Towards an Integrated Methodology for the Development and Testing of Complex Systems - with Example

Philipp Helle and Wladimir Schamai
Airbus Group Innovations
Hamburg, Germany
Email: {philipp.helle,wladimir.schamai}@eads.net

***Abstract*—This article reports on a framework for the development and testing of complex systems. The framework provides a meta-model for the description of systems at different levels of abstraction, which is used as a basis for the combination of model-based testing (MBT) techniques for automated test case generation with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to reduce the total development and testing effort for complex systems. This is accomplished by enabling a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle. The framework is illustrated using an example from the aircraft systems domain: the door locking system.**

Keywords*—Model-based Systems Engineering, Model-based Testing, Monitor-based Testing, SysML.

I. Introduction

This article is a revised and extended version of the article [1], which was originally presented at the The Fifth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2013).

The ever-increasing complexity of products has a strong impact on time to market, cost and quality. Products are becoming increasingly complex due to rapid technological innovations, especially with the increase in electronics and software even inside traditionally mechanical products. This is especially true for complex, high value-added systems in the aerospace and automotive domain - the methodology was developed and is therefore embedded in an aeronautic context but generally is independent of a specific domain - that are characterized by a heterogeneous combination of mechanical and electronic components. System development and integration with sufficient maturity at entry into service is a competitive challenge in the aerospace sector. Major achievements can be realized through efficient system testing methods.

”Testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: observable differences between the behaviours of implementation and what is expected on the basis of the specification”[2].

The typical testing process is a human-intensive activity and as such it is usually unproductive and often inadequately done. It requires human test engineers to manually write test cases. A test case contains a series of test inputs and expected results. Nowadays, the test execution especially on lower levels of testing is largely automated. Nevertheless, this process is cumbersome and costly. Therefore, testing is one of the weakest points of current development practices. According to the study in [3] 50% of embedded systems development projects are months behind schedule and only 44% of designs meet 20% of functionality and performance expectations. This happens despite the fact that approximately 50% of total development effort is spent on testing [3], [4]. This shows the importance and desirability of reducing test effort by advances in the testing methodologies.

Testing needs to be applied as early as possible in the lifecycle to keep the relative cost of repair for fixing a discovered problem to a minimum. This means that testing should be integrated into the lifecycle model so that each phase in the development contributes to the verification of the product as Figure 1 shows. Laycock claims that ”the effort needed to produce test cases during each phase will be less than the effort needed to produce one huge set of test cases of equal effectiveness on a separate lifecycle phase just for testing”[5].

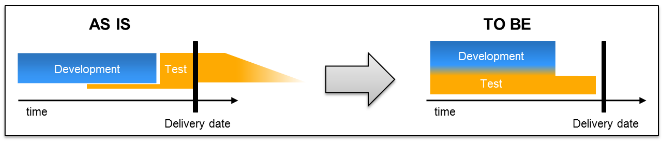

Fig. 1: Envisaged process change

This paper reports on a framework to further automate the system testing process. It is a continuation of the work earlier reported in [6]. The framework provides a meta-model for the description of systems on different layers of abstraction and combines model-based testing (MBT) techniques for automated test case generation based on a whitebox SysML model of the system with executable requirement monitors that continuously observe the status of the System under Test (SuT) during test execution. The overall goal is to achieve a high degree of automation and reuse of engineering artefacts throughout the systems engineering lifecycle.

Paper structure: First, we present background information on SysML, MBT and monitor-based testing (Section II) before we will explain the methodology in detail (Section III). Next, the methodology will be illustrated using an example from the aeronautic domain (Section IV). Finally, we propose a number of ideas for future research (Section V) and close with a summary of the current status (Section VI).

II. Background

This section provides background information on SysML, Model-based testing, Monitor-based testing and related work.

A. SysML

The Unified Modeling Language (UML) [7] is a standardized general-purpose modelling language in the field of software engineering and the Systems Modeling Language (SysML) [8] is an adaptation of the UML aimed at systems engineering applications. Both are open standards, managed and created by the Object Management Group (OMG), a consortium focused on modelling and model-based standards.

SysML is not a methodology, i.e., it does not define what steps need to be performed in what order and which diagrams should be used for which step. Estefan [9] provides an overview of existing methodologies used in industry, some of which use UML-based languages. SysML is a graphical modelling language, i.e., diagrams are used to create and view model data. However, the graphical representation is decoupled from the actual model data. The model data and its graphical representation are typically stored in different files in UML/SysML tools.

Neither UML nor SysML define complete model execution semantics in their core specification. This is different from modelling and simulation languages, such as Modelica [10], which specify the syntax (textual notation) as well as the execution semantics. However, work is underway to resolve that [11], [12], [13]. In the mean time, SysML tool suppliers often provide their own execution semantics [14], so it is possible to include action code into models, generate code from the models and then execute them.

B. Model-based testing

The term MBT is widely used today with slightly different meanings. Surveys on different MBT approaches can be found in [2], [15], [16]. One of them is that ”Model-based testing (MBT) relates to a process of test generation from an SuT model by application of a number of sophisticated methods”[17].

Model-based testing is a variant of testing that relies on explicit behaviour models that encode the intended behaviour and expected failure states of a system and possibly the behaviour of its environment. The use of explicit models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the creativity and expertise of single engineers. The idea is that the existence of an artefact that explicitly encodes the intended behaviour can help mitigate the implications of these problems [2].

Intensive research on MBT and analysis has been conducted in recent years, and the feasibility of the approach has been successfully demonstrated, e.g., in [18], [17]. Yet, Boberg [19] shows that most studies apply model-based testing at the component level, or to a limited part of the system while only few studies focus on the application of the technique at the system or even aircraft level. The main difference being that the goal of modelling at system level aims at generating a specification whereas modelling at component level aims at generating code that runs on target. Giese [20] explains that this slow adoption is not only due to scalability reasons but he also claims that ”to benefit from formal verification and early simulation, a model must be precise and detailed with respect to all aspects that are the subject of verifiation. This can usually be carried out in the detailed design phase at the earliest”[20].

A major distinction between the different available MBT approaches can be made by looking at the source of the generated test cases [20]. Some approaches rely on separate explicit test models that are disjunct from the system or specification model, as depicted by Figure 2 while other approaches do not make that distinction and generate test cases from the defined system behaviour as shown by Figure 3.

The usage of explicit test models reflects the different objective (validation vs. solution) and point of view (tester vs. implementer) in creating a test model rather than a specification model [21]. A test model is a model representing all possible stimulations of input of the system interacting in various usage contexts and normally also includes verification points stating what is a correct response from the system to an input and what not. It thereby follows a tester’s view who also has to think of how to combine the possible input stimuli of a system to achieve a high confidence in its correctness.

The main benefit of this approach is the degree of independence it naturally entails between the generated test cases and the system. The generated test cases can thus be used directly to test any form of the SuT, either a model or the implementation. Additionally, as the test model is not a part of the design it can be optimised for validation and verification purposes thereby increasing the chance to uncover defects that are outside the focus of the design artefacts [20]. A drawback of the approach is that there are two models that have to be kept consistent with the requirements at all time, which requires further effort.

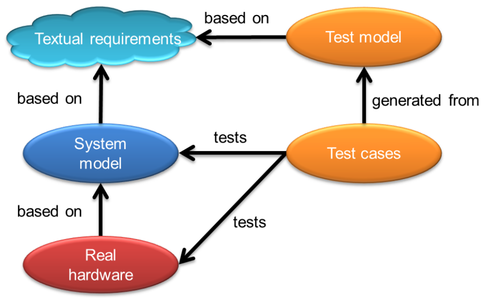

Fig. 2: Model-based testing using explicit test models

One example for an approach that does not rely on explicit models is the work from Lettrari [22] that is the basis for the commercially available IBM Rational Rhapsody Automatic Test Generator (ATG) tool. Test cases are generated from a behaviour model of the SuT using model coverage as test selection criteria. Automated test case generation uses constraint based symbolic execution of the model and search algorithms.

The main benefit is that the approach does not require the creation and maintenance of a separate test model. On the other hand, since the test case generation is not guided by a test engineer it cannot distinguish between ”good” and ”bad” test cases. The only goal for the generator is to achieve a high degree of model and/or code coverage by generating stimuli that force the executable system model to visit all states and transitions and call all functions of the system's components. Furthermore, there is no independence between the generated test cases and the system model. This means that the test cases cannot be used to test the model they were generated from if the test success criteria is that the observed behaviour and the test case behaviour are the same.

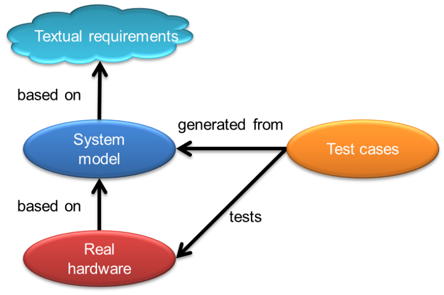

Fig. 3: Model-based testing using design/specification models

C. Monitor-based testing

The idea for formalizing a natural language requirement statement into a requirement monitor is similar to the monitor concept used in runtime verification [23], [24]. A more formal definition states, that "Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property"[23].

Runtime verification itself deals with the detection of violations of correctness properties. Thus, whenever a violation is observed, it typically does not influence or change the programs execution, say for trying to repair the observed violation. Checking whether an execution meets a correctness property is typically performed using a monitor. In its simplest form, a monitor decides whether the current execution satises a given correctness property by outputting either yes/true or no/false [23].

D. Related Work

In [25], Artho et. al. propose a method for combining test case generation and runtime verification for software systems. In their framework they combine automated test case generation, which is based on a systematic exploration of the input domain of the tested software system using a model checker that is extended with symbolic execution capabilities with runtime verification techniques, that monitor execution traces and verify them against properties expressed in a temporal logic notation. They include further capabilities for the analysis of concurrency errors, such as deadlocks and data races. The paper also provides a description of the application of the method using a NASA rover controller.

Our work differs from the work by Artho et. al. in some major points. Firstly, the test oracles are written as temporal logic formulas whereas we use SysML for both the modelling of the system as well as the requirement monitors. Secondly, the test scenarios are generated based on a definition of all possible inputs using a model checker, whereas we generate the test scenarios from a whitebox model of the system under test.

Drusinsky calls the usage of statecharts for the automated verification of models *execution-based model checking* and compares it to classical model checking, i.e., static analysis, as follows: "[execution-based model checking] seldom yields 100% test coverage, whereas classical model checking consists of a mathematical proof that does yield 100% coverage. The truth, however, is that both classes of techniques require compromises. Execution-based model checking compromises in the achieved test coverage; classical compromises in the size and type of programs that can be verified, and in the kinds of assertions that can be verified to begin with"[26]. In our work, we follow a concept that is similar to what Drusinky calls *execution-based model checking* but embed the idea in an overall framework for the development and testing of systems.

III. Methodology for Development and Testing of Complex Aircraft Systems

This section provides a description of our methodology in terms of the overall concept, the underlying metamodel and the envisaged process.

A. Concept

Our methodology combines monitor- and model-based testing to test the system model and the resulting system. Our aim is to achieve a high degree of reuse of artefacts from early development stages at later development stages and a high degree of automation throughout the process. Since we consider multiple levels of abstraction in our metamodel it is necessary to provide means, which can verify a model at any abstraction level or the final product without the need for redeveloping the verification artefacts for each verification stage. To this end, we use executable requirement monitors, which can be built as soon as the first requirements are defined. The formalized reuquirement monintors can be reused and adapted easily for verifying the models or the product. Also, these monitors can be reused for testing different variants and/or design alternatives.

Figure 4 provides an overview of the main artefacts involved and their relations.

A requirement monitor is an exectuable model representing one requirement that, at any point in time, indicates the requirement violation status. The status should be enumerated with at least the following values [27]:

- Not evaluated (default value), to indicate that the requirement has not been evaluated yet. Typically, this means that a necessary precondition has not been met yet or that the monitor is currently evaluating but could not make a verdict yet.
- Not violated, to indicate that no violation has happened and implying that the requirement has been evaluated.
- Violated, to indicate a violation of the requirement and implying that the requirement has been evaluated.

This enumeration is referred to as ”three-valued semantics” in [23] with the literals ”inconclusive”, ”false” and ”true” respectively.

The monitor status can be obtained from a monitor at any point in time and can change between not evaluated, not violated and violated in any possible way. Following this approach, the status of the individual reuqirement monitors that are instantiated during one test can be used in aggregation to derive the test verdict. Removing the test verdict from the test cases will enable the reuse of test cases, that we now call *test scenarios*, for the verification against several requirements.

The task of converting the natural language statement into a formal language will require a correct interpretation of the requirement statement and the ability to translate the meaning into a model that expresses exactly the same. The general systematic way for deriving a monitor from natural language requirement is as follows:

1) Read the requirement statement
2) Identify properties that can be quantified either by explicit numbers or by logical conditions
3) Identify pre-conditions (if any), which must be satisfied before the requirement can be evaluated
4) Express when the requirement is violated and when not

Neither a particular design of the system nor scenarios are needed for formalizing a requirement. The resulting monitor can be used for the verification of any design alternative of the system using any scenario. Generally, the task of formalizing a requirement into a requirement monitors can be accomplished in many different ways using different formalisms. We decided to use SysML for the task because using the same notation for design and testing artefacts enables integrated development and testing without the need for additional tools or data converters.

We drive the tests using scenarios that we generate from the system models using MBT technology. Since we derive the test verdict from the requirement monitors independently from the system model we can use the scenarios derived from the system model to actually verify the system model as well as the final product.

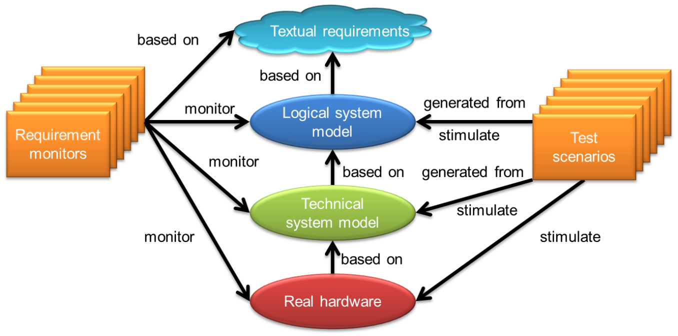

Fig. 4: Model-based testing using monitors

B. Meta-model

For our purpose, we extended the already established meta model for functional and systems architecture modeling [28] to allow a distinction between the functional, logical and the technical architecture of the system as depicted by Figure 5.

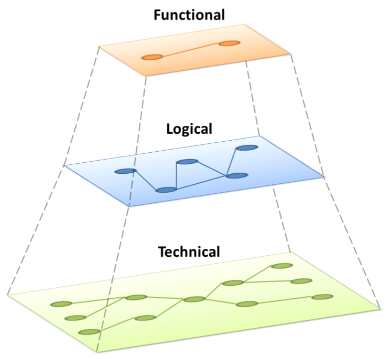

Fig. 5: Levels of abstraction

The main rationale for the distinction between these different layers is reusability. Between different aircraft programmes the functional architecture of a system is quite stable whereas the implementation can differ drastically. For a given aircraft programme the logical architecture is fixed quite early but different technical implementations might be considered and compared in trade studies. Ideally, we can now reuse the same functional architecture that is mature and proven and derive different logical and even more possible technical implementations that satisfy these functional needs.

The functional architecture, consisting of functions and data exchanges via functional dependencies is mapped to a logical system architecture, consisting of logical components that are instances of logical component classes and logical links between these components. This logical architecture can then in turn be mapped to the technical architecture of the system, which contains technical components, i.e., devices, and technical connectors, i.e., cables that connect the components. As can be seen from Figure 6, the relations between the elements in the different modelling layers allow a full traceability. This is crucial especially for maintaining the consistency of the models after changes.

While the modelling of the functional architecture in our approach is purely descriptive, the logical and the technical system architecture models are fully executable. Typically, the complexity of the models increases from the functional over the logical to the technical model. This is mainly due to two reasons: Firstly, when following this top down approach for systems modelling the level of abstraction decreases, which in turn increases the level of detail and complexity. Secondly, most aircraft systems require a certain degree of redundancy to abide by the safety constraints. A fact, which is normally not considered during the functional analysis, only partly in the logical design but has the most impact on the technical architecture.

C. Process

The overall process underlying our methodology is straight forward and consists of the following steps:

1) Formalize requirements: create a violation monitor for each requirement

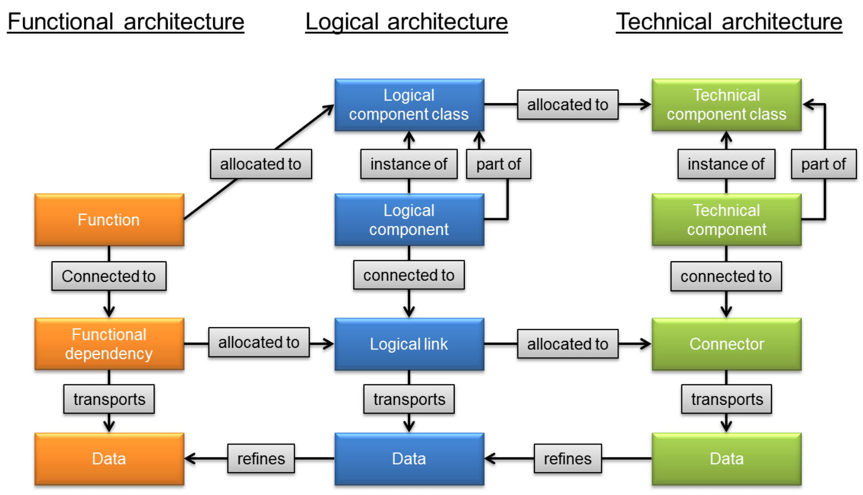

Fig. 6: Meta-model for current approach

2) Build system models
3) Generate test scenarios from system models using MBT
4) Prepare the test environment: instantiate the monitors of the requirements that can be tested using the available scenarios and connect them to the SuT (models or real hardware) appropriately
5) Execute tests: run all defined scenarios
6) Evaluate tests: aggregate the individual statuses of the requirement monitors that were active during a test to derive a test verdict
7) Analyze violations: Find root cause for violation in order to fix it

IV. Example - Door Locking System

We will illustrate steps 1 to 4 of our approach using a simple yet representative example from a passenger aircraft: the Door Locking System (DLS). The DLS controls the locks that fix the aircraft doors in a latched position and prevents unauthorised and unwanted door openings in flight and on ground in case of an existing pressure difference, so-called residual pressure, between the outside of the aircraft and the aircraft cabin. While the rationale for keeping the doors locked in flight at high altitude can be justified by common sense, incidents show that even on ground left-over residual pressure may cause harm [29]. Subsequently, the DLS is a safety-critical part of the aircraft and has to adhere to the according regulations, e.g., the DO-178 and DO-256[30]. The tool Rhapsody by IBM Rational is used for all modelling activities.

A. Initial requirements

The initial requirements of the DLS are provided in Table I. Please note, that this set of requirements serves as an example and is therefore not necessarily complete.

TABLE I: Description of initial requirements

Req.	Text
REQ-01	If the aircraft doors are unlocked, the DLS shall lock the aircraft doors when receiving the lock door command within 3 seconds.
REQ-02	The DLS shall calculate the residual pressure as the absolute difference between the cabin pressure and the outside pressure.
REQ-03	Once a door is locked, the DLS shall keep the door locked at all times, if the residual pressure exceeds 2.5 mbar.
REQ-04	If the aircraft doors are locked, the DLS shall unlock the aircraft doors when receiving the unlock door command within 3 seconds if the residual pressure is at or below 2.5 mbar.

Without any information regarding the implementation of the DLS, a requirement analyst can start to formalize the requirements into requirement monitors. The subsequent sections provide the implementation of the requirement monitors for REQ-03 and REQ-04. The other requirements can be formalized in a similar fashion.

1) REQ-03: Following the steps described in Section III-A, reading the requirement yields the following properties that can be quantified either by explicit numbers or by logical conditions:

- isDoorLocked (bool, input): locked status of the door
- residualPressure (real, input): amount of residual pressure relevant for this door's control decision
- residualPressureThreshold (real, constant 2.5 mbar): threshold for the residual pressure above which the doors need to be kept locked

Using these identified properties, Figure 7 shows the statechart of the requirement monitor that is used for checking if the system adheres to REQ-03.

2) REQ-04: As before, reading the requirement leads to the identification of the following properties of the requirement that are needed to determine if the requirement is violated:

- isDoorLocked (bool, input): locked status of the door

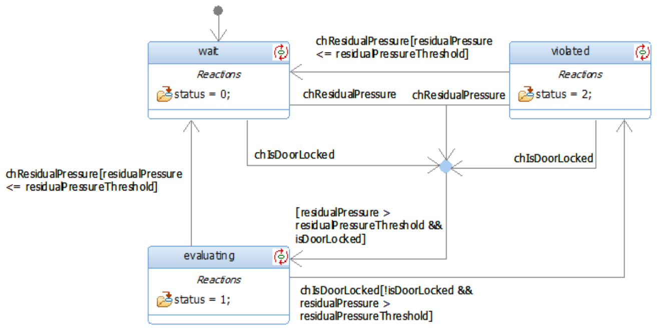
wait
Reactions
status = 0;
violated
Reactions
status = 2;
evaluating
Reactions
status = 1;
chResidualPressure[residualPressure <= residualPressureThreshold]
chResidualPressure
chResidualPressure
chIsDoorLocked
chIsDoorLocked
chResidualPressure[residualPressure <= residualPressureThreshold]
[residualPressure > residualPressureThreshold && isDoorLocked]
chIsDoorLocked[!isDoorLocked && residualPressure > residualPressureThreshold]

Fig. 7: Requirement monitor statechart for REQ-03

- evUnlockDoorCmd (event, input): unlock command has been send to the DLS
- maxWaitTime (int, constant 3000 ms): time available to open the door after a user sends an unlock command
- residualPressure (real, input): amount of residual pressure relevant for this door's control decision
- residualPressureThreshold (real, constant 2.5 mbar): threshold for the residual pressure above which a door needs to be kept locked

Figure 8 shows the statechart of the requirement monitor that is used for checking if the system adheres to REQ-04. Multiple unlock commands might be send to the DLS one after another and the requirement monitor needs to consider all of them. So, the monitor has an internal queue in which receptions of the *evUnlockDoorCmd* event are stored with a timestamp. The state *evaluating* now continuously (self-transition with timeout *pollTimeOut*) polls the queue and checks whether within 3 seconds (*maxWaitTime*) after the reception of a command by the monitor the door is unlocked or not.

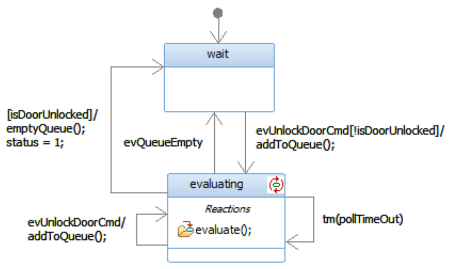

Fig. 8: Requirement monitor statechart for REQ-04

Figure 9 shows the algorithm, modelled as a flowchart, that is used to determine if requirement REQ-04 has been violated.

B. Functional model

Starting with the initial requirements the system engineer can create a functional model. The goal is to identify all functions that need to be performed by the system and the functional dependencies, i.e., data flows, between them. Table II provides a description of all the identified functions and Figure 10 shows the complete functional model. Note, that the functional model is not formal and not executable.

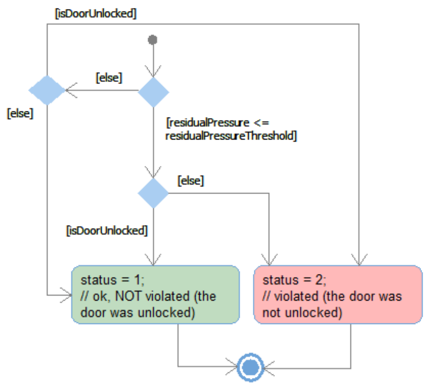
[isDoorUnlocked]
[else]
[else]
[residualPressure <= residualPressureThreshold]
[else]
[isDoorUnlocked]
status = 1;
// ok, NOT violated (the door was unlocked)
status = 2;
// violated (the door was not unlocked)

Fig. 9: Flowchart for status evaluation of REQ-04

TABLE II: Description of functions

Function	Description
Issue door commands	The issue door commands function is an interface function that allows the users of the system, i.e. the crew members, to issue commands to open or lock the aircraft doors.
Sense outside pressure	The sense outside pressure function measures the atmospheric pressure outside the aircraft.
Sense cabin pressure	The sense cabin pressure function measures the athmospheric pressure inside the aircraft cabin.
Control door locks	The control door locks function issues controls to the actuate door lock functions according to the user requests taking into account the athmospheric pressure outside and inside the aircraft.
Actuate door locks	The actuate door lock function moves the aircraft door locks between the locked and unlocked position and provides the status of the door locks to the control door locks function.

C. Logical model

The logical model is a much more sophisticated refinement of the functional model geared towards providing an actual specification while still keeping an appropriate level of abstraction.

D. Additional logical requirements

The additional complexity of the logical model compared to the purely descriptive functional model requires further design decisions and allows taking further external requirements into account. The additional requirements of the logical model that have not been taken into account in the functional model are provided by Table III. They are mostly motivated by the actual design of the aircraft that the system will be used in, while the segregation of the left and right side of the aircraft as postulated by REQ-05 is typically motivated by safety considerations and enforced by the airworthiness authorities.

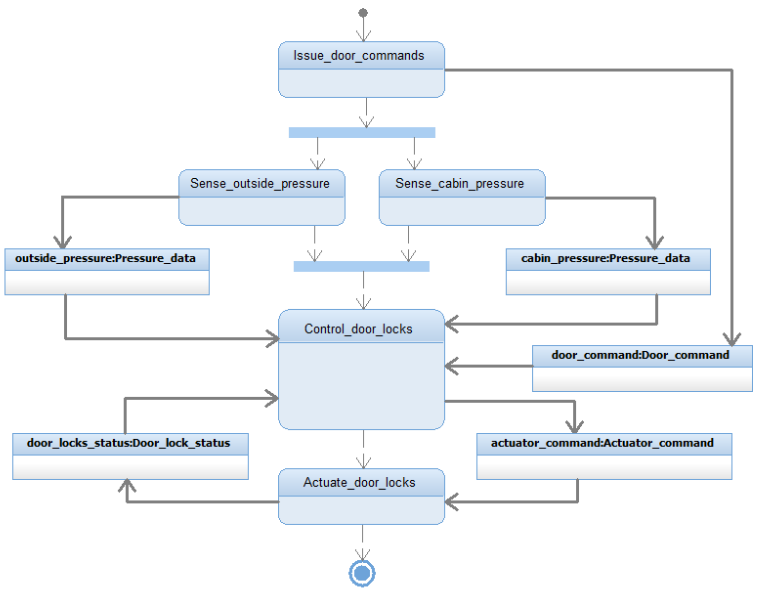
Issue_door_commands
Sense_outside_pressure
Sense_cabin_pressure
outside_pressure:Pressure_data
cabin_pressure:Pressure_data
Control_door_locks
door_command:Door_command
door_locks_status:Door_lock_status
actuator_command:Actuator_command
Actuate_door_locks

Fig. 10: Functional model

Note, that the functional model and the requirements from that level of abstraction remain valid for the logical model while the additional requirements from the logical level are not considered in the functional model.

TABLE III: Description of additional logical requirements

Req.	Text
REQ-04	The DLS shall control 2 doors on each side of the aircraft, one in the front and one in the back.
REQ-05	Doors on either side of the aircraft shall be controlled separately.
REQ-06	The cabin pressure shall be measured in the front of the cabin and at the back of the cabin.
REQ-07	For determining the residual pressure relevant for the operation of the front doors, the cabin pressure measured in the front of the aircraft shall be used primarily. If the pressure data from the front of the cabin is not available, then the data from the back of the cabin shall be used as backup.
REQ-08	For determining the residual pressure relevant for the operation of the back doors, the cabin pressure measured in the back of the aircraft shall be used primarily. If the pressure data from the back of the cabin is not available, then the data from the front of the cabin shall be used as backup.

Figure 11 shows the internal structure of the logical DLS. As can be seen the additional requirements from Table III have been taken into account, e.g., REQ-06 lead to the multiple instantiation of the *Sense cabin pressure Function* for measuring the pressure in the front as well as in the back of the cabin.

Figure 12 defines the mapping between the functions from the functional model to the logical components of the logical model. Note, that this mapping is at class level. The function *Actuate door locks*, which was responsible for moving all door locks in the aircraft from the functional model is mapped to the logical component class *Actuate door lock Function*, which moves a single door lock in the aircraft and therefore has to be instantiated multiple times in the logical DLS model, once for each door.

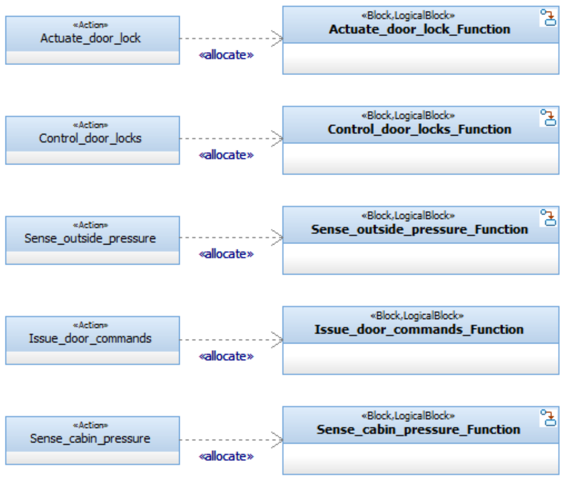

Fig. 12: Mapping between functional and logical model

Likewise, there exists a mapping of the functional dependencies from the functional model to the logical links of the logical model. This mapping can be one to one or one to many. For example, the functional dependency between the function *Sense outside pressure* and the function *Control door locks* that transports the *outside pressure* data is mapped to two logical links in the logical model: the link between *itsSense outside pressure Function* and *itsControl door locks Function Left* and the link between *itsSense outside pressure Function* and *itsControl door locks Function Right* as Figure 13 shows. Formally, this mapping is represented by the fact that the interfaces of the ports of the involved logical blocks contain the data that was previously associated with the functional link; in the example case the *outside pressure* data.

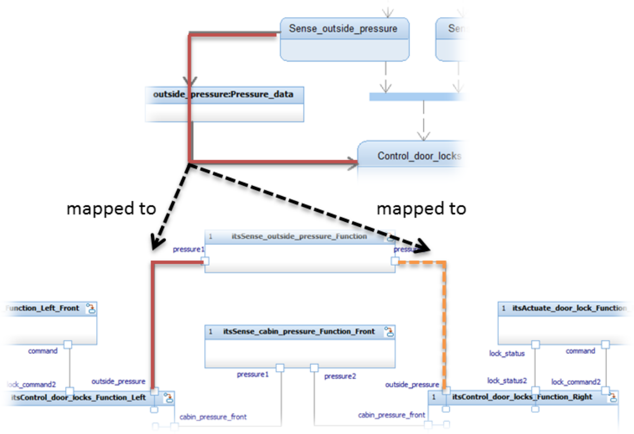
Sense_outside_pressure
outside_pressure:Pressure_data
Control_door_locks
mapped to
mapped to
itsSense_outside_pressure_Function
pressure1
itsSense_cabin_pressure_Function_Front
itsControl_door_locks_Function_Left
itsControl_door_locks_Function_Right
itsActuate_door_lock_Function
command
outside_pressure
cabin_pressure_front
lock_status
lock_status2
lock_command2

Fig. 13: Mapping between links in the functional and the logical model

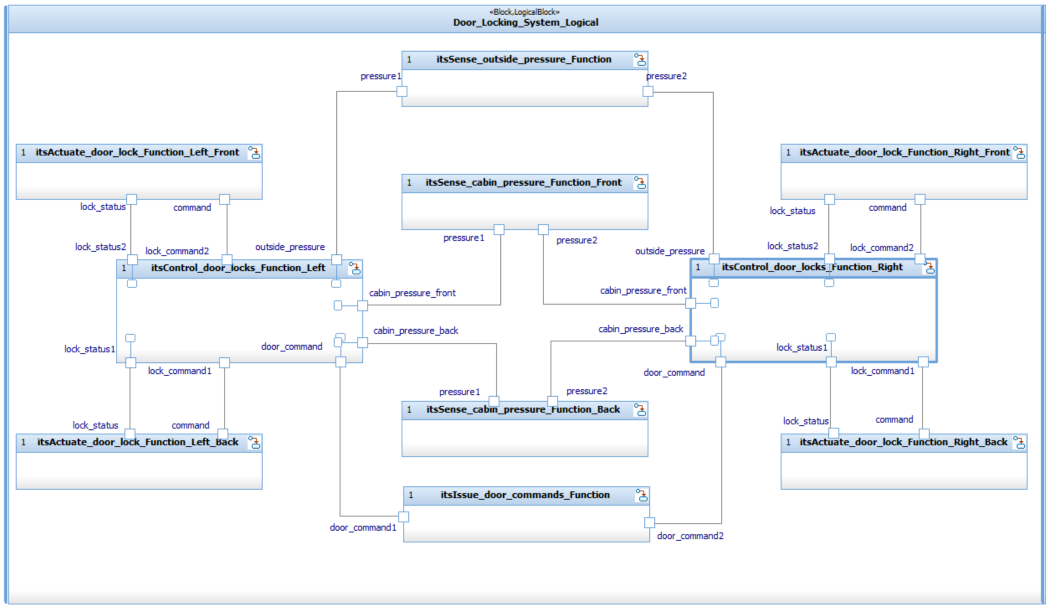
«Block,LogicalBlock»
Door_Locking_System_Logical
itsSense_outside_pressure_Function
itsActuate_door_lock_Function_Left_Front
itsActuate_door_lock_Function_Right_Front
itsSense_cabin_pressure_Function_Front
itsControl_door_locks_Function_Left
itsControl_door_locks_Function_Right
itsSense_cabin_pressure_Function_Back
itsActuate_door_lock_Function_Left_Back
itsActuate_door_lock_Function_Right_Back
itsIssue_door_commands_Function
pressure1
pressure2
lock_status
command
lock_status2
lock_command2
outside_pressure
cabin_pressure_front
cabin_pressure_back
lock_status1
lock_command1
door_command
door_command1
door_command2

Fig. 11: Logical model

E. Technical model

The technical model is again a refinement of the logical model taking into account physical aspects of the system. Also, further design decisions have been made, i.e., it would be possible to create different versions of the technical model that satisfy the given requirements to represent different design alternatives.

F. Additional technical requirements

The additional requirements of the technical model that have not been taken into account in the functional or logical model are provided by Table IV. Most of them directly influence the choice of technical components, e.g., REQ-12 excludes all actuators that fail the given test. Not all of these requirements can be represented adequately in a SysML model, in our example this is probably true for the requirements 12 to 16. Others require extensions of the modelling profile and external tools for their evaluation, e.g., [28] provides an extension to SysML, the metamodel from Figure 6 and a tool for the evaluation of safety requirements like REQ-09.

Table V provides a description of all components of the technical model and Figure 14 shows the internal structure of the technical DLS. As can be seen, the technical model also includes technical components that are not motivated by the logical model and/or any external requirements: the remote data concentrators (RDC) and the switches. They have been added due to a design decision that the system will make use of the existing aircraft data network, which is based on the Ethernet standard and requires data concentrators to convert data between the network and discrete sensors and actuators.

Figure 15 defines the mapping between the logical components from the logical model to the technical components of the technical model. Keep in mind, that this mapping is again at class level. The mapping is not necessarily a one to one mapping. One logical component might require several technical components to implement the required behaviour. In the example, the logical component class *Actuate door lock Function* from the logical model is mapped to two components of the technical model: the Door Lock Actuator, responsible for moving the door lock, and the Door Lock Sensor, responsible for monitoring the status of the door lock. This is again due to a design decision. It would be perfectly possible to select an actuator that provides its status as an output without the need for an additional sensor.

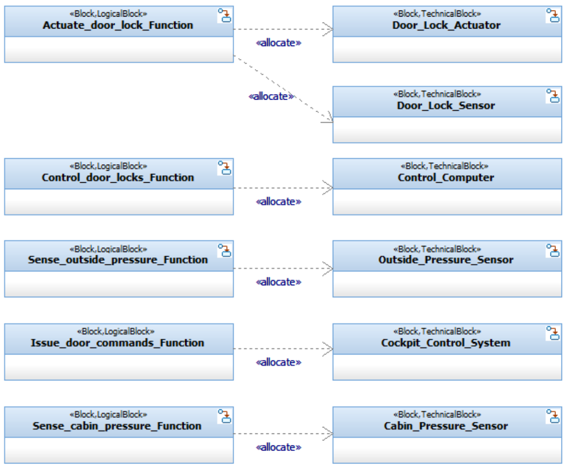

Fig. 15: Mapping between logical and technical model

Likewise, there exists a mapping of the logical links from the logical model to the physical connectors of the technical model. Picking up the example from earlier on, where the mapping of the functional link between the functions *Sense outside pressure* and *Control door locks* to the logical model has been shown, the logical link between *itsSense outside pressure Function* and *itsControl door locks Function Left* can now be mapped to a number of connectors in the technical

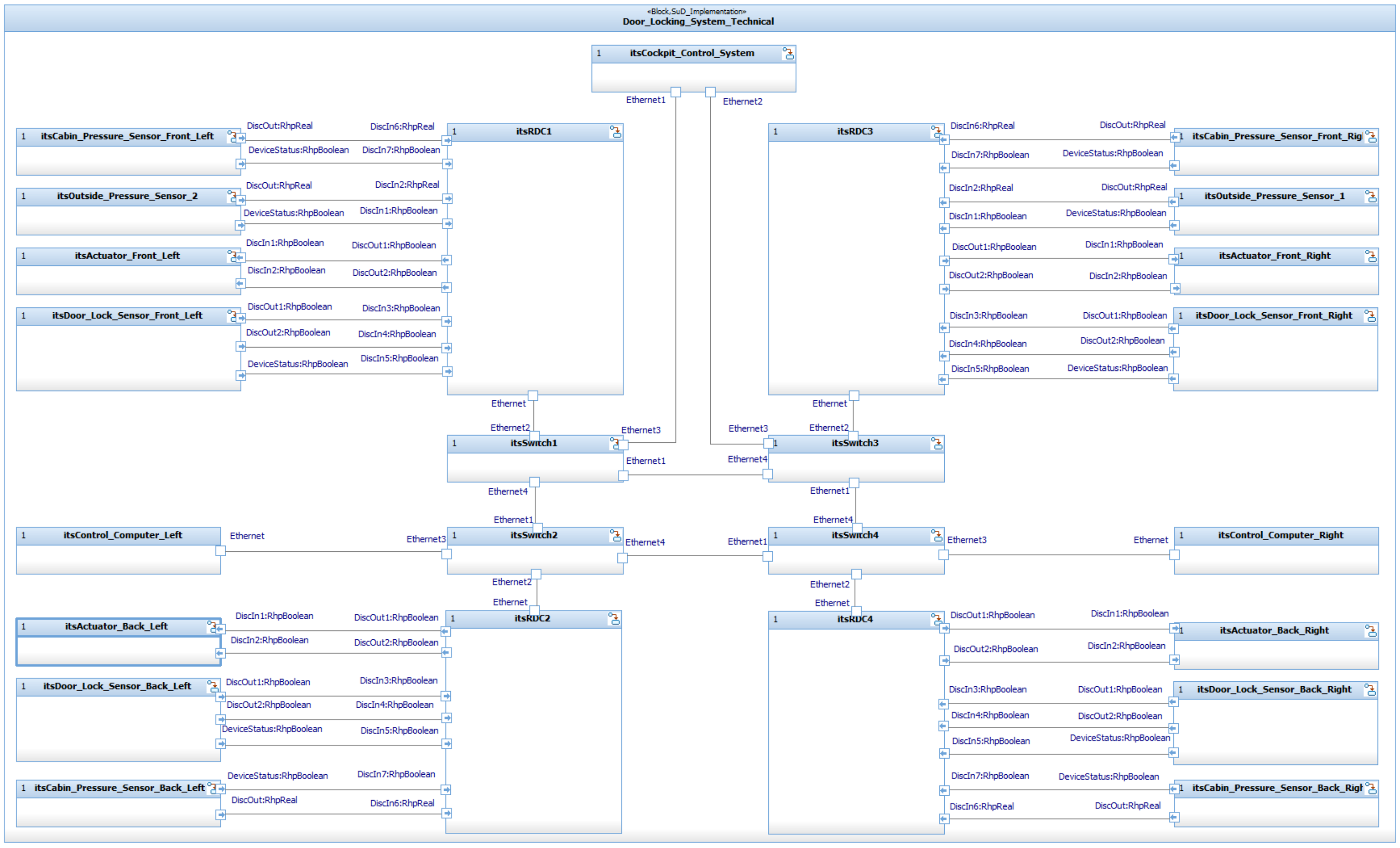

Fig. 14: Technical model

model as depicted by Figure 16.

The UML provides no natural construct to represent this mapping formally. So the information is stored by extending the logical links in the logical model with tags that contain the precise mapping of the logical link to all technical models in which it is implemented as XML data.

G. Testing the models

The logical and the technical model are executable, i.e., all blocks have a statechart that defines their behaviour and, using the execution framework that is provided by the modelling tool, it is possible to generate executable code, which when compiled simulates the model. This allows us to test both models using our defined requirement monitors.

There are two prerequisites for doing so: Firstly, the instantiation of the model under test (MuT) and the requirement monitors in a verification model. Note, that the requirement monitors may have to be instantiated several times, depending on the structure of the model that is to be tested. In the DLS case, the requirements and subsequently their requirement monitors are applicable for each door of the aircraft, hence they have to be instantiated four times in the verification model. Secondly, the connection of the instantiated requirement monitors and the model. The requirement monitors and the MuTs do not necessarily have matching interfaces so an additional entity is required, a so-called mediator [27]. This mediator pulls all the relevant data from the MuT, converts the data to the format that is expected from the requirement monitors and sends it to the requirement monitor instances. Given that the logical and the technical model, or even different implementations, i.e., design alternatives, of one of the models can have quite a different structure, the mediator allows reusing the same requirement monitors for testing all the models.

For the verification of the technical model, we connect a graphical user interface (shown by Figure 17) to the executable verification model. This interface allows stimulating the model and visualises various parameters of the models at runtime, i.e., it enables playing with the model. Since the requirement monitors are active all the time, this kind of testing may lead to uncovering errors that would not have been found with fixed test scenarios.

A more structured verification of a model can be achieved by defining fixed test scenarios. The UML Testing Profile (UTP) [31] is an extension to the UML that provides additional type definitions, such as *test case*, which can be used to manually define test scenarios and the implementation of the UTP in modelling tools allows the automatic execution of these scenarios to verify a model.

And of course, as described in Sections II-B and III-C, it is possible to derive the test scenarios directly from the logical or technical model using a white-box test case generator, such as the ATG for Rhapsody. This tool will systematically stimulate

TABLE IV: Description of additional technical requirements

Req.	Type	Text
REQ-09	Safety	The failure rate for wrong residual pressure determination in a controller shall be no greater than 1E-6/flight.
REQ-10	Weight	The weight of the DLS shall not exceed x kg.
REQ-11	Cost	The costs for purchase and installation of a DLS shall not exceed 0.5 Million Euro per aircraft in serial production.
REQ-12	Environmental	The door lock actuator shall be able to withstand the salt spray test as defined by the applicable standard DO-160E, Section 14, CAT. S
REQ-13	Maintenance	It shall be possible to replace the door lock sensors without removal door lock actuator from the aircraft and without recalibration.
REQ-14	Operational	The DLS shall be designed for a 10000 cycles life in normal operations.
REQ-15	Reliability	The guaranteed meantime between failure (MTBF) of all DLS components shall be at least 30000 flight hours.
REQ-16	Installation	The DLS shall be designed such that persons with a height of between 155 cm and 200 cm are able to install any component without the use of non-standard tools.

the MuT and find test scenarios that cover all states and transitions in the model.

V. Future Work

This section provides a couple of topics for current or future work for extending the approach described in this paper. Apart from extensions to the framework, we are also working on the application of the methodology for a concrete industrial-based use case to validate the framework.

A. *Combination of model-based testing and model-based analysis*

Dijkstra's famous aphorism holds that tests can only show the presence of errors not their absence [32]. Analysis techniques, e.g., model checking can be used to proof required characteristics of a system. Model-based analysis (MBA) and testing are complementary quality assurance techniques since static and dynamic analysis provide altogether different types of information: typically, static analysis provides general information about a model of the system while dynamic testing provides specific information about the system under test itself. Substantial quality and cost improvement can be obtained when they are systematically applied in combination.

TABLE V: Description of technical components

Technical Component	Description
Cabin Pressure Sensor	Measures the athmospheric pressure inside the aircraft cabin.
Cockpit Control System	External system that has a user interface that allows the users, i.e., the crew members, to issue commands to open or lock the aircraft doors.
Control Computer	Hosts the door lock control function.
Door Lock Actuator	Moves the door lock between locked and unlocked position.
Door Lock Sensor	Monitors the position of the door lock.
Outside Pressure Sensor	Measures the athmospheric pressure outside the aircraft.
RDC	Remote Data Concentrator converts between discrete and network data.
Switch	Ethernet switch for routing data in the aircraft data network.

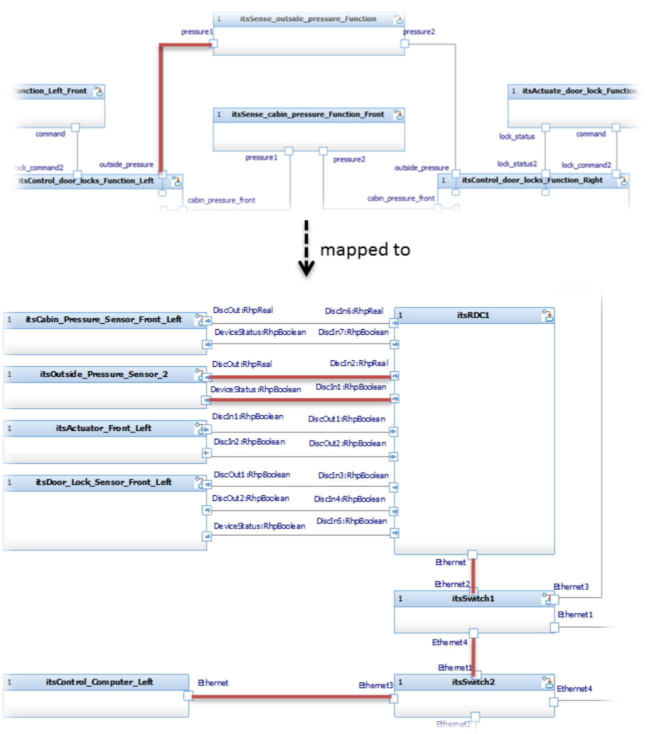

Fig. 16: Mapping between links in the logical and the technical model

One example for such a combination of MBT and MBA is the application of MBA in form of a model checker to improve the completeness of a test suite generated from a whitebox model using MBT as Figure 18 shows. The problem that is addressed by this method is that the automatic test scenario generator does not always achieve to generate a test suite with 100% coverage (coverage for this scenario means model/code coverage). At the moment, manual effort is required to complete a test suite to achieve 100% coverage. This manual effort can be replaced by the application of a model checker. If a test case generator manages to cover 95

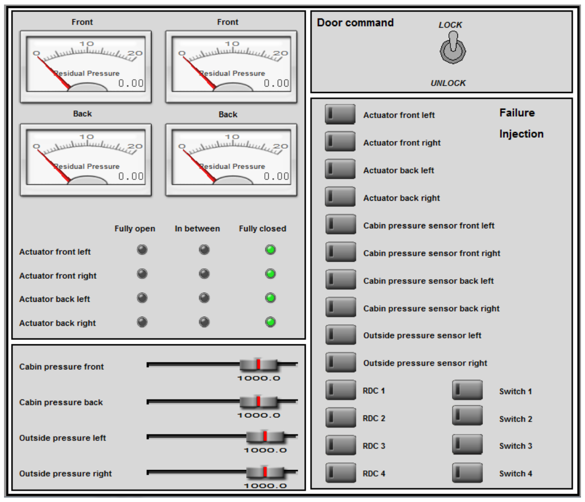

Fig. 17: User interface for simulation control

out of 100 states of a model using test scenarios then we can write properties that check the reachability of the remaining five states. If the model checker manages to reach a state then the proof trace provided by the model checker can be directly added to the test suite as a new test scenario. If the model checker cannot find a solution for reaching a state then the model needs to be adapted.

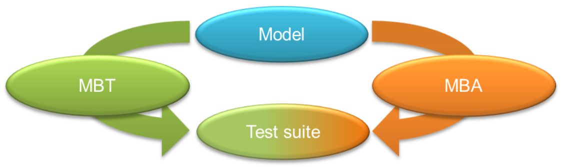

Fig. 18: Combination of model-based testing and model-based analysis

B. Combination of test scenarios obtained from different sources

Evluation of different MBT approaches and tools in the recent past, e.g., [33], [34] showed, that each tool has specific strengths and weaknesses and almost none of them can replace additional manual test scenario creation completely. If we use more than one test scenario generation approach and if we allow test scenario generation at different levels of abstraction as Figure 19 shows, then there is a high probability that the resulting test suite contains a high amount of redundant test scenarios. In order to test efficiently, especially when we are in the phase of hardware testing where a test run is much more expensive than a test run on a model, the redundancy in the test suite must be reduced to find an optimal test suite. Adaptation of previous work, e.g., [35], on that topic to our overall development and testing approach is currently being investigated.

C. Automated model-composition and results evaluation

The creation of verification models, i.e., models that integrate requirement monitors, a SuT system model and scenarios, i.e., the finding of suitable combinations of system model, scenarios and requirements, can be automated. Such a combined

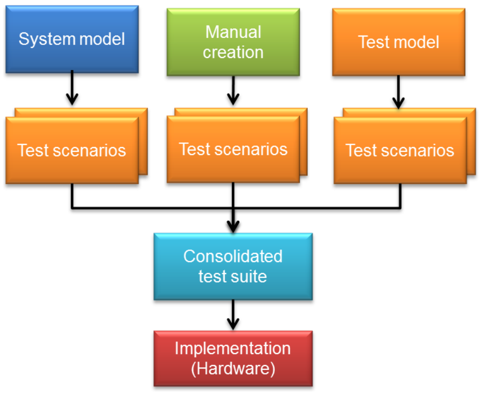

Fig. 19: Optimal test suite from different sources

verification model consists of one system model, which can be logical or technical, one scenario that can stimulate the design alternative, a set of requirements, which can be tested using the selected scenario and a mediator that ensures the compatibility between the involved models. To automate the process further information is needed to evaluate the suitability of a combination of a test scenario and a design model, a test scenario and a requirement or a requirement and a system model. An approach for encoding this information and thereby enabling the automated composition of such verification models is presented in [36]. Combining this approach with the one presented here is ongoing work. Additionally, running the tests, post-processing of the test results, and presenting the verification results appropriately can also be done in an automated fashion.

VI. Conclusion

We presented a framework for an integrated development and testing approach of complex systems and showed its application using an example from the aircraft system domain, the Door Locking System. The main driver behind this development was the need for more efficient testing. This was succesfully achieved by increasing the degree of reusability of engineering artefacts and automation of the testing process in the following way:

- Reusability
  - Explicit modelling of different architecture levels enables reuse of architectures.
  - Requirement monitors can be reused for testing different architecture levels as well as the real hardware product.
  - Removing verdicts from test cases allows using the same test scenario for testing multiple requirements. Additionally, testing a requirement in different test scenarios increases the confidence in the conclusions drawn from the test results.
- Automation
  - Executable requirement monitors allow automated test verdict derivation.
  - Generation of scenarios using MBT.
  - Automated test execution of formal test scenarios.

The approach requires, as most model-based approaches, a frontloading of effort, a personnel shift and a different education of the involved people compared to the current state of practice. While evidence suggests that, through the high degree of reuse and automation, the effort for model-based testing is only slighly higher than the one for traditional testing [37] the adoption of the presented approach in an industrial environment nevertheless requires a rethinking of traditional roles and process steps.

Acknowledgment

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 269335 (ARTEMIS project MBAT) and from the German BMBF.

References

[1] P. Helle and W. Schamai, “Towards an integrated methodology for the development and testing of complex systems,” in VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle, 2013, pp. 55–60.

[2] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297–312, Aug 2012.

[3] V. Encontre, “Testing embedded systems: Do you have the GuTs for it,” IBM: http://www.ibm.com/developerworks/rational/library/459.html, Nov 2003, last visited on 5.5.2014.

[4] A. Helmerich et al., “Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area,” European Commission: http://www.artemis-austria.net/uploads/media/FAST_final-study-181105_en.pdf, Nov 2005, last visited on 5.5.2014.

[5] G. Laycock, The theory and practice of specification based testing. Department of Computer Science, University of Sheffield, 1993.

[6] P. Helle and W. Schamai, “Specification model-based testing in the avionic domain - Current status and future directions,” in Proc. Sixth Workshop on Model-Based Testing (MBT 2010), ser. ENTCS, vol. 264, no. 3, 2010, pp. 85 – 99.

[7] Object Management Group, “OMG Unified Modeling Language (OMG UML), v2.3,” 2010.

[8] ——, “OMG Systems Modeling Language (OMG SysML), v1.2,” 2008.

[9] J. Estefan, “Survey of Model-Based Systems Engineering (MBSE) methodologies,” INCOSE: https://www.incose.org/ProductsPubs/pdf/techdata/MTTC/MBSE_Methodology_Survey_2008-0610_RevB-JAE2.pdf, 2008, last visited on 5.5.2014.

[10] P. Fritzson and V. Engelson, “Modelica - a unified object-oriented language for system modeling and simulation,” in Proc. European Conference on Object-Oriented Programming (ECOOP98). Springer, 1998, pp. 67–90.

[11] B. Selic, “The less well known UML,” in Formal Methods for Model-Driven Engineering, ser. LNCS, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, vol. 7320, pp. 1–20.

[12] Object Management Group, “Semantics Of A Foundational Subset For Executable UML Models (FUML, v.1.0),” 2011.

[13] ——, “Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), v1.0.1 beta,” 2013.

[14] D. Harel and H. Kugler, “The Rhapsody Semantics of Statecharts (or, on the executable core of the UML),” in Integration of Software Specification Techniques for Applications in Engineering, ser. LNCS, H. Ehrig et al., Eds. Springer, 2004, vol. 3147, pp. 325–354.

[15] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[16] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer, 2005.

[17] J. Zander-Nowicka, Model-based testing of real-time embedded systems in the automotive domain. Fraunhofer FOKUS, Berlin, 2009.

[18] T. Bauer, H. Eichler, A. Rennoch, and S. Wieczorek, Model-based Testing in Practice - Proc. 2nd Workshop on Model-based Testing in Practice (MoTiP 2009). University of Twente, The Netherlands, 2009.

[19] J. Boberg, “Early fault detection with model-based testing,” in Proc. 7th ACM SIGPLAN workshop on ERLANG. New York, NY, USA: ACM, 2008, pp. 9–20.

[20] H. Giese, G. Karsai, E. A. Lee, B. Rumpe, and B. Schatz, Model-Based Engineering of Embedded Real-Time Systems, ser. LNCS. Springer, 2010, vol. 6100.

[21] H. Le Guen, F. Valle, and A. Faucogney, Model-Based Testing - Automatic Generation of Test Cases Using the Markov Chain Model. Wiley, 2012, pp. 29–81.

[22] M. Lettrari, “Using abstractions for heuristic state space exploration of reactive object-oriented systems,” in FME 2003: Formal Methods. Springer, 2003, pp. 462–481.

[23] M. Leucker and C. Schallhart, “A brief account of runtime verification,” Journal of Logic and Algebraic Programming, vol. 78, no. 5, pp. 293–303, 2009.

[24] J. Levy, H. Saïdi, and T. E. Uribe, “Combining monitors for runtime system verification,” ENTCS, vol. 70, no. 4, pp. 112–127, 2002.

[25] C. Artho et al., “Combining test case generation and runtime verification,” Theoretical Computer Science, vol. 336, no. 2, pp. 209–234, 2005.

[26] D. Drusinsky, Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. Newnes, 2011.

[27] W. Schamai, “Model-based verification of dynamic system behavior against requirements : Method, language, and tool,” Ph.D. dissertation, Linkoeping University, Department of Computer and Information Science, The Institute of Technology, 2013.

[28] P. Helle, “Automatic SysML-based safety analysis,” in Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, ser. ACES-MB ’12. New York, NY, USA: ACM, 2012, pp. 19–24.

[29] National Transportation Safety Board, “Safety Recommendation, A-00-23 through -27,” August 2, 2002.

[30] V. Hilderman and T. Baghi, Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware). Avionics Communications, 2007.

[31] Object Management Group, “UML Testing Profile (UTP), v1.2,” 2013.

[32] E. W. Dijkstra, “The humble programmer,” Communications of the ACM, vol. 15, no. 10, pp. 859–866, 1972.

[33] M. Shafique and Y. Labiche, “A systematic review of model based testing tool support, Technical Report SCE-10-04,” Carleton University: http://people.scs.carleton.ca/ jeanpier/404F13/T7 - survey papers/SystematicReviewOfMBT-2010.pdf, 2010, last visited on 5.5.2014.

[34] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in Proc. 1st Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech’07). ACM, 2007, pp. 31–36.

[35] G. Fraser and F. Wotawa, “Redundancy based test-suite reduction,” in Fundamental Approaches to Software Engineering, ser. Lecture Notes in Computer Science, M. Dwyer and A. Lopes, Eds. Springer, 2007, vol. 4422, pp. 291–305.

[36] W. Schamai, P. Fritzson, C. J. J. Paredis, and P. Helle, “ModelicaML value bindings for automated model composition,” in Proc. 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, ser. TMS/DEVS ’12. Society for Computer Simulation International, 2012, pp. 31:1–31:8.

[37] T. Bauer, F. Böhr, and R. Eschbach, “On MiL, HiL, statistical testing, reuse, and efforts,” in 1st Workshop on Model-based Testing in Practice (MoTiP 2008). Fraunhofer, 2008, pp. 31–40.