Abstract Test Execution for Early Testing Activities in Model-Driven Scenarios

Reinhard Pröll*[0000−0002−3979−5483], Noël Hagemann[0000−0001−9441−9889], and Bernhard Bauer[0000−0002−7931−1105]
{reinhard.proell, noel.hagemann, bauer}@informatik.uni-augsburg.de

Software Methodologies for Distributed Systems
University of Augsburg, Germany

Abstract. The continuous improvement of the performance of computing units makes it possible to cope with increasingly complex tasks. This results in more complex software systems. However, the development of such highly complex systems is difficult to achieve using traditional approaches. Concepts like model-driven software development can weaken this problem in these constructive phases. However, new challenges arise for the testing of development artifacts. In order to be able to perform a real shift left of verification and validation tasks towards early phases of development, we present a semi-formal approach that enables users to execute test cases against the system under development (SUD) on the model-level. Grounded on an Integrated Model Basis which is created and maintained during development, test reports are automatically derived. This opens up a wide range of possibilities for early and targeted troubleshooting.

Keywords: Test Execution, Model-Based Testing, Domain-Specific Modeling, Integrated Model Basis, Model-Driven Software Development

1 Introduction

Due to the rapid development of hardware, more and more complex tasks can be mastered. As the complexity of the tasks steadily increases, the complexity of the software that handles these tasks is growing. In order to handle this increased complexity of the software development, a new trend has emerged in development practices.

In contrast to purely code-based approaches, many development tasks are nowadays handled by model-based ones. These techniques are characterized by the concepts of abstraction and automation, thus reducing complexity for the user. Considerable progress has been made in the areas of executable models, especially in the formal verification of suitable models, and in model-based testing. This gives insights into the planned system in early phases of development and enables developers to take appropriate and possibly early (counter-)measures,

* Corresponding author

since defects introduced into the system in early phases of development usually cause significantly higher costs for elimination (time and money) [23][5]. Further, Jones et al. [13] show how the worst-case scenario of very late discovery of such defects is the rule rather than the exception. Thus, verification in early phases of development is significant.

1.1 Problem Statement

The mentioned approaches in the model context either place high demands on the modeling languages used or work effectively on code artifacts derived from models or even platform specific artifacts. If one tries to put this into the context of Model-Driven Architecture (MDA), instances of the Platform-Specific Model (PSM) or Implementation-Specific Model (ISM) level are usually used for this purpose [15]. Especially for semi-formal test activities no effective shift left towards early development phases can be achieved, due to missing execution or simulation concepts.

We want to achieve this kind of functionality by implementing an approach to perform tests against the system based on model artifacts associated with the Platform-Independent Model (PIM). This enables the possibility to detect certain types of defects even earlier and thus reduce the overall costs. In contrast to purely specification-based tests (black-box), information about the implementation can be used at this point, allowing more targeted testing in the sense of gray-box testing. This can be seen as a kind of guidance for the modeler in addition to classical test results.

Up to our knowledge, there is no semi-formal approach that offers such functionalities. Based on an integrated set of model artifacts of different domains, like system development and testing, our *Abstract Test Execution (ATE)* approach is implemented. Therefore, the modeling expert specifies correlations between elements of the system model and the test model. From this *Integrated Model Basis*, an analysis-specific representation can be derived by using Model-to-Model (M2M) transformations. On the basis of these transformed model representations as well as the updated mapping information captured by the Integration Model, the concept of ATE is applied. Similar to classical testing, reports are created for the test runs, documenting the results of the execution to support troubleshooting.

In contrast to the related conference paper [10], the following sections draw a holistic picture of the concepts around the ATE approach. Furthermore, the ATE itself is more detailed and reworked with a lightweight formalization. This is followed by a critical discussion.

1.2 Outline

Following the previously presented introduction and problem statement, the remaining contents are structured as follows. In the course of section 2 the foundations are described. This includes the introduction of our running example

(section 2.1) which demonstrates the details of our approach. Further, the *Integrated Model Basis (IMB)* is presented throughout sections 2.2 and 2.3, using the running example to give a more intuitive understanding of the concepts. The main contribution, namely the *Abstract Test Execution*, is introduced in section 3. Thereby, the structure of subsections 3.2 to 3.6 reflects the overall structure of the underlying process. Following section 3, a mixed qualitative evaluation/discussion of concepts is presented in section 4. In order to set our approach in context to other research, related work is discussed in section 5. Finally, a conclusion is drawn and a road map for future topics is elaborated.

2 Foundations

In order to detail the approach outlined in the previous chapter, the necessary foundations, including the running example, are presented. In particular, the different model artifacts which are part of the concept are explained. An overview is given in Figure 1.

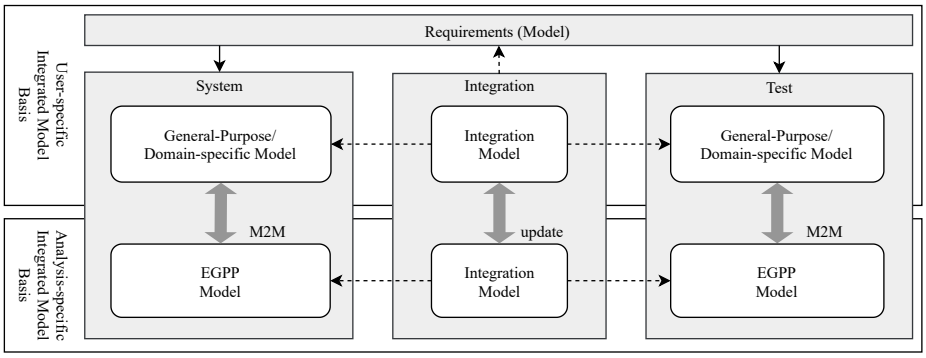

Fig. 1. Model Artifacts in the Context of Abstract Test Case Execution

In the upper part of the figure the *Requirements Model* is shown, representing the starting point of any development. In the context of Model-Driven Software Development (MDSD), both *System* and *Test Models* are built upon this basis. Different types of modeling languages can be applied to create these artifacts, such as General Purpose Modeling Languages (GPML) or Domain-Specific Modeling Languages (DSML). In this case, the use of two separate model artifacts is considered to support the automation of subsequent processing steps. Therefore, it is necessary to define a well-formed relation between these model artifacts, achieved by the integration component placed in the middle of the figure, namely the *Integration Model*. All of these models represent possible interaction points with the user (*User-Specific Integrated Model Basis*) For details see section 2.2.

In addition, derived from the *User-Specific Integrated Model Basis* through Model-to-Model (M2M) transformations, a so-called *Analysis-specific Integrated*

Model Basis is introduced (for details see section 2.3). Essentially, the System and Test Models are mapped to an internal metamodel, which was designed for the subsequent automated processing (for details see section 2.3).

2.1 Running Example - Automatic Door Control System

As already mentioned, the running example serves for illustrating the different aspects of our approach. In order to keep this intuitively understandable and clear, we have chosen a simple example from everyday life. Due to its simplicity, there are no structural artifacts of the System Model. It is a door control system, where its control logic is represented by the state machine in Figure 2.

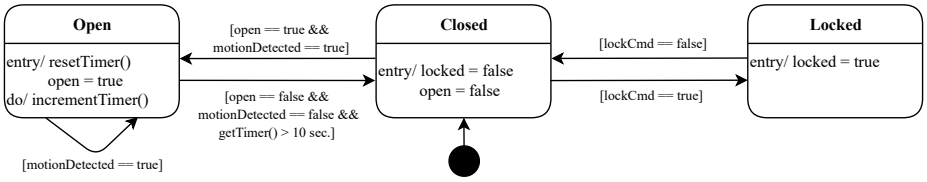

Fig. 2. Behavior System Model for the Automatic Door Control System

The control logic defines three different states, namely `Open`, `Closed` and `Locked`. The door is able to query the status of a sensor, which reveals if the door is open or closed (`open`). This is supplemented by a sensor for detecting movement at close range (`motionDetected`). Besides the sensors of the system, a conventional lock is provided for manually locking of the door, which sends a `lockCmd` to the door control unit on actuation. Apart from the event-driven points of interaction, a time-triggered component is part of the door control unit. Precisely, as soon as a timer of ten seconds has elapsed, the closing of the door is initiated, unless motion is detected by the sensor.

Starting with this System Model artifact, the proposed running example is extended and constantly used in the following sections to illustrate certain aspects of our contributions.

2.2 User-specific Integrated Model Basis

As already mentioned above, the so-called *User-specific Integrated Model Basis* (*Omni Model* for short) represents the data side of the approach. In particular, this combination of model artifacts represents the action point for the users of the subsequent automated processing chain. In principle, any development model can be integrated, provided their metamodels are completely available. This approach forces the separation of concerns on the model-level, which has a positive impact on the significance of the resulting tests [25]. An essential role and the minimal amount of model information is given by the System Model,

the Test Model and the integration of these two artifacts modeled within the Integration Model. In this context, most of the relevant aspects of these artifacts have already been introduced in earlier publications by Rumpold and Pröll (for details see [28][27]).

System Model - In the context of MDSD, this model covers both structural and behavioral aspects of the SUD. Different modeling languages can be used, depending on the application domain and the expertise of the developers. E.g. GPMLs such as the SysML [19] or DSMLs used in the context of the embedded MDSD tool radCase [7] from IMACS can be used.

Test Model - The same applies to the model artifacts concerning test modeling. The choice of the metamodel should be based primarily on aspects such as expertise of the test engineers and sufficient tool support. E.g. GPML-based approaches such as the UML Testing Profile [21] or proprietary modeling languages can be used.

Other Domain-Specific Development Models - The combination of different models is not limited to the two domains already mentioned. If information about e.g. temporal behavior, safety or security of the SUD is considered in separate DSMLs, it can be linked to the Integrated Model Basis. Even if this information has no direct influence on the processing and thus results of the ATE, such information can be included in the context of post-processing. For subsequent troubleshooting, correction and, under certain circumstances, transitive defect effects can be determined.

Integration Model - This model artifact provides the link between the sets of different domain-specific models. In order to map the relationships between the models, the structural decomposition of the instantiated SUD is primarily modeled. This enables purely structural mappings between instance- and component-related parts of the different models. Beyond the structural mappings, additional information on elements of the structural decomposition can be added to the Integration Model (see aspects concept in [26]).

In addition, these mappings between models participating in the Integrated Model Basis can be specified for behavioral model elements. These mappings represent a key concept in the implementation of the ATE, since this enables the synchronization of different types of models of the same behavioral aspect. I.e. a series of synchronization points (so-called `IMSyncPoints`) is defined for each behavior model, which are connected to elements of the System Model as well as the Test Model. Furthermore, different types of synchronization points are distinguished. Besides the conventional `IMSyncPoints` there are `IMSyncEntryPoints` and `IMSyncExitPoints` representing the entry and the exit of a synchronization sequence respectively. The level of detail or completeness of these mappings has a decisive influence on the test results determined in the context of the ATE. Therefore, careful modeling must be carried out to avoid deducing false conclusions from the corresponding results.

Application to the Running Example In order to illustrate the concepts of the Integrated Model Basis, the concrete instances of the models in the context of the Automatic Door Control System will be discussed below.

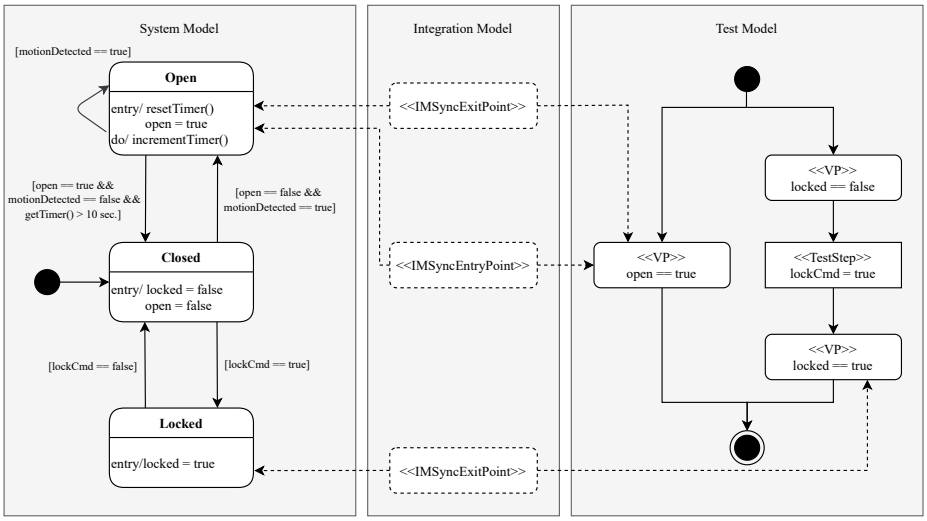

Fig. 3. Excerpt of the User-specific Integrated Model Basis for the Automatic Door Control System

Figure 3 shows the relationships between the individual model instances. On the left side of the figure, the state machine previously introduced in Figure 2 represents the behavioral model of the SUD. The model elements arranged in the middle of this figure show synchronization points and thereby specify the behavioral mappings across the development domains. On the right side, a activity chart based Test Model consisting of *Test Steps* and *Verification Points (VP)* is conducted. In principle, two different test cases can be derived (see right side of Figure 7) from this model, which are evaluated against the system as part of the ATE.

2.3 Analysis-specific Integrated Model Basis

In Figure 1 the user-specific and the analysis-specific view of the model basis was shown. Basically, the analysis-specific representation of the Integrated Model Basis decouples the algorithmic implementation from the specifics of the respective application context. In contrast to the manual modeling of the model artifacts, the analysis-specific variant is derived automatically. This is done by a set of transformation steps describing a model transformation. These steps are specified on the metamodel-level. The components of this target metamodel, relevant for ATE as well as the concepts of the transformations, are explained in the following.

Analysis-specific Metamodel In order to be able to map multiple models in the context of model-centric testing, the *Execution Graph ++* (EGPP) metamodel was developed. In the context of Pröll et al. [26] the metamodel simultaneously representing the structure, control flow and data flow information of a system, was introduced. Basically, the control flow information is described by nodes (`EGPPNode`) and edges (`EGPPTransition`). In addition, structure information can be modeled by nesting these control flow structures, since special nodes (`EGPPGraph`) can contain control flow (sub-)graphs. The data flow information (`EGPPTaggedData`) completes the information set and annotates nodes and edges of the model. In particular, the annotated data are atomic with regard to the included expressions, such that only one expression is captured per node/edge.

In general, an instance of the EGPP describes behavior by sequences of states, which can be applied to both a Test Model and a System Model. Each of these states consists of an active node of the control flow and a set of variable assignments. A variable state is updated by the assignments of the active node, assuming the intermediate guards could be fulfilled. If an assignment is made in the node, it is further identified by a `EGPPInputNode`, if only a condition is checked, a `EGPPOutputNode` is modeled. The latter type of node is used when mapping Test Models, particularly to check the current state from a data flow perspective. In case of the control flow perspective, `IMSyncPoints` concept of the Integration Model is evaluated. If such a connection between `EGPPNodes` of a Test Model and a System Model is specified, an additional condition is imposed on the control flow. Especially in case of `IMSyncEntryPoints` or `IMSyncExitPoints` this means that the evaluation of the test case against the system has to start or end at the referenced points of the control flow.

Model-to-Model Transformations As shown in Figure 1, both the original System and the Test Model are transformed into the EGPP metamodel by a horizontal exogenous M2M transformation. Different patterns are implemented such that a uniform representation is created for both aspects independent of the user-specific metamodel. The basic concepts of these patterns are shown in Figure 4. Further, it is crucial that the transformation rules defined for this purpose do not change the semantics of the original models, but at best refines them.

On the left side of Figure 4, the System Model is considered. In case of purely structuring model elements of the original model, a construct is created in the EGPP context, which cyclically embeds all included components (SSM) and behavioral models (SBM) in its flow. In contrast, the behavioral models are not enriched with any synthetic control structures, as long as the original model already provides a defined initial and final state. At this point, it is important to preserve the original specification of system states.

In contrast, the right side of the graph shows the pattern for the Test Model. Here, the different test levels are aligned to the integration levels of the System Model. In this hierarchy the highest level model is the *System Test Model (STM)*, which specifies consistent test cases at the integration level, but can include lower

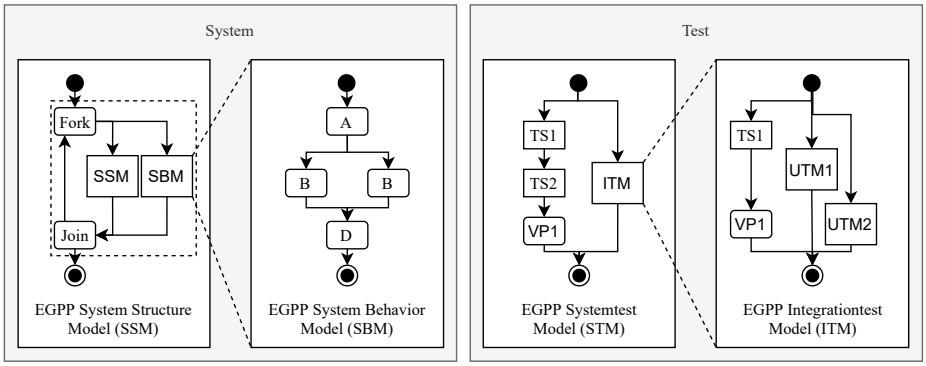

Fig. 4. EGPP Model Patterns for System and Test Context

integration levels, such as an *Integration Test Model (ITM)*. This is done through all the integration levels considered down to the unit level, which is illustrated in the right-hand part of Figure 4.

In addition to the control flow and structure-giving patterns, the data flow is transformed realized in the form of a pseudo-code-like language, which is out of scope for this contribution. According to this language the `EGPPTaggedData` elements are filled with information during the transformation.

In addition to the M2M transformations of the System and Test Models, the mapping information is updated between artifacts of these modeling domains captured by the integration model. This update includes the creation of new mappings between transformed model elements of the respective EGPP instances, provided that their original model elements are already part of a mapping relation specified by the Integration Model. Normally, no manual intervention is necessary, provided that the M2M transformations are specified completely regarding the conventions mentioned above.

Application to the Running Example The application of these transformation rules converts the Integrated Model Basis already introduced in Figure 3 into the following variant (see Figure 5).

Basically, the two graphs are very similar, since the original model elements have been transformed into `EGPPNodes`. Model elements with sharp corners represent `EGPPInputNodes` and round corners represent `EGPPOutputNodes`. Exceptions are the dashed model elements of the Integration Model, arranged in the middle of this figure. The annotated data flow information is stored in the code fragments, which reflects the `EGPPTaggedData`. Furthermore, an explicit final state was added to the original System Model, which can be reached by all original states.

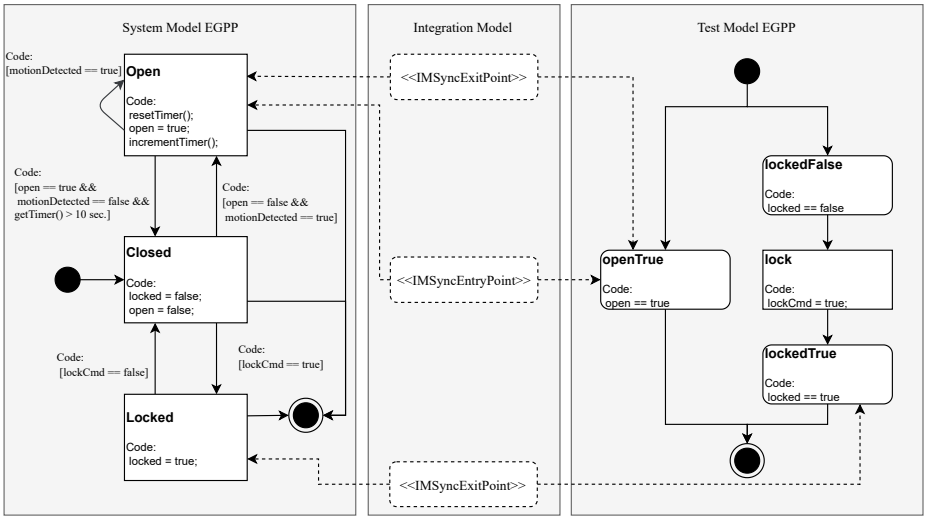

Fig. 5. Excerpt of the Analysis-specific Integrated Model Basis for the Automatic Door Control System

3 Abstract Test Execution

Several input artifacts are required to perform the process of *Abstract Test Execution*. These input artifacts are given by the Analysis-Specific Integrated Model Basis. As mentioned in section 2.3, this model basis mainly consists of two model artifacts which are interconnected by a third model artifact, namely the Integration Model.

3.1 Overall Process

Both main artifacts, namely the System Model and the Test Model, are instance models of the EGPP metamodel. The System Model describes the structure and behavior of the system, while the Test Model represents the intended behavior of the system. The Integration Model can be used to define so called `IMSyncPoints` and is based on a corresponding metamodel. `IMSyncPoints` are used to define entry points and exit points of the execution of abstract test cases.

The test cases are derived from the Test Model. The process is detailed in the following section 3.2.

The process of ATE uses the System Model, the Integration Model and the generated set of abstract test cases to perform testing. The general approach is visualized in Figure 6. Every abstract test case contained in the generated set is evaluated one after the other. This evaluation process of each abstract test case consists of multiple steps, which ultimately result in a test report.

At first, the abstract test case is merged into the System Model. Different cases for the merging exist which results in a vast range of different paths

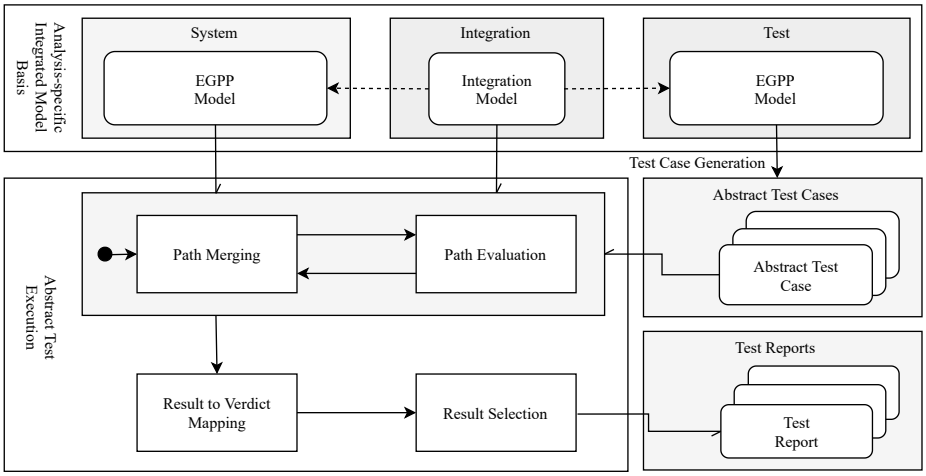

Fig. 6. Overall ATE Process

describing the majority of all possible data flows and control flows. We identify some basic merging rules, which are described in section 3.3. Besides the merging process, the data flow and the control flow of these paths are assessed. Therefore, basic data flow specific faults are taken into account as well as control flow specific characteristics. This baseline is detailed in section 3.4. In addition, the preliminary results gathered by this analysis are collected and classified. The result classes are described in section 3.5.

Then, one of these preliminary results is chosen to be the representative result of the ATE. Finally, a human-readable test report is generated from the representative result. These steps are detailed in section 3.6. For further understanding, all the mentioned process steps are illustrated along the running example.

3.2 Preprocessing and Derivation of Abstract Test Cases

A model is viewed as a graph consisting of nodes and edges. A Test Model comprises two kinds of nodes and unidirectional edges. As described in 2.2, such models are structurally based on activity diagrams that can preserve a chain of events by transforming them into a fixed sequence of nodes enclosed by an initial and a final node. Contained nodes are connected by unidirectional edges. Generally, a distinction is made between nodes that contain instructions which either send stimuli to the SUT or check whether certain outputs of the system meet predefined conditions. Due to the abstract nature of the ATE approach, we distinguish between instructions that modify variables of the SUT or check for certain variable values. Nodes included in a Test Model can either contain one instruction that is capable of modifying exactly one system variable or any number of conditions to challenge the system state. The former are called *Test*

Steps while the latter are referred to as *Verification Points*. During transformation from the user-specific input model to its analysis-specific EGPP-based form, it is ensured that non-atomic nodes are transferred into an atomic form. Furthermore, nodes of type *Test Step* are transformed into `EGPPInputNodes` while *Verification Points* are converted into `EGPPOutputNodes` as described in section 2.3. Depending on the Test Model as an EGPP-based artifact, a data flow analysis is performed which is able to mimic combinations of structural as well as data flow coverage metrics to derive sets of abstract test cases.

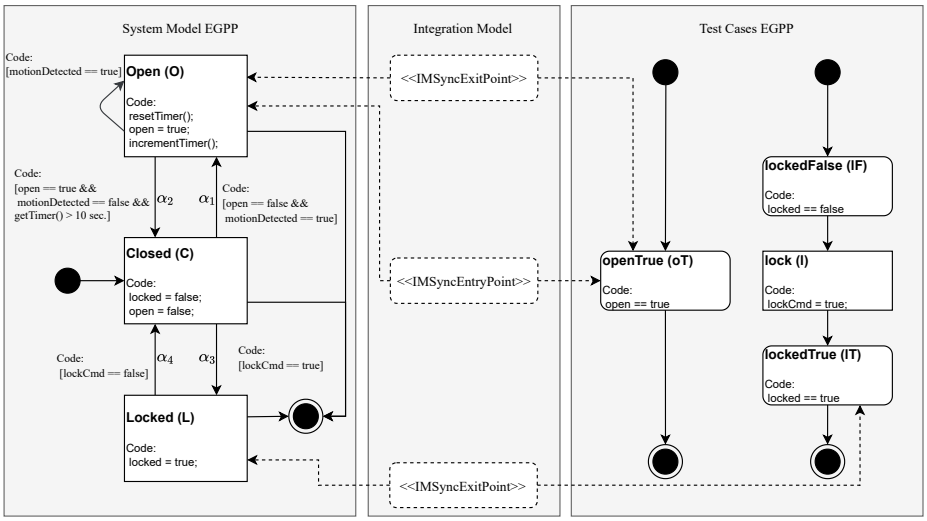

Fig. 7. EGPP Representation of Running Example

In the context of the running example, two test cases are extracted from the Test Model (see Figure 5) and presented in Figure 7. The test case on the left hand side checks whether the internal system variable `open` is set to `true` while the system state is initialized as `Open`. The other test case determines whether the system switches correctly into the `Locked` system state after initializing the system to the `Closed` state and sending the `lockCmd = true` stimulus to it.

3.3 Path Merging based on Integrated Model Basis Mappings

Like the Test Model described in section 3.2, the System Model supports different model elements. Generally speaking, the System Model can have nodes that contain instructions that modify the system state and conditional transitions that restrict the change of system states (referred to as *Guards*).

Nodes of the System Model and Test Steps of the Test Model contain instructions capable of altering the system state. Verification Points of a Test Model and instructions of guarded transitions of the System Model share the same kind

of instructions to validate the system state. Overall, the Test Model and the System Model contain two different kinds of instructions which represent the basic blocks of the merging process.

Furthermore, the EGPP-based structure of the System Model and Test Model is defined by an initial node and a final node. This common structure of the models naturally specifies the entry point and exit point for ATE. However, due to the potentially multi-layered structure of the input models (cf. section 2.3) more than two entry and exit point pairs may exist. Therefore, we introduce the possibility to specify explicit connections between both models to determine the entry point as well as the exit point of the ATE to restrict the number of model artifacts taken into account. In contrast to the fact that end connections are mandatory for the control flow analysis of ATE, the definition of entry connections are optional. In addition, entry connections can be used to initialize the system state different to the initial system state as visualized in Figure 7.

After the determination of an entry point for the ATE, the merging process is carried out step by step. Every step inserts one node of the test case into the system. The process starts by merging the first node of the test case into the System Model and ends with the test case being completely merged. Depending on the System Model, several possibilities exist for inserting a node of the test case into the System Model. The most basic merging approach is to take every permutation into account but this can lead to a state explosion. In order to tackle this challenge, the merging process performs the following rules:

1. The sequence of nodes of the test case and System Model is kept.
2. Incoming transitions of a system node are not separated by nodes of the test case
3. Verification Points are inserted after nodes of the System Model
4. Test Steps of a segment are inserted directly after the leading Verification Point

Generally speaking, these rules limit the set of permutations without loss of generality on the final result of the ATE. The effect of the rules is discussed in section 4. Moreover, we name the set of all permutations the path space P and the subset of permutations created by applying the rules as the limited path space P_{lim}.

The first rule ensures that the control flow of the system is maintained by keeping the general structure of the test case during merging. Subsequently, nodes of the test case can only be inserted into the System Model, if the given order of test nodes is preserved, which represents the first step of limiting the path space.

The second rule is used to imitate more classical testing approaches. From a classical testing point of view, stimuli are sent to the system to initialize a change of system state. In more detail, the stimuli are used to satisfy some condition which guard the change of system state. From our abstract testing point of view, nodes of the test case can be inserted directly before a node of the System Model or before an incoming transition of a node of the System Model. In order to preserve the behavior of classical testing, rule two allows the

insertion of nodes of the test case before an incoming transition of the system model which is used to imitate the classical testing approach.

The third rule aligns with entrenched code-based testing activities. From a classical testing point of view, a test case must interact with the SUT to verify its functionality. In consequence, if the test case does not interact with the SUT, the test case fails. Due to the abstract nature of this approach, test cases may be defined that rely on induced variables by the test cases rather than variables induced by the system. This could potentially lead to a test case that is falsely successful. To reduce the risk of such test cases, rule three is defined (cf. Section 4).

The last rule leads to a significant reduction of the path space. Two aspects come into play. First, due to the abstract nature of this approach, time-dependent variables are out of scope. As a result, the sequence of Test Steps of a Segment can be ignored since Test Steps represent stimuli to the system which are affected by timing. Here, a *Segment* refers to all nodes between two successive Verification Points. Second, stimuli to the system are bound to a change of system state. Therefore, if more than one node of the System Model is contained in a Segment, the test may fail by mistake. To cope with this problem, we allow the over-assignment of variables to declare all assignments of variables induced by Test Steps of a Segment as valid. A more in-depth description of the over-assignment of variables and the resulting effect can be found in section 4.

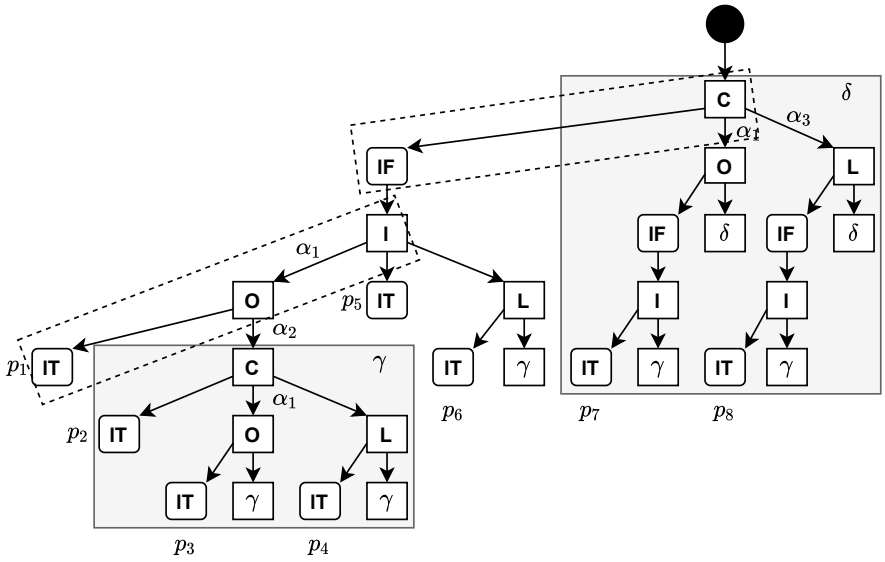

Fig. 8. Limited Tree of Paths of the Running Example

To visualize the merging process, a tree can be formed which contains the limited subset of permutations P_{lim}. Figure 8 shows the path tree of the running example. For better understanding, the Segments of the path p_1 are visualized by dashed boxes. In addition, nodes of the System Model and Test Model are visualized by squares which have rounded or sharp corners. The former represent nodes that contain instructions which verify the system state. The latter illustrates nodes that consist of instructions which modify the system state. The naming refers to Figure 7 while the following sections utilize the abbreviations in brackets. Guarded transitions are referenced by their respective identifier α_x, which can also be found in Figure 7. Due to loops in the representative System Model, the tree of paths contains an infinite number of paths. Therefore, nodes labeled with γ or δ can be substituted by nodes enclosed by the box with the same label.

3.4 Evaluation of Path Space

In reality, paths are not evaluated as a whole. Instead, the evaluation process is triggered after a new Segment is formed by injecting a Verification Point into the System Model.

The analysis of such a segment is carried out with the help of a combination of data flow and control flow analysis. The former is used to determine whether the instructions of guarded transitions and Verification Points can be fulfilled. The latter is used to check if the test case is solvable from a structural point of view and if the final system state is reached after the data flow is completely analyzed.

Generally, data flow analysis is an approach of collecting information about possible values of system variables. We use this analysis to execute and evaluate the instructions contained in the nodes of the segment currently being analyzed. During this process, several faults can be detected. We define $D := \{d_1, d_2, d_3, d_4, d_5, d_6\}$ as the set of data flow based faults. In the following, these cases are described.

d_1 Instruction of node not solvable
d_2 Guard of transition not solvable
d_3 Undeclared or uninitialized variable
d_4 Missing end point for data flow analysis
d_5 Guard contains time-dependent variable
d_6 Guard fulfilled by over-assigned variable

As previously detailed, several paths emerge which are likely to solve the test case. Test cases contained in the resulting P_{lim} consists of nodes of the EGPP Test Model and nodes and transitions of the EGPP System Model. The merging process removes the boundaries between these models, which causes the evaluation to distinguish only between variable-verifying instructions (VVI) and variable-modifying instructions (VMI). Such verifying instructions are Boolean expressions which can be evaluated to `true` or `false`. On the one hand, we consider the latter result as unwanted behavior and register a fault in the event of

such a case. On the other hand, if the Boolean expression results in `true`, the associated Guard or Verification Point is successfully solved.

Algorithm 3.1: evalSegment(s)

procedure evalSegment(s)
 for each $e \in$ getElements(s)
 do { **if** instanceof(getInst(e), VMI)
 then storeValuesOfElement(e)
 else if instanceof(getInst(e), VVI)
 then verifyElement(e)
persistLastStoredVariableValues()

procedure verifyElement(e)
 if newFaultsRegistered(checkPreconditions(e))
 then return
 if instanceof(e, *node*)
 then { **if** ! verifyInst(getInst(e)) XOR isOverassigned(getInst(e))
 then registerFault(d_1, e)
 else if instanceof(e, *edge*)
 then { **if** verifyInst(getInst(e))
 then { **if** isOverassigned(getInst(e))
 then registerFault(d_6, e)
 else registerFault(d_2, e)

procedure checkPreconditions(e)
 for each $v \in$ getVariables(getInst(e))
 do { **if** isTimeDependentVariable(v)
 then registerFault(d_5, e)
 if size(getStoredValues(v)) == 0
 then registerFault(d_3, e)

procedure isOverassigned(i)
 for each $v \in$ getVariables(i)
 do { **if** size(getStoredValues(v)) > 1
 then return (`true`)
 return (`false`)

procedure storeValuesOfElement(e)
 if instanceof(e, *node*)
 then store(getLHS(getInst(e)),
 eval(getRHS(getInst(e)),
 getPermutatedVariableAssignments(getInst(e))))

procedure verifyInst(i)
 for each $va \in$ getPermutatedVariableAssignments(i)
 do { **if** eval(i, va)
 then return (`true`)
 return (`false`)

Algorithm 3.1 presents the procedure to evaluate a Segment which is detailed in the following. However, one of the requirements for the ATE is that an end connection is specified in the Integration Model that defines the desired exit point as specified in section 3.3. If no end connection is defined, the fault d_4 is reported and the segment evaluation of the path is skipped.

If a variable-verifying instruction fails as part of a transition, the fault d_2 is registered, otherwise if it is part of a node, the fault d_1 is registered. However, as a first step of evaluating instructions, affected variables need to be resolved. If they are not initialized or undeclared, the fault d_3 is listed.

Due to the fact that this approach is based on data flow and control flow analysis, time-dependent variables are out of scope. In general, however, there are test cases that rely on such variables. In order to be able to evaluate such test cases, the over-assignment of variables within a segment is allowed, which leads to multiple valid values at a time. However, after a segment is evaluated, only the value last-set remains valid, while the others are invalidated. Further, the usage of over-assigned variables is only allowed to evaluate verifying instructions of transitions, but in any case a fault is logged. If the instruction can be fulfilled by over-assigned variables, the fault d_6 is noted, otherwise the fault d_5 is captured. If an over-assigned variable is used to solve such an instruction of a node, the fault d_1 is added to the set of registered data flow faults for this path. As presented, the topic of over-assigned variables is addressed in detail in section 4.

On the one hand the data flow of the path is analyzed, on the other hand control flow analysis is used to structurally evaluate the path. Due to the end connections contained in the Integration Model, it can be distinguished if the system has reached the desired system state after the last node of the test case is merged into the System Model. The result is categorized into one of four fault classes. The set of the characteristics based on control flow is defined as $C := \{c_1, c_2, c_3, c_4\}$.

- c_1 All verifying instructions of path are fulfilled and the last verification point is solved by the instructions of one of the marked system nodes
- c_2 All verifying instructions of the path are fulfilled and the last verification point could be satisfied using the instructions of one of the marked system nodes
- c_3 At least one verification point of the path could not be fulfilled, but a system node marked as exit point is part of the path
- c_4 At least one verification point of the path is not solvable and no system node marked as exit point is part of the path

In general, we distinguish between test cases that can or cannot be fulfilled. If the test case can be fulfilled, it is differentiated whether the last verification point and the instruction of the system node used to fulfill the VP are connected by an end connection. If the test case is not solvable, it is determined whether an end node is generally found or not. These cases result in the four control flow specific fault classes listed above.

Overall, a set $O := \{R_1, \ldots, R_{|P_{lim}|}\}$ is iteratively formed containing result sets $R := D_H \cup C_{NI}$ derived from the segments of the paths p included in the

limited path space P_{lim}. The set R consists of the set $D_H := D \times H$ containing the detected data flow faults D combined with hints H on their cause and a set $C_{NI} := C \times \{NI\}$ of control flow characteristics C with the symbol NI as a pair. This symbol signals the absence of a hint resulting in the extended set $H_{NI} := H \cup \{NI\}$. Generally, hints can be instructions or variables of the System Model and Test Model.

$$
\begin{aligned}
R_1 &= \{(d_2, \alpha_1), (d_1, lT), (c_4, NI)\} \\
R_2 &= \{(d_2, \alpha_1), (d_2, \alpha_2), (d_1, lT), (c_4, NI)\} \\
R_3 &= \{(d_2, \alpha_1), (d_2, \alpha_2), (d_2, \alpha_1), (F1, lT), (c_4, NI)\} \\
R_4 &= \{(d_2, \alpha_1), (d_2, \alpha_2), (d_1, lT), (c_3, NI)\} \\
R_5 &= \{(d_1, lT), (c_4, NI)\} \\
R_6 &= \{(c_1, NI)\} \\
R_7 &= \{(d_2, \alpha_1), (d_1, lT), (c_4, NI)\} \\
R_8 &= \{(d_2, \alpha_3), (d_1, lT), (c_3, NI)\}
\end{aligned}
$$

In context of the running example, the set of detected faults and characteristics stated above are based on the paths p_x given by Figure 8. In this case, the set of hints is defined as $H = \{O, C, L, \alpha_1, \alpha_2, \alpha_3, \alpha_4, oT, lF, l, lT\}$. The result sets R_x are directly derived from their respective paths by their identifier $p_x \rightarrow R_x$ with $x \in \{1, \ldots, 8\}$.

For Example, the set R_1 consist of three elements. The first elements gives information that the guard `open == false && motionDetected == true` could not be satisfied. The second element can be interpreted in that way that the Verification Point `locked == true` is not solvable. The last element marks the evaluation of this path as finished and states that at least one Verification Point could not be fulfilled and the test case could not be solved structurally, since an end connection exits which connects the Verification Point `lockedTrue` with the system state `Locked`, but this system state is not part of the analyzed path.

3.5 Result to Verdict Mapping

The next step covers the classification of the result sets $R \in O$. For this purpose, we define the set of verdicts $V := \{v_1, v_2, v_3, v_4\}$. In general, we distinguish between the four verdicts *Passed* (v_1), *Probably Passed* (v_2), *Inconclusive* (v_3) and *Failed* (v_4). The test verdicts *Passed*, *Inconclusive* and *Failed* are based on TTCN-3's verdict set [8], extended by the new test verdict *Probably Passed*. We justify the extension of the classical verdict set to signal the existence of aspects which cannot be evaluated due to the abstract nature of this approach.

A path that is classified as *Passed* fulfills all variable-verifying instructions contained in the path based on its data flow. The classifications *Probably Passed* and *Inconclusive* indicate that some information is missing. In the case of *Inconclusive*, these information can be added to the source models by the modeler. Otherwise, this information cannot be provided in the case of *Probably Passed*,

as the exact runtime behavior of the system cannot be determined by the ATE. We leave this feature over to code-based testing mechanisms. The last verdict marks paths where the evaluation of variable-verifying instructions leads to a negative result (`false`).

$$M : R \to V \begin{cases} v_1, & \text{if } \exists (f,h) \in R.\ f = c_1 \wedge |R| = 1 \\ v_2, & \text{if } \exists (f,h) \in R.\ f = d_i \text{ such that } i \in \{5,6\} \wedge \\ & \quad \forall (f,h) \in R.\ f \neq d_j \text{ such that } j \in \{1,2,3,4\} \\ v_3, & \text{if } \exists (f,h) \in R.\ f = d_i \text{ such that } i \in \{3,4\} \wedge \\ & \quad \forall (f,h) \in R.\ f \neq d_j \text{ such that } j \in \{1,2\} \\ v_4, & \text{otherwise} \end{cases}$$

The presented verdicts are concluded by the cases of the function shown above. It is used to derive a verdict from a result set. Generally, a pessimistic approach is chosen for the determination of verdicts. For example, a missing end connection d_4 does not necessarily lead to a failing test case, but considering the displayed function, it is marked as Inconclusive, although the missing connection has no effect on the data flow on the one hand. On the other hand, this feature can significantly impact the runtime of the ATE, which may result in the test case not being able to be analyzed by the ATE in the worst case. To prevent such behavior, the classification process is based on very strict and pessimistic rule set, in the sense that the worst possible result is always expected which reflects the core classical testing approaches.

R_x	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
$M(R_x)$	v_1	v_1	v_1	v_1	v_1	v_4	v_1	v_1

Table 1. Mapped Results of the Running Example

Table 1 shows the results of determining the verdicts of the result sets of the running example. Here, seven out of eight result sets are classified as *Failed*. The remaining result set R_6 derived from the path p_6 is marked as *Passed*.

3.6 Result Selection and Test Report

The last step of the ATE is the selection of one result set as the final result of the ATE. In contrast to the pessimistic approach of the verdict determination, the process of result selection follows a more optimistic approach. Here, the best result set is selected based on their classification. The best case describes result sets that are identified as *Passed*, in contrast to the worst case, which is a result set marked as *Failed*. In addition, test cases rated as *Inconclusive* can be improved by enriching the model in that way that the test case may pass

later. Furthermore, test cases assessed as *Probably Passed* cannot be improved by adding information. Therefore, we define that the verdict *Probably Passed* represents a better case than the verdict *Inconclusive*. As a result, the verdicts are weighted as follows: $v_4 > v_3 > v_2 > v_1$.

$$\Sigma : O \to O_{best} := \{R_i \mid \exists R_i \forall R_j.\ i, j \in \{1, \ldots, |P_{lim}|\} \text{ such that } i \neq j \wedge M(R_i) \geq M(R_j) \wedge |R_i| < |R_j|\}$$

In general, the selection of the best result set as the final test report for the test case is performed in two steps. For this purpose, the function Σ is defined to select in the first step the happy cases $O_{best} \subseteq O$. Second, if $|O_{best}| > 1$ the result sets $R \in O_{best}$ which represents the path with the least steps is chosen as the test report. In context of the running example $O = O_{RE}$, $\Sigma(O_{RE}) = \{R_6\}$ with $M(R_6) = v_4$ which indicates that the test case presented as the running example passed.

Subsequently, the test report reflects the faults and characteristics derived by the ATE to give the modeler hints on possible causes. Therefore, our approach can be seen as Gray-Box Testing as detailed in section 1.

4 Qualitative Evaluation and Critical Discussion

Following the introduction of basics and implementation of the Abstract Test Execution, the approach will be further evaluated qualitatively and critically discussed in the course of this chapter. At the beginning the evaluation of Hagemann et al. [10] should be mentioned, which has already been carried out in the context of the conference contribution. In the course of that evaluation, excerpts from the Automotive Light Control System, originally utilized by Peleska et al. [22], were used to demonstrate the proof of concept. There, a wide variety of defects were introduced into the model through a mutation analysis. Then test cases capable of detecting these defects were tested against the mutated system models using our approach. As a result, this demonstrated the ability of the approach to verify test cases against the system model in an abstract way. This was subsequently done for other parts of the system model, which supports the drawn picture. The same approach was applied to the Ceiling Speed Monitoring model of the University of Bremen, again showing the same possibilities and limitations [3].

In order to provide a meaningful extension of the previous findings on our approach to *Abstract Test Execution*, a qualitative evaluation is carried out. Here, the results gathered so far are compared to the state of the art and put in relation to the technical background of the approach. In order to be able to conduct such a discussion in a reasonable manner, a brief overview of the state of the art is given in advance, which should be seen in relation with the content of the section on related work.

In today's software development, different kinds of tests are performed depending on the applied development process and the desired level of integration

of the software. Depending on the integration level, different knowledge bases are assumed and usually special techniques are used to derive possible test cases from development artifacts. The palette here ranges from black box to white box procedures. In order to execute such test cases, the SUT must be available in a (partially) executable version. Depending on the test level, concepts such as mocking or stubbing are often used to simulate system parts which are missing or lie outside the development context. In contrast to this, the concept of *Abstract Test Execution*, where only model artifacts are used to derive the test results, is used.

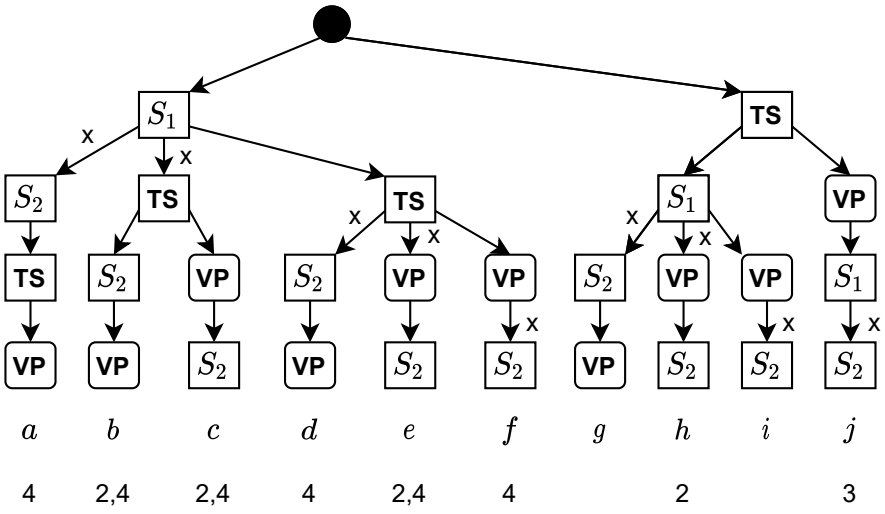

Fig. 9. Path Space limited by rule one

Based on the findings in section 3, we identify that the limitation of path space by merging rules and subsequently their potential impact on the test report needs to be discussed. For better understanding, a path space is generated from the minimal System Model $Init \rightarrow [S_1] \xrightarrow{x} [S_2]$ and Test Model $Init \rightarrow [TS] \rightarrow (VP) \rightarrow End$. The System Model consists of the two system nodes S_1 and S_2 with the exception that S_2 is guarded by x. The Test Model includes the node TS as a Test Step and the node VP as a Verification Point. The representative and slightly limited tree of paths is shown in Figure 9. It is derived by taking merging rule one into account. This fundamental tree of paths represents all cases that could come into play. The included paths can be identified by a letter attached to the end node of each path. In addition, paths that violate the remaining rules are flagged by the numerical identifier of the violated rule. Since the first rule guarantees the consistency of the test cases, the importance of this rule does not need to be discussed further.

The second rule is another approach to reduce the level of abstraction and to bring the approach of ATE more in line with classical testing approaches. Unlike rule one which focuses on the sequence of nodes, this rule focuses on the sequence of edges or guarded edges in particular. From the viewpoint of classical testing, stimuli applied to the system are used to trigger a change of the system state. In case of ATE, such stimuli are expressed as Test Steps. Furthermore, the change of system state is usually bound to conditions. The Test Steps are then used to fulfill the conditions bound to a specific system state to change the system state to that state. Such conditions are modeled by enriching edges of the System Model with instructions. Previously, we referred to such enriched edges as guarded transitions or Guards. Unguarded edges can therefore be ignored, since the change of system state is not bound to any condition. Subsequently, if a guarded edge is inserted before the Test Step that simulates stimuli required to satisfy the guard of the edge, the path can never be fulfilled as represented by the paths b and c. In addition, paths e and h exist where the guarded edge is inserted before the Verification Point *VP*. In this case, the paths are able to meet the requirements of the test cases, since instructions contained in guards generally cannot change the data flow. However, the paths g and i exists which are not excluded by appliance of rule two. As a result, the paths e and h can be excluded without harming the final result of ATE.

Since test cases in which a Verification Point can be fulfilled without the use of system nodes can be considered a bad test design, such test cases can be excluded. Structurally, this can be done by forced insertion of Verification Points after nodes of the System Model. For this purpose, rule three is conducted. The enforcement of this rule prevents verification points from being injected between the initial node and the first system node during merging process. Generally, this leads to the path space always being shortened by exactly one path. In case of the generic path space, the path j is therefore excluded.

The fourth and last merging rule excludes the most paths from the path space shown in Figure 9, but may affect the outcome of the ATE as described in section 3.4. In this context, this rule has the power to exclude six of the represented ten paths. In general, rule four is used to dictate the structure of segments. A segment consists of system nodes and Test Steps followed by a Verification Point. This rule forces test steps of a segment to be inserted before the system nodes of the segment. This results in the structure that a segment starts with Test Steps followed by system nodes and ends with a Verification Point. This change is generally not problematic and mimics a more natural approach of testing. However, if more than one guarded system node is included in the segment the analysis may inadvertently fail. We justify this rule with the abstract character of this approach and the resulting incompatibility with time-dependent system states. Since Test Steps simulate stimuli to trigger system changes that are inseparably linked to time aspects, the concept of the over-assignment of variables is introduced to overcome the incapability of temporal considerations. Here, variables can have more than one valid value during segment analysis as presented in section 3. This solves the problem of temporal incompatibility, since

the needed stimuli to solve a test case are delivered at the right time, but due to the uncertainty factor, test cases solved with the help of such variables are marked to maintain the pessimistic evaluation of the ATE approach.

In conclusion, the limited path space P_{lim} of the generic Test Model and the generic System Model holds the two paths g and h which underlines the possibility that the instruction of the Verification Point `VP` verifies either the system state S_1 or the system state S_2.

5 Related Work

In the context of test execution at model level, the execution of the modeled functionality itself plays a central role. This was originally applied in the engineering context and is known as Model-In-The-Loop Testing [24]. Furthermore, it is important to be able to manipulate the execution of the model with stimuli, as well as to verify the system state (internal or external). In literature, there are many ways of doing this, but there are some parts that differ significantly from our approach.

First, approaches are discussed that consider the model artifacts as input and convert the model into code for execution. For example, this is the basis for the simulation/execution of Matlab/Simulink models, which are therefore converted into C code [14] [6]. The same applies to the approach of Anlauf et al., which is based on so-called Extensible Abstract State Machines [1]. In comparison to the approach presented, however, this type of execution is not applicable to other original models. Zentai et al. have implemented this in a similar way in the context of the MDA-oriented test methodology using the IBM Rhapsody tool and its simulation capabilities [29]. This mitigates the above mentioned problem of input models, but still requires the detour via code representation.

A similar variant for the execution of model artifacts is given by the Foundational Subset for UML (fUML), which represents a subset of UML that has been substantiated with clean semantics [20] [18]. In particular, execution engines have been implemented for this modeling language, which no longer requires upstream translation into code artifacts [9]. This was implemented by Arnaud et al. and extended by more formal concepts like symbolic execution [2]. Similarly, Iftikhar et al. introduced a virtual machine for the execution of timed automata [12]. The disadvantages of such approaches are the same as those mentioned above.

In the context of execution engines, there are approaches that rely on model interpreters. In most cases, a internal model artifact is created for this purpose, which is derived from the input model. Within the MoMuT::UML project, for example, UML models are translated into Object Oriented Action Scripts (OOASs) for the purpose of mutation analysis, which in turn are animated by an interpreter [16]. The test cases are evaluated in this context by means of conformance checks on these representations. This evaluation is realized in particular by formal approaches, which in turn entails limitations.

In contrast to these approaches are the formal verification approaches, which are not the same as executing and testing a SUT, but have a similar goal. In particular, such approaches place special demands on the input models, which usually severely limits their applicability. Various model checking approaches have been presented for decades, but most of them are strongly optimized for the respective application context [17][12][11][4]. At this point, again, our presented approach is much more flexible and does not require the detailed knowledge of formal technologies.

6 Conclusion and Outlook

Within the scope of this work, we have presented a promising approach to the challenges initially displayed. Especially the ability to perform tests in early phases of model-centric software development represents a significant improvement. Based on the presented foundations regarding modeling and analysis-specific constructs, the concept of *Abstract Test Execution* was presented, which performs a comprehensive analysis of the System Model in the context of previously generated test cases. In particular, the concept behind the integrated evaluation of control and data flow properties was presented in detail. The results of this analysis can be compared to a classical test report. In contrast, the range of verdicts has been extended to explicitly represent novel evaluation results in the modeling context and to introduce no room for interpretation within the set of possible results. In the course of the discussion on the presented approach, the meaningfulness of the concept as well as its limitations were particularly emphasized.

At the same time, these limitations indicate possible starting points for improvements and extensions of the current approach. On the one hand, the approach could be extended by concepts that allow time considerations to be carried out on the basis of runtime estimates. For this purpose, however, the model has to be enriched with information or at least given access to data about execution times on the target platform or target technology used. However, this is in some ways contradictory to our overarching goal of applying the approach as early as possible in order to receive early and automated feedback.

On the other hand, the approach could be improved in such a way that not only a set of test cases is evaluated against the System Model, but a complete test model. From a technical point of view, this could result in a significant performance gain, since test sequences that appear in several test cases could be evaluated once.

An abstract view on the presented approach could be a possible development towards an automated decision support for model-centric software development approaches. Based on the collected test results of the *Abstract Test Execution*, this could for example include concrete suggestions for improving or extending the current model.

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 numbers 16KN044137.

References

1. Anlauff, M.: X asm-an extensible, component-based abstract state machines language. In: International Workshop on Abstract State Machines. pp. 69–90. Springer (2000)
2. Arnaud, M., Bannour, B., Cuccuru, A., Gaston, C., Gerard, S., Lapitre, A.: Timed symbolic testing framework for executable models using high-level scenarios. In: Complex Systems Design & Management, pp. 269–282. Springer (2015)
3. Braunstein, C., Haxthausen, A.E., Huang, W.l., Hübner, F., Peleska, J., Schulze, U., Hong, L.V.: Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In: International Conference on Formal Engineering Methods. pp. 380–395. Springer (2014)
4. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**(4), 410–425 (2000)
5. Galin, D.: Software quality assurance: from theory to implementation. Pearson Education India (2004)
6. Gambarotta, A., Morini, M., Saletti, C.: Development of a Model-based Predictive Controller for a heat distribution network. Energy Procedia **158**, 2896–2901 (2019)
7. GmbH, I.: radCase - Model-Driven Generation (2020), `http://www.radcase.com/`
8. Grossmann, J., Serbanescu, D.A., Schieferdecker, I.: Testing Embedded Real Time Systems with TTCN-3. In: ICST. pp. 81–90. IEEE Computer Society (2009)
9. Guermazi, S., Tatibouet, J., Cuccuru, A., Dhouib, S., Gérard, S., Seidewitz, E.: Executable modeling with fUML and alf in papyrus: tooling and experiments. strategies **11**, 12 (2015)
10. Hagemann, N., Pröll, R., Bauer, B.: Towards abstract test execution in early stages of model-driven software development. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, February 25-27, 2020, Valletta, Malta (2020). https://doi.org/10.5220/0008934802160226
11. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. In: International Conference on Computer Aided Verification. pp. 460–463. Springer (1997)
12. Iftikhar, M.U., Lundberg, J., Weyns, D.: A model interpreter for timed automata. In: International Symposium on Leveraging Applications of Formal Methods. pp. 243–258. Springer (2016)
13. Jones, C.: Applied Software Measurement: Global Analysis of Productivity and Quality. McGraw-Hill Education Group, 3rd edn. (2008)
14. Khalesi, M.H., Salarieh, H., Foumani, M.S.: 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 pp. 1–20 (2019)

15. Kleppe, A.G., Warmer, J., Warmer, J.B., Bast, W.: MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional (2003)
16. Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H.: Momut:: UML model-based mutation testing for UML. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). pp. 1–8. IEEE (2015)
17. McMillan, K.L.: Symbolic model checking. In: Symbolic Model Checking, pp. 25–60. Springer (1993)
18. Mellor, S.J., Mellor, S., Balcer, M.J.: Executable UML: a foundation for model-driven architecture. Addison-Wesley Professional (2002)
19. OMG: OMG Systems Modeling Language (OMG SysML), Version 1.3 (2012), `http://www.omg.org/spec/SysML/1.3/`
20. OMG: About the Semantics of a Foundational Subset for Executable UML Models Specification Version 1.4 (2018), `https://www.omg.org/spec/FUML/About-FUML/`
21. (OMG), O.M.G.: UML Testing Profile 2 (UTP 2), Version 2.0 (Dec 2018), `https://www.omg.org/spec/UTP2/2.0/PDF`
22. Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In: IFIP International Conference on Testing Software and Systems. pp. 146–161. Springer (2011)
23. Planning, S.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology (2002)
24. Plummer, A.R.: Model-in-the-loop testing. Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering **220**(3), 183–199 (2006)
25. Pretschner, A., Philipps, J.: 10 methodological issues in model-based testing. In: Model-based testing of reactive systems, pp. 281–291. Springer (2005)
26. Pröll, R., Bauer, B.: A model-based test case management approach for integrated sets of domain-specific models. In: O'Conner, L., Feldt, R., Yoo, S. (eds.) Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), ICSTW 2018, 9-13 April 2018, Västerås, Sweden (2018). https://doi.org/10.1109/icstw.2018.00048
27. Pröll, R., Rumpold, A., Bauer, B.: Applying integrated domain-specific modeling for multi-concerns development of complex systems. Communications in Computer and Information Science **880**, 247 – 271 (2018). https://doi.org/10.1007/978-3-319-94764-8_11
28. Rumpold, A., Pröll, R., Bauer, B.: A domain-aware framework for integrated model-based system analysis and design. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development: February 19-21, 2017, in Porto, Portugal (2017). https://doi.org/10.5220/0006206301570168
29. Zentai, A.: Model-in-the-Loop Testing of a Railway Interlocking System. In: Model-Driven Engineering and Software Development: Third International Conference, MODELSWARD 2015, Angers, France, February 9-11, 2015, Revised Selected Papers. vol. 580, p. 375. Springer (2016)