View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by OPUS Augsburg

Towards Abstract Test Execution in Early Stages of Model-driven Software Development

Noël Hagemann[a], Reinhard Pröll[b] and Bernhard Bauer[c]

Software Methodologies for Distributed Systems, University of Augsburg, Augsburg, Germany

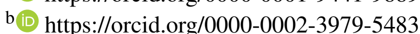
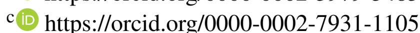
Keywords: Test Execution, Model-based Testing, Domain-specific Modeling, Model-driven Software Development.

Abstract: Over the last decades, systems immanent complexity has significantly increased. In order to cope with the emerging challenges during the development of such systems, modeling approaches become an indispensable part. While many process steps are applicable to the model-level, there are no sufficient realizations for test execution yet. As a result, we present a semi-formal approach enabling developers to perform abstract test execution straight on the modeled artifacts to support the overarching objective of a *shift left* of verification and validation tasks. Our concept challenges an abstract test case (derived from test model) against a system model utilizing an integrated set of domain-specific models, i.e. the omni model. Driven by an optimistic dataflow analysis based on a combined view of an abstract test case and its triggered system behavior, possible test verdicts are assigned. Based on a prototypical implementation of the concept, the proof of concept is demonstrated and further on put in the context of related research.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

The steadily raising complexity of application software may hardly be tackled by traditional development techniques. In order to reduce the complexity, the concepts of abstraction and automation are used in many development phases. While concepts and tools in the areas of executable modeling languages, model simulation, formal verification and model-based testing (MBT) show promising results on the way towards model-centric development methodologies, the main focus is on generating code from these models. Apart from formal approaches for verification, none of the mentioned techniques provides any significant verification steps during development until the model-to-code transformation is performed, either in a manual or automated way. It is a well-known fact, that faults introduced in early stages of development, demand significantly more money and time for fixing, than faults induced in later phases (Planning, 2002) (Galin, 2004). Further, Jones et al. gave insights about the most prominent development phases, where defects are revealed, namely the late testing phases (Jones, 2008).

[a] https://orcid.org/0000-0001-9441-9889

[b] https://orcid.org/0000-0002-3979-5483

[c] https://orcid.org/0000-0002-7931-1105

1.1 Problem Statement

Based on the insight on impacts of design time faults together with the steadily raising complexity of today's software, we follow a real shift left of verification and validation (V & V) activities in model-driven development processes. Apart from formal verification approaches like model checking, and informal techniques like reviews, which most of the time close the gap between specification and code, up to our knowledge there is no semi-formal and (semi-) automated technique for early stages of model-driven software development (MDSD). Therefore, we see a strong need for a semi-formal approach to perform abstract test execution in model-driven development processes.

In order to achieve this ambitious goal, we combine concepts of model-based testing with model interpretation mechanisms powered by dataflow analysis on models. Starting from an integrated model basis, made up of domain-specific models used during software and test development together with an integration component, the combined dataflow is analyzed. This means the model artifacts specified in the constructive phases are linked with the test model to perform an abstract execution of test cases. Depending on the level of concreteness and completeness of model artifacts, the analysis results may ei-

Hagemann, N., Pröll, R. and Bauer, B.
Towards Abstract Test Execution in Early Stages of Model-driven Software Development.
DOI: 10.5220/0008934802160226
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 216-226
ISBN: 978-989-758-400-8
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

ther be seen as a problem indication during ongoing modeling work or as a first verification step before code-based processing. In a more general context, the possibility to evaluate abstract test case specifications against models of the system may fit into the context of a model-based software testing lifecycle as sketched in earlier work of Pröll et al. (Pröll and Bauer, 2018b).

1.2 Outline

Starting off with Section 2, basic terms, definitions, and a case study are given in order to narrow down the context of our work. Based on these basic building blocks, Section 3 presents the core concept for the abstract test execution, which splits up into three phases featuring the major processing steps of our approach. Based on the case study and a prototypical implementation, a proof of concept for our approach is done (Section 4). Therefore, Section 4.1 introduces the setup, before the results are presented in Section 4.2. In order to be able to classify the presented approach in the context of other approaches, related work is presented in Section 5. Finally, Section 6 draws a conclusion and gives an outlook on future work in this area.

2 PREPARATORY STEPS

The term *domain* is used in multiple contexts, with different semantics. In this paper the term domain is used in a sense of a technical domain of a model-driven development approach, like requirements modeling or safety engineering, which may also be carried out in a model-based fashion.

For MDSD as well as MBT the number of domain-specific modeling languages steadily raises. Apart from the specific language constructs, all the MDSD approaches share the goal of ending up with an adequate specification of the System Under Development (SUD), i.e. System Under Test (SUT). We call this specification artifact the *system model*.

In parallel, the testing domain makes use of descriptions of the intended system behavior to subsequently derive tests for the SUD in an automated way. Such descriptions may either be integrated in the system model or modeled separately. Both possibilities have pros and cons (see Pretschner et al. (Pretschner and Philipps, 2005)). Having in mind the trend towards automation of error-prone steps, e.g. replacing the manual transformation of specifications to code with code generators, we decided to follow a separated model approach. Therefore, the term *test model* represents a model artifact apart from the system model, derived from a shared set of requirements with the intention to generate test cases for subsequent V & V tasks (Apfelbaum and Doyle, 1997).

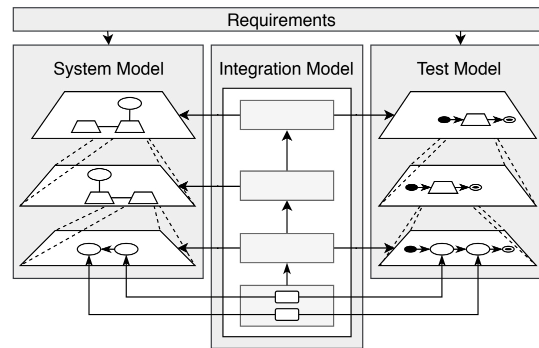


Figure 1: Orchestration of domain-specific models.

The system model as well as the test model specify the actual and intended sequences of actions. As originally defined by Apfelbaum et al., a common concept is the so called *path*, determining “a sequence of events or actions that traverse through the model” (Apfelbaum and Doyle, 1997). The intended sequences of actions derived from a test model, i.e. paths, are better known as *test cases*. Closely linked to the path concept, the so called *guards* are an inevitably contained concept for conditional parts of the specified system model or its test counterpart.

2.1 Metamodel Concepts

The separation of the system and test model, demands for additional concepts managing the interaction of models across multiple domains. Introduced by Proell et al. (Pröll et al., 2017), the *omni model* approach deals with the orchestration of a flexible set of model artifacts. Figure 1 illustrates the role of the model artifact bridging the conceptual gap, namely the *integration model*.

2.1.1 Integration Model

Basically, the integration model is designed for two main purposes. On the one hand, the model artifact should give the modeler the possibility to specify a coarse sketch of how the SUD may be hierarchically decomposed into its basic building blocks. The iterative alignment of the hierarchical decompositions defined in the integration model as well as the system model allows us to maintain model mappings without affecting the original system model, in a sense of separation of concerns. Further, it gives the flexibility to manage varying levels of granularity and interpretation across the domain-specific model artifacts.

On the other hand, the integration model includes concepts for the mapping of abstract states of behavioral models across domains (lower part of Figure 1). Thereby, model artifacts included in a certain path of the system model may explicitly be mapped to elements of a test case, derived from the respective test model. Especially in scenarios, where system and test modeling is carried out on different domain-specific languages the mapping of concepts cannot be achieved automatically. All in all, this allows us to perform an abstract execution of a test case, which is detailed in Section 3.

Depending on the applied development methodology, the creation and maintenance of the integration model artifacts may either be achieved in an automated fashion or requires some manual modeling by the respective systems engineer. Especially, the maintenance of the mapping information across modeling domains demands for manual adjustments, in case of separated models for system and test modeling. The extra work to be done here, is expected to pay off during test iterations on the model-level, revealing conceptual defects of the SUD and thereby drastically reducing the cost for fixing defects.

2.1.2 System Model

Derived from an initial requirements model (see Figure 1), there are many domain-specific and general-purpose modeling languages, which serve the purpose of MDSD. For example, the Unified Modeling Language (UML) represents the most prominent general-purpose modeling language used in software development (OMG, 2011). Being published as a standard alongside the Meta Object Facility (MOF), it gained popularity and therefore is widely known (OMG, 2002). Due to the huge amount of modeling capabilities and its vague semantics, many profiles and subsets for specific needs have emerged.

All these modeling languages have in common, that they share concepts for the specification of structural as well as behavioral descriptions of the SUD. Thereby, the palette for modeling behavior may further be categorized into concepts, either *verifying* or *modifying* the current system state. For example, the case study analyzed in Section 4 uses UML state charts to specify its behavior. Therein, state nodes' actions represent *modifying* model elements, while state transitions with annotated guards mark a *verifying* concept.

2.1.3 Test Model

The same holds for the test domain, where many modeling languages are usable or at least adaptable for test specification tasks. SysML is one possibility for modeling V & V concepts of a SUD. Further, there is an UML profile named UML Testing Profile (UTP), which was explicitly designed to serve as a modeling profile for MBT activities (Object Management Group (OMG), 2004). Throughout the case study, we use a reduced and customized version of UML activity charts to specify the test models, taking the core concepts of UTP and SysML into account. Thereby, the activity elements are distinguished by their stereotype. On the one hand, the stereotype *verification point* determines activities, checking the current system state against specified criteria. On the other hand, the stereotype *test step* marks an activity, which sends stimuli to the SUD by manipulating variables. Apart from the activity elements, the connectors may specify guards, in turn controlling the subsequent generation of test cases via path extraction.

To summarize, the included model elements may again be categorized, either following a *verifying* or *modifying* purpose, later on reflected by appropriate Execution Graph ++ (EGPP) Model elements.

For the system and test model information, a sensitive point in the process is given by the M2M transformation from the respective modeling languages to our internal representation, which is introduced in the following section.

2.1.4 Execution Graph++ Model

As already mentioned in the previous sections, it is desirable to lift the original model to an independent representation, which serves for internal analysis purposes. Therefore, we use the EGPP metamodel originally presented in prior work of Pröll et al. as target meta model for processing (Pröll and Bauer, 2018a). It also represents a basic concept for the Architecture And Analysis Framework (A3F) prototype, carrying out the functionality presented throughout this contribution.

The metamodel includes concepts for modeling hierarchical control flow graphs to capture the structure and behavior of mentioned test and system models (see *EGPPNode*, *EGPPGraph*). Further, a concept dealing with additional or already processed information from source model artifacts is included (see *EGPPTaggedData*, *EGPPAttribute*).

Altogether, the EGPP metamodel marks the central metamodel artifact which allows us to flexibly apply to all kinds of constellations of domain-specific modeling languages taking part in the respective development setup. Although, the EGPP instances are currently not used for implementation purposes, but may be used in future versions, the potential for semantic gaps between the original model artifact and

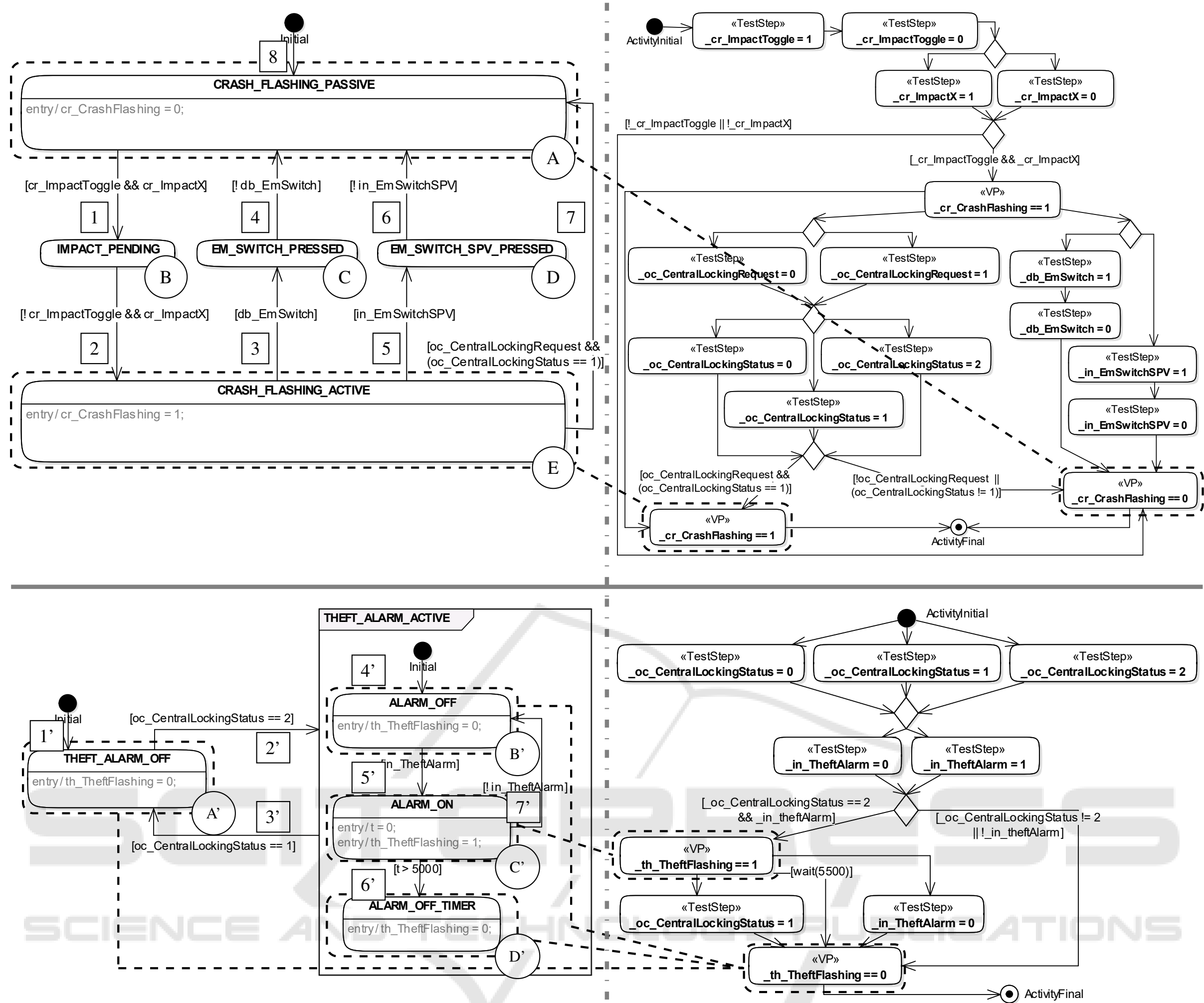


Figure 2: Integrated Model basis for CtrlCrashFlashing (**M1**, top) and TheftFlashing (**M2**, bottom).

the EGPP artifact demands for a proper and sensible specification of the M2M transformations involved. Nevertheless, once adapted to the set of applied modeling languages, engineers may integrate real MDSD analyses built upon this model representation, like for example test case management related model scoping.

2.2 Case Study

In the software testing discipline, there are several benchmark suites. Most of them aim at code-based testing approaches, which means that the compiled or interpreted source code - possibly derived from model artifacts - is stimulated by a test environment executing the set of test cases. However, Paleska et al. published "A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain" (Peleska et al., 2011), which ships with the underlying model artifacts and therefore is used as our running example and basis for our proof of concept (Section 4.2).

The presented *Automotive Light Control System* from an industrial use case splits up into two major parts, the *SUT* and the *Test Environment (TE)*, the latter being out of scope. The SUT, which comprises left/right indication, emergency flashing, crash flashing, theft flashing, and open/close flashing functionality, represents our system model. In addition to the system model, we specify the test model as well as the integration model making up the omni model basis (Pröll et al., 2017).

Figure 2 illustrates four pairwise integrated source model artifacts, which represent the scope of investigation on the automotive light control system. The upper part of the figure shows the omni model parts of the subsystem *CtrlCrashFlashing* (**M1**). The left part determines the state chart specifying the subsystem's functionality, while the right part represents the

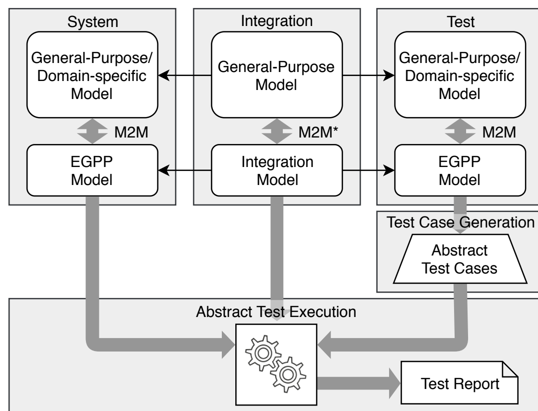


Figure 3: Processing pipeline for Abstract Test Execution.

test model. Further, this figure reduces the integration model elements to connectors between both sides (dotted and dashed boxes and lines). For example, the system model state `CRASH_FLASHING_ACTIVE` is connected by the integration model to the test model element `_cr_CrashFlashing == 1`. The same applies to the lower part of Figure 2 addressing the *Theft-Flashing* subsystem (**M2**). What both of the models reveal is, that there is no need for a complete specification of mapping information in the integration model, which is further detailed in Section 3.

3 THE ABSTRACT TEST EXECUTION CONCEPT

Based on the orchestration of domain-specific models, we introduce the context for the *Abstract Test Execution* (ATE) on the basis of Figure 3. The upper part revisits the split into domain-specific models and their transition to the internal representations via M2M transformations. These M2M transformations are specified in QVTo with a focus on preserving the original mapping information across involved domains as well as the included semantic of source models. Based on the EGPP instance for the test domain, a set of abstract test cases is derived. The test case generation is implemented by a dataflow analysis on the test model, which is guided by structural as well as dataflow oriented coverage metrics (Ammann and Offutt, 2016). In this contribution, we applied the bounded path-coverage criteria, realized by a specific dataflow analysis, to maximize the set of abstract test cases, evaluated by our ATE approach. The derived abstract test cases complete the set of inputs for the ATE processing pipeline. Herein, each test is challenged against the system model taking into account the information specified in the integration model. At

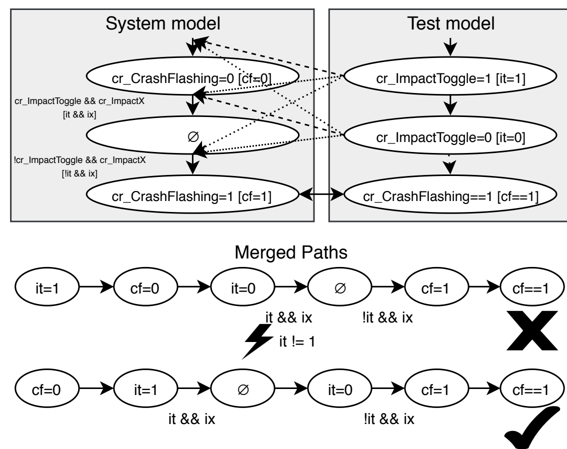


Figure 4: Running Example for ATE based on Figure 2.

the bottom of Figure 3 ATE's test report, a purpose-specific textual representation of the verdicts including explanation, is conducted.

Before we elaborate the internals of ATE, we introduce a running example. The running example (see Figure 4) marks a compacted excerpt from M1 introduced in Section 2.2. It is meant to illustrate the core concepts applied during the dedicated steps of the ATE. The affected paths of the system and test model are shown in the upper part and the lower part includes an intermediate analysis artifact further detailed in Section 3.2. The statements in square brackets represent a more compact version of the original statements included and are used in the lower part of the example. Further, the solid double arrow indicates an integration model based mapping of concepts, whereas the dotted and dashed arrows mark the possibilities for merging paths during the ATE.

In general, the ATE is made up of three steps. First of all the system model is prepared for the ATE. Therefore, the affected sub models of the system model will be detected and relevant start or end nodes of the execution are determined.

Second, every test case will be verified. The system model is not fully analyzed in one step, instead it is split into its paths triggering separate analyses. Finally, the results captured during the joint analysis of an abstract test case and a system model path are categorized as one of the four *test verdicts* to determine the quality-level of the afflicted model artifacts. Three of those verdicts are based on the test verdict set introduced by TTCN-3 (Grossmann et al., 2009). A fourth test verdict is made up to state the abstract computability of a test case (see Section 3.3).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

3.1 Preprocessing

As previously described in Section 2.1.1, the integration model enables modelers to specify connections between elements of the system and test model. Those connections are used to determine the start node and possible end nodes for the verification process of this test case. For example, the initial system node of a test case execution is either explicitly specified by a linked model artifact of the respective system model part, or implicitly determined by the initial node of the system model part. The same holds for final system nodes of a test case execution, which may be specified explicitly. All the mapping information given for intermediate model artifacts are processed according to the merged paths rule set detailed in the following section

Basically, a connection is relevant for the process, if it includes a test model element of the test case under analysis. Therefore, at least one relevant connection needs to be specified to validate the test case.

3.2 Abstract Test Case Verification

The first step of the abstract test case verification (ATCV) is to integrate the test case into one of several associated paths of the system model. Thereby, each element of the test case is merged into paths of the system model, with respect to the previously determined start node and possible end nodes. These so called *merged paths* are derived with regard to the following rules. First, the sequence of the elements of a path is always retained. Second, test steps are always inserted before guards of the system model. Third, verification points are always added after guards of the system model, if the associated verification point is not connected to the integration model. Otherwise, in consideration of rule two, the verification point is inserted after the connected system node and subsequent guard if the affected transition is guarded.

In consideration of the running example, the merging process is shown in Figure 4. There, a partial path of the test model is merged into a sub path of the system model by following the predefined rules. The possible merge spots are accentuated by dotted arrows. As a result, six valid merged paths exist. It should be mentioned that the instructions of the merged paths are compacted versions of the instructions of the test and system model, clarified by brackets. In general, the combination of one test case and one path of the system model can possibly result in multiple merged paths as seen in Figure 4. We restrict the number of generated merged paths to be considered in this case, since the sheer number of node combinations can easily explode. Please note that this restriction does not limit the merging of the test case with other paths of the system model. For short, every test case can be merged with several paths of the system model, whereby every merging of two different paths results in exactly one merged path. Thus, the validation of one test case can take several merged paths into account.

This lazy evaluation minimizes the workload, but due to path restriction can lead to falsely unfulfillable test cases, as shown in Figure 4. To compensate this, we allow the over-assignment of variables. Thus, every variable can have multiple valid values. In general, the system state is determined by variables introduced by modifying instructions contained in the merged path. Modifying instructions modify the value of a contained variable. In contrast, verifying instructions evaluate the system state by comparing the set of variables with their expected values. Due to over-assigned variables, multiple system states can be valid at the same time, which solves the problem of falsely unfulfillable test cases.

In general, a merged path consists of instructions, which modify or verify the system state. Overall, modifying instructions alter the system state, whereas verifying instructions check if a certain system state is reached. The system model consists of guards, which are verifying instructions and nodes, which possibly contain modifying instructions. The test model holds test steps and verification points. The former are able to contain modifying instructions, whereas the latter can hold verifying instructions.

When analyzing a merged path, guards are generally ignored during path exploration, but a fault is registered whenever a guard may not be evaluated positively, i.e. a guard may not be fulfilled. A fault is also registered if a mapped end node can not be reached. Another fault is the detection of an unknown variable in an instruction. Whenever a variable is initialized or updated, the new value is interpreted and then assigned to the identifier. The uninterpreted value is able to contain variable identifiers and mathematical or logical operators. All variable identifiers are resolved and their associated values are merged by using the operators. Whenever a variable identifier is not resolvable, a fault is registered, except when a verification point contains an unknown variable then the analysis of the merged path will be terminated and the resulting report is not considered as result of the test case. Therefore, we differentiate between the following faults (F_x):

(F_1) A verification point is unfulfillable or challenged with an over-assigned variable

(F_2) A guard could not be fulfilled

(F_3) No end connection is specified

(F_4) Unknown variable in test model

(F_5) Unknown variable in system model

(F_6) Guard contains time-dependent variable

(F_7) Guard fulfilled by using over-assigned variable

An analysis of a merged path ends as soon as one of the following conditions is fulfilled. These conditions further guarantee that ATCV always terminates. First, the maximum distance between two verification points is limited to a certain threshold (default: 5 steps) to prevent deadlocks. Whenever this threshold is exceeded, the verification of this merged path is terminated. Furthermore, the analysis is also terminated, if the end of the merged path is reached or the last verification point is satisfied.

Whenever the analysis of a merged path is terminated, a report is created. This report contains information about the merged path, detected faults and associated priority group and is linked with the original test case. Alongside the prescribed information, every report is categorized as one of the four following priority groups (P_x):

(P_1) All verifying instructions of the merged path are fulfilled and the last verification point is fulfilled with the instructions from one of the marked end nodes

(P_2) All verifying instructions of the merged path are fulfilled and the last verification point is not fulfilled with the instructions from one of the marked end nodes

(P_3) At least one verification point of the merged path is unfulfillable, but a marked end node is found

(P_4) At least one verification point of the merged path is unfulfillable and no marked end node is found

The priority groups are needed to determine if the entire merged path has been processed or only part of it, due to premature termination. A report associated to priority group P_1 and P_2 indicates that the processed merged path was entirely taken into account, while P_3 and P_4 indicate a premature termination of the verification process.

The prescribed faults and priority groups are used to determine the representative result of the test case. The report containing the highest detected priority group and least registered number of unfulfilled verifying instructions marks the representative result of the test case, which serves as the basis for the subsequent test verdict categorization.

Table 1: Test Case Result to Test Verdict Mapping.

	F_1	F_2	F_3	F_4	F_5	$P_{3/4}$	F_6	F_7	$P_{1/2}$
V_4	•	•							
V_3			•	•	•	•			
V_2							•	•	
V_1									•

3.3 ATCV Result to Test Verdict Mapping

In the last step, the representative results are mapped to test verdicts. We differentiate between these four verdicts (V_x):

(V_1) *Passed*

(V_2) *Probably Passed*

(V_3) *Inconclusive*

(V_4) *Failed*

The test verdicts *Passed*, *Inconclusive* and *Failed* are based on TTCN-3's verdict set, extended by a new test verdict *Probably Passed*. A test cases that is classified as Passed fulfills all verifying instructions in their order. As Probably Passed categorized test cases may be able to fulfill the associated verification points, but there is at least one verifying instruction which can not be evaluated with the provided information of the model. In this case, it is not possible to further detail the model with the information needed. One can only determine whether the test case can be fulfilled if the actual program code is examined. A test case that is classified as Inconclusive can not fulfill all verifying instructions either. However, the model artifacts can be enriched with the required information. A test case categorized as Failed is not able to fulfill at least one verifying instruction because a expected value is not met.

As described in the definition of the test verdict Probably Passed, the exact run-time of a program may not be determined by the ATE. In order to avoid the approximation of system states during test case verification, we skipped this feature and left it over to code-based testing mechanisms. In this context, the modification of time-dependent variables can not lead to a result change, because these variables did not contribute to the system state in the ATCV. That is why, the related result may not be categorized as Passed, but as Probably Passed.

Table 1 represents the mapping from faults and priority groups to a test verdict. The test verdicts are further weighted as follows: $V_4 > V_3 > V_2 > V_1$. The

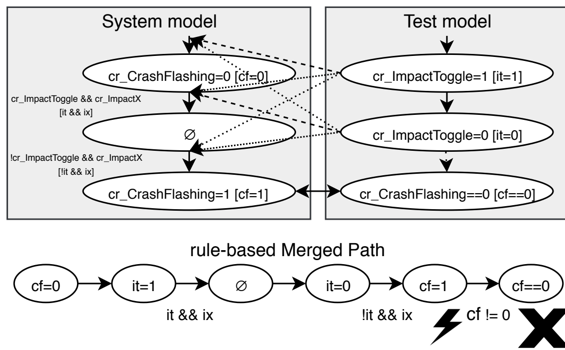


Figure 5: Example for a failing ATE based on Figure 2.

test verdict with the highest weight leads to the worst possible outcome, whereas the test verdict with the least weight leads to the best possible outcome. By following a pessimistic approach of test verification, the verdict with the highest weight is always prioritized. Thereby, a representative result is always categorized as the worst possible test verdict in consideration of the detected faults and associated priority group, as shown in Table 1.

In case of a representative result categorized as Passed no faults were detected during the verification of the respective merged path, consequently assigned to the priority group P_1 or P_2. Further, an incomplete verification leads to a best case categorization as Inconclusive, e.g. if the maximum allowed number of exploration steps of the merged path is exceeded and all verifying instructions are met at this point. The combination of a representative result and its categorization leads to the final report of the abstract test case. These reports are bundled in a final stage representing the *Test Report* (see Figure 3).

Contrasting the happy case of tests being categorized as *Passed*, *Probably Passed*, or *Inconclusive*, Figure 5 shows a test case leading to a merged path, which is categorized as *Failed*. Originated in an ATCV fault of the F_1 category, which means a verification point could not be fulfilled by the dataset (`cf (0) != 0`), the representative result of the test case is consequently mapped to the *Failed* verdict.

All in all, the concept of ATE gives testers and developers an indication about the quality-level of model artifacts contributing to the development process. In order to demonstrate the practical relevance of the approach, a prototypical implementation has been done. The implementation is based on a framework, which was previously introduced by Pröll et al. (Pröll et al., 2017), namely A3F. The analysis pipeline shown in Figure 3 gives insight into the prototypical realization.

4 PROOF OF CONCEPT

Based on the omni model of the case study introduced in Section 2.2, we further do a proof of concept for the ATE approach. Therefore, the processing pipeline originally presented in Figure 3 is extended.

4.1 Analysis Pipeline for Proof of Concept

The model artifacts from the three domains *System*, *Integration*, and *Test* remain constant and again mark the starting point (see Figure 6). Although the M2M transformations are not included here, we like to point out the importance for the flexibility of the ATE approach and for the validity of results produced by the ATE. The EGPP-based system model is further consumed by two separate ATE instances. While one instance takes the original version of the system model, the second instance processes a slightly modified version, named *EGPP model'*. The modified version is created in a controlled way, widely known as model mutation in the context of fault-based testing (Morell, 1990).

In parallel, the respective test model is processed, determined by mapping information of the integration model. Here, the test case generation step derives a set of abstract test cases, the second input for both of the ATE instances.

Finally, both instances of the abstract test execution are performed, leading to a test report each. Both artifacts, namely *Test Report'* and *Test Report*, are then compared based on the derived test verdicts. Based on the assumption, that the extracted test cases cover the system in a sense of edge coverage, every manipulation of the system model should be challenged by at least one of the generated abstract test cases.

For example, Figure 4 and 5 visualize the evaluation of one particular test case on two different system models as part of the previously presented analysis pipeline. The system model shown in Figure 5 is a altered version of the system model presented in Figure 4. Here, the constant of the model element E is changed. This leads to a unfulfillable test case. This particular case is part of the shift from unchanged to V_4 categorized test cases, as summarized in Figure 7 (c) M1 E.

4.2 Analysis of Results

In general, one can distinguish between modifications leading to structural or instruction-based changes of

the original system model. Figure 7 shows all the results produced during the ATE of the previously introduced models M1 and M2 including labels annotated to elements of these models, specified in Figure 2. In our evaluation, structural modifications, like the deletion (7a) or insertion (7b) of edges, as well as instruction-based modifications, like the replacement of instruction constants (7c) or exchange of guards with `true` (7d) are taken into account.

The mentioned modifications were consequently applied to appropriate model elements of Figure 2. One pillar of the figure represents a run of the previously introduced analysis pipeline, which means the height is determined by the number of test cases generated during the pipeline execution. Thereby, the model mutation modifies exactly one model element of the original system model. In general, every modification aims at exactly one element of the system model, except for the edge insertion case, which integrates new edges into the original system model. All permutations are taken into account, but only the most significant impacting edge is represented.

Furthermore, we observed that some modifications did not trigger any change of the resulting test verdict. We identified two reasons for that. First, there are cases which can not lead to a result change, which is due to the structural characteristics of the models M1 and M2. This holds for the modifications of B, C, D and D' shown in Figure 7b, as well as the modifications of 1 to 6 shown in Figure 7d. Second, the replacement of a constant of element C' and $6'$ also did not lead to a result change which is shown in Figure 7c (target modifications underlined). The reason is given by the variable t which represents a time-related variable, which is not taken into account by ATE.

All in all, we showed that a modification of the system model in most cases leads to a change of the assigned test verdicts. Therefore, the prototypical implementation is able to abstractly evaluate test cases with the limitation that time-dependent variables are ignored and in the best case, such test cases are classified as Probably Passed.

5 RELATED WORK

Our concept focuses on providing a flexible semi-formal model-based testing approach that is designed for early stages of software development. In general, MBT aims at generating test cases and executing them on the SUT. Behavioral models are used to automatically derive test cases that check whether a system is performing certain intended behaviors correctly. Traditionally, these test cases are executed on the SUT in

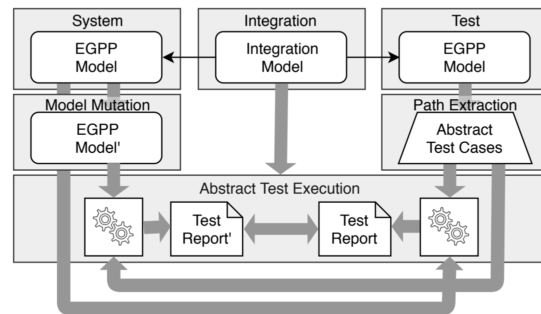


Figure 6: Analysis pipeline for the evaluation of abstract test execution within the framework.

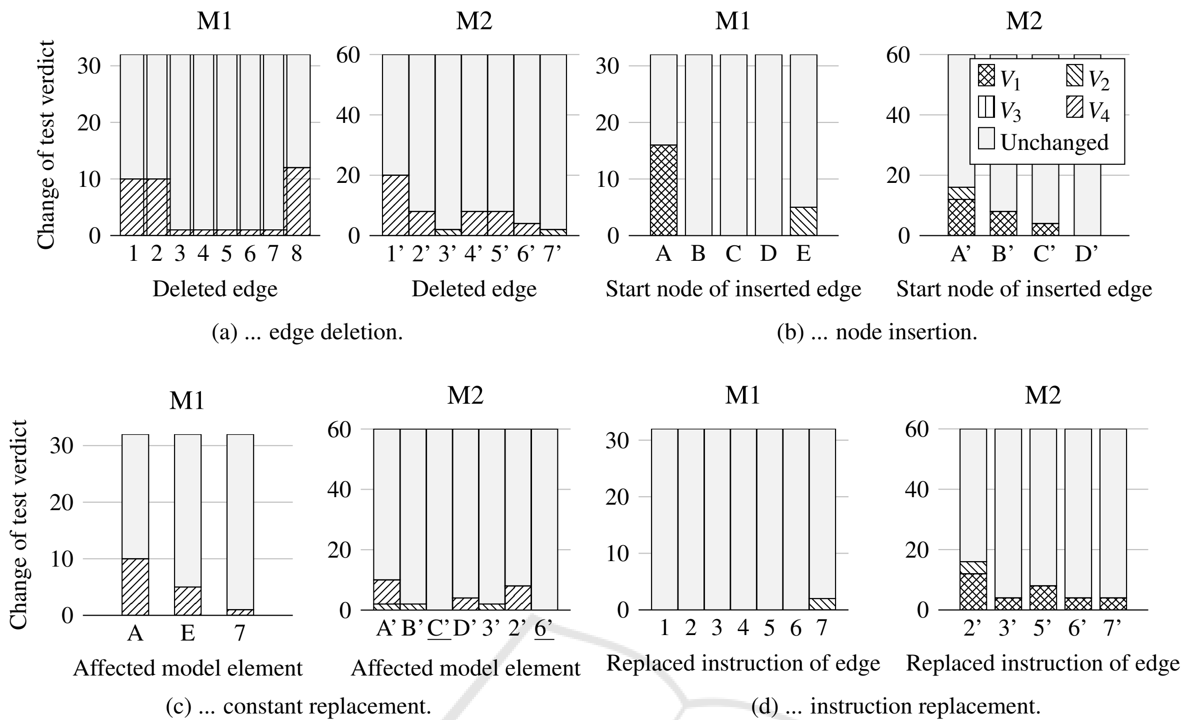
a classical *black box* testing manner. (Apfelbaum and Doyle, 1997)

In contrast, our approach may be seen as a *gray box* testing technique as previously presented by Khan et al. (Khan et al., 2012). Therefore, we can not only distinguish *if* a test case fails, but also *where* the challenged test case fails.

Model checking is another approach for model validation. In contrast to our concept, the validation process commonly based on formalized requirements, which are compared with every possible system state of the SUT (Baier and Katoen, 2008). Whereas, we follow a semi-formal and data-flow-oriented way of modeling, and comparing requirements with the SUT. Moreover, model checkers are preferably applied in later stages of development. Our approach meant to be applied in early stages of development as described in (Pröll and Bauer, 2018b).

Another approach dealing with the formal execution of model artifacts is represented by the OMG specification *Semantics of a Foundational Subset for Executable UML Models* (fUML) (OMG, 2018). Our approach shares the same basic concept of model execution as fUML, as demonstrated by Guermazi et al. (Guermazi et al., 2015). Further, both approaches are not generating program code, instead the model is directly interpreted most likely like script engines interpret scripts. While fUML is based on and limited to a subset of UML, our approach can easily be adapted to support other domain-specific modeling languages, like SysML.

A more general approach for simulation-based validation of development artifacts is given by x-in-the-loop approaches. This concept has its origins in the engineering sector (Plummer, 2006). For example, MiL (Model-in-the-loop) testing is used to evaluate the behavior of a mechanical system by a simulation based on mathematical descriptions, where the SUT is represented by a model. Another variant is given by Sil (Software-in-the-loop), where the simulation is based on compiled artifacts of the system, as



(a) ... edge deletion.

(b) ... node insertion.

(c) ... constant replacement.

(d) ... instruction replacement.

Figure 7: Results of ...

e.g. demonstrated in (Demers et al., 2007). Today, MiL simulations are often part of the design process of embedded systems. In terms of tooling, these simulations are mostly created with Simulink (Khalesi et al., 2019) or MatLab (Gambarotta et al., 2019), two well known engineering tools. In the context of testing, the mentioned simulation approaches show a black box characteristic. In contrast to that, our concept involves continuous monitoring of the system state, therefore categorized as a gray box technique.

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented a valuable approach tackling the twofold challenge, namely the steadily raising level of software complexity along with the insights on cost and time intensity of fixing induced faults. Therefore, we establish a basic approach towards a real shift left of V & V efforts. Based on a varying set of domain-specific models, an integrated model basis is build. A subsequent dataflow-based analysis of the involved test and system related models, driven by a set of extracted abstract test cases, enables developers to draw conclusions about the conformance of specified and intended functionalities of the system. To underline the abstract computability of test cases, a new test verdict was introduced.

Due to merging of test steps into a system path, the number of merged paths can easily explode. We addressed this problem by reducing the quantity of all possible merged paths to one, thereby allowing the over-assignment of variables under certain circumstances.

As the research successfully demonstrated, we see future work in a better formalization adressing the potential for semantic gaps originated in the M2M transformations, as well as in technical improvements of the prototypical implementation, but also in the integration and adaption to other application contexts. The technical improvements may include the development of more sophisticated preprocessing steps, leading to a better scalability and an optimized memory footprint. Moreover, the use of heuristic approaches for picking the most relevant system path during ATE may also represent a valuable extension of the current work. Focusing possible new application contexts, our approach could play a central role in a model-centric realization of mutation testing. Herein, a scalable mechanism for abstract test execution probably has the potential to push the acceptance of such technologies to the next level. Further, we see another application scenario of our approach in the area of semi-automated modeling support. Therefore, the results gained from the abstract test execution may be used as a knowledge base for a dynamic

and aligned set of modeling guidelines, addressing the weakness of vague guidelines due to missing context information.

ACKNOWLEDGEMENTS

The research in this paper was funded by the German Federal Ministry for Economic Affairs and Energy under the Central Innovation Program for SMEs (ZIM), grant number 16KN044137.

REFERENCES

Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Apfelbaum, L. and Doyle, J. (1997). Model based testing. In *Software Quality Week Conference*, pages 296–300.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Demers, S., Gopalakrishnan, P., and Kant, L. (2007). A generic solution to software-in-the-loop. In *MILCOM 2007-IEEE Military Communications Conference*, pages 1–6. IEEE.

Galin, D. (2004). *Software quality assurance: from theory to implementation*. Pearson Education India.

Gambarotta, A., Morini, M., and Saletti, C. (2019). Development of a model-based predictive controller for a heat distribution network. *Energy Procedia*, 158:2896–2901.

Grossmann, J., Serbanescu, D. A., and Schieferdecker, I. (2009). Testing embedded real time systems with ttcn-3. In *ICST*, pages 81–90. IEEE Computer Society.

Guermazi, S., Tatibouet, J., Cuccuru, A., Dhouib, S., Gérard, S., and Seidewitz, E. (2015). Executable modeling with fuml and alf in papyrus: tooling and experiments. *strategies*, 11:12.

Jones, C. (2008). *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill Education Group, 3rd edition.

Khalesi, M. H., Salarieh, H., and Foumani, M. S. (2019). Dynamic modeling, control system design and mil–hil tests of an unmanned rotorcraft using novel low-cost flight control system. *Iranian Journal of Science and Technology, Transactions of Mechanical Engineering*, pages 1–20.

Khan, M. E., Khan, F., et al. (2012). A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6).

Morell, L. J. (1990). A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857.

Object Management Group (OMG) (2004). Uml 2.0 testing profile specification.

OMG (2002). Meta object facility specification.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1.

OMG (2018). About the semantics of a foundational subset for executable uml models specification version 1.4.

Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., and Zahlten, C. (2011). A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *IFIP International Conference on Testing Software and Systems*, pages 146–161. Springer.

Planning, S. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*.

Plummer, A. R. (2006). Model-in-the-loop testing. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 220(3):183–199.

Pretschner, A. and Philipps, J. (2005). 10 methodological issues in model-based testing. In *Model-based testing of reactive systems*, pages 281–291. Springer.

Pröll, R. and Bauer, B. (2018a). A model-based test case management approach for integrated sets of domain-specific models. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 175–184.

Pröll, R. and Bauer, B. (2018b). Toward a Consistent and Strictly Model-Based Interpretation of the ISO/IEC/IEEE 29119 for Early Testing Activities. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: AMARETTO*, pages 699–706. INSTICC, SciTePress.

Pröll, R., Rumpold, A., and Bauer, B. (2017). Applying integrated domain-specific modeling for multi-concerns development of complex systems. In *International Conference on Model-Driven Engineering and Software Development*, pages 247–271. Springer.