2018 44th Euromicro Conference on Software Engineering and Advanced Applications

Model-Based Testing of Software-Based System Functions

Imke Drave*, Timo Greifenberg*, Steffen Hillemacher*, Stefan Kriebel†,
Matthias Markthaler†, Bernhard Rumpe*, Andreas Wortmann*
* *Software Engineering*, *RWTH Aachen University*, www.se-rwth.de
† *Development Electric Drive*, *BMW Group*, www.bmw.de

***Abstract*—Software is the most crucial innovation driver for cyber-physical systems and validating its correct behavior is crucial to many application domains from automated vehicles to medical systems, smart homes, and smart cities. Typical for these systems is that software errors can cause severe ramifications and can be very costly to fix. Consequently, facilitating a priori software validation and verification is of crucial importance for cyber-physical system development. We hence present a model-based method which facilitates test creation, test maintenance, and improved test coverage in automotive software engineering. To this end, we translate specification models used in the BMW Group's SMArDT engineering methodology into activity diagrams, apply optimizations, and ultimately generate test cases. We realize these transformations using the MontiCore language workbench and report on ongoing evaluation at the BMW Group. First results indicate that model-based test case creation can provide great benefits to all stakeholders, once the initial learning efforts have been mastered.**

Keywords–Model-Based Testing, Test Case Creation

I. Motivation

Engineering software-intensive cyber-physical systems (CPS) is a tremendous challenge that often requires the participation of experts from multiple domains. Such domain experts are rarely software engineering experts. Hence, their contribution is hindered by the conceptual gap between problem domains and the solution domain of software engineering. Model-based software development aims to reduce this gap by lifting models to primary development artifacts. As such, they capture domain concepts and abstract from technical details. As model artifacts, they are better comprehensible than general-purpose programming language artifacts and enable automated analysis and transformation.

Model-based testing (MBT) [1] employs models to generate test cases for software-intensive systems. Thereby, the models themselves describe the behavior of the system under test (system model), its environment (environment model), or a process of possible test steps (test model).

Today, software is the most crucial innovation driver for CPS [2], including autonomous robots, cars, medical systems, smart homes, and smart cities. Typical for CPS is that software errors can cause severe real-world ramifications and - due to the entanglement of hardware and software - can be very costly to fix. Consequently, facilitating an early validation and verification is of crucial importance to CPS development. At the BMW Group, MBT has become a mean of tackling this issue. Besides, by deriving test cases from model-based specifications, test engineers can benefit from automated test case creation. Therefore, we developed a method for *Model-Based Test Case Creation* (MBTCC) [3], based on a survey, our experience and a close cooperation with test engineers. In this paper, we present the detailed MBTCC method that enables automatic generation of test cases from a system model for verification of system behavior. Our method facilitates test creation, test maintenance, and improved test coverage for automotive software. To this end, we present a method to transform the BMW Group's SMArDT [4] (specification method for requirements, design, and test) SysML [5] behavior models into executable test cases. For efficient model transformation and manipulation, the input models are translated into UML/P activity diagrams (AD) [6]. With the UML/P infrastructure for model analysis and transformation already in place, engineering the required code generator and integrating additional transformations requires little effort. Hence, the contributions of this paper are

- A transformation from UML/P ADs into system level test cases which can be executed on various test beds.
- A transformation from UML/P SCs into UML/P ADs which enables MBTCC using SCs.
- A tool-chain to efficiently generate test cases for automotive software.

In the following, Section II sketches the outline of our method and its integration into the SMArDT methodology, before Section III introduces preliminaries. Afterwards, Section IV describes a method to derive test cases from UML/P ADs. Section V leverages this to derive test cases from SCs containing embedded ADs. Section VI presents lessons learned from applying MBTCC, Section VII discusses our method, and Section VIII highlights related work. Section IX concludes.

II. Overview

This section highlights the MBTCC processes at the electric drive development department at the BMW Group. The department uses SMArDT [4], [7], a specification technique for requirement, design, and testing of systems engineering artifacts which fulfils ISO 26262. SMArDT applies modeling techniques to describe functional behavior by means of SCs and ADs. Moreover, the method supports a combination of both, in which ADs describe entry, do, and exit actions of SC states.

The MBTCC method targets the system level of the SMArDT development process, *i.e.,* it aims at specifying the system's functional behavior and its high-level architecture. Prior to MBTCC establishment, test cases at the system level

978-1-5386-7383-6/18/$31.00 ©2018 IEEE
DOI 10.1109/SEAA.2018.00032

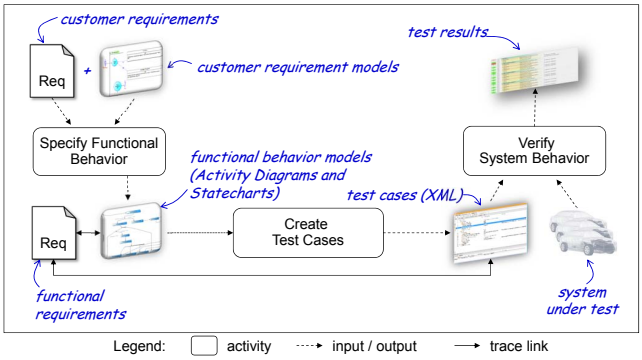

Figure 1. Overview over the overall test case creation process.

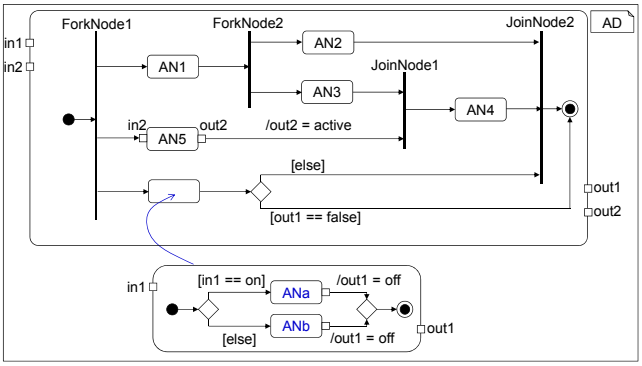

Figure 2. UML/P activity diagram presenting quintessential AD elements.

were derived manually from requirements. With SMArDT at hand, high quality function models are created as part of the system requirements. Leveraging the formal nature of the models enhances automated test case creation and yields several advantages: (1) Test cases can be generated according to the path coverage criterion. The generator produces a minimal number of test cases fulfilling the coverage criterion. This allows for systematic testing without overhead by executing redundant test cases. (2) Test engineers can focus on creating the most complex test cases, which are not available as function models or cannot be created systematically; and (3) automated MBTCC is faster than manual test case creation. Hence, test cases derived from the latest development model are instantly available independent of the state of the process and thus, errors can be fixed at an early stage.

Figure 1 illustrates the quintessential activities and artifacts of MBTCC. First, the functional behavior models are derived from customer requirements. They specify the black-box behavior of the system, from a customer's point of view. Requirement engineers utilize ADs, SCs and Use Case Diagrams for costumer requirement models. Among others, SMArDT proposes ADs and SCs for functional specifications. A functional behavior specification describes the activities, actions, and states needed to fulfill customer requirements. For example, a functional behavior model describes how a car fulfills the customer requirement for a superior driving experience. Moreover, functional behavior models describe how a car fulfills its functional requirements. These requirements are linked to the corresponding models. Both, ADs and SCs, can be used as input for MBTCC, which derives XML test cases. Linking test cases to the functional requirements enables horizontal traceability. For validating and and verifying system behavior, the test case execution tool runs the test cases on various test beds (system under test). The horizontal traceability allows to backtrace test results to their corresponding functional requirements.

The test case creation takes ADs as input. The respective model processing tool-chain comprises various model-to-model (M2M) transformations and a model-to-text (M2T) transformation from ADs to test cases, which is described in Section IV. Afterwards, Section V describes an M2M transformation from SCs to ADs. This supports integrated model processing of both models, which reduces the number of different tools necessary to generate test cases.

III. Preliminaries

Our MBTCC method takes advantage of the benefits of the MontiCore language workbench by applying its already implemented UML/P AD language. The following two sections introduce basics about the workbench and UML/P ADs.

A. MontiCore

MontiCore [8], [9] is an extensible workbench for engineering compositional, textual modeling languages. Developers define languages as context-free grammars that specify integrated concrete and abstract syntax. MontiCore generates model-processing infrastructure (*e.g.,* parsers) with respect to the grammar to facilitate model checking and transformation of the language into code. Various MontiCore languages have been engineered for and applied to different domains[1], including automotive, cloud computing, smart homes, robotics, and software engineering itself. For the latter, we have developed the UML/P family of modeling languages [10], which is a subset of UML [11] that is refined to enable pervasive model-driven engineering without the underspecification discrepancies of UML. This subset includes activity diagrams, class diagrams, statecharts, and sequence diagrams. All of them are well-integrated with another and come with comprehensive tooling (model checkers, model transformations, code generators, *etc.*).

B. Activity Diagrams

To leverage the benefits of MontiCore's model processing infrastructure, we chose to preprocess test case generation by transforming the input SMArDT ADs into UML/P ADs. This step enables to reuse tooling, such as model checkers and transformations. Figure 2 shows an UML/P AD example including all quintessential modeling elements. We describe MBTCC for ADs, however, application to SCs is also possible by a translation into an AD (*cf.* Section V).

For automotive software testing, it is crucial to validate execution paths. As we deal with UML/P ADs, *i.e.,* directed graphs, it is possible to determine these paths by graph traversal. In the following, we therefore consider an execution path to be a path of the UML/P AD, which is a connected sequence of nodes and edges starting with the initial node and ending at the final node. A path may be split into concurrent

[1]See http://monticore.de/languages/

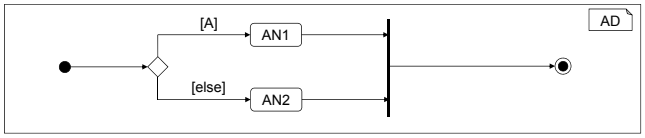

Figure 3. Deadlock: Exactly one token is placed at the initial node and no elements create additional tokens. The join node waits until two tokens arrive but will only ever receive one token. Thus it can never continue.

sub-paths by fork nodes. The sub-paths can be synchronized by a join node or ended by a flow-final node. If one sub-path of a concurrent region reaches the final node, all other concurrent paths are aborted.

We need to impose strict well-formedness constraints on the input diagrams to ensure correct path generation. Among others, we require that SMArDT ADs are free of infinite loops and deadlocks. The former occur whenever a node is reachable from itself and no exit condition to leave the loop is given. Finite loops, on the other hand, are allowed in MBTCC. The latter arise in cyclic waiting situations, as illustrated in Figure 3. The join node will continue iff two incoming flows have arrived [12]. The absence of deadlocks and infinite loops is required for the transformations performed on the ADs.

In SMArDT ADs which model functional behavior use ports to define input and output signals for system functions. Furthermore, local signals which are neither input nor output and are only processed within a specific function of the system can be defined. Transitions may hold guards in the form of Boolean conditions, *i.e.,* `[expression]`, or signal assignments that assign values to output or local signals, *i.e.,* `/signal = value`. A combination of both is also possible, *i.e.,* `[expression] /signal = value`. For MBTCC, we prescribe that the values of input signals are not changed within the diagram.

We define the following AD correctness constraints: (1) Names and IDs of nodes and edges must be unique. (2) Diagrams may not contain deadlocks that occur in cyclic waiting situations. Figure 3 illustrates an example where a join node will continue if and only if two incoming flows have arrived [12]. (3) Infinite loops cannot prevent path calculation. They occur, whenever a node can be reached from itself and no exit condition will ever come true. (4) Input signals values must not be changed on any path of an AD. (5) Edges originating from a `DecisionNode` need to always specify a Boolean guard condition. (6) Branches must always be complete, *i.e.,* its guard conditions must cover the diagram's entire value range. This is necessary for path validation because branching is not decidable otherwise.

The above-mentioned correctness constraints are implemented as context conditions and checked prior to code generation. Consequently, erroneous models will not be released for implementation before all guidelines are passed.

IV. From ADs to Test Cases

Figure 4 shows the steps of the automated MBTCC process. First, two transformations are applied to simplify the diagram and thus, enable easy path calculation. The optional value collection step gathers all signals and possible values used in

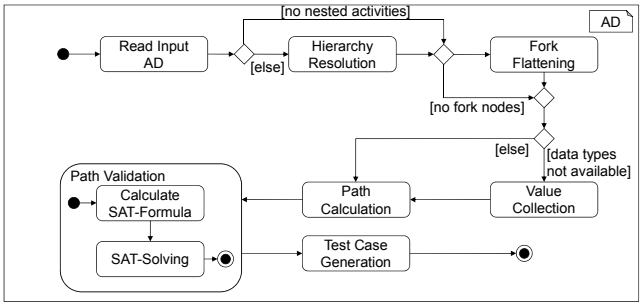

Figure 4. AD showing the intermediate steps of the automatic test case generation implemented by the test case generator.

the diagrams. Thus, it defines the data types used for path validation. This step is only required when data types of the signals are not available during the generation process. Path calculation is performed according to the path coverage criterion C2c [13] to prevent state state explosion. The following path validation step checks whether an assignment of values to input variables exists such that a given path will be executed. If no such assignment exists, the respective path is declared invalid and discarded. In the end, test cases are created for each valid path using the calculated values for input signals. To illustrate the test case generation process, the diagram shown in Figure 2 will serve as running example throughout this chapter.

A. Hierarchy Resolution and Fork Flattening

We perform two transformation steps, Hierarchy Resolution and Fork Flattening, to enable a simplified path calculation algorithm later on. First, the Hierarchy Resolution resolves nested activities by integrating them into the uppermost activity. The Fork Flattening transformation interprets parallel flows modeled by fork nodes. As they simulate concurrent flows in ADs, their order is interchangeable as long as the sequence of each individual flow is preserved. Additionally, we assume that parallel paths are linearly independent. The high degree of abstraction of the diagrams allows to make this assumption. Therefore, we apply a sequential arbitrary order interpretation of concurrency to obtain a single flow. In the following, we perform both transformations using the example AD of Figure 2. The AD also contains a complex fork-join structure with inner branching behavior which is valid according to SysML although forked flows are not joined at exactly one join node. SMArDT ADs often contain such constructs and hence, need to be handled correctly by MBTCC.

Hierarchy Resolution resolves nested activities by removing the initial node of a nested diagram and adding a new edge from the last node in the parent diagram to the first node of the nested. Equally, it removes the final node of the nested diagram and connects the last node of the nested activity to the first activity after the nested node in the parent diagram. Multiple hierarchy levels are resolved recursively. Figure 5 depicts this step for the running example AD.

The Fork Flattening transformation applies a sequential arbitrary order interpretation for parallel flows. For nested fork nodes, it starts at the innermost fork node. Starting at the first encountered fork node, the transformation traverses

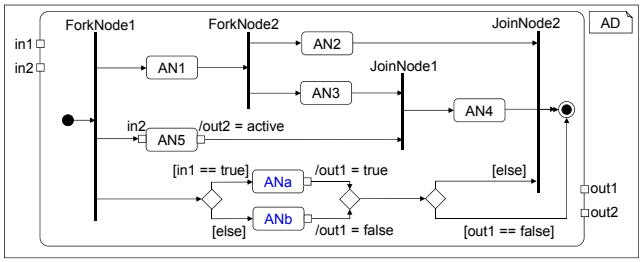

Figure 5. The UML/P activity diagram of Figure 2 after the first step, the hierarchy resolution transformation.

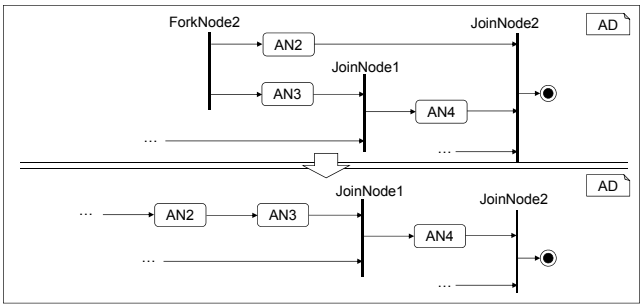

Figure 6. Excerpt of the first intermediate fork flattening step. Based on the UML/P AD of Figure 2 the `ForkNode2` is transformed into a basic node. The two forked paths are connected at their common join node `JoinNode2`

.

the AD recursively until it finds the innermost fork node. All parallel paths originating from this node are calculated. The transformation takes two parallel paths and connects them at their first common join node until only one path is left. UML/P AD semantics prescribe that a path starts at the initial node and ends at the final node. Hence, in a well-formed AD, parallel paths must be synchronized at some join node prior to the final node. However, if no such node can be found, either one of the parallel paths must end at a flow final node or both must end at the final node. In the first case, the flow final node simply marks the end of that forked path. Therefore, it can be connected to the other by changing the target of its last edge to the first node of the other path. The second case is handled similarly by changing the target of the last edge of either one of the parallel paths. The final or a flow final node, therefore, may also be taken as a synchronization point.

Considering the example in Figure 2, the transformation flattens `ForkNode2` first as it is the innermost fork node of the AD (*cf.* Figure 6). Flattening is possible at `JoinNode1` by changing the target of the edge from `AN2` to `AN3` and removing the edge from `ForkNode2` to `AN2`.

Flattening continues at `ForkNode1`. As the bottommost fork also contains a branch which is not merged, this fork will be considered last in the flattening process. We find the first common join node of the other two forks at `JoinNode2` and connect them in the same manner as before, illustrated in Figure 7. This leaves two forks (*cf.* Figure 8, top). One branch of the decision node shares `JoinNode2` with another fork while the other edge of the branch leads to the final node which will end all running threads immediately. Thus, it is

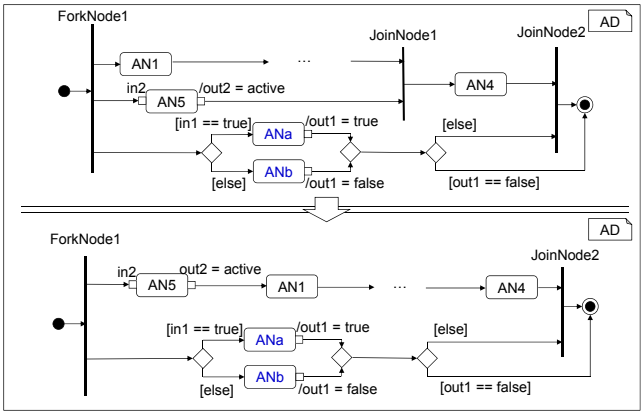

Figure 7. Excerpt of the second intermediate fork flattening step. `ForkNode1` is flattened.

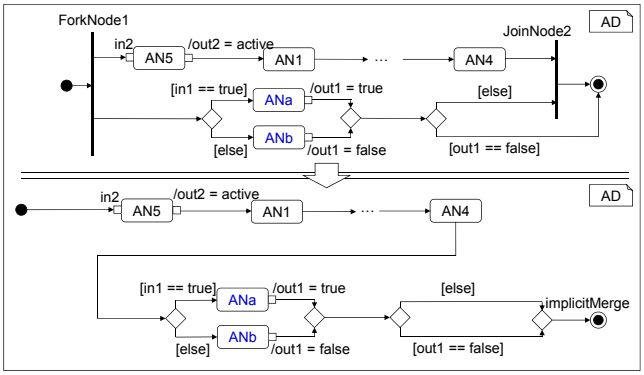

Figure 8. Excerpt of the resulting fork flattening process, resolving `ForkNode1` and `JoinNode2`.

correct to connect node `AN4` to the decision node, as shown in Figure 8.

After performing both transformations, nested activities, join and fork nodes have been eliminated, which allows presuming with the test case generation.

B. Value Collection, Path Calculation and Validation

The next step of the MBTCC process is the optional value collection. This step collects signals and possible values to calculate data types (*i.e.,* possible ranges of values) for signals. This compensates for unspecified data types of signals in input ADs. In SMArDT, ADs may contain signals for which no value specification in a global system scope is present due to the high abstraction level. Consequently, the generator creates test cases based only on values present within the input diagram. The value collection collects all possible values for each of the diagram's signals, by traversing the diagram. During this traversal, ports, guards, and assignments are checked for the needed information. For signals, the set of possible values is complemented by the corresponding value once the variable appears in a guard or an assignment. In case a signal is assigned to another signal, *e.g.,*, `signalA = signalB`, the data-type of `signalA` is extended by that of `signalB`. If a data-type contains only one element,

Variable	Actual Value Range	Negation
in1	true	false
in2		
out1	true, false	
out2	active	not_active

TABLE I. Result of the value collection shown in Figure 2.

the algorithm automatically adds a negated value, which is necessary to handle [else] guards in a correct way.

For Figure 2, the value collection returns the results presented in Table I. The diagram contains a single guard condition that holds whenever in1 is set to true. The value collection adds this value to the value range of in1. In the end, it also appends the negation false, as only one value for in1 is present in the diagram. As in2 is not used within a guard or assignment, its value range remains empty and the generator warns about an unused input. For out1, there are two values present in the diagram, hence no negations are added by the value collection. For out2 the procedure generates the value range active and adds not_active as its negation.

Next, paths are calculated according to a path coverage strategy. A Depth First Search graph traversal algorithm, retrieves all paths of the graph. Test cases are only generated for valid paths. An AD path is valid if all its guard conditions can be satisfied. Checking whether a signal interpretation exists, such that a path is valid corresponds to a Boolean satisfiability problem. The Boolean formula to be satisfied is given by the conjunction of all guard conditions and assignments of a path. If the value of a signal is changed multiple times on a path, this is handled by creating multiple versions of the signal for each assignment. The Boolean formula is built up according to the version of the signal along the specific path. Hence, after changing the value of a signal, the rest of the formula is built by substituting each occurrence of the variable by its new version. We use the Z3 solver [14] to solve the problem, which outputs a value for each variable used in the formula.

In Figure 2, the diagram contains four paths. The upper path will be taken iff in1 == true and out1 != false. The expected values for the output variables out1 and out2 can be retrieved by evaluating the actions in ANa, which sets out1 to true and AN5 setting out2 to active.

C. Test Cases

Test cases always correspond to one execution path of the activity diagram and are structured in three blocks containing preconditions, actions, and postconditions, which is common practice in software testing [15].

Within the preconditions, additional conditions required before executing the test case can be specified. The test case will not be executable if the precondition does not hold. Postconditions contain test steps to check value assignments to output signals. Specific pre- and postconditions are not derived from the diagrams, but can be added manually after the generation process finished. The action block will contain test steps to set all calculated input values and to check the assignments of intermediate results. Thus, each of the test steps comprises an access action, *i.e.,* setting or reading signal information, the signal name and its current or expected value.

Test Case 1

Action	Name	Value	Expectation
set	out2		active
set	in1	true	
set	out1	true	
check	out1		true

Test Case 2

Action	Name	Value	Expectation
set	out2		active
set	in1	false	
set	out1	false	
check	out1		false

TABLE II. Result of the test case generation. It belongs to the two valid paths in the AD shown in Figure 2.

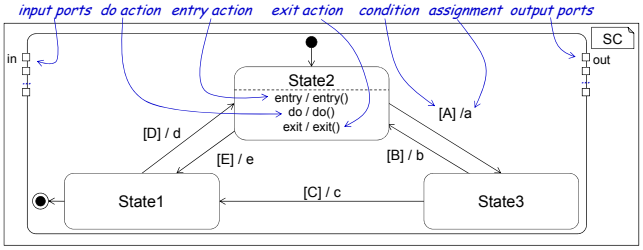

Figure 9. SC to be transformed into an AD.

The test case can be expressed in a tabular format, which is parsed into XML.

For the AD in Figure 8, two test cases will be generated, as the diagram comprises only two valid paths. The first path starts with AN5 after the initial node, which sets the output variable out2 to its value active. At the first decision node, either in1==true or in1!=true must hold. The conditions are checked in one test case each. The tables of Table II show the resulting test cases in a simplified way. Each of their rows represents one test step executable on several test beds. The other two paths have been marked invalid. Since in case in1 == true holds, the else condition of the following decision never holds. Analogously, if in == false, out1 == false can never be true.

V. From SCs To Test Cases

In addition to ADs, SMArDT allows SCs to describe functional behavior to lay focus on different states of a system and their transitions. We therefore designed our MBTCC method such that it is also applicable to SCs. In contrast to regarding ADs and SCs separately, our approach also supports a combination of both model types. This is advantageous because modelers can integrate specific purposes of both diagram types. While SCs describe the states of a system with its transitions, developers can use ADs to describe the internal behavior of the states. This enables a more detailed function behavior modeling than just using one of the diagram types.

Figure 9 shows an example of an SC that contains all quintessential modeling elements. The SC comprises input ports and output ports, states and transitions. Each state can feature an *entry*, *do*, and an *exit* action. These actions can either be assignments, analogous to signal assignments as described in Section IV, or contain internal ADs. Similar to edges and guard conditions in ADs, a transition in an SC holds

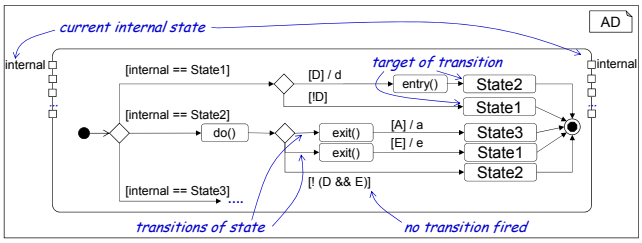

Figure 10. AD generated from the SC of Figure 9.

a condition described inside brackets (*cf.* Figure 9) and allows further assignments.

Depending on the current state and the conditions of its outgoing transitions, the SC determines which transition fires. We assume that outgoing transitions of a state are always deterministic, *i.e.,* it is not possible for any two transitions to fire at the same time. To reuse the AD-MBTCC approach presented above, we transform SCs to ADs prior to the actual test case generation. This also facilitates test case generation for combinations of both diagrams. Figure 10 shows the AD derived from the SC of Figure 9. The figure depicts the representation of states and transitions of the SC within the AD. The main idea of this transformation is to store the information about the current internal state of the SC in the AD. Therefore, the AD contains designated ports to determine the current state of the SC. The new ports are denoted `internal` (*cf.* Figure 10).

Depending on the current state of the SC, different paths through the transformed AD are possible. That is, the paths through the AD represent the transitions of each state of the SC. The transformation from SC to AD also takes *entry*, *do* and *exit* actions of each state into account and adds them to the transformed AD. First, the transformation adds a decision node with an outgoing edge for every state of the SC. Each outgoing edge of this decision uses the `internal` port in its condition to determine the current SC state in the AD. Afterwards, the transformation adds the *do* action of each state to the corresponding edge in the AD. Next, the transformation models the outgoing transitions of each state using decision nodes. If the SC transitions into a new state, the *exit* action of the current state, the assignment of the respective transition, and finally the *entry* action of the new state are added. The transformation also takes into consideration that none of the outgoing transitions fires.

When describing state behavior by means of ADs, any of the *entry*, *do* or *exit* actions may be linked to an AD. Furthermore, any information about signals used in the parent SC, including their possible values, is also available in the respective AD. Hence, the nested ADs can influence which outgoing transition of a state fires. This is important when creating a functional model using SCs combined with ADs. Model correctness is given, if signals and their data types are consistent throughout the parent SC, its child ADs and vice versa. This consistency between SCs and ADs ensures the accurate description of the corresponding functional behavior and the correctness of the generated test cases.

The test generation strategy for SCs in our approach is different to the one applied to ADs. For test case generation from SCs, each transition is covered by exactly one test case and each test case covers exactly one transition. As a consequence, sequences of transitions, *i.e.,* paths through the SC, are not regarded.

To illustrate the test strategy for state transitions, we look at the transition from `State2` to `State3` as shown in Figure 9. Assuming that the current state is `State2`, with regard to the AD of Figure 10, the *do* action of this state is covered. If an outgoing transition fires, the *exit* activity of `State2` will also be covered. Subsequently, the current state of the SC changes to `State3` and the respective transition is covered. Finally, in case `State3` contains an *entry* action, it is also covered.

In general, the test case calculation works as follows: (1) Determine the current state of the SC. (2) Cover the *do* action of this state. (3) Depending on its outcome, determine which transition fires. If none can fire, the current state does not change. (4) If one of the outgoing transitions fires, first cover the *exit* action of the current state. Afterwards, cover the corresponding transition. (5) Finally, cover the *entry* action of the target state of the transition.

The test cases derived from an SC cover all transitions and all ADs nested within its states. Thus, in cases where ADs serve as input in addition to SCs, a single transition can be tested by more than one test case. As a consequence, the actual number of generated test cases heavily depends on the nested ADs of the states. Since Section IV already explained test case generation for ADs in more detail, it is easy to see that nested ADs, with a high number of decisions, yield a higher number of test cases generated out of SCs.

VI. Lessons Learned from Applying MBTCC

The close cooperation with the BMW Group broadly enhanced the conception and development of our MBTCC method. Immediate feedback given by practical application of the automatic test case generation by developers showed its advantage in the ongoing development process. In this section, we present the resonance within BMW and our experiences from applying MBTCC.

In its introduction phase MBTCC faced organizational challenges. As specified in BMW's development process, a development branch of the main productive model contains the inputs of for MBTCC. In this branch, elements cannot be linked to their corresponding requirements, as this would increase maintenance efforts. This prevents automatic linking of the generated test cases to the respective requirements which would be necessary for regular reports on test coverage of requirements. BMW plans to integrate models into the main productive model as soon as a first draft has been accepted by a committee of responsible authorities and test engineers. On the main branch, links are maintained which enables the test case generator and automatic requirement test coverage reports.

As mentioned before, current input models do not contain complete data-type information for all signals. We therefore developed the value collection workaround. However, when considering individual points of view, this step might not mirror what is intended by the modeler. An example is given by

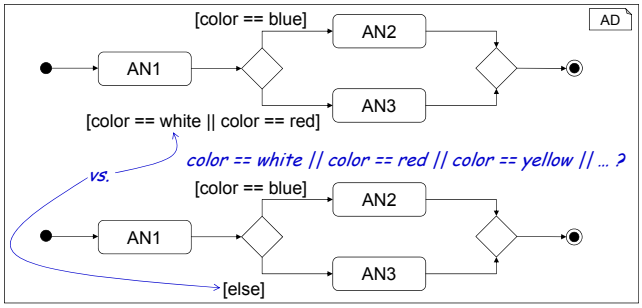

Figure 11. Problem of condition understanding: the generator's proposed value collection (`color==white || color==red`) versus modeler's point of view (`else`).

Figure 11 which shows two similar activity diagrams. To the domain expert modeler, the expression `color == white || color == red` is equivalent to `else`. However, since the diagram is not complemented by `color`'s data-type, the test case generator will assume it to be fully represented by the set `blue, not_blue`. In a realistic test, however, the system's behavior should be tested in case `color == white` and `color == red`. Instead of three test cases, the generator is only able to produce two where one of them puts a value, that is not realistic due to the lack of data-type specification. This challenge can be overcome by establishing and maintaining a central database containing signals and their data-types. Additionally, such a database would also facilitate maintenance of diagram consistency within SMArDT.

MBTCC application at BMW faces other challenges in providing support for platform specifications. Different test platforms may require specific pre- and postconditions which are valid only for a certain pair of system and test platform. For example, conditions for *Vehicle in the Loop* or hardware system information for *Hardware in the Loop* still need to be added manually by test engineer. Providing this information together with the productive model would enable automatic processing of test platform and system specific pre- and postconditions.

Within the BMW group, our method has been perceived positively among engineers. Current experiences show that semi-automatic MBTCC as proposed here is capable of reducing test case creation times and whenever the linked requirement set is complete and well-formedness is ensured. The generator validates input models by means of efficient model checking and thus, reveals modeling mistakes which are not visible to the human's eye even if it belongs to an experienced modeler or test engineer.

VII. Discussion

The hierarchy resolution and fork flattening transformations (*cf.* Section IV) require strict adherence to the correctness constraints described in Section III, *i.e.,* logical correctness needs to be assured and is therefore checked prior to test case generation. The generator rejects models which are not well-formed, which is a major advantage because even invisible modeling errors can be detected at an early stage of the development process. Within SMArDT, erroneous models can be detected at its second stage, *i.e.,* before test cases are generated. Thus, no testing activities are required to detect logical mistakes which might not have been detected until later in the system development process without MBTCC. Currently, our tool-chain only supports flat SCs without nested or concurrent states. The limited support for only basic SC elements suffices to describe function models in MBTCC. Generally, as discussed in [10], it is possible to transform SCs into a reduced form which only contains the basic elements the prototype already supports. This transformation can be done without the loss of information. Hence, the application of this pre-existing transformation enables the prototype to support all UML/P statecharts if needed. Also, our approach for SCs supports test case generation based on single transitions only. While this suffices for current SMArDT function models, we also look into extending the test case generation to full path coverage. As a first step, we plan to generate test cases for successive transitions.

VIII. Related Work

Most approaches for MBTCC focus on deriving test cases from a specific diagram type only. Some studies also generate test cases from ADs [16], [17]. The authors of [16] present a direct technique, which ignores the state of objects during system execution. The method only regards external input and output signals, but disregards internal or local signals. Therefore, the generated test cases do not allow to investigate system behavior in case of internal errors.

Generating test cases from SCs has also been subject to current research. As such [18], [19], [20] have developed approaches based on a transformation of SCs into extended finite state machines (EFSM) [21], [18]: The transformation proposed by [21] eliminates events after broadcast and flattens hierarchical and concurrent structures of states. Test case generation based on this transformation targets class level testing only and is not suitable for system level testing. The approach proposed by [18] translates an EFSM into an extended flow control graph prior to the generation process. Incomplete guard conditions are handled by changing the field variable or choosing appropriate parameter values obtained from a database. The approach does not overcome the data type underspecifications as present in our input models and is thus not applicable. The approach presented here, is able to overcome data type underspecification in an efficient and understandable way.

The MBTCC method proposed here is able to handle ADs and SCs as well as a combinations of both which is not addressed by existing research. Using different diagram types within SMArDT allows to leverage their strengths to model specific system attributes and behavior. Thus, models are created to be more comprehensible, less complex and to contain clear system specifications.

Approaches which leverage UML diagram syntheses for test case generation exist, as presented in [22], [23]. Another recent approach takes ADs combined with sequence diagrams as input [22]. The approach takes the AD and transforms it to a directed graph and the sequence diagram into a sequence graph. A combined activity sequence graph is taken as source for the generated test cases. [22] enables concurrency testing in activity and sequence diagrams separately which would not be able to handle concurrent joining constructs as presented here.

Such structures, however, conform to the SysML specification and should therefore not be prohibited to preserve modeling liberties. Furthermore, the approach presented here, is able to handle `else` guard conditions by complementing incomplete data types.

Transforming integrated ADs and SCs into the intermediate representation of state activity diagrams (SADs) prior to test case generation [23] is closely related to MBTCC. Based on the resulting SAD, path operations and test case generation according to coverage criteria can be executed. In contrast to [23] for cluster-level and integration-level testing of software, our approach targets testing on a level where software and hardware are already combined in a real-world system or subsystem. The test cases in [23] are derived from a fault model whereas our approach takes the actual productive system model as input.

IX. Conclusion

We have presented a systematic method to derive test cases for automotive software from specifications already present through the BMW Group's SMArDT method. As these specifications can be statecharts with conceptually embedded activity diagrams, they are homogenized into UML/P activity diagrams first. Afterwards, the resulting activity diagrams are optimized and transformed into a tabular, XML-based format providing the individual test actions and variable valuations in an executable format. Using a modular transformation tool-chain based on the language workbench MontiCore, our approach is easily extensible for future challenges in automotive software testing and first results from evaluating it at the BMW Group indicate that all stakeholders expect it to deliver great benefits in test case creation.

References

[1] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in Proceedings of the 21st international conference on Software engineering. ACM, 1999, pp. 285–294.

[2] M. Broy, "Challenges in Automotive Software Engineering," in Proceedings of ICSE 2006, 2006.

[3] S. Kriebel, M. Markthaler, K. S. Salman, T. Greifenberg, S. Hillemacher, B. Rumpe, C. Schulze, A. Wortmann, P. Orth, and J. Richenhagen, "Improving model-based testing in automotive software engineering," in 40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, 2018.

[4] Stefan Kriebel, Vincent Moyses, Georg Strobl, Johannes Richenhagen, Philipp Orth, Stefan Pischinger, Christoph Schulze, Timo Greifenberg, Bernhard Rumpe, "The next generation of bmw's electrified powertrains: Providing software features quickly by model-based system design," 26th Aachen Colloquium Automobile and Engine Technology, 2017.

[5] S. Friedenthal, A. Moore, and R. Steiner, A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann, 2011.

[6] H. Grönniger, D. Reiß, and B. Rumpe, "Towards a Semantics of Activity Diagrams with Semantic Variation Points," in Conference on Model Driven Engineering Languages and Systems (MODELS'10), ser. LNCS 6394. Springer, 2010, pp. 331–345.

[7] S. Hillemacher, S. Kriebel, E. Kusmenko, M. Lorang, B. Rumpe, A. Sema, G. Strobl, and M. von Wenckstern, "Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology," in Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18). SciTePress, 2018, pp. 163 – 178.

[8] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann, "Composition of Heterogeneous Modeling Languages," in Model-Driven Engineering and Software Development, ser. Communications in Computer and Information Science, vol. 580. Springer, 2015, pp. 45–66.

[9] R. Heim, P. Mir Seyed Nazari, B. Rumpe, and A. Wortmann, "Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors," in Conference on Modelling Foundations and Applications (ECMFA), 2016.

[10] B. Rumpe, Modeling with UML: Language, Concepts, Methods. Springer International, 2016.

[11] O. M. Group, "OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03)," 2010.

[12] P. Sugunnasil, "Detecting deadlock in activity diagram using process automata," in 2016 International Computer Science and Engineering Conference (ICSEC), 2016, pp. 1–6.

[13] P. Liggesmeyer, Software-Qualität: Testen, Analysieren und Verifizieren von Software. Springer Science & Business Media, 2009.

[14] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008, pp. 337–340.

[15] A. Spillner and T. Linz, Basiswissen Softwaretest. dpunkt, 2012.

[16] Generating test cases from UML activity diagram based on Gray-box method: 11th Asia-Pacific Software Engineering Conference, 2004.

[17] H. Kim, S. Kang, J. Baik, and I. Ko, "Test cases generation from uml activity diagrams," in Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007). IEEE, 2007, pp. 556–561.

[18] M. Shirole, A. Suthar, and R. Kumar, "Generation of Improved Test Cases from UML State Diagram Using Genetic Algorithm," in Proceedings of the 4th India Software Engineering Conference. ACM, 2011, pp. 125–134.

[19] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in UML 99 — The Unified Modeling Language: Beyond the Standard Second International Conference Fort Collins, CO, USA, October 28–30, 1999 Proceedings. Springer Berlin Heidelberg, 1999, pp. 416–429.

[20] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," Software Testing, Verification and Reliability, 2003, pp. 25–53.

[21] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," IEE Proceedings - Software, 1999, p. 187.

[22] Monalisha Khandai, Arup Abhinna Acharya, and Durga Prasad Mohapatra, "Test case generation for concurrent system using uml combinational diagram," International Journal of Computer Science and Information Technologies, 2011, pp. 1172–1181.

[23] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test case generation based on state and activity models," The Journal of Object Technology, 2010.