Electronic Communications of the EASST
Volume 47 (2012)


# Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

## Coverage Criteria for Testing DMM Specifications

Svetlana Arifulina, Christian Soltenborn, and Gregor Engels

14 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Coverage Criteria for Testing DMM Specifications

**Svetlana Arifulina[1], Christian Soltenborn[2], and Gregor Engels[3]**

[1] svetarif@mail.uni-paderborn.de
[2] christian@uni-paderborn.de
[3] engels@uni-paderborn.de

Research Group "Database and Information Systems"
Department of Computer Science, University of Paderborn, Germany*

**Abstract:**

Behavioral modeling languages are most useful if their behavior is specified formally such that it can e.g. be analyzed and executed automatically. Obviously, the quality of such behavior specifications is crucial. The rule-based semantics specification technique Dynamic Meta Modeling (DMM) honors this by using the approach of Test-driven Semantics Specification (TDSS), which makes sure that the specification at hand at least describes the correct behavior for a suite of test models. However, in its current state TDSS does not provide any means to measure the quality of such a test suite.

In this paper, we describe how we have applied the idea of test coverage to TDSS. Similar to common approaches of defining test coverage criteria, we describe a data structure called invocation graph containing possible orders of applications of DMM rules. Then we define different coverage criteria based on that data structure, taking the rule applications caused by the test suite's models into account. Our implementation of the described approach gives the language engineer using DMM a means to reason about the quality of the language's test suite, and also provides hints on how to improve that quality by adding dedicated test models to the test suite.

**Keywords:** Dynamic Meta Modeling, DMM, Test-Driven Semantics Specification, coverage, graph transformation, test quality.

## 1 Introduction

The development of large modern software systems requires participation of experts from different problem domains and development on high quality level with least time efforts. This process cannot be separated from the usage of models. Models provide different beneficial properties, for example, a successful use for communication purposes. In this case, visual modeling languages serve as a convenient means to enable experts to model on their own and exchange information among diverse experts, so that it is possible for them to understand each other. In this context,

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (SFB 901).

one should be able to judge about the quality of developed models, since error-free artifacts contribute to the quality of software as well. In order to check the correctness of models designed in a visual modeling language, the language should be analyzable, i.e. have a formal definition.

One way to define the abstract syntax of a visual modeling language formally is in the form of a metamodel. In this case, its dynamic semantics can be specified formally using *Dynamic Meta Modeling (DMM)*. DMM enhances the syntactic metamodel with concepts referring to behavior of language constructs at runtime that results in a so-called runtime metamodel. The instances of the runtime metamodel express states of execution. The behavioral semantics is represented as a set of graph transformation rules (GTRs) operating with these instances and defining how they change over time. Given a model and a set of GTRs, the model's behavior can be computed and automatically checked for possessing some properties of correctness.

However, a DMM specification containing erroneously specified behavior may lead to unreliable results. So, in order to improve its quality, the *Test-Driven-Semantics Specification (TDSS)* approach is applied. A language engineer starts with creating a set of test cases. The test input for such a test case consists of a model written in a considered modeling language. The test result is a formalized expected behavior of the model. Then, for each test case, the actual behavior of the model is computed based on the DMM specification and checked for conformance with the expected one. If they do not comply, the language engineer fixes the given DMM specification. Thus, continuous testing and, as a result, error detection during the development process of DMM specifications yields their higher quality, since the behavior for the tested models is proven to be correct.

But the deficiency of TDSS is that the quality of the used test cases is not taken into consideration. Since models from the used test cases exercise a certain part of a DMM specification applicable on them, they determine parts of this specification which have been tested for correspondence with the expected behavior. Thus, good tests influence the quality of DMM specifications, since they exercise them to a larger extent. This paper handles the described problem by enriching the TDSS approach with a technique for assessing the quality of tests.

One means to assess the quality of tests in white-box testing of software systems is *test coverage*. It expresses a degree to which structural elements of a system are exercised by a test suite. Full coverage means that all structural elements were tested at least once. Reaching a coverage value predefined by the language engineer guarantees system's minimum level of quality.

Since the implementation of DMM specifications is accessible, the notion of test coverage can be applied in TDSS for judging about the quality of tests as well. In the proposed approach, structural elements are operational rules of a DMM specification. During the execution of a single test case, only a subset of all rules are used for the input model. So, coverage is a proportion between the number of used rules to the whole number of rules. The contribution of the paper is the definition of different coverage criteria for assessing the quality of tests used in TDSS. The higher is the coverage value, the better is the test suite used. The test suite can be improved by adding test cases executing the untested parts of the system.

The quality of a DMM specification is a correspondence between this specification and an informal semantics of the modeling language which it is supposed to model. Testing a DMM specification on an improved test suite helps to find more errors in that specification. The language engineer will correct these errors, which should bring the DMM specification closer to the desirable behavior. Thus, improving the quality of the test suite by means of raising the

coverage (indirectly) influences the quality of the DMM specification and should improve it in comparison to a situation without coverage analysis, where judging about the quality of the tests is completely a task of the language engineer.

The remainder of the paper is structured as follows: A theoretical basis needed to comprehend the content of the paper is explained in Section 2. Based on the existing ideas, the application of test coverage in the field of DMM, definition of new coverage criteria, and their classification are discussed in Section 3. In Section 4, an overview of a related work is presented outlining similarities and differences to the proposed approach. Finally, Section 5 summarizes our main results and proposes some perspectives for future research.

Please note that the paper arose from a master's thesis [Ari11]. Due to space limitations, the paper omits some details, which can be found therein; we mention this fact at the according places.

## 2 Foundations

This section will briefly present the foundation needed to understand the rest of this paper. We start by introducing *DMM* and techniques used by it, followed by an introduction to *TDSS*. Finally, we explain the basics of *test coverage* in software engineering.

**Overview of Dynamic Meta Modeling (DMM)** In order to enable automatic analysis of models' behavior and to facilitate a precise understanding of their semantics, a formal representation of this semantics is needed. *Dynamic Meta Modeling (DMM)* [EHHS00, Hau05] is an approach for formal specification of behavioral semantics of visual modeling languages whose abstract syntax is represented by a metamodel.

An overview of the DMM architecture is presented in Figure 1. In a nutshell, DMM works as follows: First, a *runtime metamodel (RMM)* is derived from the syntax metamodel enhancing it with concepts needed to express states of execution. Second, the actual dynamic semantics is defined by means of *graph transformation rules (GTRs)* which describe how instances of the RMM change over time. Given a model and a DMM specification, a *labeled transition system (LTS)* can be computed, where states are instances of the RMM, and transitions are applications of GTRs. That LTS can then be analyzed e.g. with model checking techniques.

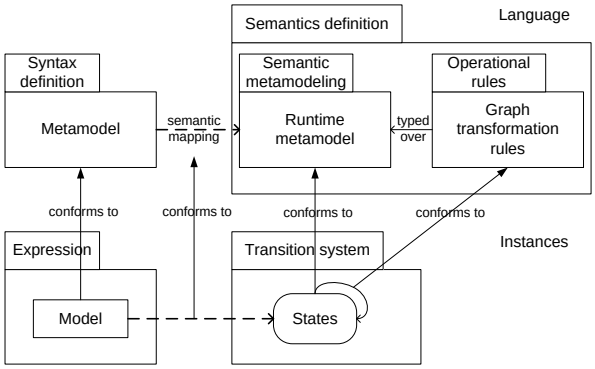


Figure 1: DMM architecture [EHHS00]

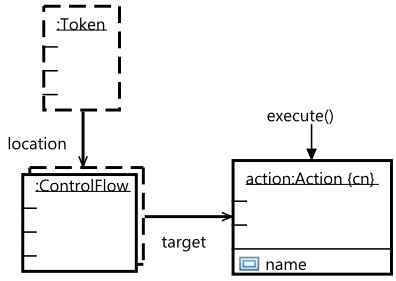


Figure 2: DMM rule *action.start.1*

**Rule Application and Invocation** Since the understanding of graph transformations is crucial for this paper, let us investigate this formalism closer. A GTR $R$ consists of a *left-hand side graph* $R_L$, a *right-hand side graph* $R_R$, and a set of *negative application conditions NAC*.[1] $R$ *matches* a state graph $s$, if $R_L$ can be found in $s$ and none of the conditions contained in *NAC* is violated; the matching part of the state graph is called *matching context*. If $R$ matches $s$, it can be *applied* to $s$, i.e. the occurrence of $R_L$ in $s$ is replaced with $R_R$ giving rise to a new state graph $s'$.

However, in our experience, working with rule-based formalisms is quite unusual for most language engineers. Thus, DMM provides the possibility to explicitly *invoke* rules out of other rules. For this, DMM supports two kinds of rules: *Bigstep rules* which are comparable to normal GTRs applying as soon as they match and *smallstep rules* which, in contrast, can only match if they are explicitly invoked by another bigstep or smallstep rule. This gives the language engineer some imperative means to realize the desired semantics making her job significantly easier.

An example DMM rule is depicted in Figure 2 (note that the concrete syntax of DMM merges $R_L$ and $R_R$ into one graph, where nodes to be deleted or created are annotated accordingly). Its semantics is as follows: The rule matches if all incoming ControlFlows of an Action carry a Token – if this is the case, rule *execute* is invoked which will perform the actual execution of the action. The rule's concrete syntax expresses this behavior by means of the multi-object of type ControlFlow and the object of type Token having a dotted outline – the whole construct can basically be read as "'Each incoming ControlFlow of the Action must carry its own Token"'.

Let us also investigate rule invocations in more detail. Each DMM rule has a name, a so-called *context node* and a (possibly empty) *list of invocations* of smallstep rules. An invocation consists of a *target node* and a rule name. Now, if rule $R$ invokes rule $R'$, the invocation's target node is mapped to the invoked rule's context node, i.e. the rule is applied in that node's context.

Note that names of DMM rule do not have to be distinct. In fact, it is quite common to have several smallstep rules with the same name. The idea is to model a kind of if-then-else structure (which is not explicitly supported by DMM). For example, if we want to model that some behavior changes depending on the existence or non-existence of some node, we can provide two smallstep rules with the same name $n$ covering these two cases, and we can invoke a rule $n$ from within another rule. When computing the LTS, it will be decided based on the available context which rule will actually apply. Note that for this very reason, each DMM rule also has a *unique name*. These are distinct within a DMM specification.

Now, a rule invocation can *fail*, i.e. at invocation time, none of the possibly invoked rules matches the current state graph (the reason either being the absence of an object contained in $L_R$ or a violated *NAC*). This results in a failed matching. If a DMM specification gives rise to such situations, this is considered to be a *specification failure*.

Finally, as mentioned above, a LTS can be computed from a DMM specification and an according model. This is done as follows: The model is translated into an instance of the RMM, which serves as the start state for the LTS. Now, all matching DMM rules are applied to the start state, giving rise to new states; the resulting transitions are labeled with the applied rule's unique name. This is repeated until no new states are found. The resulting LTS can then be analyzed, e.g. for functional requirements [ESW07].

Let us capture the above in a definition:

[1] See e.g. [HHT95] for details.

**Definition 1** A LTS is a graph: $LTS = (V, E)$, where $V$ is a set of instances of the RMM, $E \subseteq V \times \Sigma \times V$ is a set of transitions. $(v_1, l, v_2) \in E :\Leftrightarrow$ rule with unique name $l$ is applied to $v_1$ and the application results in $v_2$.

**Test-driven Semantics Specification** A DMM specification is only useful if the specified semantics correctly captures the desired behavior (e.g., when formalizing the UML semantics which is provided as natural language), and if it does not give rise to situations where failed invocations occur. However, the experience from software engineering has shown that performing proofs against complex systems is not feasible; the solution is to test the according system. Since DMM specifications can become quite complex too, we have developed the approach of *Test-driven Semantics Specification (TDSS)* [SE09].

A comparison of TDSS with the testing of software systems is presented in Figure 3. On the left-hand side, a typical test scenario is depicted: A test consists of an input for the system under test (SUT) depicted as an oval and a desired output. The input is fed into the SUT, and the computed result is compared with the output.

In contrast, the SUT in TDSS is a DMM specification. A test case consists of a model and its expected behavior. The test is executed by the DMM interpreter using the specification and the test model to calculate a LTS representing the actual behavior. Then, it is verified whether this LTS contains the expected (and only the expected) behavior. For this, model checking techniques are used (for the details, please refer to [SE09]).

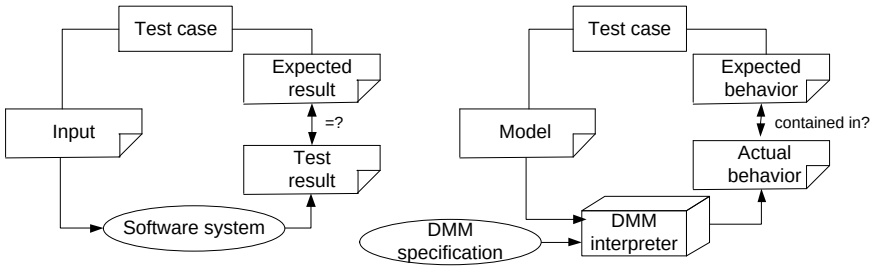


Figure 3: Testing of software systems and DMM specifications [SE09]

To sum up, TDSS is a pragmatic means to ensure a high-quality DMM specification. However, we did not consider the quality of the tests themselves. This is to be changed as a result of this paper, where we will define *coverage criteria* which indicate how well a certain DMM specification is tested. Therefore, let us now briefly investigate test coverage in software engineering.

**Test Coverage** One approach to evaluate the quality of white-box testing in software engineering is *test coverage*. According to the standard glossary of terms used in software testing [IST10], coverage is the degree, expressed as a percentage, to which a specified *coverage item* has been exercised by a test suite. One common way to define such coverage items is to compute a *control flow graph*, where e.g. statements of the program code are nodes, and two nodes $n_1, n_2$ are connected by an edge if there is a possible execution of the program such that first statement $n_1$ is executed followed by statement $n_2$. As an example, an if-then-else construct will result in a node representing the if-statement, and that node will have two outgoing edges.

The actual coverage produced by a set of tests is then computed by firstly counting the cove-

rage items under investigation (e.g. nodes of the control flow graph). Secondly, while executing the tests, we keep track of all coverage items which have been used. Finally, the coverage is computed by dividing the number of used coverage items by the number of all coverage items.

# 3 Coverage Analysis for DMM Specifications

In the previous section, we have seen that one way to define test coverage in software engineering is to use a control flow graph, and to cover that graph. The control flow graph is computed based on the underlying program's structure. Fortunately, DMM specifications contain comparable structures which can be used for coverage definition, as we will see in Section 3.1. Section 3.2 will then use our notion of a control flow graph to define the actual coverage criteria. Finally, Section 3.3 will relate the coverage criteria defined in Section 3.2 and will provide some hints for the language engineer on which coverage criterion to choose.

## 3.1 Invocation Graph

Analogously to control flow graph, an auxiliary data structure called *invocation graph* is defined for the coverage analysis in DMM. The invocation graph reflects the control flow in DMM modeled by the invocation mechanism. According to it, a bigstep or smallstep rule can invoke other smallstep rules which again can invoke further ones. This process repeats until all the invoked rules have no more invocation. So, DMM rules serve as nodes of the invocation graph. Two nodes representing rules which can apply directly one after another during execution are connected with an edge. The final rules applied serve as leaves in the invocation graphs.

The invocation graph represents all possible sequences of rules which can result out of execution of every possible model. For example, if a rule has several implementations corresponding to different application contexts (see Section 2), all cases appear in the invocation graph. For a certain input model, however, only a selection of all possible rule execution orders will be contained in the corresponding LTS. So, our goal is to create such input models that all implementations of all rules are exercised during testing. Only that way we can ensure that each rule works as desired in different contexts – otherwise, this indicates an error in the DMM specification which should be corrected by the language engineer.

In order to perform the coverage analysis, a DMM specification is transformed into a set of invocation graphs, one for each bigstep rule having at least one invocation (the case without invocations is trivial). Invocation graphs computed for the running example are depicted in Figure 4. A single invocation graph consists of nodes corresponding to DMM rules and edges connecting two rules which follow each other directly in the DMM control flow. The rule *start.1* invoking the rule *execute*, which has two implementations *execute.1* and *execute.2*, forms a fork in the invocation graph. A typical example for the recursion is the rule *supplyStreamingToken* invoking the rule *destroy*, which has two implementations: *destroy.1* calls itself recursively and *destroy.2* stops the recursion.

In order to formulate the definition for the invocation graph, let us first introduce some helper definitions. The set of smallstep rule with different unique names invoked by a certain invocation is defined as follows:


Figure 4: Invocation graphs for the running example

**Definition 2** Let $R$ be the rules of some DMM specification, let $i$ be an invocation. The set of smallstep rules invoked by $i$ is as follows: $invokedRules(i) := \{r \in R \mid r\text{'s signature matches } i.\}$[2].

Furthermore, it will be useful to define the set of rules which might be (transitively) invoked during the execution of a particular rule including the rule itself. It is defined as follows:

**Definition 3** Let $r$ be a rule, let $r_I$ be the list of invocations of $r$. The set of smallstep rules transitively invoked by $r$ is defined as

$$transitiveRules(\{r\}) := \{r\} + \bigcup_{i \in r_I} transitiveRules(invokedRules(i))$$

Finally, we need to define the set of rules which are (transitively) invoked by some rule $r$, and which are executed as the very last ones (i.e., immediately before the invocations of $r$ have been completely finished).

**Definition 4** Let $\{R\}$ be a set of rules, for which final rules have to be computed.[3]

```
leafRules({R}) :=
    FOR EACH r IN R
      IF r has invocations THEN
        remove r from R
        LET i be the last invocation of r
        FOR all r' in invokedRules(i)
          ADD leafRules({r'}) to R
    RETURN R
```

We are now ready to define our notion of invocation graphs. The idea is that the execution of a rule $q$ can follow that of a rule $p$ in the following cases: firstly, $p$ invokes $q$ as its first invocation;

[2] As explained in Section 2

[3] This definition is a simplification not considering loops and sharing of identical subgraphs in the invocation graph.

secondly, $q$ matches after the execution of some rule which (transitively) invokes $p$ as its very last rule. This leads to the following definition of an invocation graph:

**Definition 5** Let $B$ be a bigstep rule. $B$'s *invocation graph* $G_{inv} = (V, E)$ is defined as follows:

$$
\begin{aligned}
V &= transitiveRules(\{B\}) \\
E &= \{(v_1, v_2) \mid \exists v_1 \in V \wedge (i_1, ..., i_m) \text{ are invocations of } v_1 \wedge v_2 \in invokedRules(i_1)\} \bigcup \\
&\quad \{(v_1, v_2) \mid \exists r \in V \wedge (i_1, ..., i_n) \text{ are invocations of } r \wedge \exists k \in 1, ..., n-1 : \\
&\quad v_1 \in leafRules(invokedRules(i_k)) \wedge v_2 \in invokedRules(i_{k+1}))\}
\end{aligned}
$$

As a result, the invocation graph of a bigstep rule contains all possible sequences of rule applications which might arise as a result of applying $B$, similarly to a control flow graph containing all possible execution paths of a program. How the coverage for DMM can be computed based on the invocation graph is presented in Section 3.2.

## 3.2 Coverage Definition

Executing a DMM specification in all possible ways is usually computationally infeasible. So, the goal of coverage analysis is to define some efficient and feasible coverage criteria, which are, on the one hand, expressive enough and, on the other hand, require reasonable number of tests.

### 3.2.1 Rule Coverage Criteria

Analogous to test coverage in software engineering, a coverage item has to be chosen at the beginning. In DMM, nodes (DMM rules) and edges (pairs of DMM rules following each other directly in the DMM control flow) of the invocation graph serve as coverage items. Having chosen rules as coverage item, the first family of coverage criteria, called *rule coverage criteria*, is explained in this section.

**Rule Coverage** The criterion serving as a minimal quality measurement of DMM specifications is *rule coverage (RC)*. The minimal quality is guaranteed, when at least one correct application of each rule is ensured. RC demands each rule being tested at least in one matching context, i.e. being used at least once during the execution of the test suite. For the running example (see Figure 4), full RC is achieved under the following condition: all rules having different unique names over the whole DMM specification have to apply at least once on models from the used test suite. In this case, 8 rules have to be covered in order to achieve a coverage of 100%.

In case coverage is lower than 100%, the DMM coverage tool provides feedback for improving the test suite. For RC, this is rather simple: The tool provides a list of rules which were not used during execution of any of the test models. Note that we are aware that it would be more helpful to provide feedback on how additional test models should look like to improve coverage (which currently needs to be figured out be the language engineer); this is a subject of our current research (see Section 5).

The LTSs for the running example, one for each example model, are presented in Figure 5. They cover 5 rules from 8 possible from the invocation graphs. So, the coverage equals 62,5%.

The formal definition for RC is based on Definition 1:

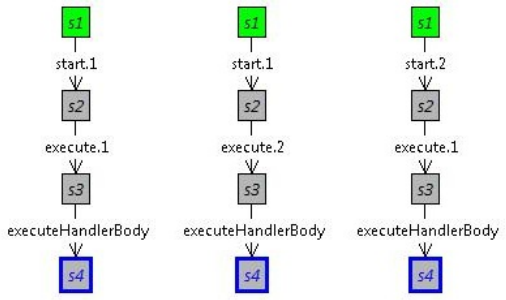


Figure 5: LTSs for the running example (excerpts)

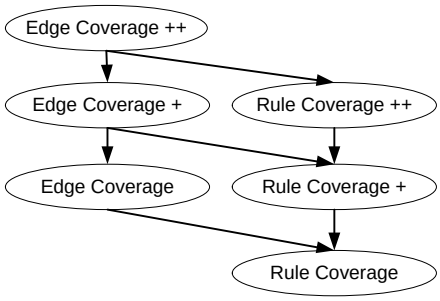


Figure 6: Hierarchy of the coverage criteria [Ari11]

**Definition 6** Let $LTS = (V, E)$ be a LTS for an input model. A rule is considered to be covered iff there exists $(v_1, s, v_2) \in E$, where $s$ is the unique name of the rule.

Assuming that full RC is achieved, there are still some application cases which were not tested, e.g. it is sufficient to check the application of a rule only in one invocation graph even if it appears in several ones. So, further rule coverage criteria with increasing level of complexity are needed, in order to detect potential errors that the previous ones cannot.

**Further Rule Coverage Criteria** A criterion which exercises more matching contexts in comparison to RC is *rule coverage plus (RC+)*. It ensures that each rule applies correctly at least in one matching context in each invocation graph where it appears. In other words, RC+ ensures RC for each invocation graph separately.

However, if a single rule appears more than once in the same invocation graph, paths containing its further occurrences may still remain untested. Therefore, a further criterion called *rule coverage plus plus (RC++)* is defined. It guarantees the coverage of execution paths, which use all nodes in each invocation graph at least once. As a result, rules are tested in more matching contexts and more possible failures can be revealed in comparison to RC+.

Because of the definitions of RC+ and RC++ (and also the definitions presented below), it does not suffice any more to only report a list of rules in case coverage is lower than 100%. As such, we from now on provide sequences of rule applications which–if triggered by an additional test model–would improve coverage. As already mentioned, providing improved feedback is subject of our current research.

### 3.2.2 Edge Coverage Criteria

Despite of the high power of RC++, paths formed from the already tested nodes following each other in another way are still not handled. Consider the case when a node has N incoming edges. In this case N paths are possible through this node. However, rule coverage criteria would require testing only one path from these to cover the necessary node. So, possible errors in the other paths can only be detected when connections among rules are taken into account as well. So, the family of edge coverage criteria has been introduced. Since the simplest edge coverage criterion is analogous to RC and is rather intuitive, we will start with a criterion having medium

expressiveness and then shortly explain the other two ones.

**Edge Coverage Plus** The criterion of medium expressiveness in this family is called *edge coverage plus (EC+)*. It requires all distinct edges being tested at least once in each invocation graph. Two edges are considered distinct, if they have different names. The name of an edge is formed from the unique names of the rules comprising this edge in the form: $source\_node\_unique\_name \rightarrow target\_node\_unique\_name$. So, to achieve the coverage value of 100%, an edge with the same name should be tested at least once in each invocation graph.

For the running example (see Figure 4), in order to achieve full EC+, 12 edges should be covered. The edges $execute.1 \rightarrow executeHandlerBody$ and $execute.2 \rightarrow executeHandlerBody$ appearing in two invocation graphs are considered as separate elements and so have to be tested in both contexts. The formal definition of EC+ is based on Definition 1 and defined as follows:

**Definition 7** Let $LTS = (V,\ E)$ be a LTS for an input model. An edge is considered to be covered iff there exist two transitions in LTS $(v_1,s,v_2),(v_2,t,v_3) \in E$ directly following one another with $s$ and $t$ equal to unique names of the rules comprising the edge.

**Further Edge Coverage Criteria** *Edge coverage (EC)* is the simplest coverage criterion in this family. Analogous to RC, it demands all distinct edges from the invocation graphs for a given DMM specification to be exercised at least once during testing. So, it guarantees that each pair of rules represented in the form of edges applies correctly at least once during the execution.

EC exercises some matching contexts which are never taken into account by the rule coverage family. For the example with a node having multiple edges, all these edges have to be checked according to EC, since they have different names, and so more paths are involved in testing. However, EC tests a single edge only in one matching context, i.e. after a particular sequence of rules. Another sequence of rules leads to another host graph and so to another possible matching context. So, similar to RC, further edge coverage criteria able to test more application contexts are needed. One of them is the presented above EC+.

Assuming that full EC+ was reached, some application cases still remain unchecked. Such case would be the same edge appearing in the same invocation graph more than once. In this case, the same pair of rules applies on different state graphs resulting in different behavior. So, a further coverage criterion involving more execution paths in the testing process should be defined. This coverage criterion is *edge coverage plus plus (EC++)*. It ensures that all edges from all invocation graphs computed for a given DMM specification are covered during testing. Thus, it tests the same edges in more matching contexts compared to EC+. However, EC++ still does not cover all application cases, e.g. edges in loops after the second iteration still remain unhandled.

## 3.3 Hierarchy

In order to judge about the power of each criterion individually and to help the language engineer to decide which criterion would be the most suitable to use, a hierarchy of the coverage criteria in DMM is developed and presented in Figure 6.

The principle of the hierarchical dependencies is the following: If tests with respect to a coverage criterion located on a higher level of hierarchy achieve a complete coverage, it implies

a complete coverage for all criteria placed lower than this one. RC guarantees a minimal level of quality by assuring the minimal part of the invocation graph to be correct. Some coverage criteria, e.g. EC+ and RC++, are not connected by any arrow at all, since they cover different parts of the invocation graph, which do not overlap with each other.

The hierarchy shows how powerful each criterion is. However, it is not always feasible for the language engineer to use the most expressive one, because of different computational complexity. With the help of the DMM coverage tool, the computation complexity of each criterion was evaluated based on several semantics specifications. A general advice for the language engineer to conduct the coverage analysis would be to start with EC+, since this criterion implies EC, RC+, and RC and requires relatively few computational effort. If the coverage analysis with EC+ succeeded, the language engineer could try RC++ or EC++. Otherwise, the language engineer should perform both EC and RC+, since they handle different kinds of errors.[4] Empirical evidence supporting the given guidelines are omitted here due to space limitations but they can be found in [Ari11].

## 4 Related Work

Different approaches to measure test quality exist and can be divided into two main kinds: Metrics which can be applied during testing, and metrics which can only be applied after delivery of the software (the latter mainly aiming at discovering general problems in software quality assurance). For example, one can measure the amount of bugs discovered after delivering the software; being put in relation with the number of bugs found during testing, this can be an indication that more effort needs to be put into testing for the next project. The obvious drawback is that the system under test itself will not benefit from such metrics. However, as an advantage, this way to measure test quality will provide feedback on the actual software quality (in contrast to the approaches described below, which measure test quality under some assumptions). Moreover, such approaches are very general and can therefore be applied in a broad variety of scenarios (including DMM specification).

Measuring test quality at testing time is more difficult. One important approach is *mutation analysis*, where flaws are intentionally injected into a software system. The quality of the system's tests is then quantified as the relation between the number of flaws introduced and the number of these flaws actually discovered by the tests. For example, Haschemi and Weißleder [HW10] present a generic approach to run mutation analysis, where the creation of mutants is separated from the mutation analysis execution environment. This approach could likely also be used in the context of DMM specifications. However, the metrics presented in our paper depends on the structural properties of DMM specifications (i.e. invocation structures); as a result, our framework is able to provide more concrete hints on how to improve test quality.

Another important approach is the area of *test coverage*. As we have seen earlier, the idea is to define parts of the system to be covered during test execution. The coverage is then defined as the ratio of the system parts actually covered when running the tests and the overall number of system parts. Many works on test coverage exist, especially in the area of software testing (see

[4] We do not claim the computational complexity always be distributed this way. The presented results are rather implementation dependent.

e.g. [MSB11] for a pragmatic introduction).

Of particular interest in the context of our paper is *model-based testing*, where different kinds of coverage criteria have been defined (a good overview of model-based testing is provided e.g. in [UL06]). For example, Friedman et al. describe in [FHNS02] different coverage criteria for state machines, which are then used to automatically generate and evaluate test cases. Another example is [FFP10] where Ferreira et al. compute the coverage of UML activities by simulating them, and provide visual information on which parts of the activity have been covered.

On a more general level, Friske et al. [FSW08] investigate an application of combinations of different coverage criteria and optimization of test suites at different abstraction levels. This approach generates tests based on the structural elements and coverage criteria formalized by means of OCL. Then, comparison, merging, and optimization of coverage criteria of different types (e.g. structural and data), structural elements to test and test suites are possible. In [WS08], Weißleder and Schlingloff define new coverage criteria as a combination of advantages from condition- and boundary-based types, and use them to automatically generate test cases. The efficiency of the introduced coverage criteria is evaluated using mutation testing.

McQuillan and Power in [MP09] examine white-box coverage criteria for model transformations specified with ATL [JABK08]. In particular, they define the notions of *rule coverage* (which is very similar to our definition of rule coverage), *instruction coverage* and *decision coverage*, the latter roughly corresponding to the code coverage metrics statement and branch coverage. This is the main difference to our work: Since ATL transformation rules are specified textually (in contrast to the visual DMM language), it is rather straight-forward for the authors to adjust existing coverage criteria for ATL transformations.

Almost all of the above approaches primary deal with evaluating the tests of *models*, whereas we are interested in evaluating the tests of *semantics specifications*. As such, the described approaches can be used in parallel to our approach (i.e., to evaluate models whose language is already equipped with a DMM specification). They have, however, served as a source of inspiration for our work.

## 5 Conclusion and Outlook

**Conclusion** This paper proposes coverage analysis to evaluate the quality of tests used for the development of high quality DMM specifications. For this purpose, we have defined *invocation graphs* as a means to describe the control flow existing in DMM specifications due to rule invocation. Based on that, we have defined several coverage criteria, *rule coverage* being the weakest and *edge coverage plus plus* being the strongest (and most difficult to achieve). Finally, we have shown how our coverage criteria relate to each other, and we have given the language engineer some simple guidelines on which coverage criterion to choose.

With the help of coverage analysis, the language engineer can obtain better tests and, as a result, test to a larger extent the developed DMM specification for correspondence with the desired behavior. This should yield DMM specifications of higher quality in comparison to a situation without coverage analysis. This contributes to the quality of models designed in these languages as well, since they can be analyzed based on a correct semantics specification. For this purpose, the DMM coverage tool was implemented within the DMM workbench (an Eclipse-

based collection of plug-ins supporting the creation of DMM semantics specifications). The DMM coverage tool computes the coverage value for selected coverage criteria, and provides (currently rather simple) recommendations on how to improve the coverage.

Techniques presented in this paper including the coverage criteria and DMM coverage tool are currently evaluated within the scope of the DFG's Collaborative Research Centre 901 "On-The-Fly Computing". They are applied to formally specify behavioral semantics for parts of a service specification language. Furthermore, in the future, we plan to investigate the following paths of research:

**Test models** Currently, recommendations for test suite's improvement our tool provides are of limited use. This is because the tool only reports which of the structural elements considered for the respective coverage criterion have not been covered during execution of the test suite. It would be much more helpful to provide the language engineer with feedback on the structure of models to be added for the sake of improving test coverage, or to even (semi-) automatically create such additional test models. For this, we will start with thoroughly investigating [GMS00] and [GBR98].

**Critical Pair Analysis (CPA)** The coverage analysis proposed in this paper only takes invocations of smallstep rules into account. However, if a DMM specification makes use mainly of bigstep rules, the analysis results are not very helpful. A trivial way to come up with possible sequences of bigstep rules to be covered would be to compute the permutations of all bigstep rules, but this will be computationally infeasible for most cases. CPA might be a way to remove sequences of rule applications which can not occur due to the rules' structures (e.g., one rule might add a node which disables the application of another rule).

## Bibliography

[Ari11] S. Arifulina. Coverage Criteria for Testing DMM Specifications. Master's thesis, University of Paderborn, 2011.

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, S. Sauer. Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceedings of UML 2000*. LNCS 1939, pp. 323–337. Springer, Berlin/Heidelberg, 2000.

[ESW07] G. Engels, C. Soltenborn, H. Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In *Proceedings of FMOODS 2006*. LNCS 4468, pp. 76–90. Springer, Berlin/Heidelberg, 2007.

[FFP10] R. D. F. Ferreira, J. P. Faria, A. C. R. Paiva. Test Coverage Analysis of UML Activity Diagrams for Interactive Systems. In *Proceedings of QUATIC 2010*. Pp. 268–273. IEEE Computer Society, Washington, DC (USA), 2010.

[FHNS02] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected State Machine Coverage for Software Testing. *SIGSOFT Softw. Eng. Notes* 27:134–143, 2002.

[FSW08] M. Friske, B.-H. Schlingloff, S. Weißleder. Composition of Model-based Test Coverage Criteria. In *MBEES*. Informatik-Bericht 2008-2, pp. 87–94. TU Braunschweig, Institut für Software Systems Engineering, 2008.

[GBR98] A. Gotlieb, B. Botella, M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. *SIGSOFT Softw. Eng. Notes* 23:53–62, March 1998.

[GMS00] N. Gupta, A. P. Mathur, M. L. Soffa. Generating Test Data For Branch Coverage. In *Proceedings of the International Conference on Automated Software Engineering*. Pp. 219–227. IEEE, 2000.

[Hau05] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2005.

[HHT95] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* 26:287–313, 1995.

[HW10] S. Haschemi, S. Weißleder. A Generic Approach to Run Mutation Analysis. In *Proceedings of TAIC PART 2010*. LNCS 6303, pp. 155–164. Springer, Berlin/Heidelberg, 2010.

[IST10] Standard Glossary of Terms Used in Software Testing, Version 2.1. Technical report, Glossary Working Party, International Software Testing Qualifications Board, 2010.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.* 72(1–2):31–39, 2008.

[MP09] J. A. McQuillan, J. F. Power. White-Box Coverage Criteria for Model Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL*. CEUR Workshop Proceedings, Aachen (Germany), 2009.

[MSB11] G. J. Myers, C. Sandler, T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[SE09] C. Soltenborn, G. Engels. Towards Test-Driven Semantics Specification. In *Proceedings of MODELS 2009*. LNCS 5795, pp. 378–392. Springer, Berlin/Heidelberg, 2009.

[UL06] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA (USA), 2006.

[WS08] S. Weissleder, B.-H. Schlingloff. Quality of Automatically Generated Test Cases based on OCL Expressions. In *Proceedings of ICST 2008*. Pp. 517 –520. IEEE Computer Society, Washington, DC, USA, 2008.