. Interfaces

The input and output interfaces of the FSB control function, parameters, and internal model variables are listed in Table I.

TABLE I
VARIABLES USED IN STATE MACHINE DIAGRAMS.

Symbol	I	M	O	Meaning
p_a	p			FSB AUTO condition variant, range 1, 2, 3
p_{ea}	p			Excessive altitude (EA) handling variant for FSB signs, range 0 (no EA handling for FSB signs), 1 (FSB signs are switched on when EA is active)
C	•			Cockpit switch for FSB signs, range 0 (FSB signs OFF), 1 (ON), 2 (AUTO)
EA	•			Excessive altitude (i.e., cabin decompression) is active, range 0 (false), 1 (true)
EM	•			Emergency mode active (normal power unavailable), range 0 (false), 1 (true)
ESG	•			Engine shutdown & aircraft on ground, range 0 (false), 1 (true)
L	•			Nose landing gear down&locked, range 0 (false), 1 (true)
S_1	•			Slats 1 extended, range 0 (false), 1 (true)
S_2	•			Slats 2 extended, range 0 (false), 1 (true)
a		•		AUTO condition active, range 0 (false), 1 (true)
f		•		FSB ON condition active, range 0 (false), 1 (true)
SC			•	System startup completed, range 0 (false), 1 (true)
F			•	Fasten seatbelts signs are switched on, range 0 (off), 1 (on), 2 (undefined)

I: p = input parameter
I: • = Input variable
M: Internal model variable
O: Output variable

The main input is the FSB cockpit switch C which is used to switch the signs on (switch position 1) or off (position 0). In switch position 2 (so-called AUTO position), the FSB signs are switched automatically on or off, depending on further inputs L, S_1, S_2, and ESG signaling the status of the nose landing gears, slats 1 and 2, and the engine status in conjunction with on-ground status of the aircraft, respectively. The normal control logic can be overridden by the occurrence of the excessive altitude condition (input EA) or by the loss of normal power (input EM).

The SUT outputs considered during FSB-related tests are represented by variables SC indicating that the SUT is in the operative state and F indicating whether the FSB signs are to be switched on. Output F is an abstraction of the status of all FSB signs which are connected to a peripheral bus and need to be controlled by sending ON/OFF commands to all device addresses where FSB signs are deployed. Since all FSB signs are switched synchronously, the test model just uses one variable aggregating their state. A subordinate software layer of the test engine monitors the individual device states and aggregates the concrete bus commands associated with FSB devices to outputs 0 (all FSB signs off), 1 (all on), or to value 2 (undefined) as long as the FSB signs are in inconsistent states.

The control logic for FSB signs depends on two configuration parameters p_a and p_{ea} which may be set only once at system startup and remain constant during the whole SUT execution. Parameter p_a has 3 different values determining the variant how FSB signs are automatically switched on or off while the cockpit switch is in the AUTO position. Boolean parameter p_{ea} indicates whether the occurrence of the excessive altitude state affects the FSB control logic or not.

B. Functional Model

The FSB control functionality is modeled by three concurrent, interacting state machines depicted in Fig. 1, 2, and 3. The first machine decides whether the condition for switching signs automatically on holds and records the decision in internal variable a (see Table I). The second decides whether FSB signs should be switched on and records this decision in the model variable f. The decision is based on inputs C and EA, and on the AUTO condition a. The third machine actually writes to the FSB control output F; the output value depends on the current value of f and the state of the EM input.

The state machines shown adhere to UML/SysML syntax as defined in [9]. The *change event* when(c) occurs when a Boolean condition switches from `false` to `true`. The transitions emanating from transient choice pseudo states are labelled with *guard conditions* in square brackets, determining the transition to be followed.

1) FSB AUTO Function: The FSB AUTO condition depends on a parameter p_a, since different airlines prefer specific variants of the automatic switch function; this is modeled by the state machine shown in Fig. 1. The machine sets the internal model variable a to 1 if and only if the AUTO condition holds. When p_a equals 1, signs shall be automatically switched on if the aircraft is not on ground with engines switched off (ESG $= 0$), and the nose landing gear is down and locked or at least one of the slats is extended. With parameterization $p_a = 2$, the AUTO condition only depends on nose landing gear and slats, and for $p_a = 3$, it depends on the nose landing gears alone.

2) FSB Control Logic: The central control logic setting the FSB on flag is shown in Fig. 2. If the excessive altitude state EA has no influence on the FSB control logic (this is reflected by parameter setting $p_{ea} =$ `false`), control of the FSB signs is specified completely by submachine FSB_NORMAL: the signs are switched on if the cockpit switch is in position 1, or if it is in position 2 and the AUTO condition a holds. Otherwise the FSB signs are switched off.

If the FSB control logic should take the excessive altitude state into account ($p_{ea} =$ `true`), occurrence of this state forces the signs to be switched on, regardless of the cockpit switch position and the state of the AUTO condition. When the

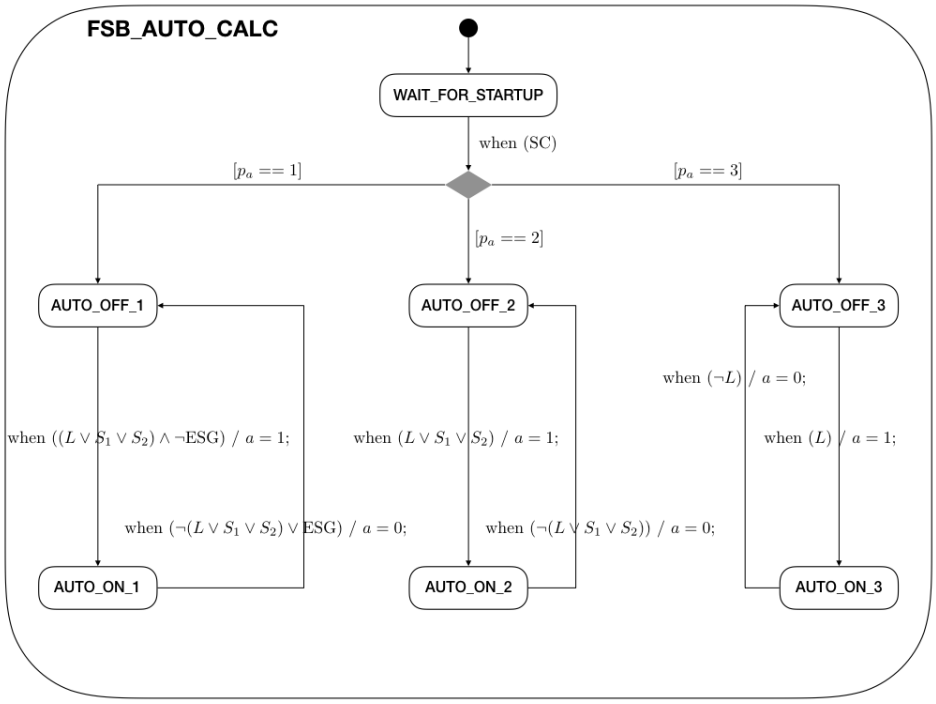

Fig. 1. State machine calculating the auto condition.

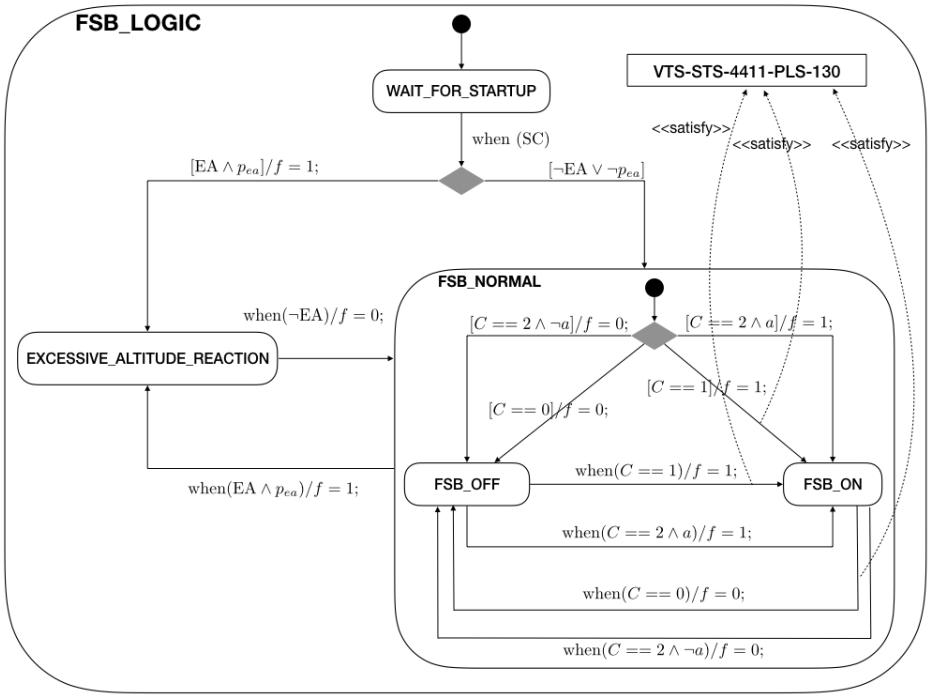

Fig. 2. State machine modeling the FSB ON/OFF logic.

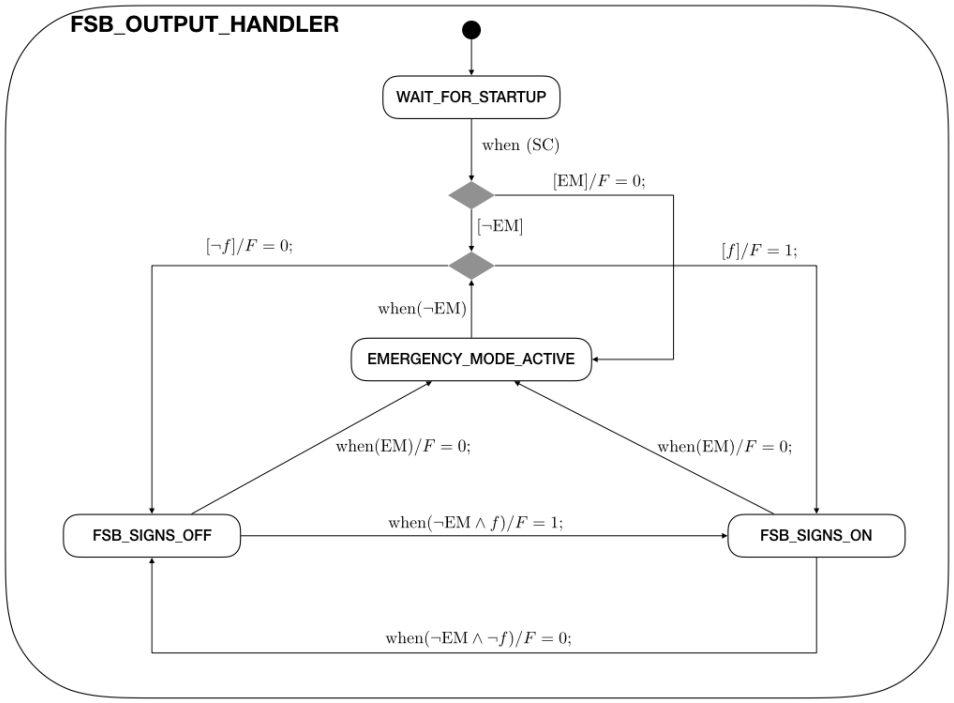

Fig. 3. State machine modeling the FSB output handler logic, taking the emergency mode into account.

excessive altitude state is no longer active, the FSB signs are switched again according to the rules of the FSB_NORMAL submachine.

The state machine in Fig. 2 also exemplifies how model elements can be traced back in SysML to the requirements they help to realize. In the example shown here, only one requirement (V-STS-4411-PLS-130) is referenced. It states that the FSB signs shall be switched on if the cockpit switch C is in position 1, and they shall be switched off if C is switched back to 0. This holds as long as the sign state is not overridden by the excessive altitude condition or the emergency mode. Obviously, all transitions setting f to 1 or 0 due to cockpit switch changes between $C == 1$ and $C == 0$ contribute to modeling this requirement. Therefore the transitions triggered by $C == 1$ and $C == 0$ conditions are connected to the requirement by means of a *satisfy relationship*. Its interpretation is that every model computation triggering one of these three transitions is a witness for requirement V-STS-4411-PLS-130. This information can be exploited to associate requirements with test cases: for each of the three transitions, a covering test case is needed, in order to test the requirement in a comprehensive way. More complex requirements can be related to constraints containing LTL formulas, expressing more complex conditions to be fulfilled by the requirement's witness computations.

Though the graphical satisfy-notation shown in Fig. 2 is part of the SysML syntax, it is not so frequently used, because it clutters behavioral diagrams if their model elements are linked to many requirements. Therefore, most tools offer a separate tabular notation for associating model elements with the requirements they help to satisfy. In the state machine diagrams presented in this paper, we have only shown this one requirement, in order to explain the SysML traceability concept. In the real-world model, more than 30 requirements need to be linked to these model elements.

3) FSB Output Logic: The state machine depicted in Fig. 3 performs the actual writes to the output interface F. As long as normal power is available (EM = `false`), output F is switched consistently with internal variable f. The occurrence of a power loss (EM = `true`) forces the FSB signs to be switched off.

IV. Model-based HW/SW Integration Testing in Practice

In this section, it is described how MBT is applied today for HW/SW integration testing of avionic systems. To this end, we first describe the capabilities of our toolset in Section IV-A and then summarize the main benefits resulting from the MBT approach in Section IV-B.

A. Technical Approach – the RT-Tester Tool

For model-based HW/SW integration testing in the avionic systems testing context described in this paper, the RT-Tester tool is used [10]–[13]. In several aspects, RT-Tester competes with tools like Rational Test Conductor and Automated Test

Generator[3], the TGV tool integrated with TorX [14], and UppAal-TRON [15]. Each of these tools has its unique selling points, but they adhere to the same model-based testing paradigm, where models specify the expected behavior of the SUT, test cases are identified in the model using some auxiliary information, and concrete test data are calculated using model checking methods for witness generation and, optionally, SMT solvers.

The RT-Tester tool parses SysML models consisting of blocks, operations, and state machines and creates an internal model representation (IMR) which is a special variant of an abstract syntax tree. The IMR is traversed to generate a transition relation $\Phi(s, s')$ relating model pre-states s to potential post-states s'. The transition relation takes dense time into account, so transitions may be discrete (performed in zero time while inputs remain stable) or delays (inputs may change and time is advanced, taking into account when discrete transitions may become enabled, due to elapsed timers). In contrast to the UppAal tool which is based on the non-urgent Timed Automata semantics [16, Chapter 9], enabled SysML transitions need to fire immediately, so our transition relation encodes *maximal progress*. More details about the encoding of the behavioral SysML semantics in RT-Tester have been described in [17, Chapter 11].

Requirements are formally represented as model properties specified in LTL. Simple requirements, as the one discussed in Section III-B2 above, can be represented by conjunctions of LTL formulas of the form $\mathbf{G}(\psi \Rightarrow \mathbf{X}\psi')$, where ψ and ψ' are propositions which do not contain any temporal operators. The requirement VTS-STS-4411-PLS-130 introduced above, for example, can be represented as

$$\begin{aligned}&\mathbf{G}\big((\text{FSB\_OFF} \wedge C = 1) \Rightarrow \mathbf{X}(\text{FSB\_ON} \wedge f = 1)\big) \wedge \\ &\mathbf{G}\big((\text{FSB\_NORMAL.CHOICE} \wedge C = 1) \Rightarrow \\ &\qquad\qquad \mathbf{X}(\text{FSB\_ON} \wedge f = 1)\big) \wedge \\ &\mathbf{G}\big((\text{FSB\_ON} \wedge C = 0) \Rightarrow \mathbf{X}(\text{FSB\_OFF} \wedge f = 0)\end{aligned}$$

This representation is automatically derived from the satisfy-relations contained in the model. More complex requirements need to be specified by user-defined LTL formulas.

Simple test cases are derived from model elements, using the following model coverage strategies: (1) simple state coverage ($\mathbf{F}$state_identifier), (2) transition coverage ($\mathbf{F}$(state_identifier $\wedge$ trigger_condition)), (3) MC/DC coverage, (4) hierarchic transition coverage, and (5) coverage of different combinations of pairs of states in concurrent communicating state machines. The details of the strategies 3 – 5 are described in [10], [18]. Simple test cases are related to requirements using the satisfy-relation: a test case covering a model element linked to a given requirement contributes to this requirement's verification. More formally, test cases leading to computations fulfilling $\mathbf{F}\psi$ contribute to the test of a requirement $\mathbf{G}(\psi \Rightarrow \mathbf{X}\psi')$, if the test execution is sufficiently long to check the fulfillment of ψ'.

Calculating concrete test data for a given test case φ is realized by solving the *bounded model checking (BMC) instance*

$$\text{BMC} \equiv I(s_0) \wedge \bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i) \wedge G(s_0, \ldots, s_k) \qquad (1)$$

using an SMT solver. Here, $I(s_0)$ specifies the initial state, and Φ is the transition relation. $G(s_0, \ldots, s_k)$ is a first-order representation of the LTL formula φ, interpreted on the finite state sequence $s_0, \ldots, s_k$ as described in [19]. From a solution of (1), the sequence of input vectors $s_i(\vec{x}), i = 0, \ldots, k$, together with their input time stamps $s_i(t), i = 0, \ldots, k$ is extracted. Model parameters are encoded in the transition relation as special inputs p satisfying $s_i(p) = s_{i-1}(p), i = 1, \ldots, k$. This implies that they can be set only once as the very first input and remain constant throughout the remaining execution.

B. MBT Benefits in Practice

In our discussions with verification and validation experts from Airbus, the following characteristics of the existing MBT technology were considered as the main advantages, when compared to conventional programming of test procedures.

a) Automated requirements tracing: A major objective of avionic systems verification is to provide evidence that each requirement has been comprehensively reviewed, analyzed, and tested. Typically, a *traceability matrix* is created which associates each requirement to the related verification activities performed, documenting the outcome of each activity. For MBT activities, the traceability matrix can be constructed in an automated way as follows. (1) We start with a requirement's representation as an LTL formula φ. (2) Collect all test cases ψ_i associated with φ by the relation $\psi_i \Rightarrow \varphi$, so that every witness generated for a test case represented by LTL formula ψ_i is also a witness for requirement φ. (3) Associate each of these ψ_i, together with their verdicts with requirement φ in the traceability matrix.

In Fig. 4, a section of the traceability matrix is shown for the FSB control function. Consider again the requirement VTS-STS-4411-PLS-130 discussed above in Section III-B2. Since the criticality level of this requirement is only C, it suffices to associate transition coverage test cases with this requirement. Test case TC-PLS-TR-174, for example, has witnesses covering the model transition from the choice pseudo state to state FSB_ON, guarded by $[C == 1]$ in submachine FSB_NORMAL shown in Fig. 2. Test case TC-PLS-TR-0177 has witnesses covering the transition from FSB_ON to FSB_OFF guarded by $[C == 0]$.

b) Automated identification of implicitly covered test cases: When programming test procedures in a manual way, one always has a fixed set of test objectives in mind that should be implemented by this procedure. In many cases, however, the resulting test execution also covers other useful test cases which could also be marked as passed or failed, if the test procedure designer would be aware of this opportunity

[3]See https://www.ibm.com/support/knowledgecenter/SSB2MU_8.2.1/com.btc.tcatg.user.doc/topics/com.btc.tcatg.user.doc.html and ftp://public.dhe.ibm.com/software/uk/itsolutions/innovate2013/12.00_Udo_Brockmeyer-003.pdf

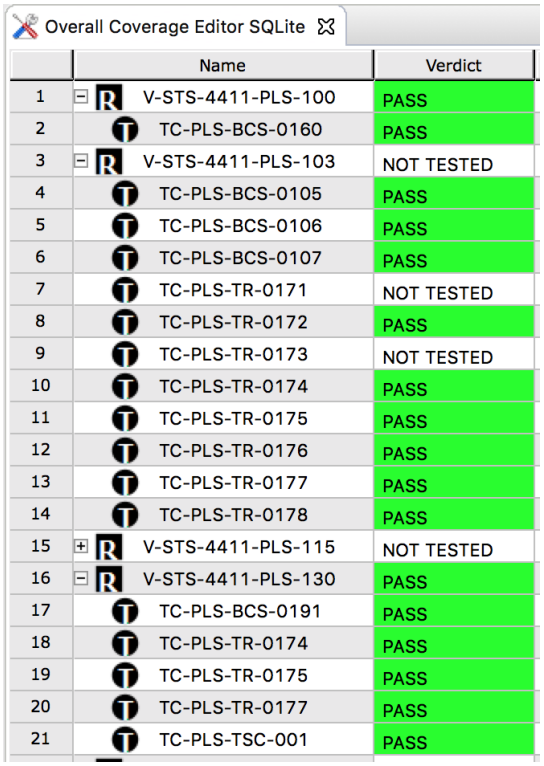

Fig. 4. Traceabilty matrix from requirements to test cases.

and insert associated checks for expected results into the procedure.

When applying MBT, test procedures can be analyzed with respect to the model computations they will trigger. This is performed in RT-Tester by a model interpreter which uses the test data calculated by the test generator and the SMT solver to simulate the model with these data. The resulting computation is then checked against *all* test cases identified in the model, so that also implicitly covered test cases can be identified, though they were not among the test objectives used to generate the procedure. Moreover, the test oracles generated from the model always monitor *all* SUT outputs and compare them to the behavior expected from the concurrent state machine model. As a consequence, it is also checked whether the implicitly covered test cases lead to SUT reactions that are consistent with the expected results.

c) 150% models: The test models typically used for testing avionic systems are so-called *150% models*. This term is used to indicate that the model comprises different, mutually exclusive behaviors depending on the system parameterization. It a considerable advantage to capture these parameterization effects in a single model and let the test generator and SMT solver automatically calculate the parameter settings required to cover a specific behavior during a test.

Consider, for example, the FSB control logic shown in the state machine of Fig. 2. When generating a test procedure covering the excessive altitude reaction modeled by the transition with trigger when$(\mathrm{EA} \wedge p_{ea})$, the test generator identifies the dependency on parameter p_{ea} and sets the appropriate parameter value (`true`) needed for this test.

d) Advantages during regression testing: Some of the main advantages of MBT become effective during regression testing. When test procedures have been conventionally programmed, all of them need to be analyzed, whether they are affected by the changes performed in the SUT and its requirements. With manually programmed procedures, this task is very time consuming, because both the test data (including parameterizations) and the software structure of each procedure may need to be adapted. When using MBT, the changed requirements directly identify the model portions to be modified, using the satisfy-relationships described above. After that, the complete test suite is generated again, and the new procedures are executed against the new SUT revision.

RT-Tester supports regression testing with a specific feature for configuring the test procedure generations: the generation directives of each procedure are automatically analyzed to check whether the symbols referenced in the procedure configuration (e.g. a target state or a transition to be covered, or the symbols used in an LTL formula) are still available in the modified model. If this is not the case, the procedure is marked as outdated for the new SUT version, and the procedure's configuration data needs to be modified. Otherwise the procedure just needs to be generated again, using the same test objectives. Depending on the model changes performed for the new SUT release, this new generation may use different test data to meet the objectives and apply modified test oracles to check the SUT behavior.

e) Integrated approach to MiL, SiL, and HiL testing: Since test models operate on interface abstractions, tests can be executed in different modes. In *model-in-the-loop (MiL)* test mode, a model simulation is executed during each test, as described above. This mode is useful during the model verification phase: tests are executed, and it is checked whether their associated computations reflect the expected SUT behavior. In *software-in-the-loop (SiL)* mode, the tests are executed against the SUT software. The model interface data referenced in the test procedures are transformed to software interfaces using simple adapters. In *hardware-in-the-loop* test mode, the SUT consists of the integrated HW/SW system, and the model interface data are transformed by more complex interface modules using the drivers needed to access the SUT hardware interfaces.

V. Open Issues for Model-based Testing of Avionic Systems

Already today, the characteristics of MBT described above offer significant advantages over conventional test procedure programming. There is, however, considerable potential to optimize the MBT approach further with respect to automated test case selection and test procedure generation. In this section, we describe two challenges of specific importance and indicate how these may be overcome in a satisfactory way; further challenges and promising solution possibilities are described in [20].

A. Two Challenges

1) Fine-grained Test Cases versus Test Scenarios: Discussions with verification experts from Airbus have shown that the test cases derived from model coverage criteria (e.g., [hierarchic] transition coverage, MC/DC coverage) are usually too fine-grained to result in expressive test procedures reflecting complete end-to-end functionality. In particular, demonstration tests for the certification authorities should rather cover a larger number of SUT interactions, illustrating comprehensive use cases of the SUT functionality. Moreover, the witnesses generated by the SMT solver for a model coverage objective are often robustness test cases, because they may apply unusual input combinations that would occur only as a consequence of failures in the SUT's operational environment. These test are of course also necessary, but usually unsuited for demonstrating normal behavior use cases.

These considerations have led to the introduction of *scenario-based testing*, where experts define important use cases to be transformed into test procedures to be run against the SUT. The fine-grained model coverage test cases are then used to complement the test scenarios by robustness tests and other test cases applying unusual input traces which may be unintuitive, but will usually increase the test strength of a suite in a considerable way, because faulty SUT behavior will often be revealed by these less intuitive test executions.

The present version of RT-Tester supports the definition of test scenarios by means of additional state machines restricting the solutions to be calculated by the SMT solver to intuitively understandable normal behavior executions. Alternatively, test scenarios can be specified using LTL formulas. Other tools (e.g., Rational Test Conductor) offer the possibility to specify test scenarios by means of SysML sequence diagrams describing the desired interface interactions between operational environment and SUT during the scenario execution.

From our experience, neither state machines, LTL formulas, nor sequence diagrams are sufficiently effective to specify test scenarios. This seems to have been observed by others as well, so that several suggestions for test scenario languages have been made; see, for example, [21], [22]. We will illustrate our own approach to test scenario specification in Section V-B.

2) Configuration Testing: The current capabilities of RT-Tester suffice to calculate both configuration parameters and timed sequences of input vectors needed to cover a given test objective. It is also desirable, however, to be able to determine whether the different parameter assignments applied in a test suite represent an acceptable coverage of parameter value combinations. The number of different parameterizations is usually so big in avionic systems, that an exhaustive enumeration of all value combinations would be infeasible. Additionally, it needs to be investigated whether a given test scenario should be executed with more than one set of configuration parameter assignments. A solution for this problem is described in Section V-C.

B. Test Scenario Specification and Scenario Generation

The idea of presenting test scenarios in the form described here is inspired by constraint programming, see, e.g., [23]: instead of explicitly specifying the inputs and expected outputs to be exercised is a test scenario, we describe a sequence of constraints to be fulfilled by the test. The test generator uses the SMT solver to construct a model of the BMC instance

$$\text{scenario} \equiv I(s_0) \wedge \bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i) \wedge \text{SC}(s_0, \ldots, s_k), \quad (2)$$

where, as before, Φ represents the transition relation and $\text{SC}(s_0, \ldots, s_k)$ is a first-order representation of the scenario specification. Models are finite sequences of model states, each pair of states connected by the transition relation, such that the sequence also fulfills the scenario constraints encoded in SC.

The formal semantics of these test scenarios is defined by transforming them into LTL formulas that are interpreted in the finite trace semantics introduced in [19]. The details are described in [20]; here we only give an intuitive explanation, using the test scenario shown in Fig. 5.

There, different *time lines* specify constraints changing at certain uniquely identified points during the test scenario execution. In the simplest case shown in the examples presented here, these points are specified by execution time stamps. In the more general case handled in [20], the points may be specified my arbitrary predicates on the model state that are unique identifiers of the execution, such as, for example, cycle counter values or other conditions occurring only once during test execution. The formal semantics of the section between two points of one time line is close to that of the TCTL Globally operator $\mathbf{G}^{[t_1,t_2]}\varphi$ which asserts that constraint φ holds in all computation states traversed between time points t_1 and t_2, see [16, pp. 698]. When changing between two sections of a time line, a finite number of zero-time transitions are allowed to ensure that the new constraint holds after these transitions have been completed. Several time lines in the same scenario specification are interpreted by logical conjunction. The FRAME constraint specifies which input symbols are allowed to change in order to ensure that the other constraints applicable at the corresponding time line sections are fulfilled. In Fig. 5, it is specified that during the first 10 seconds of the execution, only the cockpit switch input C is allowed to change. After that, inputs C, L, S_1, S_2 are allowed to change until the end of the test.

During the first 10 seconds of the scenario execution, the cockpit switch C remains in position 0. Since C is the only input symbol in the applicable frame, this automatically implies that FSB signs shall remain in state off during this period. From test seconds 10 to 25, the cockpit switch is set into the AUTO position, and it is required that the AUTO condition shall not hold, so that the FSB signs remain off, because the EA interface is not in the current frame, so the signs cannot be switched on due to the excessive altitude condition. At second 25, the auto condition shall become true, and we expect the FSB signs to be activated, since the EM

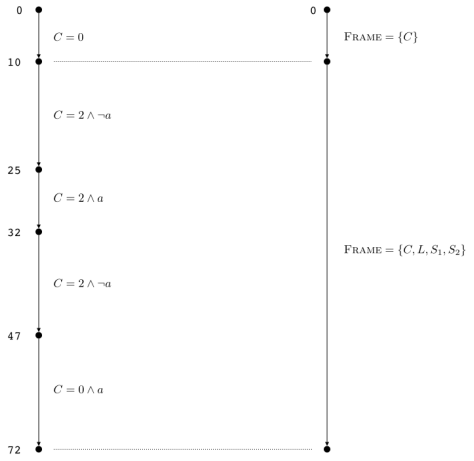

Fig. 5. Test scenario specification 1 for testing the FSB AUTO function.

interface is not in the frame, so the emergency mode cannot be activated. This illustrates the advantages of the constraint programming style used in scenario specifications, in contrast to imperative test procedure programming: instead of explicitly setting inputs in a way leading to a becoming `true`, we just require that a should become `true` in zero time at point $t = 25$ and leave the task of setting the appropriate inputs and parameterizations to the SMT solver. At second 32, the auto condition shall become false again, and we expect the FSB signs to be switched off again. At second 47, the AUTO condition shall become true again, but we expect the FSB signs to remain off, since the cockpit switch C is in the off-position.

C. Automated Configuration Testing

To explain the method for automated generation of parameter-dependent test procedures, consider the test scenario shown in Fig. 6. This is a more complex variant of the scenario shown in Fig. 5, with the following changes. (1) The landing gears input L is no longer contained in any frame, so it has to remain 0 throughout the scenario execution. (2) Excessive altitude may occur, and it is checked between time 32 and 47 that the FSB signs remain active when the AUTO condition is no longer true, but excessive altitude is reached.

The following algorithm describes how several test procedures are generated from this scenario, each time with a different parameterization.

1) **Input.** Transition relation Φ and scenario representation as LTL formula ψ.
2) **Output.** A set of test procedures $P = (\overline{x}, \vec{p})$, where $\overline{x}$ is a timed trace of input vectors for stimulating the SUT and $\vec{p}$ is a parameterization vector, such that each P corresponds to a model of formula (2) with $SC(s_0, \ldots, s_k)$ a valid first-order representation of ψ.
3) Extract all symbols from ψ representing internal model variables or outputs; this results in a set S.
   For our example scenario, this results in the symbol set $S = \{a\}$.
4) Perform a writer/reader analysis across all state machines to determine which parameters influence the symbols in S. This results in a set Q of parameter symbols.
   In our example, the actual value of a is influenced by parameter p_a, as follows from a writer/reader analysis of the state machine in Fig. 1. The other state machines do not write to a, so there are no further dependencies, and $Q = \{p_a\}$.
5) Perform a writer/reader analysis to determine which transitions depending on symbols from S finally affect output values. Insert these transitions into set T. Extract parameters from both triggers and actions of these transitions and add them to Q.
   In the example, a backward analysis to this end starts in the state machine from Fig. 3, and we see that (apart from certain inputs) the only output F is affected by the value of internal model variable f. This is traced back to the state machine in Fig. 2, where we identify two transitions t_1, t_2 depending on the trigger events when$(C == 2 \wedge a)$ and when$(C == 2 \wedge \neg a)$, respectively. The transitions do not contain any parameters to add into Q.
6) For each transition $t \in T$, check which other transitions u enable or prevent that t is triggered. Extract additional parameters occurring in trigger conditions or actions of each u and add them to Q.
   In the example, the transitions t_1 and t_2 are enabled or prevented by transitions depending on the parameter p_{ea}. This results in $Q = \{p_a, p_{ea}\}$.
7) For each value assignment for the parameters in Q, check whether the scenario BMC instance from formula (2) can be solved. Each solution gives rise to a new test procedure $P = (\overline{x}, \vec{p})$, running the scenario with a new set of parameter assignments.
   In our example, formula (2) can be solved for each combination of $p_a = 1, 2$ and $p_{ea} = 0, 1$. For $p_a = 3$ there is no solution, because a will never be set to `true`, since this would depend on input value L which cannot be changed due to the frame definition in this scenario.

As can be seen from the algorithm specification above, the parameter and value identification technique takes into account which parameters are really logically related, and which value combinations for these parameters can lead to an instance of the scenario. This contrasts to "naive" n-way testing [24], where parameter combinations are selected without further analysis of their logical dependencies.

Further optimization heuristics can be applied if the parameter set Q gives rise to too many value combinations. In this situation, n-way testing can prove to be very valuable for systematically reducing the number of parameter combinations to be tested in separate procedures, because it ensures that at least all subsets of Q containing n parameters are exhaustively tested.

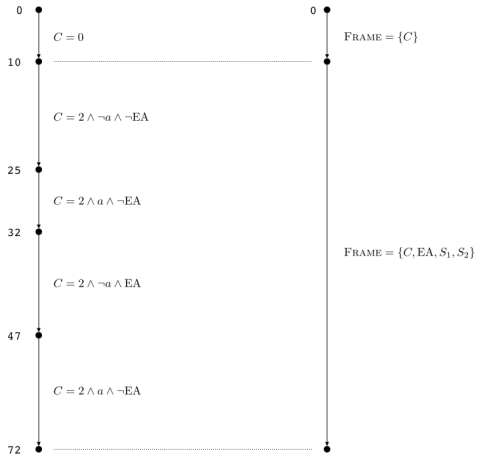

Fig. 6. Test scenario specification 2 for testing the FSB AUTO function.

VI. Conclusion

We have described an approach to model-based testing, as it is applied today for HW/SW integration testing of avionic controllers in Airbus aircrafts. The main benefits of the MBT approach have been described and compared to conventional automated testing based on manually programmed test procedures. Two challenges for future improvements regarding test scenario specification and automated configuration testing have been described, and promising solutions for these challenges have been sketched.

Acknowledgment

I would like to sincerely thank the organizers of the ETS 2018 conference for the invitation to present this paper.

References

[1] RTCA,SC-167, *Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B*, RTCA, 1992.

[2] RTCA SC-205/EUROCAE WG-71, "Software Considerations in Airborne Systems and Equipment Certification," RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, Tech. Rep. RTCA/DO-178C, December 2011.

[3] J. Brauer, M. Dahlweid, and J. Peleska, "Tool-supported structural coverage analysis for DO-178C compliant software," in *Proceedings of the SAE Aerotech conference, Seattle, September 2015*. SAE Technical Paper, 2015, doi:10.4271/2015-01-2558.

[4] J. Brauer, M. Dahlweid, T. Pankrath, and J. Peleska, "Source-code-to-object-code traceability analysis for avionics software: Don't trust your compiler," in *Computer Safety, Reliability, and Security - 34th International Conference, SAFECOMP 2015 Delft, The Netherlands, September 23-25, 2015. Proceedings*, ser. Lecture Notes in Computer Science, F. Koornneef and C. van Gulijk, Eds., vol. 9337. Springer, 2015, pp. 427–440. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24255-2_31

[5] RTCA SC-205/EUROCAE WG-71, "Model-Based Development and Verification Supplement to DO-178C and DO-278A," RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, Tech. Rep. RTCA/DO-331, December 2011.

[6] ——, "Formal Methods Supplement to DO-178C and DO-278A," RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, Tech. Rep. RTCA/DO-333, December 2011.

[7] ——, "Software Tool Qualification Considerations," RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, Tech. Rep. RTCA/DO-330, December 2011.

[8] J. Brauer, J. Peleska, and U. Schulze, "Efficient and trustworthy tool qualification for model-based testing tools," in *Testing Software and Systems. Proceedings of the 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 2012*, ser. Lecture Notes in Computer Science, B. Nielsen and C. Weise, Eds., no. 7641. Heidelberg Dordrecht London New York: Springer, 2012, pp. 8–23.

[9] Object Management Group, "OMG Systems Modeling Language (OMG SysML), Version 1.4," Object Management Group, Tech. Rep., 2015, http://www.omg.org/spec/SysML/1.4.

[10] J. Peleska, "Industrial-strength model-based testing - state of the art and current challenges," in Proceedings Eighth Workshop on *Model-Based Testing,* Rome, Italy, 17th March 2013, ser. Electronic Proceedings in Theoretical Computer Science, A. K. Petrenko and H. Schlingloff, Eds., vol. 111. Open Publishing Association, 2013, pp. 3–28.

[11] J. Peleska, A. Honisch, F. Lapschies, H. Löding, H. Schmid, P. Smuda, E. Vorobev, and C. Zahlten, "A real-world benchmark model for testing concurrent real-time systems in the automotive domain," in *Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011*, ser. LNCS, B. Wolff and F. Zaidi, Eds., vol. 7019, IFIP WG 6.1. Heidelberg Dordrecht London New York: Springer, November 2011, pp. 146–161.

[12] J. Peleska, E. Vorobev, and F. Lapschies, "Automated test case generation with SMT-solving and abstract interpretation," in *Nasa Formal Methods, Third International Symposium, NFM 2011*, ser. LNCS, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Pasadena, CA, USA: Springer, April 2011, pp. 298–312.

[13] J. Peleska and W. Huang, "Industrial-strength model-based testing of safety-critical systems," in *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, Eds., vol. 9995, 2016, pp. 3–22. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-48989-6_1

[14] L. D. Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R. G. d. Vries, "Formal Test Automation: The Conference Protocol with TGV/TORX," in *Testing of Communicating Systems*, ser. IFIP Advances in Information and Communication Technology. Springer, Boston, MA, 2000, pp. 221–228. [Online]. Available: https://link.springer.com/chapter/10.1007/978-0-387-35516-0_14

[15] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 299–306. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086283

[16] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[17] W.-l. Huang, J. Peleska, and U. Schulze, "Test automation support," COMPASS Comprehensive Modelling for Advanced Systems of Systems, Tech. Rep. D34.1, 2013, available under `http://www.compass-research.eu/deliverables.html`.

[18] *RT-Tester Model-Based Test Case and Test Data Generator RTT-MBT*, Verified Systems International GmbH, Bremen, 2012.

[19] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Logical Methods in Computer Science*, vol. 2, no. 5, Nov. 2006, arXiv: cs/0611029. [Online]. Available: http://arxiv.org/abs/cs/0611029

[20] J. Peleska, J. Brauer, and W. Huang, "Model-based testing of avionic systems," in *Proceedings of the ISoLA 2018.* Springer, 2018, to appear.

[21] R. Tommy, N. Prasannakumaran, K. V. Sumithra, and S. Kesav, "Test scenario modeling: Modeling test scenarios diagrammatically using specification based testing techniques," in *Proceedings of the 2015 Third International Conference on Computer, Communication, Control and Information Technology (C3IT)*, Feb. 2015, pp. 1–7.

[22] J. Ryser and M. Glinz, "Using Dependency Charts to Improve Scenario-Based Testing -Management of Inter-Scenario Relationships: Depicting and Managing Dependencies . . ." in *Proceedings of the 17th International Conference on Testing Computer Software (TCS2000), Washington D.C*, 2000.

[23] K. Apt, *Principles of Constraint Programming*. Cambridge: Cambridge University Press, 2009.

[24] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC Press, 2013.