Application of Model-Based Prioritization

D4.7 Deliverable

PUBLIC

2019-09-30

Edited by:

Andre Bergmann, Felix Jakob, Julian Becker, Karsten Meinecke, Xabier Valencia, Martin Reider

Revision history

Date	Version	Comments
2019-05-20	0.1	Initial document, draft structure

Introduction

Definition of Model-Based Testing

Contribution by AKKA

In the last four centuries software became more complex than ever and software testing became an important step in the software development process. Today the question is not if a software was testet, the question is what has been tested, how was it tested, and how much was it tested. [Winter2016]

Model-Based Testing (MBT) has the intention to enable and improve test automation in an early state of a development process. In these early states the requirements are often written in an everyday language. Testing in these states is usually based on manual methodologies. Model-Based Testing focuses on creating a formal model, which is then being used to derive test-cases. [Winter2016]

Definition of Model-Based Test Prioritization

Contribution by ifak

Test case prioritization techniques organize the test cases in a test suite by ordering such that the most beneficial are executed first thus allowing for an increase in the effectiveness of testing. One of the goals is a measure of how quickly faults are detected during the testing process. In test case prioritization, each test case is assigned a priority. Priority is set according to some criterion and test cases with highest priority are scheduled first. For instance, a criterion may be that the test case which has faster code coverage gets the highest priority [Srivastava2008]. For model based test prioritization, similar criterions can be derived from model representations of the SUT.

Model-based test prioritization is a subset of test prioritization methods in which prioritization metrics are derived from model-based representations of the system under test (SUT) or requirements. Common modelling notations are graph-based models such as UML State Machines, UML Activity Diagrams and high level Petri nets. Requirements models are usually created by using notation languages that allow for the formalization of the requirement.

A number of suitable metrics are available for model-based test prioritization. Similar to code coverage metrics, model-based coverage metrics can be utilized for test prioritization. These are derived from coverage of model elements (e.g. nodes, edges, decisions, paths). For model-based coverage of a test case, a differentiation between between total and additional coverage can be made [Korel2009]. The first criterion provides information about the total coverage of a single test case for the underlying model. Additional coverage assumes an execution order and compares model coverage of for all test cases. Test cases that activate parts of the model not yet covered are assigned a higher priority. In addition to coverage

metrics, test cases can be prioritized based on model changes in the development process. Korel and Koutsogiannakis [Korel2007] propose a prioritization technique that compares changes made to the model representation of the SUT. Test cases that cover the modified parts of the model are assigned a higher priority.

To utilize model-based test prioritization techniques, linking of test cases to model elements or requirement models is necessary. For scripted test cases, linking usually requires the execution of test cases for the model using a suitable test adapter. Test cases can then be linked to model elements activated during test execution. Model-based test prioritization is particularly suitable for test cases generated from a model of the SUT or the requirements. By utilizing information provided during the test case generation, linking of test cases to the model elements they were derived from is possible. For more complex toolchains, that e.g. derive the model of the SUT from requirement models, generated test case can be linked to model elements and the requirements they were derived from by ensuring full traceability of generated artifacts during the model based test process [Reider2018].

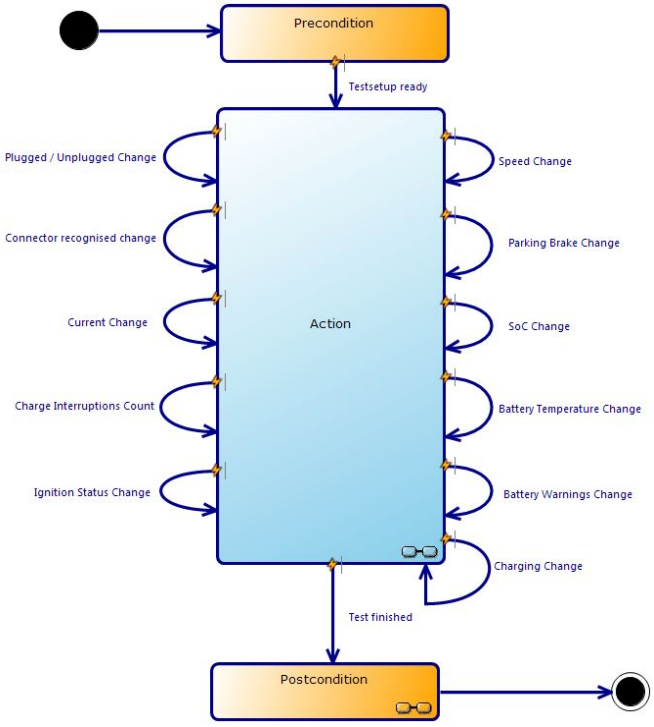
Industry Application

Report of usage of Modica within Automotive Use Case

Contribution by AKKA, Expleo

MODICA is a tool for creating usage models of a test object. From these usage models, test cases can be automatically deduced based on the possible sequences of interaction with the test object. The generated test cases can be transferred to an automatic test execution environment and are immediately ready for use in the target system.

MODICA model creation



AKKA's UseCase is a software module, implementing the approval of charging of an electric cars battery. It decides upon evaluating several inputs, if the charging process of the battery may start or not.

The requirements were modeled into the usage model within MODICA. The default concept of the 3 main states was kept.
The functional behaviour was implemented in the inner state of the Action state. All input values needed for the functions are modeled as self-transitions to re-trigger the actual function execution within the action state.

Figure X. Usage model of Use Case in MODICA

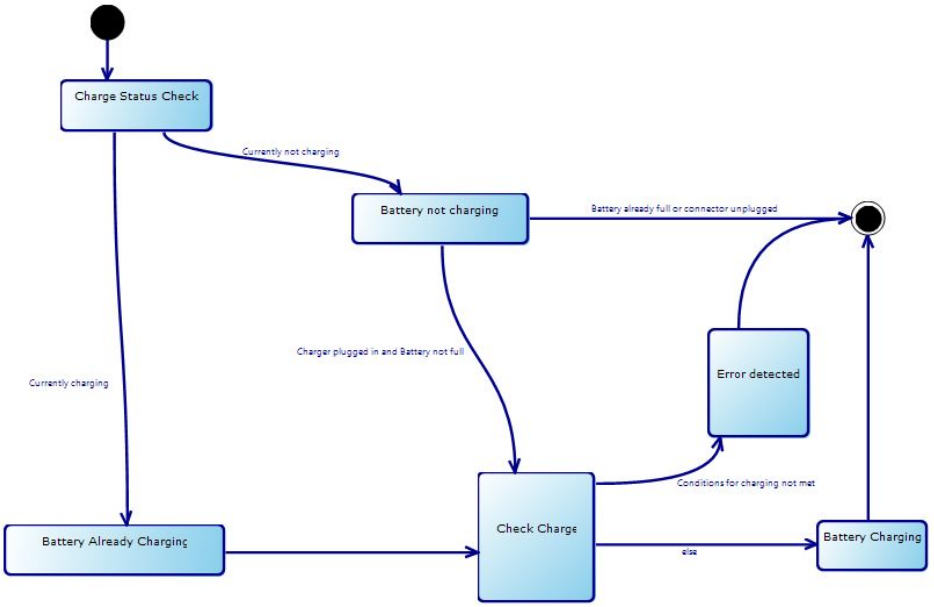


Figure X. Inner behaviour of the Action State

Generate test suites

To enable full test case generation, MODICA needs some information. Code snippets are added to transitions and states and a test case template is also added. Prioritization features are added as well. Executable test suites can now be generated, according to desired rules.

The screenshot shows the overview for the target 100% state coverage.

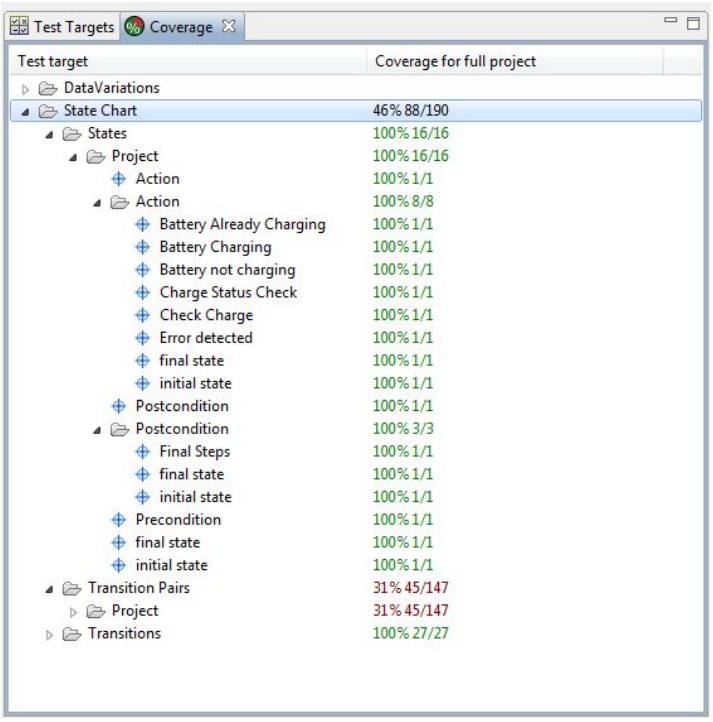


Figure X. Result of geneated tests with target 100% state coverage

Report of usage of MBTCreator within Automotive Use Case

Contribution by ifak

The MBTCreator is a tool developed by ifak that combines various functionalities for model-based testing. MBTCreator offers editors and a graphical user interface for a toolchain that covers all steps from requirements to test case generation and prioritization. In the following chapter, the tool and it's features will be described and the results for it's application to the AKKA use case will be presented.

MbtCreator for model based testing generation and prioritization

In a first step, the tool features methods for formalization of requirements using a notation language, the IRDL (Ifak Requirement Definition Language). This notation for requirement description was developed on the basis of UML sequence diagrams and is specially adapted to the needs of describing requirements as sequences. IRDL defines a series of model elements (e.g. components, messages) with associated attributes (name, description, recipient, sender, etc.). This enables a modular modelling of the requirements. Functional, behavior-based requirements are described textually using ifakRDL and can then be visualized graphically as sequence diagrams (Fig. X). Individual requirements can be combined to features.

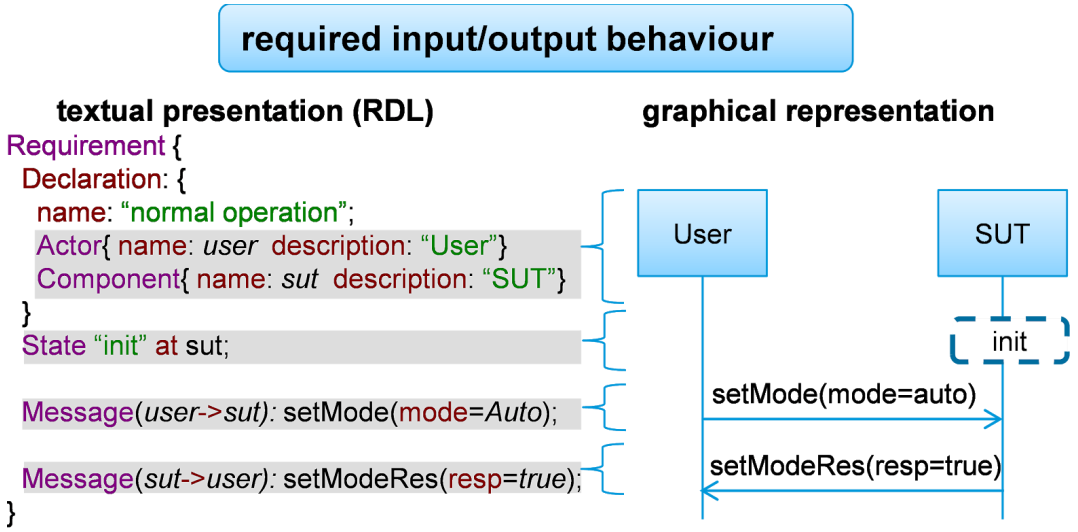


Figure X. Example for ifakRDL and graphical representation as a sequence chart

In a next step, the formalized requirements or features of the SUT can be combined into a specification model using model synthesis. The sequence elements described there, consisting of lifelines, messages, state statements, attribute statements and fragments for time-dependent and alternative processes, are transformed using a rule-based algorithm into equivalent elements of a UML state machine. The resulting specification model describes the behavior of the test object from the requirements perspective and models it in the form of

states and transitions with start and end states, events as triggers and internal attributes to describe data values. The approach formulated at ifak by Magnus et al. [Magnus2017] for the formal description of requirements and model synthesis was implemented within the academic tool ModgenApp in the project MASSIVE and is applied within the work in TESTOMAT.

Based on a model of the demanded behavior (specification model) and with respect to specified test goals, test cases with test data are generated in a systematic way by using a suitable algorithm. Common modeling notations are graph based models like UML State Machines, UML Activity Diagrams and high level Petri nets. Additionally, textual modeling languages are used as input models of the used test generation tool. The specified test goals are very important for the test generation results. Normally test goals are coverage criteria regarding the used specification (requirements) including a specification model as the basis for the test generation. Generally the specification model is a graph based model, so graph based coverage criteria (all nodes, all edges, all paths, etc.) are very common as test goals.

Additionally other coverage criteria regarding the coverage of requirements properties have also been established as test goals. These are linked to associated elements of the specification model (usually nodes, edges) in order to enable the used test generation method to generate the required test cases for an efficient test suite with regard to the test goals. For test generation, the academic tool TcgApp (Fig. 1) developed at ifak was used in, which is based on the methods developed in [Krause2011] on the basis of Petri net unfolding for SPENAT models.

Basis of the model synthesis are formal behavioral requirements of selected features of the test object. The result of the model synthesis is an UML State Machine. For test generation, this UML State Machine is mapped to a SPENAT to generate test cases with respect to the specified test goals.

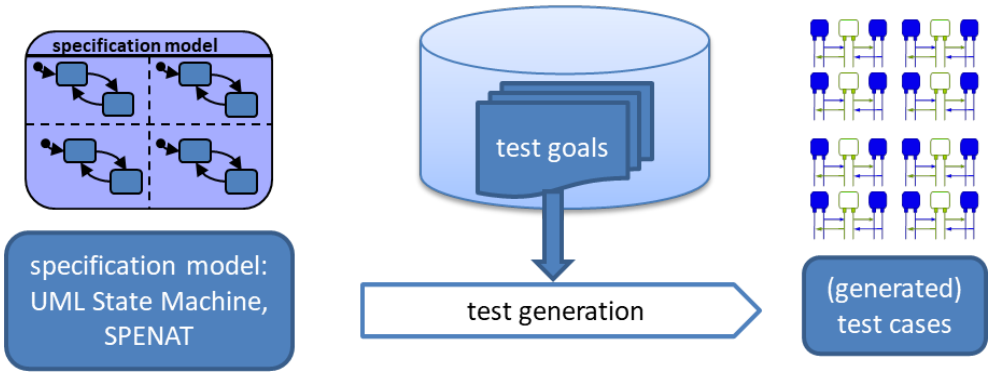


Figure X. Process of model based test generation

Within TESTOMAT an extension of the toolchain in the form of a separate tool called MBTP (Model Based Test Prioritization) was developed, with which the generated test cases are prioritized via a combination of model-based cluster analysis and a requirements-based evaluation procedure (see figure X). The model-based test process allows the seamless linking of all information across the test process, from the behavioral requirements to the generated test cases. This allows the use of metrics for test prioritization derived from the

SUTs requirements and other artifacts from the test process. The goal of the test prioritization method presented here is to optimize test execution with regard to both test coverage and the error detection rate in the early test process, thus enabling early feedback on faults to the test engineer.

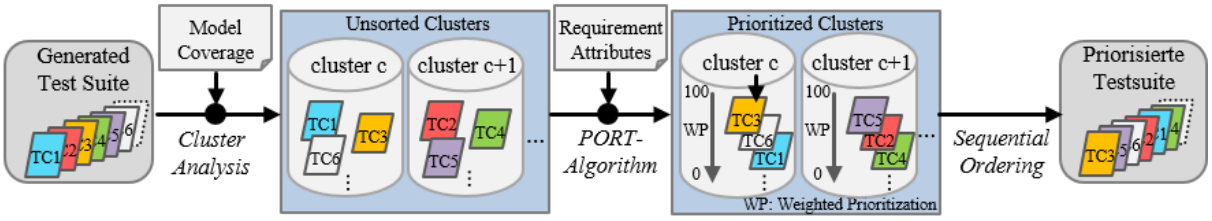


Figure X. Process of model based test prioritization

The first step is a cluster analysis (of generated test cases) using coverage metrics in regard to the specification model. Here, all test cases are evaluated regarding the selected coverage metric (e.g. state-, transition-, path-coverage etc.). In a second step, the test cases in each cluster are prioritized according to one or several requirement-based criteria (e.g. complexity, volatility, customer priority). The prioritization goal is to improve test coverage and fault detection rate during early testing.

Application of toolchain within Automotive Use Case

The toolchain for model-based testing presented in the previous chapter was applied to the use case provided by AKKA. The toolchain was applied separately for each of the two SUTs (Charging Management and Charging Approval). As a result of the application, an extensive set of abstract prioritized test cases could be generated. In the following chapter, the application of the toolchain will be demonstrated.

1. Formalization of requirements

As part of use case 8, AKKA has provided functional and non-functional requirement documents describing the expected behavior for each of the SUTs. As part of this report, only the functional requirements were relevant for test in work package 4. Testing for the non-functional requirements will be part of the collaboration in WP5 and are not further detailed in this document.

Each of the statements describing functional behavior of both SUTs were treated as a separate requirement. Overall, the Module Charging Approval is described by 10 separate requirements and the module Charging Management by 11 separate requirements. Using IRDL language, each of the requirement statements were formalized as separate requirement models. In figure X, one of the requirement models is visualized as a sequence diagram. The diagram belongs to a requirements of the module Charging Approval that defines that the approval has to be withdrawn if the connection to the charging station is lost for more than a predefined amount of time (here 500ms).

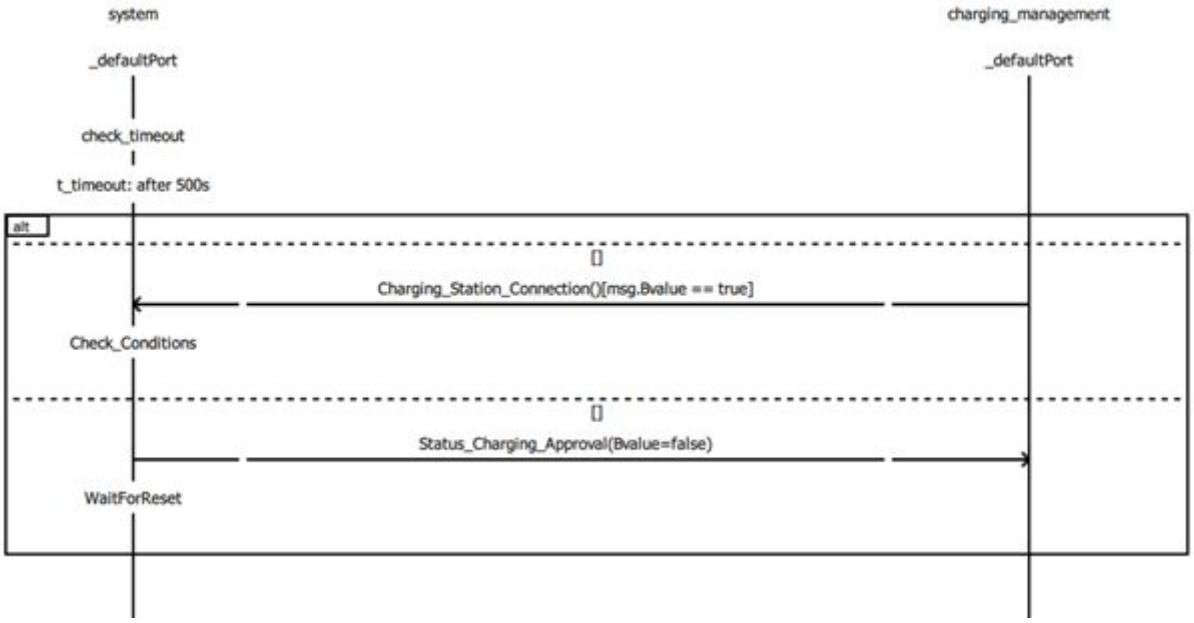


Figure X. IRDL model for withdrawal of charging approval in case of a connection timeout

2. Model synthesis

For the next step, the requirement models of each module were used as the input for model synthesis. For model synthesis, the academic tool ModgenApp developed by ifak was used. As a result, a graph-based representation of the functionality of each module as described by their respective requirements was generated in the form of two UML statemachines. The model of the module Charging Approval contains 6 states and 20 transitions, while the model of the module Charging Management contains 10 states and 23 transitions. The validity of each generated UML statemachines was manually validated based on the requirement documents. In figure X, the UML statemachines of both modules are shown.

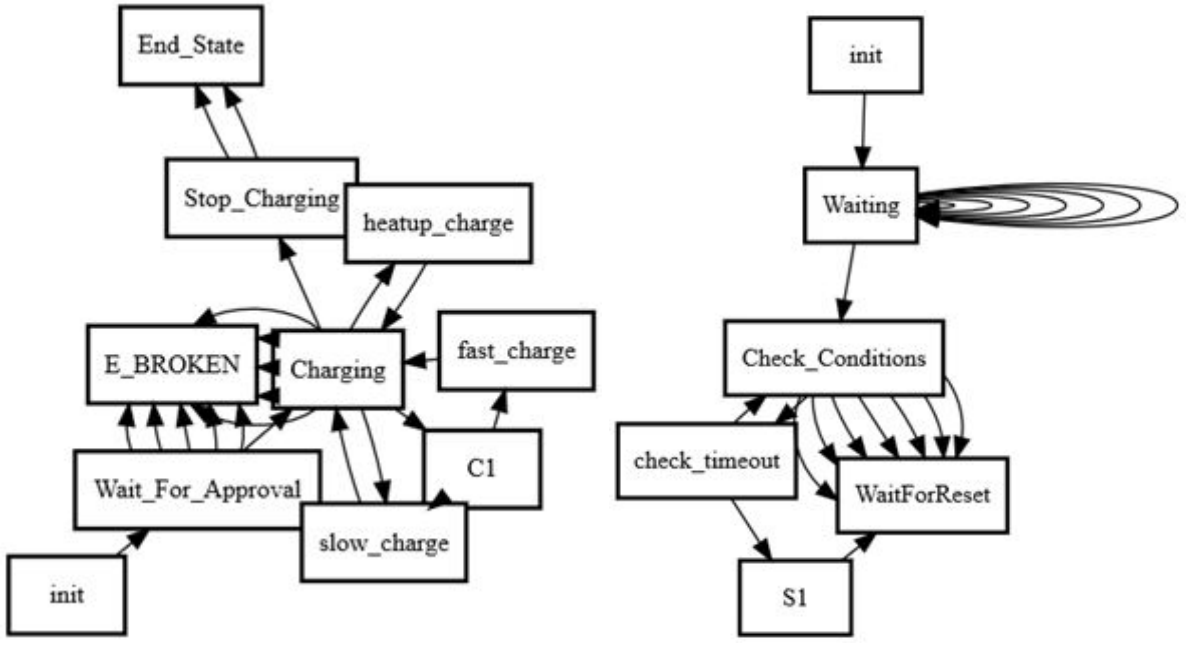


Figure X. UML Statemachines of Charging Management (left) and Charging Approval (right)

3. Test Generation

After model synthesis, the generated UML statemachines of both modules were used as an input for test generation using the academic tool tcgApp developed by ifak. For test generation, a set of test goals in the form of coverage criteria can be selected as shown if figure X. Here, the coverage criteria “all-paths” was selected, which ensures that each possible path in the UML statemachine is covered by at least one test case.

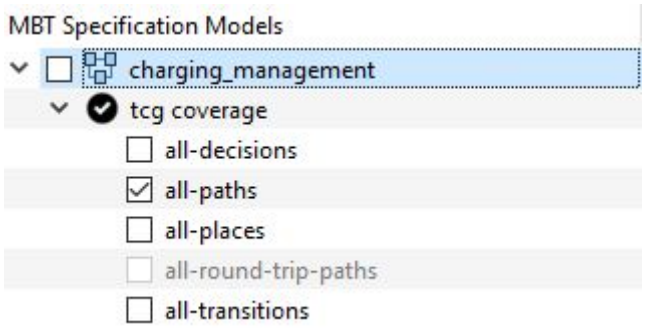


Figure X. Selection of test goals in the form of coverage criteria in the tool MbtCreator

By utilizing our test generation algorithm, a total of 780 test cases were generated for module Charging Approval and 33 test cases were generated for module Charging Management. The large difference in the number of test cases between both modules can be explained by the higher input complexity of the charging approval. Approval is given when a large number of different criteria is fulfilled, therefore tests are generated for all possible combinations of the criteria.

In figure X, one of the generated test cases is visualized in the form of a sequence diagram. Here, the test system interacts with the SUT and provides a number of valid criteria. Afterwards, it is checked if Charging is approved by the module. Afterwards, one of the criterias for charging approval is changed (here the vehicle speed) to be no longer valid and the test system checks if the module withdraws the approval.

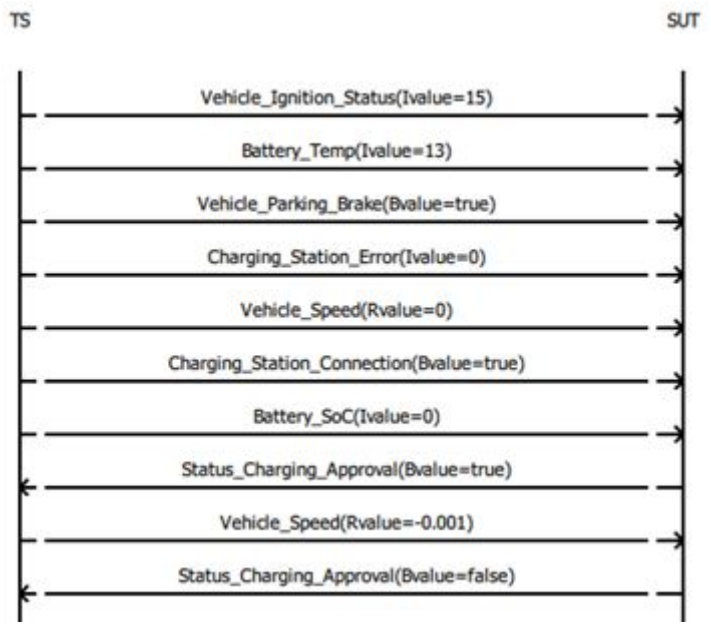


Figure X. Test case for module Charging Approval visualized as a sequence diagram

4. Test Prioritization

At last, the test cases generated for both of the modules were prioritized using the academic tool MbtpApp developed by ifak. For the hierarchical, agglomerative cluster analysis, which is the first step of the prioritization process, a coverage criteria has to be selected for which the similarity of all test cases is calculated. Here, transition coverage in regard to the underlying UML Statemachine of each module was selected because it allows for an adequate comparison of test cases. The result of the cluster analysis can be visualized by a dendrogram, as shown in figure X for the charging management.

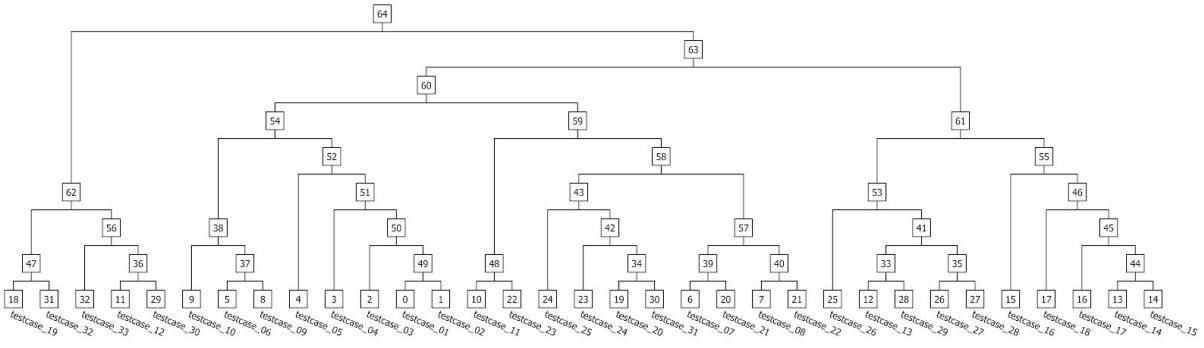


Figure X. Dendrogram based on hierarchical, agglomerative cluster analysis of test cases

The next step of the test prioritization step is a requirement-based prioritization of test cases in each cluster. Here, the attributes volatility, customer priority and complexity of a requirement were selected for prioritization. The attribute values for volatility and customer priority were provided by AKKA. The complexity of each requirement was calculated by the tool for test prioritization. Here, the cyclomatic complexity of each requirement is measured

automatically by transformation of each individual requirement model into separate UML statemachines. The cyclomatic complexity is then calculated for the number of edges and nodes in the model.

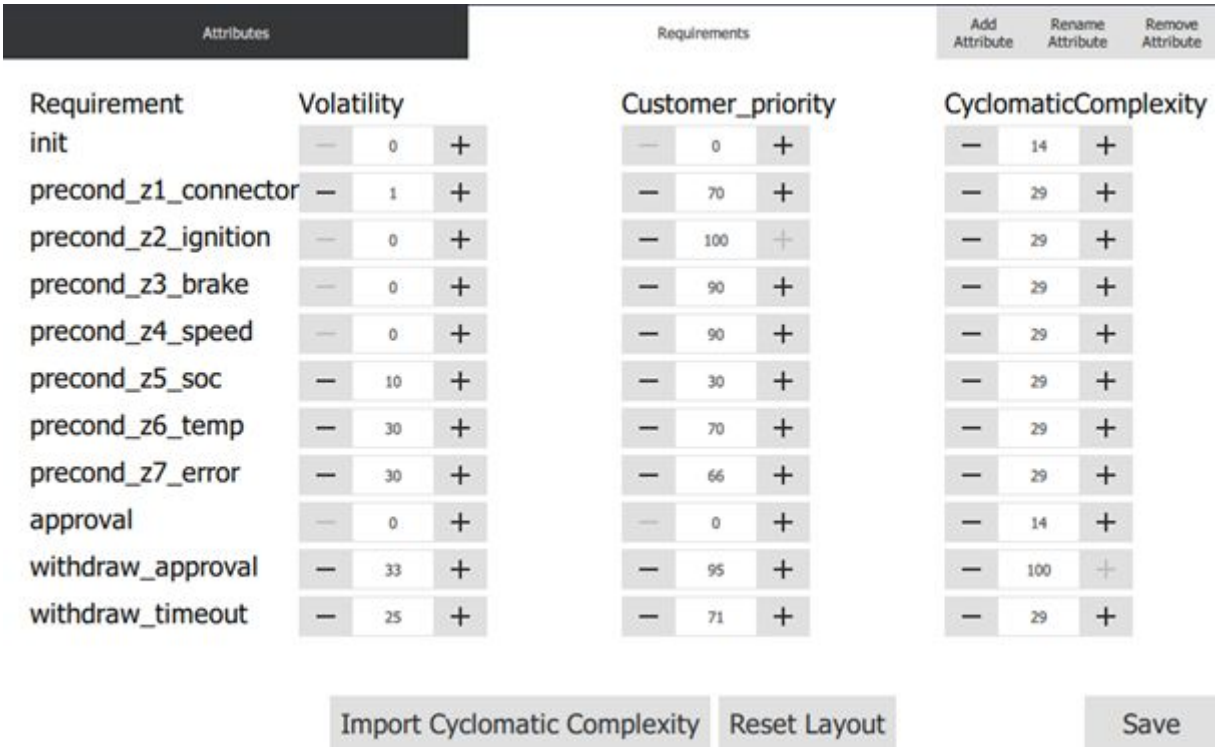


Figure X. Assignment of values to requirement attributes in MbtCreator

As a result of the prioritization tool, a list of all test cases and their respective relative priorities are provided for each module.

Outlook

In future work, a test adapter will be developed to connect the test system (MbtCreator) to the SUTs provided by AKKA. The prioritized test cases will then be executed to evaluate the effectiveness of the prioritization approach.

Industry Application of Model-Based Prioritization by AKKA Technology

Contribution by AKKA Technology

Depending on certain requirements on customer projects, AKKA Technology applies a methodology to do a prioritization on Model-Based Testing [Jakob2012]. This methodology requires a specific test-model, which can also be derived from a system model. Since AKKA is not a tool vendor the methodology does not depend on a specific tool. In the past the test model was done in different tools such as PTC Integrity Modeler, MID Innovator, Sparx Enterprise Architect or Expleo MODICA.

The test model has to be developed in such a way, that a test case generator can read the test model and can generate test cases. Even though the test case generator applies specific strategies for generating the test cases, the number of generated test cases can easily reach a number, which are not handable anymore. For that purpose AKKA Technology combines the test model with some analysis. These analysis can focus on different aspects such as risk, security, likeliness, or severity. For its automotive customers AKKA Technology often uses existing FMEAs and combine them with additional analysis.

The combination of the test model and analysis allows the test case generator to optimize its generation. Depending on the technology used for the test case generator the additional analysis can be seen as weights on a graph while applying graph theory. The analysis tells the generator which test cases are more relevant, and therefore does a prioritization of test cases.

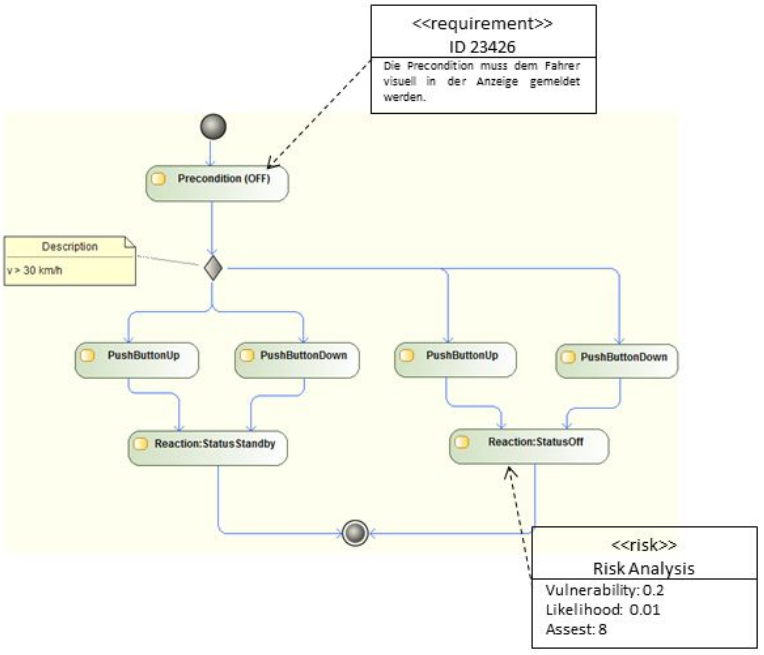


Figure x. Test model extended with results of a risk analysis

Using this methodology AKKA Technology was able to realize test case generation for very specific purposes with optimization on running time, risk, security, costs, known errors, requirements and other attributes, with coverage regarding functionality, transitions, requirements, risk, functional safety, user experience, likelihood and many more.

Industry Application for Expleo

Contribution by Expleo

Model-Based Prioritization is done with the inhouse tool MODICA. Therefore the generation of the test cases based on the model can be altered to produce better or less test cases that cover the same requirements or testing criteria.

The test set to be generated should often reach a desired coverage but should remain compact in the number or length of test cases. This is an initially very abstract requirement

which has a wide range of impacts and presents a challenge for the manual and automatic (or model-based) test case finding. In many cases, it may be useful to involve the domain knowledge of the tester and, for example, to provoke particularly interesting partial sequences in a more generic model. Such guidance of the generator is possible in MODICA via the sequence rules. This can be a useful tool for increasing the test depth, especially in a combination of model-based approaches and black-box testing.

Report of usage of SBTTool within Aerospace Use Case

Contribution by Alerion and Mondragon University

Alerion Technologies develops and commercializes turnkey small aerial autonomous robots for industrial applications, including infrastructure inspection and automatic damage detection. To achieve this, it is necessary to develop an autonomous high-performance embedded computer software and navigation system for unmanned aerial vehicles as well as real-time image processing and data analytics. What is more, the platforms are designed to fly very close to structures or objects, which can have disastrous consequences when the software has not been thoroughly tested.

The system testing is divided into four phases: Unit testing (Phase 1), Integrity testing (Phase 2), Simulation-based testing (Phase 3), Real flight testing (Phase 4). Currently, Phases 1 and 2 are performed automatically, unlike Phase 3. Simulations must be launched manually which is extremely time consuming and inefficient as it requires constant human operation. With the TESTOMAT project, the aim is to automate Phase 3. This can be achieved by automating (i) the generation and prioritization of test cases, (ii) configuration of the simulation environment and (iii) the execution and evaluation of the simulation. As a result, the time necessary to prepare, run and evaluate the simulations will be significantly reduced. This in turn will enable the system to carry out more test cases, leading to more robust software. In the following section, the tool SBTTool is detailed, which will allow Phase 3 testing automation process.

SBTTool

The SBTTool is composed of the Jenkins tool, which is in charge of orchestrating the different components: the simulation environment (ROS [2019] + Gazebo [2019]), the test suite generator, the test case prioritizer and the Oracle (see Figure 1). To automate the setup, deployment and launch of the simulations and evaluations, each component will be delivered in a Docker container.

The SBTTool will allow Phase 3 testing automation process as follows. First, a test suite must be created (see step 1 in Figure 1) and after that, test cases are prioritized based on the established criterion (see step 2 in Figure 1). Afterwards, the simulation docker container receives as input the software under test and the test case to execute (see step 3 in Figure 1). Once the simulation has finished, the *Oracle* (see step 4 in Figure 1) evaluates whether

the result of the simulation was satisfactory or not and stores its data for further analysis (see step 5 in Figure 1).

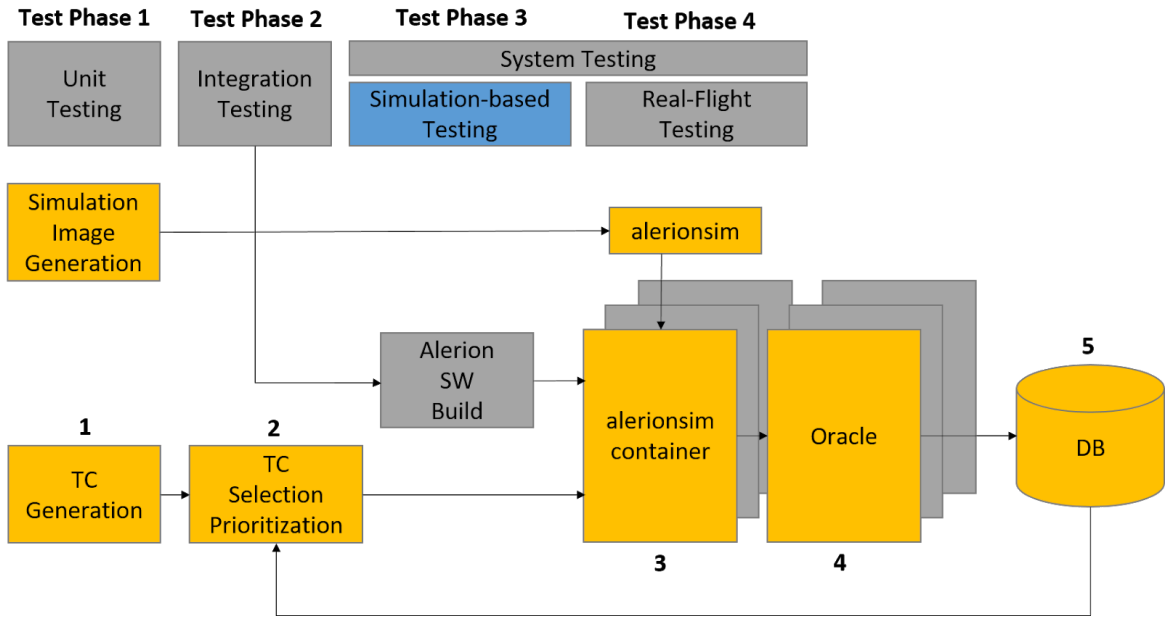


Figure 1. Phase 3 automation process

Model Based Test Case Generation and Prioritization for Simulation Environments

The test case generator is based on a model (Scenario Specification Model) that defines the relevant parameters of the simulation scenario to be tested, such as, the inspection of petroleum pipelines, hydroelectric dams, bridges or wind turbines among other infrastructures.

Despite the fact that the model can be used for different purposes, inspection of bridges, gas pipelines, etc., the use case provided by Alerion in the TESTOMAT project named *Test Prioritisation to handle Automated Tests of Robotic Platforms* will be used as an example.

Scenario Specification Model (SSM)

The SSM, as can be seen in Figure 2, is formed by a *Test Scenario* that comprises a set of initial parameters (*Initial Param Set*) and execution parameters (*Execution Param Set*).

The *Initial Parameters Set* is composed of those parameters that must be set before running the simulation. A few examples of these kinds of parameters are, the height of a bridge abutment, the angle of the blade in a windmill, the width of a dam floodgate, etc. The *Initial Parameters Set* has a unique *name*, a *description*, *units* and can have either a range (*Range*) or enumeration (*Enum*) values.

The *Range* type parameter has a *minimum*, *maximum* and a *step* value. Thus, it is possible to define all the range values in which the drone should be able to perform its job properly. In the use case provided by Alerion, the initial range type parameters are the inspection distance (*d1*), initial drone yaw (*a1*), initial drone shift (*d3*), and blade position (*a2*) (see

Figure 3). Conversely, when the number of possible parameter values is reduced or does not follow a pattern, the type would be *Emum*. An example of this type of parameter in the use case is the size of the blades (*d2*).

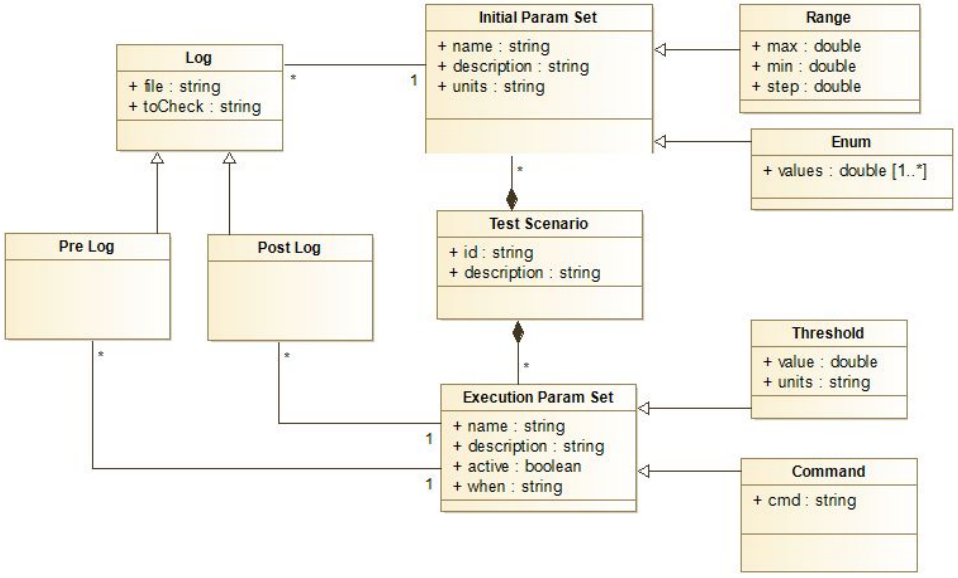


Figure 2. Scenario Specification Model

The *Execution Param Set* type parameters are used to define those events that can occur during the simulation. Two types of execution parameters have been differentiated: those that can be defined with a value (*Threshold*) and those that require a command to make it happen (*Command*). *Threshold* type parameters can be used, for example, (i) to simulate communication problems, such as a time that has passed without communicating, or (ii) to define a constant atmospheric phenomena, like rain with a predefined intensity during the simulation. *Command* type parameters, on the other hand, can be used to simulate problems in the modules under test.

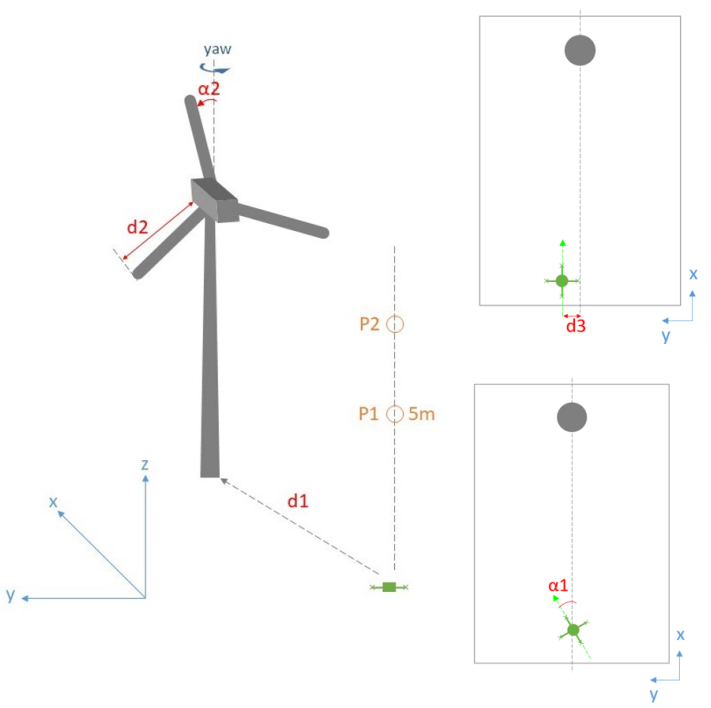


Figure 3. Windmills inspection scenario, Initial Param Set type parameters description

Execution Param Set parameters require establishing either the specific moment in which it will occur (time) or when a specific state of a state machine has been reached. For the use case, two *Threshold* parameters have been defined: *Communication lapse* and *LIDAR noise*. Furthermore, four *Command* type parameters have been defined: *autopilot crash*, *autopilot abort*, *LIDAR crash* and *windsurveryor crash*.

The parameters can have in addition, a *log* associated with the expected value in a satisfactory simulation for the given parameter. In such a manner, the *Oracle* will be capable of assessing the outcome of the simulation. The *Execution Param Set* can have two types of logs: Pre log for the results expected before the execution, and Post log for the results expected after its execution. The *Oracle* will use this information, among others, to determine the result of the simulation carried out.

Test Suite Generation

The test case generator uses an adaptive random algorithm to create test cases. This algorithm was selected to cover the widest spectrum with the fewest test cases, which will allow more faults to be found. To create a test suite, the test case generator needs four inputs: (i) the Scenario Specification Model, (ii) the number of test cases to generate, (iii) the number of test case candidates and (iv) the algorithm to measure the distance between the test cases.

First, the scenario parameters have to be defined with the SSM. Once the parameters have been defined, it is necessary to translate this to an XML file, so the test case generator can interpret it. The transformation from the visual model to the XML format nowadays is done

manually. However, a tool to translate the information automatically will be developed, to facilitate the creation of the XML-based Scenario Specification Models.

The number of test cases to include in the test suite, the number of test case candidates and the distance algorithm must be selected. Although the system currently calculates the distance between test cases using Euclidean distance measurement, other measurements such as Jaccard coefficient or Hamming distance will be added as well.

Once all inputs are set up, the test case generator reads each parameter and randomly selects a value from the range of the possible values for that parameter. When all parameters are accomplished, a test case candidate is created when it is checked that it does not already exist in the test suite or candidates set. Once the candidate is valid, the distance is calculated against the test cases that are already part of the test suite. When the required candidates are generated, the candidate with the largest distance becomes part of the test suite. This process is repeated until the desired number of test cases for the test suite is reached. Finally, as a result, a YAML is created with the test suite.

Test Case Prioritization

The prioritization will be performed in two phases. In the first phase, the primary goal is to create historical data about the simulations, failures found, execution times, coverage achieved, and so on. On the other hand, in the second phase, historical data generated in the first phase will be considered for prioritization.

In the first phase, the prioritizer receives as input the test suite in a YAML format, the number of test cases to execute and the algorithm to calculate the distances between test cases. The system first selects a test case randomly from the test suite and inserts it in the selected test cases. Then it searches for the most different test case in the test suite, considering all test cases selected until that moment. This process is repeated until the desired number of test cases is obtained. As a result, a YAML file with the prioritized test cases is obtained. Once the simulations have been executed, the Oracle evaluates the results and saves the historical data.

The aim of the second phase is to carry out prioritisation based on the historical data obtained from the first phase. The prioritizer will receive a test suite in YAML format and the desired number of test cases. Unlike in the first phase, the YAML will also include historical data about previous test case executions which will be used to carry out prioritization. Although the prioritization algorithm of this second phase is still undefined, algorithms based on weights or multi-objectives are being studied to this end.

Bibliography

[Winter2016] Winter, M., Roßner, T., Brandes, C., Goetz, H., Basiswissen Modellbasierter Test, dpunkt.Verlag, Heidelberg, 2016

[ROS 2019] ROS, 2019, https://www.ros.org

[GAZEBO 2019] Gazebo, 2019, http://gazebosim.org/

[Jakob2012] Jakob et al, Risk-based Testing of Bluetooth Functionality in an Automotive Environment, http://publica.fraunhofer.de/documents/N-263969.html

[Srivastava2008] Srivastava, P. R. (2008): “Test case prioritization,” Journal of Theoretical and Applied Information Technology , vol. 4, pp. 178-181.

[Korel2007] Korel, B., Koutsogiannakis, G. and Tahat L.H. (2007): “Model-based test prioritization heuristic methods and their evaluation”. Proceedings of the 3rd International Workshop Advances in Model Based Testing, AMOST 2007

[Korel2009] Korel, B. and Koutsogiannakis, G. (2009): “Experimental Comparison of Code-Based and Model-Based Test Prioritization,” IEEE International Conference on Software Testing, Verification, and Validation Workshops.

[Reider2018] Reider, M.; Magnus, S.; Krause, J. (2018): „Feature-based testing by using model synthesis, test generation and parameterizable test prioritization,“ *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* S. 130-137.

[Magnus2017] Magnus, S.; Ruß, T.; Krause, J.; Diedrich, C.: „Modellsynthese für die Testfallgenerierung sowie Testdurchführung unter Nutzung von Methoden zur Netzwerkanalyse,“ in *at-Automatisierungstechnik, Volume 65(1)*, 2017, S. 73-86.

[Krause2011] Krause, J.: Testfallgenerierung aus modellbasierten Systemspezifikationen auf der Basis von Petrinetzentfaltungen, Magdeburg: Shaker Verlag, 2011.