Technical Report 2010-01:

Model Transformations as a Strategy to Automate Model-Based Testing – A Tool and Industrial Case Studies, Version 1.0

S. Ali, H. Hemmati, N.E. Holt, E. Arisholm, L.C. Briand

{shaukat,hemmati,ninaeho,erika,briand}@simula.no

Simula Research Laboratory and University of Oslo, Norway

Abstract

In recent years, Model-Based Testing (MBT) has attracted an increasingly wide interest from industry and academia. The beneficial use of MBT, however, requires tools that not only automate the testing process, but that also rely in an extensible and configurable architecture that make them adaptable to various contexts of application. Though a number of tools have been developed to support MBT, this technical report introduces a new approach for designing and developing MBT tools that is based on model transformation technology. We report on the experimental development of a novel MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST), which software architecture and implementation strategy supports configurable and extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages. Based on two industrial case studies, we demonstrate the configurability and extensibility of TRUST. We also investigate the challenges and likely cost savings when compared to manual test generation.

Keywords: model-based testing tool, model transformation, automatic test-case generation, model-driven development, state-based system, UML state machine, extensibility, configurability

Contents

1. Introduction

Deriving test cases from a behavior model of a system, known as Model-Based Testing (MBT) [1], is not a new domain of research in software engineering [2]. However, in recent years, the level of interest in industry and academia has been rapidly increasing. This interest can be seen from the many academic studies [1, 3-7] and industrial projects [8-11] on model-based testing being reported. This suggests that there is an increasing awareness of the benefits offered by MBT[1] . But these benefits cannot be realized without having proper tool support, matching the needs of each specific context, for automating MBT [3].

Many tools have been developed to support MBT [10-16]. However, all of them have at least one of the following drawbacks:

- They do not support well-established standards for modeling the System Under Test (SUT). This makes it difficult to integrate MBT with the rest of the development process, which in turn makes the adaptation and use of MBT more costly.
- They cannot be easily customized to different needs and contexts. For example, a tester may want to experiment with different test strategies to help target specific kinds of faults. Furthermore, constraints can evolve, e.g., the test script language in a company can change.

Thus, we propose an MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST), whose software architecture and implementation strategy facilitate its customization to different contexts by supporting configurable and extensible features such as input models, test models, coverage criteria, test data generation strategies, and test script languages. We define configurability as the ability of selecting among different options provided by the tool for a feature. For example, the tool is configurable with respect to coverage criteria if it lets the user select among several coverage criteria such as all transitions and all round trip path coverage criteria [17]. We define extensibility as the ability of providing more options for a feature without any modification in the components that are not responsible for the feature. For example, providing support for generating test scripts in more languages is considered as extending the tool.

Our approach, which is inspired from the Model-Driven Architecture (MDA) standard [18], relies on a series of model transformations to generate test cases. The main idea is to design a tool in such a way that its different components provide and require standard interfaces with input and output models based on standard metamodels. Each component in this tool is responsible for one feature (e.g., test model, test data, etc.) involved in the process of generating test cases. This separation of concerns and provision of standard interfaces make TRUST configurable and extensible. In addition, model transformation technology helps the developer upgrade the components with a new set of transformations from standard inputs into well-defined outputs.

The approach allows instantiating new, context specific MBT tools by extending or configuring TRUST with customized features, such as input models, test models, coverage criteria, test data generation strategies, and test script languages. To demonstrate the configurability and extensibility of TRUST, we instantiated two tools for two case studies, by extending and configuring TRUST with different test models, coverage criteria, and test scripting languages. On the basis of the case studies, we also evaluate the costs, challenges, and likely benefits of TRUST in particular and MBT in general. Since the case studies concern systems whose behavior is mostly state driven, they are largely based

on input models represented as Unified Modeling Language (UML) state machines [19]. However, the description of our approach, many of our results, and lessons learned are not specific to state-based testing (SBT).

The remainder of this technical report is organized as follows. Section 2 provides the relevant background on model-based testing and model-driven development, with a focus on SBT. Section 3 introduces existing model-based testing tools and provides an analysis of their extensibility and configurability. Section 4 presents key requirements, architecture and implementation details of TRUST. In Section 5, we present experiences from two industrial case studies, in which we applied TRUST. Benefits and limitations of applying TRUST are discussed in Section 6. Finally, Section 7 outlines future work and concludes the technical report.

2. Background

In this section, we define basic concepts relevant to model-based testing, with an emphasis on automated testing based on UML state machines. In addition, model transformation concepts from the model-driven development domain will be introduced since these are the basis for our test case generation approach.

2.1 Model-based and state-based testing

The general process of MBT that we use in this technical report starts with modeling the SUT and making it ready for test generation. The next step is deriving abstract test cases from the test ready model according to a test strategy, which is typically defined based on a test model and coverage criteria to guide its traversal [20]. Finally, executable test cases are generated using abstract test cases and input test data.

In our case studies, we will apply this general MBT approach to state-based testing (SBT). Many systems, such as embedded real-time systems [21], telecommunication systems [2, 22], and multimedia systems [23], exhibit state-driven behavior. UML state machines, which are extensions of traditional Finite State Machines (FSM), can be used to model such behavior. Traditional FSMs cannot model software systems with concurrent behavior. Concurrency in UML state machines is modeled using composite states with two or more regions [19]. When modeling complex software systems with FSMs, the number of states and transitions can grow exponentially with system size. This can be handled by UML state machine features for modeling submachines. Many tools (e.g., [24, 25]) support the modeling of UML state machines.

To apply MBT on UML state machine as the input model, several test strategies are presented in the literature, such as piecewise, all transitions, all transitions k-tuples, all round-trip paths, M-length signature, and exhaustive coverage [17]. For example, the all round-trip strategy requires that all paths in a state machine that begins and ends with the same state must be covered. To cover all such paths, a test tree (consisting of nodes and edges corresponding to states and transitions in a state machine) is constructed by depth-first traversal of the state machine. The test tree corresponding to the all round-trip strategy is called a transition tree. A node in the transition tree is a terminal node if the node already exists anywhere in the tree that has been constructed so far or is a final state in the state machine. Now, by traversing all paths in the transition tree, we cover all round trip paths and all simple paths (the paths in the state machine that begins with the initial state and ends with the final state) [17]. Another stopping criterion for the transition tree construction is proposed in [26], where a node is terminal if (i) it is a final state of the state machine or (ii) it is a node that already exists on the path that leads to the node. This stopping criterion makes the all round-trip strategy more rigorous, and

thus gives more coverage. This strategy has been experimentally evaluated to be more cost-effective than the all transitions and all transition pairs criteria [26]. Henceforth, the transition tree or all round-trip paths coverage criterion refer to the modified versions proposed in [26].

To automate testing based on UML state machines, test data must be generated to fire triggers associated with transitions, and the triggers typically require parameter values. Test data can be generated randomly from the possible set of values, or using more sophisticated techniques such as constraint solvers [27], or search-based techniques (for example using genetic algorithms for test data generation [28]) .

Constraints defined on UML state machines, such as state invariants, guards, and pre/post conditions of triggers, should be evaluated during the execution of the generated test cases. As shown by many studies, this is a very effective way to detect failures [3, 29]. These constraints are usually written as OCL expressions in the context of UML. Examples of available OCL evaluators are OCLE 2.0 [30], OSLO [31], IBM OCL parser [32], and EyeOCL Software (EOS) evaluator [33].

2.2 Model-Driven Architecture

Model-Driven Architecture (MDA) is a software development approach initiated by Object Management Group (OMG) [34]. The MDA approach focuses on developing software based on the incremental development of models at various levels of abstraction. MDA aims at providing a set of guidelines with tool support to create, process, and transform models. This entails the reliance on a standardized modeling framework, such as the Eclipse Modeling Framework (EMF) [35]. Being an Eclipse-based modeling and code generation framework, EMF enables the construction of tools and applications based on models [35].

2.2.1 The MDA Process

A high level process illustrating the main phases of MDA development is shown in Figure 1. MDA defines three types of models at three levels of abstractions. A Computation Independent Model (CIM) focuses on the requirements of the system and its environment [18]. This model is independent from implementation and platform specific details such as information about the programming language to be used for the system implementation and the operating system on which the system will be deployed. Some well-known languages for CIM modeling are the Business Process Modeling Notation [36] and Business Process Definition Metamodel (BPDM) [37].

A Platform Independent Model (PIM) is a system model that focuses on the operation of the system [18]. A PIM, however, is independent from how the system is going to be implemented (i.e., independent from the technical details of the development platform). PIMs can be modeled in generic languages such as UML or domain specific languages (DSL) such as WSDL [38] and PIM4Agents [39]. The transformation from CIM to PIM is manual. However, efforts are being made to automate the process [40, 41].

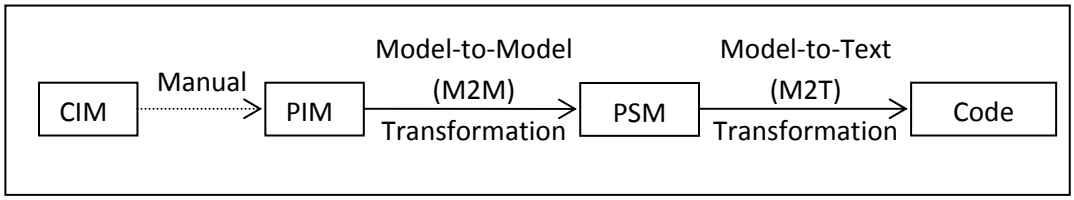

FIGURE 1. MDA PROCESS

The third type of model defined in MDA is the Platform Specific Model (PSM). This model describes a system in terms of implementation constructs. For instance, if the implementation language of the system is Java, then the PSM will be defined in terms of Java constructs. The transformation from PIM to PSM models is mostly automated and performed using Model-to-Model (M2M) transformation languages. Well known M2M transformation languages include Kermeta [42] and ATL [43]. A typical process for M2M transformation is illustrated in Figure 2.

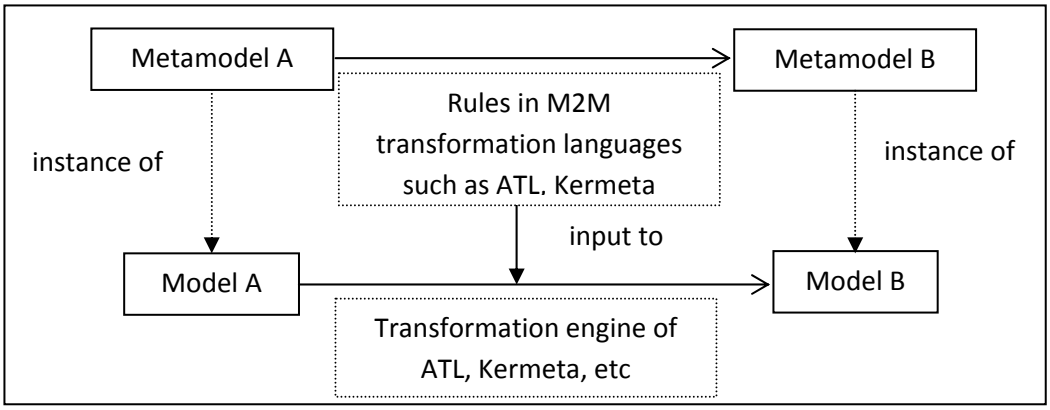

FIGURE 2. THE PROCESS OF MODEL-TO-MODEL (M2M) TRANSFORMATION

Using M2M transformation languages, we can define mapping rules that map elements of one metamodel to the elements of another metamodel. Once the rules are defined, a transformation engine uses these rules and transforms an instance of *Metamodel A* (*Model A* in Figure 2) into an instance of *Metamodel B* (*Model B* in Figure 2). We provide below a brief description of the languages that we used in the current implementation of TRUST.

Atlas Transformation Language (ATL): ATL is a hybrid (declarative and imperative) language and a toolkit used to produce a set of target models from a set of source models [43]. It is recommended to use a declarative style to write rules, although complex rules may require using imperative features of the language. An advantage, though, is that the declarative style results in less transformation code as compared to writing transformations using imperative languages. This language therefore supports both styles of writing transformation rules. ATL is available as an open-source plug-in for Eclipse and uses EMF to handle the models.

Kermeta: Kermeta is an environment for metamodeling based on an object-oriented DSL (Domain Specific Language). It is built as an extension to the Essential Meta Object Facility (EMOF) [44] and is available as an open-source Eclipse plug-in [42]. EMOF is used to specify the structure of a model but not its behaviour. Kermeta, on the other hand, allows operation semantics to be added to such object-oriented metamodels. Kermeta is aspect-oriented and strongly typed [42]. It can be used for many purposes, including the simulation of metamodel behaviour, the verification and validation of models against metamodels, the development of model-driven tools, and the transformation and weaving of models [45].

The final step is the transformation of PSM into code [18]. There exist languages for Model-to-Text (M2T) transformations such as JET [46], Acceleo [46], Xpand [46], and MOFScript [47]. A typical process of M2T transformation is illustrated in Figure 3.

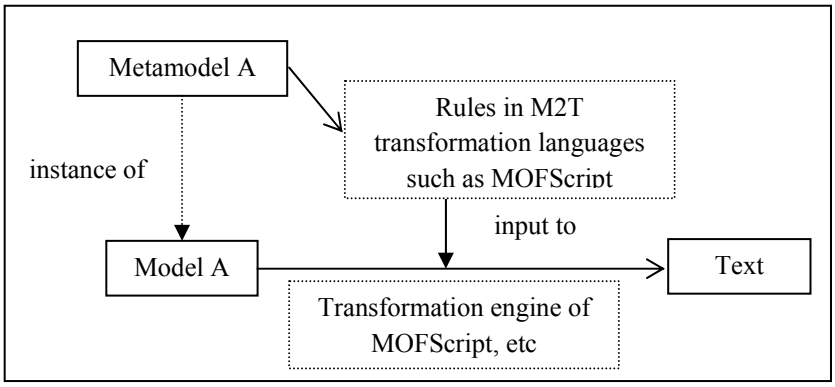

FIGURE 3. THE PROCESS OF MODEL-TO-TEXT (M2T) TRANSFORMATION

M2T transformation languages such as MOFScript can be used to define transformation rules on a metamodel (*Metamodel A* in Figure 3). These rules map elements of a metamodel to textual elements such as constructs of an implementation language. The transformation engine of a M2T transformation language uses rules to transform an input model (*Model A*), which is an instance of *Metamodel A* (see Figure 3), into text according to a specific grammar. We provide below a brief description of MOFScript, a M2T transformation language that we used in TRUST to generate executable test scripts from a test model.

MOFScript: MOFScript is a language for generating text (such as implementation code, documentation, reports, etc) from models using M2T transformation rules. Rules (also referred to as operations) are called explicitly and are written in an imperative way similar to programming and scripting languages. There exists an eclipse plug-in built on EMF, which supports the MOFScript language [47]. This means that any EMF-based model can be transformed into text by MOFScript. It also has the ability to call external Java functions.

2.2.2 Classification of transformations

Huber [48] describes two categories of transformations based on the *abstraction level* and *metamodel*. Vertical transformations change the abstraction level of a model from a higher to a lower level of abstraction. In contrast, in horizontal transformations, the abstraction level remains the same. PIM to PSM, or PSM to code transformations are examples of vertical transformation. For instance, generating Java code from the Java metamodel is a vertical transformation. Refactoring is an example of horizontal transformation because a refactored model remains at the same level of abstraction as the original model.

A second way to classify transformations depends on the *metamodels* on which the transformation is defined. If the transformation is defined on source and target models instantiated from the same metamodel, it is labeled as endogenous; otherwise it is exogenous. The transformation from PIM to PSM is an example of exogenous transformation because metamodels for PIM and PSM will necessarily be different.

3. Related Work

Several well-known, model-based testing tools have been developed in recent years, such as TDE/UML (Siemens) [10], SpecExplorer (Microsoft) [11], and IBM Rational Functional Tester [9].

Based on just three sources, we were able to find references to more than 50 model-based testing tools. In this technical report we focus on configurations of TRUST with state machines and thus we are interested in comparing this SBT version of TRUST with other SBT tools. Consequently, we focus our discussion of related work to tools that (*i*) are (partly) based on UML state machines, (*ii*) automatically generate executable test cases including test oracles, and (*iii*) have at least some support for extensibility and configurability.

After applying these criteria on more than 50 model-based testing tools [1, 8, 49], we were left with five tools [12-15, 50]. We then collected information regarding the extensibility and configurability of their different features (input model, test strategy, and output language of the tools). Since TRUST generates test cases from UML state machines, we collected the following information related to UML state machines from the tools to determine the degree to which their input model can be extended and configured:

- UML metamodel: As the UML metamodel undergoes changes on a regular basis, a test tool must have the ability to accommodate these changes with reasonable effort.
- Constraint evaluation: A UML state machine may contain various types of constraints such as state invariants and guards. These constraints can be defined in different languages such as OCL, Java, or any other tool-specific language. Therefore, the tool architecture should easily accommodate changes in constraint language or evaluation technology.
- Support for UML profiles: UML profiles provide an extension mechanism to support modelling for particular domains and platforms, for instance the MARTE profile for modelling real-time and embedded systems [51]. An extensible tool should be able to accommodate models based on different UML profiles.

The last part of the analysis considered the tools' extensibility and configurability regarding test models and coverage criteria, test data generation techniques, and test script languages since these are the components of a test strategy (Section 2). We will now provide a summary of our analysis regarding the extensibility and configurability of these five tools.

Qtronic [12] is an eclipse-based tool used to generate test scripts from system models specified in QML (Qtronic Modelling Language). This language is based on UML and Java/C# compatible syntax and is supported by a tool called Qtronic Modeler [12]. Qtronic can be configured for the following state machine coverage criteria: state coverage, transition coverage, 2-transition coverage, all paths coverage, and implicit consumption (criterion to check that system ignores transitions that are not explicitly defined on a state), branch coverage, atomic condition coverage, and boundary value pattern (to cover boundaries of decisions in guards) [12]. Qtronic supports extensibility by means of plug-ins, which can be coded in C++ or Java. These plug-ins can be written for changing the test scripting language, logging formats, and test execution type (online vs. offline test execution). This means that Qtronic can only be configured for predefined coverage criteria and cannot be extended to additional test models, coverage criteria, and test data generation other than what is already provided. However, Qtronic can be extended for different test scripting languages by implementing specific plug-ins.

The tool Automatic Test Generator/Rhapsody (ATG) [15] is a module of I-Logix STATEMATE and Rhapsody products. ATG can be configured to generate test cases from models based on a set of coverage criteria such as state and transition coverage, and modified condition/decision coverage on guards in state machines ATG doesn't provide any extension mechanisms.

AGEDIS is a tool for automated model-driven test generation and execution for distributed systems. It has been made usable and interoperable with external tools by defining clear external interfaces in the tool. In addition, well-defined internal interfaces make AGEDIS more reusable. For instance, the user can define or select a coverage criterion through a test generation directives interface which includes coverage criteria, constraints (which are additional criteria) on the test suite, and test purposes [52]. There is also a defined interface for abstract test cases that makes the tool open for later extensions to other test script languages than TTCN-3. However, most of these interfaces are not defined by standard well-known languages. For example, standard OCL constraints on an input UML model should be transformed into their interface language (IF) format [14].

ParTeg (Partition Test Generator) [50] is a test generation tool dealing with the reuse of state machines for automatic test case generation in the context of product lines. Regarding configurability, ParTeg allows the user to choose from a set of coverage criteria: state coverage, decision coverage, and modified condition/decision coverage as well as boundary value coverage criteria for input data to cover boundaries of decisions in guards. No attempt was made for making it extensible and configurable with respect to input models, test data generation, and output languages.

MOTES [13] is a model-based testing tool for generating TTCN-3 tests. MOTES accepts Extended Finite State Machines (EFSM) as input and requires test data to be prepared manually before the test case generation phase. However, it provides some extensibility opportunities for output models by having a standard input interface. For example, UML state machines created using third party CASE tools (for example Poseidon [53]) can be imported to MOTES, although state machines must be flat without concurrent and hierarchical states. MOTES provides a configurable set of coverage criteria such as selected states, selected transitions, and all transitions.

Table 1 gives a summary of the abovementioned tools regarding their extensibility and configurability for respectively the input model, test strategy, and test scripting output language. A clear conclusion from the summary is that current tools usually support configurable coverage criteria (within a limited set) and three of them are extensible regarding new output languages. However, hardly any SBT tool provides support for extending it to new input models or test data generation strategies. Therefore, we were encouraged to develop an extensible and configurable tool with well-defined interfaces and simple extensibility mechanisms. The next section describes the requirements for such a tool and explains our solution in details.

TABLE 1 EXTENSIBILITY AND CONFIGURABILITY OF THE CURRENT SBT TOOLS

Tool name	**Input model**			**Test strategy**		**Test scripting output language**
	UML metamodel	**Constraint evaluator**	**UML profiles**	**Test Model and coverage criteria**	**Test data generation**	
Qtronics	-	-	-	Configurable	Configurable	Extensible and Configurable
ATG	-	-	-	Configurable	-	-
AGEDIS	-	Extensible	-	Extensible and Configurable	-	Extensible

MOTES	-	-	-	Configurable	-	Extensible
ParTeg	-	-	-	Configurable	-	-

4. Requirements, Design, and Implementation of TRUST

In this section, we discuss the main requirements of TRUST, define and justify our architectural decisions, and choices of technologies. In addition, details regarding the test case generation and test case execution procedure will be provided.

4.1 Requirements and Approach

To be optimal from a practical standpoint, we want a tool that supports all features of UML. However, as we discussed in Section 3, this may not be enough – extensibility to various UML extension profiles, such as MARTE [13] and UML QoS profile [54], may also become a requirement in future applications. What we need is a tool that shows versatility in various contexts. Adding different output scripting languages (such as C++, Python, and Java), test models (such as transition trees and testing flow graph [55]), coverage criteria on test models (such as all transition and all round-trip path coverage), and test data generation techniques (such as random and adaptive random search [56]) for different application domains and systems are examples of useful extensions for TRUST.

In addition, the tool should be easily configurable to satisfy varying requirements, which means that the user should be able to easily configure features such as input model, coverage criteria and test data generation strategies. High configurability enables testers to experiment with various techniques without significant effort and changes in the tool implementation. This is of practical importance as different test models, coverage criteria, and test data generation techniques helps in targeting different types of faults. Figure 4 shows the summary of the tool requirements.

The approach that we have taken for implementing TRUST (Figure 5) is based on model transformations. The idea (inspired by MDA concepts, introduced in Section 2.2.1) is to generate a test model using a series of horizontal (endogenous and exogenous) model–to- model transformations on an input design model, modeling the PIM of a system. Then, a vertical, exogenous model–to-text transformation is used to generate test scripts.

We found this approach very well-suited for developing TRUST, firstly due to the fact that it addresses our extensibility and configurability requirements. Each component of TRUST implements one set of transformation rules (e.g., transformation from test tree to test cases). Each component has well-defined interfaces with other components. More specifically, each interface provides output to, or requires inputs from, other components by means of intermediate models conforming to metamodels. Secondly, separation of concerns among components has made each transformation responsible for providing one feature such as test model, test data, and test scripts. Therefore, adding a new feature (for example, outputting test scripts in a new language) can be achieved by writing a new set of transformation rules in one of the components, without affecting the other components. Thirdly, in the model transformation-based approach, the transformation language provides the developer with direct support for navigating, creating, and manipulating a model, based on its metamodel. Generally, the transformation rules are relatively compact and easy to read, write, and change.

REQ1. The tool should handle UML diagrams such as state machines, sequence diagrams, activity diagrams, and OCL as the constraint language for UML diagrams

REQ2. The tool should have a configurable and extensible input model

REQ2.1. The tool should be extensible for new UML profiles and configurable for existing UML profiles

REQ2.2. The tool should be extensible for changes in the UML metamodel and configurable

FIGURE 4 REQUIREMENTS OF TRUST

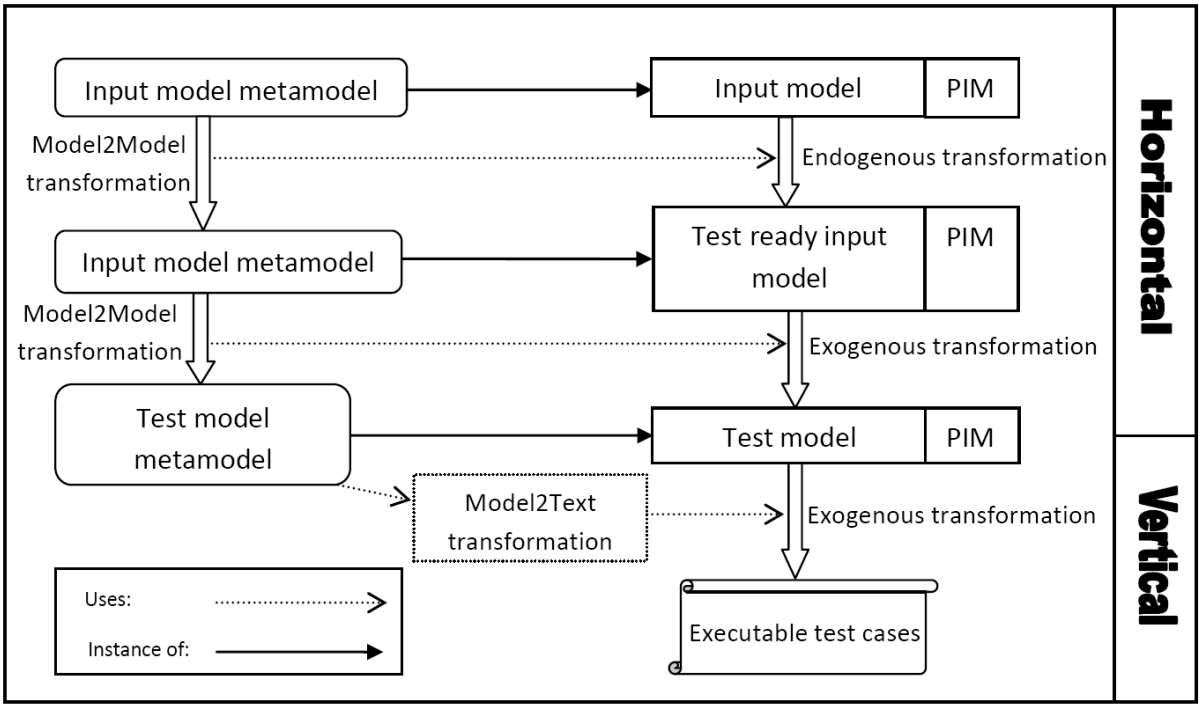

FIGURE 5. MODEL TRANSFORMATION-BASED APPROACH FOR TEST CASE GENERATION

In this technical report, we configured TRUST with UML 2.0 state machines as the input model. *REQ1* in Figure 4 is refined accordingly as follows: The tool should accept UML 2.0 state machines with support for concurrency and hierarchy. Constraints on state machines may be written in OCL because it is an OMG standard for writing constraints on UML diagrams. Furthermore, the general model transformation-based approach, given in Figure 5, is instantiated on UML 2.0 state machines, as shown in Figure 6. Required activities, technologies, and the procedure for this approach will be explained in the remainder of this section.

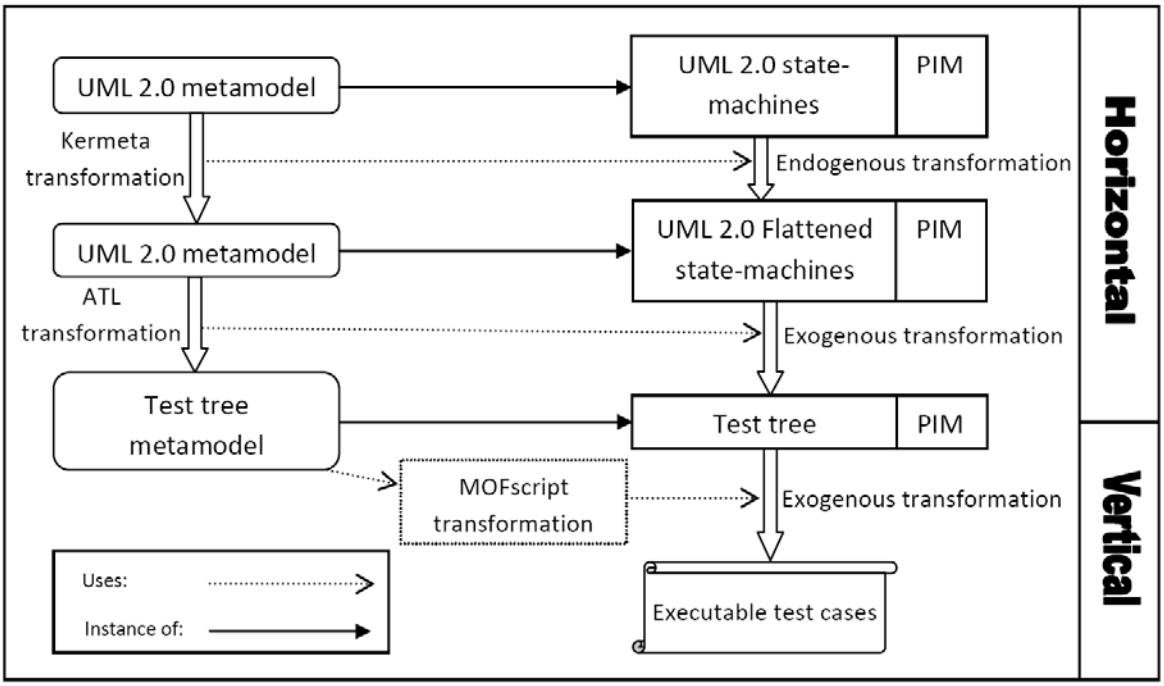

FIGURE 6. MODEL TRANSFORMATION-BASED APPROACH FOR TRUST WHEN CONFIGURED FOR STATE MACHINES

4.2 Developing TRUST using a model transformation approach

In this section, we describe the activities involved in developing TRUST when configured for SBT. Firstly, transformations must be specified and implemented for various activities (activities A1, A3, and A6 in Figure 7). Since these transformations are applied on metamodels, we may need to define input metamodels (activity A2 in Figure 7). However, since some metamodels already exist (e.g., the UML 2.0 metamodel), we do not need to define all metamodels from scratch. Secondly, to make the test cases executable, test data is required (activity A5 in Figure 7). Finally, we need to develop a method for evaluating the OCL constraints that are defined on UML state machines (activity A4 in Figure 7). In Figure 7, the notes associated with the activities show the technologies selected to implement each activity. These choices will be justified below.

4.2.1 From UML state machines to test models

The first SBT activity is *flattening* the input state machine. As explained, our input behavioral model is a UML 2.0 state machine that allows complex structures like simple-composite states, orthogonal states, and submachine states. Testing can be performed directly on such state machines, but this requires rather complex strategies, because such structures complicate the traversal and analysis of the state machine. An alternate approach is to flatten the state machines first, by removing concurrency and hierarchy, and then apply a test strategy. We implemented the latter for obtaining a better separation of concerns and lesser analysis complexity.

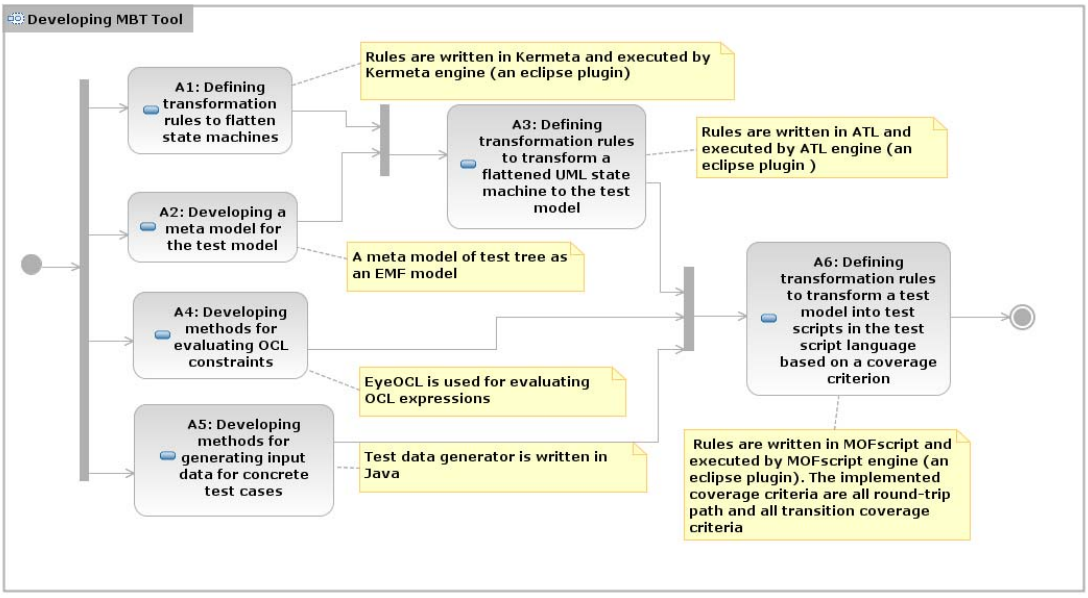

FIGURE 7. ACTIVITIES AND TECHNOLOGIES FOR DEVELOPING TRUST USING MODEL TRANSFORMATIONS

Several algorithms are reported in the literature to flatten concurrent and hierarchical state machines [17, 57]. However, to the authors' knowledge, these algorithms are partial and do not provide flattening of *both* hierarchy and concurrency. Thus we decided to implement our own flattening algorithm for UML 2.0 state machines. The implemented algorithm is a stepwise process that allows the user to modify the UML model at several points during the transformation towards the flattened version. The first step in the flattening process is to search for all nested levels for submachine states and transform these into a set of simple-composite states. Next, all simple-composite states with one region are transformed to a set of simple states or orthogonal states. If there are orthogonal states present in the model, these may now be transformed to simple-composite states. Finally, the simple-composite state(s) created in the previous step are transformed to a set of simple states.

The result is a state machine consisting of an initial state, simple state(s) and possibly a final state. The flattening follows a set of transformation rules implemented in Kermeta [42]. The key aspects in these rules address 1) how to combine concurrent states, and 2) how to redirect transitions. Redirecting transitions may require duplicating transitions, changing source or target states, combining transition information (triggers, guards, transition activities, and state entry/exit activities). Interested readers may consult the report by Holt et al. [58] for more detailed information about the flattening algorithm and its corresponding transformations.

Figure 8 below shows a Kermeta rule used for flattening of simple composite states. Incoming transitions to an entry point in a simple composite state are redirected to each of the outgoing transitions of the same entry point. The small example below provides the Kermeta rule that identifies the incoming transitions that will be redirected by calling another Kermeta rule.

Once the flattened state machine is generated, it is transformed into a test model. This transformation requires three inputs: the source metamodel, the source model (which is an instance of the source metamodel), and the target metamodel. The source metamodel is a metamodel for the flattened state machine, which is the same as the target metamodel for the flattening transformation step in Kermeta. The output of the Kermeta transformation, a flattened state machine, provides the second input, which

is the source model. The last input, the target metamodel, is a metamodel for a test model. The expected structure and content of a test model is strongly dependent on the selected test strategy. In the current version of TRUST, test model conforms to a test tree metamodel. Figure 9 shows the metamodel developed as an Ecore file, based on EMF (Eclipse Modeling Framework). The metamodel represents a tree with a starting node, called alpha, and its outgoing edges. Each edge in the tree has a target node and may have several children that are the target node's outgoing edges. Similar to transitions in a UML state machine, each edge in the test tree metamodel is associated with at least one trigger and may have an associated guard and effect. In addition, each node may have an associated state invariant.

```
/**
* Rule getTransitionsTargetedInEntryPoint identifies the incoming transitions to this
* vertex which is an entry point.
*/
operation getTransitionsTargetedInEntryPoint (r: Region) : Set<Transition> is do

        //create a set of all incoming transitions to this entry point
        var IN : Set<Transition> init Set<Transition>.new

        //add each incoming transition to the set
```

FIGURE 8. EXAMPLE OF KERMETA FLATTENING RULE.

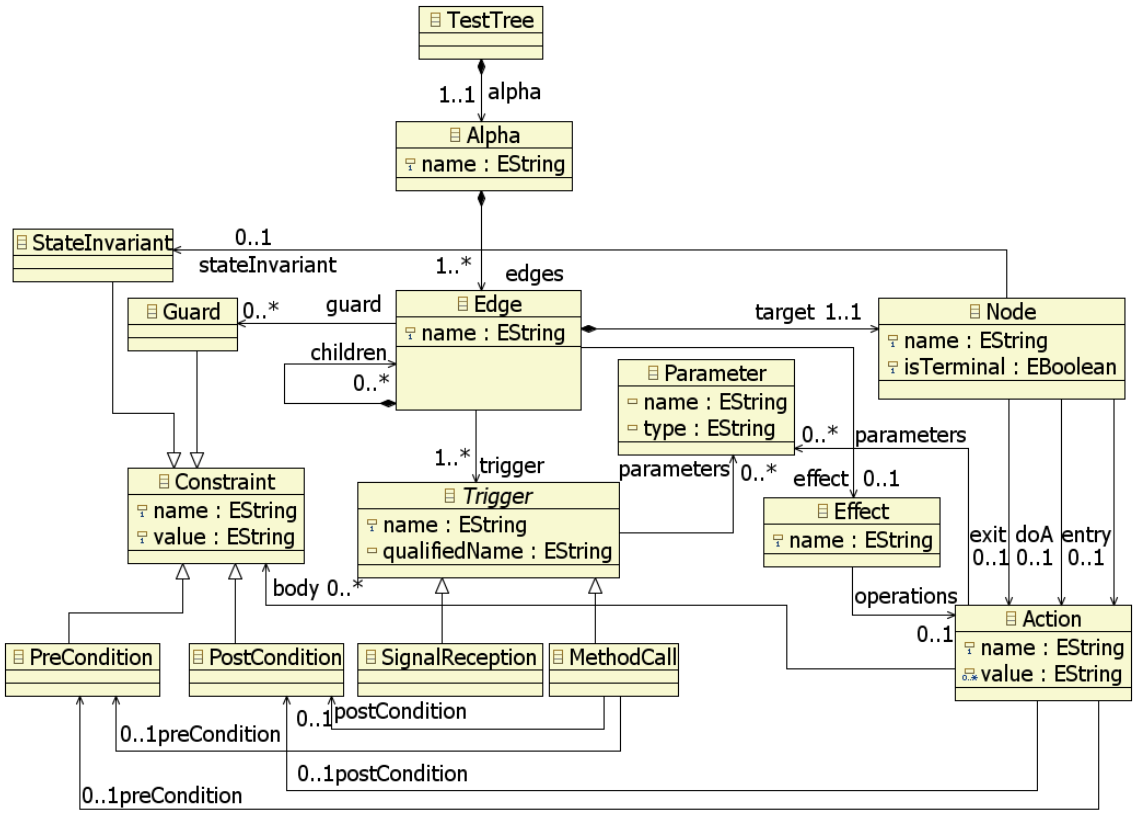

FIGURE 9. TEST TREE METAMODEL FOR THE EMF

The flattened state machine is transformed into a test tree (e.g., a transition tree for all round-trip paths coverage criterion) by a set of ATL transformation rules that take the three abovementioned inputs as parameters. We chose the ATL transformation language because most mappings in this step are simple (mainly one-to-one), which makes a declarative approach the best choice. In declarative approaches, as opposed to imperative ones, control flow and the application order of rules are not explicit. Therefore, usually the transformations in declarative languages have less transformation code and are more comprehensible than imperative languages [48]. This transformation could also have been implemented with any other declarative, hybrid, or even imperative language. We decided, however, to stick with an Eclipse-based technology so as to develop the entire tool on a consistent platform. For example, the transformation rules, instantiating a transition tree from the test tree metamodel, start from the initial state of the state machine which is mapped to the *Alpha* node in the test tree. All outgoing transitions of the initial state are also mapped to the outgoing edges from the *Alpha* node. This process of mapping transitions to edges is applied recursively on the target state of all transitions in the state machine. Finally, the recursive rule stops when it reaches a leaf or a state that has already been visited in the same path starting from the *Alpha* node (all round-trip paths coverage [26]). An example of ATL transformation rules is shown in Figure 10 that maps the *Constraint* metaclass (*SM!Constraint*) including its *name* and *value* properties from UML metamodel (*SM*) to the *Constraint* class (*transitiontree!Constraint*) in test tree metamodel (*transitiontree*) and its *name* and *value* properties.

```
rule Constraint2Constraint{
        from
                a:SM!Constraint
        to
```

FIGURE 10. EXAMPLE OF TRANSFORMATION RULE IMPLEMENTED IN ATL

4.2.2 From the test model to executable test cases

The generated test tree is the input for the next transformation, which generates the executable test cases. We chose the MOFScript [47] transformation language for several reasons. Firstly, it supports the MOF standard [44], which means that it can transform any MOF-based model-to-text. Secondly, it is an imperative language for writing transformation rules similar to many programming and scripting languages. This makes the MOFScript language easy to use and understand. Thirdly, MOFScript provides access to external Java libraries. This makes the language very suitable for our context because we need to access a test data generator (implemented in Java) during the transformation to obtain test data. MOFScript transformations require the source model and its metamodel, which are readily available from the previous step. There is no need to provide the grammar of the output language as an input to TRUST, but of course defining transformations requires its definition.

Each path in the test tree represents an abstract test case. Thus, an abstract test case consists of a sequence of nodes and edges. Nodes are mapped from states in the state machine and states are defined by state invariants, which are OCL constraints serving as test oracles. An edge contains all the information related to the trigger including event (e.g., an operation call or a signal reception), a guard, and an effect from the state machine's transitions.

The MOFScript transformation traverses the test tree (e.g., the transition tree) to obtain the abstract test cases and transforms them to concrete (executable) test cases, which are written in a test scripting language. However, it is possible to generate several concrete test cases from an abstract test case by using different test data values. There are many possible test data generation [59, 60] approaches which are applicable in different situations. What we implemented in the first version of TRUST is the simplest method, which is random data generation for operation calls. This test data generator is written in Java and provides random values for the parameters of triggers. However, such a test data generation technique is not suitable when transitions are guarded and parameters of the triggers are used in the guards. The MOFScript rule shown in Figure 11 illustrates how a trigger is mapped from the test tree for all transition to C++ test cases. The rule maps the name of the trigger event operation and then uses another mapping rule, *'mapParameter'*, to map the parameters for the trigger event operation.

```
/**
 * rule 'mapTrigger' generates the C++ code to invoke the operation implementing
 * the trigger event. Rule 'mapParameter()' is called to map parameters in the
 * trigger event operation call. Each trigger is either a MethodCall,
 * SignalReception, or Timer.
 * @param triggerWithParam List, the total generated output for a trigger as String.
  * @param noOfParam Integer, temporary helper variable used for counting parameters.
 */
transitiontree.Trigger::mapTrigger(){

    var triggerWithParam : List;
    var noOfParam : Integer = 0;

    if(self.oclGetType().equals("MethodCall") or  self.oclGetType().equals("SignalReception") or
    self.oclGetType().equals("Timer"))
```

FIGURE 11. EXAMPLE OF TRANSFORMATION RULE IMPLEMENTED USING MOF SCRIPT

When executing test cases, OCL expressions in guarded transitions should be evaluated at runtime to detect failures. For the same reason, the state invariants associated with states must also be evaluated at runtime. One way to evaluate such OCL constraints is to translate them into a test-scripting language. The constraints will then be evaluated during the execution of the test scripts. Compiler-compilers technologies [61, 62] may be used to translate constraints in one language to constraints in another language. This approach however is not reusable across contexts with different test-scripting

languages. For example, if we have transformation rules that transform OCL constraints to C++ and the test script language changes to Java, we then need to define new transformation rules from OCL constraints to Java expressions. An alternate approach is to use an existing OCL evaluator [30-33] that is called during the execution of a test case to evaluate the OCL constraints. This approach requires an object model of the SUT at runtime, representing the current state of the system. This model along with the constraint to be evaluated is passed to the OCL evaluator, which in turn returns the result of the evaluation. This approach is reusable across contexts because the only required change for each output language is to use its appropriate invocation method for calling an external library (the OCL evaluator). On the other hand, this approach may slow down the test execution as the OCL evaluators are being called at runtime. In both approaches, we need a mechanism to query the current state of the SUT and evaluate constraints on the current state of the SUT. Querying the current state of the system depends on the implementation of the SUT and the test script language's facility to access the state of the SUT. For instance, if the SUT is implemented in C++ and test script language is C++, the state of the SUT may be queried using getter methods of the SUT. In one of our case studies, we did not have direct access to the code of the SUT. Instead, special macros provided by the test script language were used to access the state of the system. Since we wanted our tool to be reusable in different contexts, we decided to use an OCL evaluator that can be invoked from test scripts. Therefore, we had to choose an evaluator that was efficient in terms of evaluating expressions, for example that does not require to be called several times for evaluating a single expression. After investigating several OCL evaluators such as OCLE 2.0 [30], OSLO [31], IBM OCL parser [32], and EyeOCL Software (EOS) evaluator [33], we chose EOS as we found this to be the most fitted evaluator for our requirements. Since EOS is a Java package, to invoke methods from its classes we need to have access to Java from a test script. For example, in one of our case studies, test scripts were in a python-based scripting language. In order to access EOS from Python, we used Jpype [63] which is an extension to Python giving access to Java libraries through interfacing at the native level in both virtual machines (Java and Python). In the other case study, we used the Java Native Interface [64] to access the EOS in test scripts in C++.

To evaluate OCL expressions, EOS requires class and object diagrams to be loaded into its memory. In order to accomplish this, we wrote another MOFScript transformation that takes the UML class diagram (modeling state variables, method calls, and signal receptions of the SUT) as input and generates a Java wrapper class that includes a set of EOS method calls for making class and object diagrams. This wrapper class is generated before executing the test tree to test scripts transformation. During test executions, we create the required object model based on the current values of system state variables.

4.3 Test case generation process

In this section, we will discuss how we designed and implemented the activities discussed in Section 4.2. Figure 12 depicts the architecture of TRUST, which consists of five components. Table 2 shows the mapping between each activity and a component. Each component has provided and required interfaces with other components to ease extensibility and configurability, as discussed in Section 4.1. Each interface provides or requires models that are instances of well-defined metamodels. For example, the TestModelGenerator component in Figure 12 requires an interface from the TestreadyModelMaker component to access the flattened state machine, which is an instance of the UML metamodel. In addition, the *TestModelGenerator* component provides an interface to the *TestScriptGenerator* component to access the test tree (e.g., the transition tree), which is an instance of the test tree metamodel. The architecture shown in Figure 12 was developed with the aim to support extensibility and configurability. For instance, if TRUST needs to be extended to handle C++ as an

output test scripting language, the only component to modify is the *TestScriptGenerator* component where new transformation rules in MOFScript must be defined. The other components do not require any change. Each component also has clearly defined configuration parameters that can easily be adjusted. For instance, if the coverage criterion is required to be changed from *All round-trip paths* to *All Transitions*, we only need to change the input coverage criterion in the *TestScriptGenerator* component.

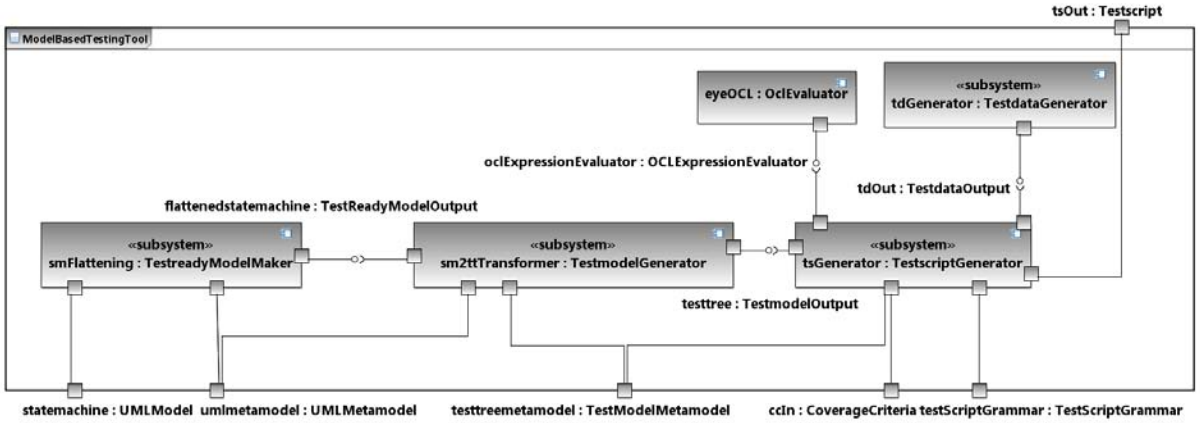

FIGURE 12. ARCHITECTURE DIAGRAM OF TRUST CONFIGURED FOR UML STATE MACHINES

TABLE 2 ACTIVITIES IMPLEMENTED BY EACH COMPONENT

Component	Activity
TestreadyModelMaker	A1
TestModelGenerator	A5
TestScriptGenerator	A6
OCLEvaluator	A3
TestDataGenerator	A4

Figure 13 shows interactions between different components that take place at runtime when TRUST is executed with an input state machine. The state machine is passed to the *TestreadyModelMaker* component, which flattens the state machine and passes the flattened state machine to the *TestModelGenerator* component. This component generates the test model from the flattened state machine and passes it to the *TestScriptGenerator* component. The *TestScriptGenerator* component determines if a trigger (a method call or a signal reception) in a test script needs static test data. Test data can be generated statically if values can be determined prior to execution of the test script, or dynamically in the other case. The parameters whose values can be determined only at runtime are obtained at this point. Section 4.4 provides more specific examples of static and dynamic test data generation.

In the current version of TRUST, static data for a parameter is generated randomly from the possible set of values. Generating random test data may not be appropriate when the parameter of a trigger is used in the associated guard. In this case, a parameter value that satisfies the guard must be chosen, so

that the trigger can be fired. However, if the value of a parameter is selected randomly from a large set of possible parameter values, then the possibility of selecting the parameter value that satisfies the guard may be very low. Hence a more efficient test data generation technique based on search or optimization techniques [60] should be considered. This will be supported in future versions of TRUST.

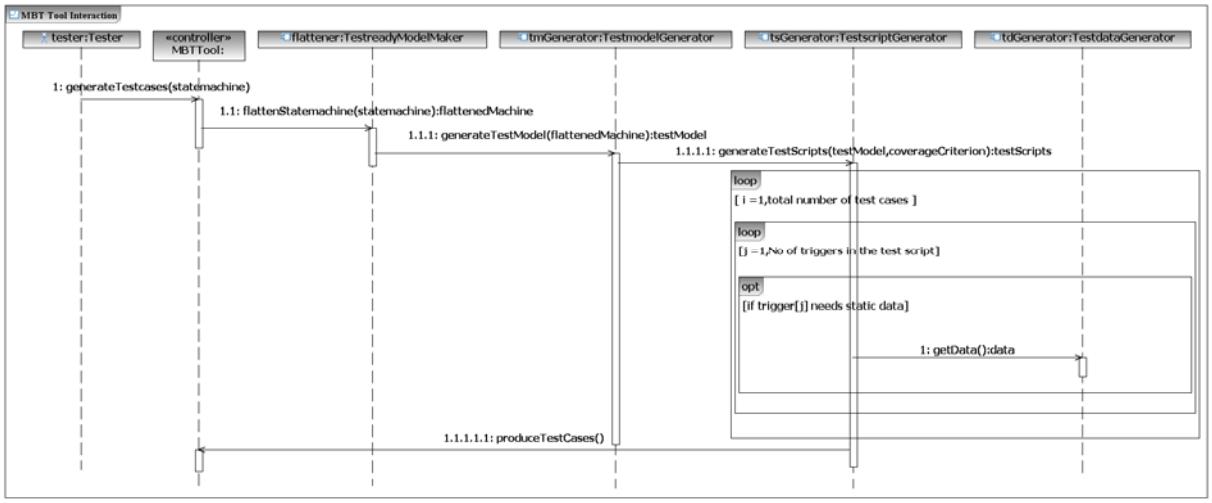

FIGURE 13. INTERACTIONS BETWEEN DIFFERENT COMPONENTS OF TRUST

4.4 Test case execution

Figure 14 shows how a test driver interacts with the SUT and other external tools when it executes a test case. Each test case consists of a series of triggers (methods or signals) with optional guards. The state of the system is checked before and after executing each trigger based on state invariants written as OCL constraints. When a test case is executed, the test driver initializes *EyeOCLWrapper* and *TestDataGenerator* (messages 1.1 and 1.2, respectively). Afterwards, the test driver obtains the state of the SUT by interacting with the SUT (message 1 in the *loop* fragment). The state of the SUT can be obtained in many ways. If the implementation of the SUT is in an object-oriented language such as in Java or C++, the state variables can be accessed using getter methods of classes if they are available. Alternatively, if the source code is available, but there are no getter methods, the source code needs to be instrumented before and after each method call to obtain the current state of the SUT. In the other case, if the source code is not available, then there are two possible options. The first option is that the system state might be obtained using some facilities of the implementation language. For example, if the Java byte code of the SUT is available, then Java's reflection facility [65] can be used to access the system state. The second option is to use a test script language that provides some mechanisms to access the state of the SUT. For instance, in one of our case studies, the test scripting language has the ability to query the state of the SUT using some predefined macros.

In our implementation, the test driver retrieves the minimum system state information by querying the values of only those state variables that are used in the OCL constraint which is to be evaluated. After obtaining the current state of the system, the test driver creates an object diagram using *OCLEvaluator* via the *EyeOCLWrapper* class (message 2 in the *loop* fragment). This class automatically generates an implementation of the class diagram and instantiates the object diagram corresponding to the implementation at runtime, based on the current state of the system (message 2.1 and 2.2 in the *loop* fragment). The test driver then evaluates the expected system state using *OCLEvaluator* via the *EyeOCLWrapper* class (message 3 and 3.1 in the *loop* fragment). Once the system state is evaluated against the expected state, the trigger, which may be guarded, should be executed. If dynamic test data

is required for the trigger, the test driver communicates with the *TestDataGenerator* class (message 1 in the *opt* fragment) to obtain required values. Whether the value for a parameter must be generated at runtime is indicated in the data model of the SUT. During the test case generation, TRUST checks if a parameter requires dynamic data generation or if static data is readily available.

In the case of guarded triggers, the associated guard must be evaluated before executing the trigger and after obtaining the dynamic test data. The guard may contain system variables and input parameters of the trigger. This means that in order to evaluate the guard, we need to obtain the system state and the values of the parameters (possibly dynamically generated) involved in the guards at runtime; The static and dynamic parameters that are used in the guard are replaced with their current values obtained from *TestDataGenerator* dynamically or statically. The guard is then evaluated in the same way as the state of the system was evaluated (messages 4, 5, 5.1, 5.2, 6, 6.1). Once the guard is evaluated, the appropriate method (i.e., the method that implements the trigger event) is invoked on (or signal is sent to) the SUT (message 7). After the execution of the method (or reception of the signal), the state of the system is evaluated (message 8, 9, 9.1, 9.2, 10, 10.1) in the same way as the previous state and guard evaluations. This process is repeated for all triggers in the test case. Finally, cleanup operations are performed on the SUT (message 1.3) once all the triggers have been executed on the system. These operations release the resources used by a test case such as memory and CPU.

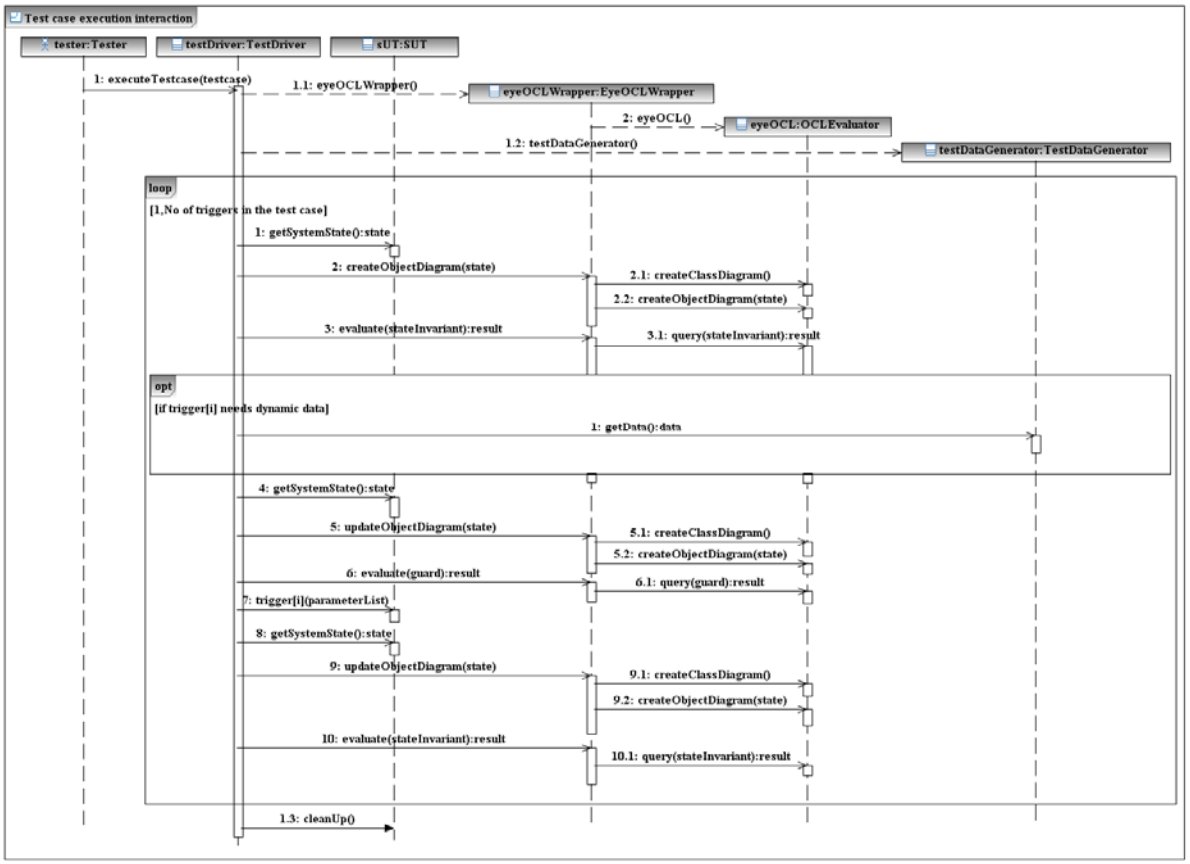

FIGURE 14. INTERACTIONS BETWEEN DIFFERENT COMPONENTS WHILE EXECUTING A TEST CASE

5. Applying TRUST on industrial cases

This section discusses the advantages and challenges of using model transformation technologies to support MBT automation by presenting two industrial usages of TRUST that cover two different contexts and application domains. Section 5.1 describes the two case studies. Section 5.2 addresses the technological issues in the two case studies. Finally, Section 5.3 provides the lessons learned from applying TRUST on realistic models. Information about the case studies is sanitized due to confidentiality restrictions.

5.1 Description of case studies

The companies where the case studies took place are international leaders in their respective fields. In both case studies, the models represent the state behavior of real world systems and the generated test cases are executable on the companies' testing platforms. Both state machines are complemented by constraints specifying state invariants which, as discussed earlier, will be useful to derive automated test oracles. The first case study (Case A) is the core subsystem of a multi-media conferencing system, whereas the second case study (Case B) is a safety monitoring component in a safety-critical control system. Both cases are suitable choices since these systems exhibit a complex state-based behavior that can be modeled as UML state machines.

The modeling process in Case A started with two presentations by the company representatives, followed by reading some specification documents. Then we had two workshops with experts from the company to better understand the system and domain. Afterwards, we built the system model in three increments. For each increment, we validated the models, with the help of company experts, both syntactically and semantically. Finally, during the development of TRUST, the model was augmented with many modeling details that were missed initially such as missing parameter type of an attribute of a class and missing connection point references on submachines. The resulting hierarchical state machine consists of four submachine states. The first submachine state hides two simple states, whereas the second contains two additional submachine states, each having two simple states. This gives in total eight simple states and 41 transitions in three levels (Table 3). The flattened state machine consists of 54 transitions and eight states. A total of 100 person-hours were approximately spent on understanding the system (60 person-hours) and modeling the SUT (40 person-hours).

In Case B, four initial meetings took place where the company representatives introduced the authors to the domain and the functionality of the SUT to be developed. In addition to the company representatives, the initial requirements specification and design documents served as sources for identifying the system behavior. Throughout the meetings, the authors made initial versions of the state-based model of the SUT. One of the company representatives took an active part in the subsequent modeling iterations. As part of the modeling process, the requirements were discussed with several of the company representatives whenever questions were raised and decisions had to be made. On many occasions, disagreements about the system specifications arose among stakeholders. In total, we spent approximately 320 person-hours on understanding (200 person-hours) and modeling the SUT (120 person-hours). It is important to note that, as opposed to Case A, where the system pre-existed the study, the modeling effort here could have been significantly smaller if the specifications had been stable to start with. At the top most level, the resulting hierarchical state machine consists of one simple state and one orthogonal state with two regions. Enclosed in Region1 are two simple states and two simple-composite states. Each of the simple-composite states contains two simple states. Region2 encloses three simple-composite states that again consist of, respectively, two, two, and three simple states. This adds up to 14 simple states and a maximum hierarchy level of two. The complete state machine contains 53 transitions (Table 3). The flattened state machine for Case B consists of 56 states and 391 transitions.

5.2 Use of the technology

In this section, we explore the feasibility of applying TRUST to two industrial case studies. One important aim is to demonstrate how the architecture of TRUST helps with configurability and extensibility. We will show how components in TRUST can be changed to fit different contexts.

TABLE 3 FEATURES SUMMARY OF THE HIERARCHICAL STATE MACHINES

State machine feature	Case A		Case B	
	Unflattened	Flattened	Unflattened	Flattened
Maximum level of hierarchy	2	-	2	-
Number of submachines	4	-	0	-
Number of simple-composite states	0	-	5	-
Number of simple states	8	8	14	56
Number of orthogonal states	0	-	1	-
Number of transitions	41	54	53	391

5.2.1 Using TRUST for test case generation in Case A

Communication from the test driver with this subsystem is done via APIs that are also used by other subsystems for requesting services. To enable test verdicts, the subsystem continuously reports its state through an XML document that can be accessed from the test scripts. Applying TRUST on Case A requires configuration values presented in Table 4.

TABLE 4 CONFIGURATION PARAMETERS OF TRUST FOR CASE A

Parameter	Value
Input model	UML 2.0 state machine
Test model	Transition tree
Coverage criterion	All round-trip paths
Test scripting language	A Python-based language
Test data generation technique	Random data generation
OCL Evaluator	EOS

A transition tree consisting of 4634 round-trip paths is generated from the flattened state machine. The test cases are generated in a Python-based test scripting language used in the company of Case A. The number of test cases in the test suite is the same as the number of paths in the transition tree (4634). However, out of 4634 test cases, only 463 test cases were found to be feasible. The infeasible test cases exist in the generated test cases because of guards on certain paths that cannot be satisfied. All feasible test cases are executable using the company's testing platform.

In order to access EOS for evaluating the state invariants and guards from the Python-based test scripting language, we used Jpype [63]. Jpype is an extension to Python, giving access to Java libraries through interfacing at the native level in both virtual machines (Java and Python).

Test case generation took place on a system with Intel Duo CPU 2.40 GHZ processor, with 4GB of RAM. The system was running Microsoft Windows Vista operating system and IBM Rational Software Architect Standard Edition 7.5.1. TRUST took 19 seconds to flatten the state machine, six seconds to generate the transition tree, and one minute and 30 seconds to generate test cases. Such execution times were deemed perfectly acceptable in the Case A context.

5.2.2 Using TRUST for test case generation in Case B

We applied TRUST with the configuration values presented in Table 5 and the flattened state machine described in Table 3 as input.

TABLE 5 CONFIGURATION PARAMETERS OF TRUST FOR CASE B

Parameter	Value
Input model	UML2.0 state machine
Test model	Test tree for all transitions
Coverage criterion	All transitions
Test scripting language	C++
Test data generation technique	Random data generation
OCL Evaluator	EOS

After applying the flattening transformation and removing unreachable state combinations due to conflicting state invariants, the flattened state machine consists of 56 states and 391 transitions, mostly guarded. In this case, TRUST was configured for another coverage criterion, the all transitions criterion, applied on a test tree which conforms to the same test tree metamodel presented in Figure 9. The tree is built differently from Case A in such a way that traversing all paths in the test tree achieves all transitions coverage. The instantiation of the metamodel for this case was implemented using Kermeta.

We chose all transitions coverage for Case B due to the much larger size of the model in order to obtain a manageable set of test cases. TRUST generated 335 test paths for Case B, where C++ was the test scripting language. Since the first version of TRUST was implemented for Case A, TRUST needed to be extended to support SBT for Case B. Therefore, a C++ test script generator was added to TRUST. Extending TRUST for C++ involved adding a new set of transformation rules according to C++ syntax to the *TestscriptGenerator* component of TRUST. This required changes in the MOFScript rules. There was some reuse of the MOFScript rules used in the tool instantiation for Case A. Only the logic for traversing the tree could be reused, however, because the mapping rules for C++ were quite different from a Python-based script language for Case A due to different language constructs.

The number of generated test cases is the same as the number of paths in the test tree: 335. However, out of 335, only 205 test cases were found to be feasible. All the feasible test cases are ready to be executed on the testing platform and do not require modifications. Once again, the infeasible test cases exist in the generated test cases because of guards on certain paths that cannot be satisfied. Test data was generated in the same manner as in Case A by randomly selecting the value for the parameters based on the data type. However, in Case B, TRUST was extended to interact with a new

TestdataGenerator component implemented in C++. We chose to do this in C++ because test scripts in Case B were in C++ and interacting with a *TestdataGenerator* component implemented in C++ was more efficient. Java Native Interface (JNI) [64] was used to access EOS from the C++ test scripts. For this case study, the platform specifications were the same as for Case A except for the 3GB of RAM. TRUST took 281 minutes to flatten the state machine, 64 minutes to generate the test tree, and 24 seconds to generate test cases. We observe that the time spent on flattening the state machine is considerably more in Case B than for Case A. The main reason for this is that the current implementation of the flattening algorithm for removing concurrency requires numerous traversals of the transition set. The traversals are necessary in order to identify possible transitions between the Cartesian product states. For each transition in each region of the unflattened state machine, it must be checked whether or not the transition triggers are also defined for transitions in the other parts of the Cartesian product state. This implies that an event could trigger transitions in several regions [58]. Improving the efficiency of the algorithm is part of our future work.

5.3 Lessons learned

Developing TRUST and then applying it to real world case studies taught us some important lessons about both modelling and model transformations. In this section, we will discuss the lessons learned for these aspects.

5.3.1 Modelling of the SUT

For our case studies, precise behavioural modelling of complex industrial systems using standard UML 2.0 state machines was a prerequisite for using TRUST. The flattening component requires that it is provided with a correctly specified state machine and currently does not provide any feedback in case of errors in the model. Modeling correctly, however, is not a trivial task and requires that the UML specification be carefully studied. Even though constructs like concurrency and hierarchy are supposed to ease the understandability of large state machines, such constructs may actually confuse the developer. In particular, we experienced that concurrency, if not carefully applied, could introduce modeling errors in practice. For example, concurrent regions sometimes make it difficult to see the set of transitions between state combinations. A typical fault is that a guard is missing on a transition, which allows for transitions to state combinations that are illegal targets from particular source states. However, we found that it helped to inspect the flattened state machine to detect such mistakes. In Case B, for instance, we detected that a missing guard on a transition from an initialization state to a system running state in Region 1 would allow transitions to be incorrectly fired in Region 2.

5.3.2 Model-to-model transformation technologies

The model-to-model transformations in TRUST used two different transformation languages: Kermeta and ATL. Kermeta appeared to be highly appropriate for flattening UML state machines. In addition to being an object-oriented language, it allows you to add behavior to the metamodel through aspect weaving. However, we experienced that navigating in the metamodel was rather time consuming. Alphabetically organized in a super-sub class structure, the UML 2.0 metamodel is a complex model that is difficult to navigate. Having tool support integrated in the Kermeta plug-in that could remove abstract classes and instead present the concrete classes relevant for a particular purpose would have been very useful.

Since the metamodel for test trees is relatively simple, the transformation from the flattened state machine to the test model was expected to be straightforward and easy to implement by depth first traversal of the state machine using a declarative language (ATL). However, we found that the declarative programming style was not intuitive to handle, perhaps because most developers are used

to imperative programming languages. Even though the final ATL code for test model generation is very short, debugging it was quite difficult especially when the input model was big. For our second case study, the input state machine was quite large and caused Eclipse to run out of memory while generating a transition tree for all round-trip paths coverage criteria. This was due to the many recursive rule calls required to generate the transition tree from the flattened state machine. Implementing transition tree generation using recursion was the only possible option when writing rules in the ATL language in a declarative fashion. Technology-wise, we also faced many problems while debugging the ATL rules, especially when the input models are large causing the debugging interface to hang.

5.3.3 Model-to-text transformation technology

Developing the final set of transformations in MOFScript was the easiest part of developing TRUST, because the rules are defined in an imperative form. MOFScript is quite similar to programming languages like Java, and provides powerful features that are easy to use for querying models, outputting text, and accessing external Java libraries. We did not face any special challenges while using MOFScript for generating test scripts.

6. Discussion

In order to achieve our requirements for an MBT tool (Section 4.1), we identified five important aspects that must be extensible and configurable. These five aspects are related to: test ready model generation, test model generation, test script generation, test data generation, and constraint evaluation. All important aspects of an MBT tool are addressed by specific components in the TRUST architecture in Figure 12. We defined clear interfaces between the components and the external tools so that communication among the components and with the external tools can take place on input models that are instances of standard metamodels. We addressed the first three aspects of TRUST using model transformation languages because using such technologies was an effective way to specify the transformations at a more abstract level, through the mapping of metamodels.

The first aspect is related to making an input model of the SUT test ready, so that the test model can be generated from the test ready model. In the case studies provided in this technical report we configured TRUST for UML 2.0 state machines, which offer features for concurrency and hierarchy. As discussed previously, such features ease modeling but complicate test automation. Consequently, many SBT tools [5, 13] assume that state machines do not have concurrency and hierarchy. Our first objective was to flatten state machines so as to simplify the implementation of different test models and coverage criteria on them. Coverage criteria can be implemented on unflattened state machines, but require a complex algorithm and thus make it difficult to implement different test models. For this reason we chose to separate flattening and test model generation so that it would be easier to implement new test models based on flattened state machines. We implemented state machine flattening by model transformations using the Kermeta language as described in Section 4. The current implementation for flattening is based on the UML 2.0 metamodel. It can, however, be extended to UML profiles and any future changes in the UML metamodel by changing the Kermeta rules for flattening state machines, without affecting the rest of the tool. In the current TRUST implementation, state hierarchies can be efficiently processed whereas concurrent states lead to scalability problems and should be further investigated, as discussed in Section 5.2.2.

The second aspect was related to the test model generator component, which transforms the test ready model (flattened state machines in our case studies) into the test model (test trees in the current version

of TRUST). This component is extensible to various test models. A test model can be defined based on the type of faults targeted or the coverage to be achieved. In order to implement other test models based on state machines, metamodels for the test models must be developed along with new mapping rules from flattened state machines to the test models in the ATL, Kermeta or any other M2M transformation language.

The third aspect, related to test script generation, is another important aspect of TRUST, which requires a test model's metamodel, a defined coverage criterion, and knowledge about the test script language's grammar. TRUST can be extended for producing test scripts in various test script languages by implementing new transformation rules in MOFScript. For example, in Case B, we extended TRUST to output test scripts in C++ in contrast to Case A, where test scripts were generated in a Python-based script language. We achieved this by changing the MOFScript transformation rules to generate the C++ syntax. We experienced that the rules addressing the logic for traversing the tree could be reused. The rest of the mapping rules that were specific to the test script language were not helpful – at least not when the developer was unfamiliar with the test script language used in the existing tool instantiation.

Even though the MOFScript rules had to be modified from Case A, the rest of the implementation of TRUST remains unchanged. It is important to note that even for a simple but real system and standard coverage criterion, the number of test cases can be very large (Case A: 4634). Furthermore, small changes in the specification of a SUT can result in dramatic changes in its test suite. For example 653 test cases are deleted by removing just two transitions from the unflattened state machine of Case A. This shows how much effect a single transition on the model can have on the test cases. Furthermore, considering the fact that specifications of such systems may change quite frequently, it is not feasible to generate and maintain such large numbers of test cases manually. Thus, to support systematic testing in a scalable way, an MBT tool that can generate test cases automatically every time the specification changes, is required.

The test data generator in the current implementation of TRUST provides only random test data generation, because in both of our case studies, guards on transitions are not constrained by the input parameters of associated triggers on the transitions. It can, however, be changed to a more sophisticated test data generation technique, such as techniques based on search-based data generation algorithms [60]; any test data generator can be easily integrated with the rest of the tool as long as the defined interfaces are used. Furthermore, some of the guards in our case studies are based on state variables, which may not be satisfied while executing some paths and thus leading to many infeasible test cases. Effectively removing such infeasible test cases is future work.

Finally, we used EOS [33] as the OCL evaluator with the current implementation of TRUST. Again, the OCL evaluator can be changed to any OCL evaluator depending on the requirements of a tool. For instance, if the tool needs to evaluate constraints written in both OCL and the Value Specification Language (VSL) of the MARTE profile [51], another suitable evaluator can be integrated with the tool without making changes to other components.

The modeling in the two case studies took approximately 100 and 320 hours for Case A and Case B, respectively. If excluding the time required to understand the system, which may not be fully necessary if the modeling is done by developers themselves, this effort comes down to 40 and 120 hours. Given the large numbers of feasible test cases required (463, 205) for systematic testing on these case studies, MBT can be considered beneficial if the manual identification and writing of test cases costs, on average, less than 6 minutes and 36 minutes per test case, respectively for the two case studies. The latter number, though being much larger than for Case A, must be interpreted with care as

the modeling effort for Case B was inflated due to frequently changing specifications leading to iterative modeling, as discussed in Section 5.1. This effort cannot therefore be solely attributed to modeling but also includes the specification effort that would have been necessary regardless of whether modeling was applied. Based on the above effort numbers per test case, under realistic assumptions, MBT is clearly likely to be beneficial in many contexts as devising test cases and writing test scripts is likely to take much more time than 6 and 36 minutes (this being probably an overestimate) in practice. Furthermore, this analysis does not account for the fact that MBT yields test cases that are more systematic, less error-prone, and that changes to the SUT behavior can simply be addressed by re-generating the test cases, thus facilitating the evolution of test suites. Ideally, a controlled experiment involving the manual writing of test cases would be warranted to demonstrate tangible benefits, but writing so many test cases manually is by all practical means out of reach, especially in the context of an experiment.

The current implementation of TRUST still has limitations that will be improved in the future:

- The flattening algorithm does not handle the following pseudo states: shallow and deep history, join, fork, junction, and terminate. Another issue is the missing evaluation of OCL expressions during the generation of the state combinations in states with more than one region. This means that impossible state combinations may be included in the flattened set of states due to conflicting state invariants. In addition, the algorithm may introduce impossible transitions due to guards that will never become true when transitions are triggered from certain states. These situations occur when state invariants hinder state variables from being initialized with the specific values required to fulfill the guard condition.
- Testing of TRUST has not been done systematically. In order to check if transformations are correct, we created models by hand and checked the transformed models manually. However, we could attempt to use tools such as [66] to generate inputs for transformations, but one important issue is that there is no tool support to generate and check expected outputs.
- We do not have any support for debugging. However, we output messages whenever a state invariant or a guard fails during test execution.
- We do not have support for model validation, and we assume that UML input models are correctly designed and are not missing any information required by TRUST.

7. Conclusions and future work

Tool support for model-based testing (MBT) has dramatically improved in recent years, but most of the tools specifically target an application context and cannot easily be adapted to others. In this technical report we report on the design and application of TRUST, a TRansformation-based tool for Uml-baSed Testing, which can be extended and configured for various application contexts. TRUST is based on model-transformation technologies and features an architecture with clear separation of concerns and interfaces, thus making it easily extensible and configurable for different context factors such as input models, test models, coverage criteria, test data generation strategies, and test scripting languages. We have illustrated this by adapting TRUST to the needs of two industrial case studies from two different companies: a multi-media conferencing system and a safety-critical control system. We also report on the costs, challenges, and likely benefits of MBT using TRUST in these two industrial contexts.

Our case studies have led to a number of results and lessons learned both related to our transformation-based approach to automating MBT and the benefits and challenges of MBT on

industrial case studies. In terms of benefits, the comparison of the cost of modelling with the number of test cases generated, in both our case studies, has shown that using TRUST should yield significant cost savings when applying standard state machine coverage criteria. In other words, the cost of writing manually the same test cases is likely to be larger than the cost of modelling the system under test (SUT) and generating the test cases. Using TRUST offers many other potential advantages which are, however, difficult to quantify. For example it should make the generation of test scripts less error-prone, enable the easy re-generation of test cases when the SUT specifications change, and ensure that testing is systematic and not redundant, an objective hard to achieve for a human tester. In terms of scalability, the only issue seems to be with the flattening of concurrent states, which may take a few hours on complex, highly concurrent state machines. Otherwise, the processing steps involved in TRUST have shown to be in the worst case a matter of minutes.

Modeling SUTs correctly, which is required by TRUST or any other MBT tool, is however not a trivial task on real systems. Future versions of TRUST should provide feedback on the likely correctness, consistency, and completeness of input models. This would save substantial modeling time and make MBT even more beneficial. For example, we experienced that inspections of flattened versions of complex state machines led to the detection of modeling errors due to error-prone constructs like concurrency.

Different Model-to-model transformation technologies, such as ATL and Kermeta, have shown different advantages. ATL tends to yield small transformation rules. However, to the use of declarative programming may not always be appropriate and, in some cases, makes it difficult to define and validate rules. Kermeta, on the other hand, follows the imperative paradigm and, as a result, complex mappings between metamodels could more easily be defined. ATL has also been shown not to scale up well to large input models. On the other hand, MOFScript, a model-to-text transformation technology, was easy and convenient to use to generate test scripts. Like Kermeta, this follows the imperative programming paradigm, which provides powerful features that are easy to use for querying models, outputting text, and accessing external Java libraries.

In the future, we are planning to implement more transformations for various state-based test models and coverage criteria. After implementing these transformations, we will conduct empirical studies to evaluate the cost effectiveness of different test strategies for our case studies. We are also devising algorithms to make SBT more scalable, for example by minimizing infeasible test paths and implementing more sophisticated test data generation techniques based on search-based algorithms. Another important area of future work would be to provide model validation support in TRUST. Finally, we are planning to extend TRUST for testing non-functional properties of a system modeled with UML and different profiles for modeling non-functional properties, such as MARTE profile for modeling real time and embedded systems.

Acknowledgement

The authors wish to thank Miran Damjanović for helping us in writing transformation rules for Case A and Simula School of Research and Innovation (SSRI) for funding this work.

8. References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*: Morgan-Kaufmann, 2006.

[2] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. 4, pp. 178-187, 1978.

[3] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001.

[4] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," *Encyclopedia of Software Engineering (edited by J. J. Marciniak), Wiley*, 2001.

[5] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem, "A State-Based Approach to Integration Testing Based on UML Models," *Information and Software Technology*, vol. 49, pp. 1087-1106, 2007.

[6] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, Georgia, 2007.

[7] D. Drusinsky, *Modeling and Verification using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, 1st ed.: Newnes, 2006.

[8] "D-MINT, Deployment of Model-Based Technologies to Industrial Testing," http://www.d-mint.org/ (September 2009)

[9] J. Feldstein, "Model-based Testing using IBM Rational Functional Tester," developerWorks, IBM, 2005.

[10] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying Model-Based Testing to Healthcare Products: Preliminary Experiences," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008.

[11] Y. Gurevich, W. Schulte, N. Tillmann, and M. Veanes, "Model-based Testing with SpecExplorer," Microsoft research2009.

[12] "QTRONIC," http://www.conformiq.com/qtronic.php (September 2009)

[13] "MOTES," http://www.elvior.ee/motes (September 2009)

[14] A. Hartman and K. Nagin, "The AGEDIS Tools for Model Based Testing," in *International Symposium on Software Testing and Analysis (ISSTA '04)*, 2004.

[15] "Automatic Test Generation," http://www.telelogic.com/products/rhapsody/test/automated-test-generation.cfm (September 2009)

[16] D. Seifert, "The TEAGER Tool Suite: Test Execution and Generation Framework for Reactive Systems," http://user.cs.tu-berlin.de/~seifert/teager.html (September 2009)

[17] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[18] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise* Addison Wesley, 2003.

[19] T. Pender, *UML Bible*: Wiley, 2003.

[20] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation," *Accepted for publication in IEEE Transactions on Software Engineering, Special issue on Search-Based Software Engineering (SBSE)*, 2009.

[21] L. Lavagno, G. Martin, and B. V. Selic, *UML for Real: Design of Embedded Real-Time Systems*: Springer, 2003.

[22] T. Weigert and R. Reed, "Specifying Telecommunications Systems with UML," in *UML for Real: Design of Embedded Real-time Systems*: Kluwer Academic Publishers, 2003, pp. 301-322.

[23] S. Sauer and G. Engels, "UML-based Behavior Specification of Interactive Multimedia Applications," in *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, 2001.

[24] "Papyrus," www.papyrusuml.org (September 2009)

[25] "IBM Rational Software Architect," http://www-01.ibm.com/software/awdtools/architect/swarchitect/ (September 2009)

[26] L. C. Briand, Y. Labiche, and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria on Statecharts," Carleton University Technical Report SCE-02-09, 2002.

[27] J. Zhang, C. Xu, and X. Wang, "Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques," in *Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, 2004, pp. 242-250.

[28] R. Lefticaru and F. Ipate, "Automatic State-Based Test Generation Using Genetic Algorithms," in *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2008.

[29] L. C. Briand, M. D. Penta, and Y. Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Transactions on Software Engineering,* vol. 30, pp. 770-793, 2004.

[30] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Cârcu, "OCLE," http://lci.cs.ubbcluj.ro/ocle/ (September 2009)

[31] C. Hein, T. Ritter, and M. Wagner, "Open Source Library for OCL," 2009.

[32] "IBM OCL Parser," http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html (September 2009)

[33] M. Egea, "EyeOCL Software," http://maude.sip.ucm.es/eos/ (September 2009)

[34] "Object Management Group," www.omg.org (September 2009)

[35] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed.: Addison-Wesley Professional, 2008.

[36] S. A. White, D. Miers, and L. Fischer, *BPMN Modeling and Reference Guide*: Future Strategies Inc., Lighthouse Pt, FL, 2008.

[37] "Business Process Definition Metamodel (BPDM)," Object Management Group, OMG Adopted Specification dtc/07-07-01, 2007.

[38] E. Christense, F. Curbera, G. Meredit, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1, 2001," http://www.w3.org/TR/wsdl

[39] C. Hahn, C. Madrigal-Mora, and K. Fischer, "Interoperability through a Platform-Independent Model for Agents," in *Enterprise Interoperability II*, 2007, pp. 195-206.

[40] W. Zhang, H. Mei, H. Zhao, and J. Yang, "Transformation from CIM to PIM: A Feature-Oriented Component-Based Approach," in *Model Driven Engineering Languages and Systems*: Springer Berlin / Heidelberg, 2005, pp. 248-263.

[41] S. Kherraf, É. Lefebvre, and W. Suryn, "Transformation from CIM to PIM Using Patterns and Archetypes," in *Proceedings of the 19th Australian Conference on Software Engineering*, 2008.

[42] "Kermeta - Breathe Life into Your Metamodels," http://www.kermeta.org/ (September 2009)

[43] "ATLAS Transformation Language (ATL)," http://www.eclipse.org/m2m/atl/ (September 2009)

[44] "OMG's MetaObject Facility," http://www.omg.org/mof/ (September 2009)

[45] P. A. Muller, F. Fleurey, and J. M. Jézéquel, "Weaving Executability into Object-Oriented Meta-Languages," in *Proceedings of MODELS/UML'2005*, 2005, pp. 264-278.

[46] "Model to Text (M2T)," http://www.eclipse.org/modeling/m2t/ (September 2009)

[47] "MOFScript Home page," http://www.eclipse.org/gmt/mofscript/ (September 2009)

[48] P. Huber, "The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches," in *Business Informatics Group*: Institut für Softwaretechnik und Interaktive Systeme, 2008.

[49] "Model-based Testing Tools," http://en.wikipedia.org/wiki/Model-based_testing_tools (September 2009)

[50] S. Weißleder, "Partition Test Generato (ParTeG)," http://parteg.sourceforge.net/ (September 2009)

[51] "MARTE: Modeling and Analysis of Real-time and Embedded systems," http://www.omgmarte.org/Specification.htm (September 2009)

[52] R. Cavarra, C. Crichton, J. Davies, A. Hartman, and L. Mounier, "Using UML for automatic test generation " in *In International Symposium on Software Testing and Analysis (ISSTA '02)*, 2002.

[53] "Poseidon for UML," www.gentleware.com (September 2009)

[54] "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," http://www.omg.org/technology/documents/formal/QoS_FT.htm (September 2009)

[55] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using UML statechart diagrams," in *SAICSIT03*, 2003.

[56] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive Random Testing: The ART of Test Case Diversity," *Journal of Systems and Software,* vol. In Press, Corrected Proof.

[57] T. Jussila, J. Dubrovin, T. Junttila, T. L. Latvala, and I. Porres, "Model Checking Dynamic and Hierarchical UML State Machines," in *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa06)*, 2006.

[58] N. E. Holt, E. Arisholm, and L. C. Briand, "An Eclipse Plug-in for the Flattening of Concurrency and Hierarchy in UML State Machines," Simula Research Laboratory, Technical Report 2009-06, 2009.

[59] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering,* vol. 17, pp. 900-910, 1991.

[60] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification, and Reliability,* vol. 14, pp. 105-156, 2004.

[61] H. Mössenböck, M. Löberbauer, and A. Wöß, "The Compiler Generator Coco/R," http://ssw.jku.at/coco/ (Spetember 2009)

[62] T. Parr, "ANTLR v3," http://www.antlr.org/ (September 2009)

[63] "JPype," http://jpype.sourceforge.net/ (September 2009)

[64] S. Liang, *Java Native Interface: Programmer's Guide and Specification*: Addison-Wesley Publishing Company, 1999.

[65] G. McCluskey, "Using Java Reflection," http://java.sun.com/developer/technicalArticles/ALT/Reflection/ (September 2009)

[66] S. Sen, B. Baudry, and J. M. Mottu, "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008.