RZ 3846 (# ZUR1302-013) 05/01/2013
Computer Science 17 pages

Research Report

Integrating Coverage Analysis into Test-driven Development of Model Transformations

Jochen M. Küster[1], Dániel Kovács[1], Eduard Bauer[2], and Christian Gerth[3]

[1]IBM Research – Zurich, 8803 Rüschlikon, Switzerland
Email: {jku, dko}@zurich.ibm.com

[2]Deutsche Bank, GTO - GT Retail, Alfred-Herrhausen-Allee 16-24, 65760 Eschborn, Germany
Email: eduard.bauer@db.com

[3]Department of Computer Science, University of Paderborn, Germany
Email: gerth@upb.de

Integrating Coverage Analysis into Test-driven Development of Model Transformations

Jochen M. Küster[1], Dániel Kovács[1], Eduard Bauer[2], and Christian Gerth[3]

[1] IBM Research - Zurich, Säumerstr. 4
8803 Rüschlikon, Switzerland {jku,dko}@zurich.ibm.com
[2] Deutsche Bank, GTO - GT Retail - Alfred-Herrhausen-Allee 16-24
65760 Eschborn, Germany eduard.bauer@db.com
[3] Department of Computer Science, University of Paderborn, Germany
gerth@upb.de

Abstract. For testing model transformations, a software engineer usually designs a test suite consisting of test cases where each test case consists of one or several models. In order to ensure a high quality of such a test suite, coverage achieved by the test cases with regard to the system under test must be systematically measured. The results obtained during coverage analysis can then be used e.g. for creating additional test cases. In addition to measuring coverage, a software engineer developing a model transformation is also confronted with how to integrate such coverage analysis results into the development process. For example, the software engineer has to decide when to measure coverage, when to investigate the results and how and when to take appropriate actions. In this paper, we present a prototypical tool which can be used for measuring coverage of test requirements for model transformations. We explain how a software engineer can make use of it in a test-driven development process for model transformations, in order to systematically develop high-quality model transformations.

1 Introduction

Model transformations are nowadays used in model-driven engineering for model refinement, model abstraction, and for code generation. Model transformations can either be implemented directly in programming languages (such as Java) or using one of the available transformation languages that have been developed in recent years (e.g. [8, 13, 17]). For testing model transformations, systematic software testing has to be applied in order to ensure the high quality of the specific model transformations. In the context of this, a software engineer usually designs a test suite consisting of test cases, where each test case consists of one or several models. One important aspect of testing is to measure and ensure a high coverage level which is used to be certain of the high quality of the test suite and the system under test [15].

Existing work on testing model transformations (cf. [4]) shows how a metamodel from the input language of a model transformation can be used to determine the quality of that model transformation. In our recent work [5, 6], we have proposed a coverage analysis approach which allows measuring coverage achieved by a test case with regard to a model transformation. Our coverage analysis approach includes coverage analysis both on the model level as well as on the code level. The result of coverage analysis can

then be used to identify untested parts of the system under test or fully-redundant test cases.

In addition to measuring coverage, a software engineer developing a model transformation is also confronted with how to integrate such coverage analysis results into the development process. For example, when should a software engineer start measuring coverage and when should he or she react if coverage levels are not high enough? Furthermore, the question arises, how coverage analysis should be integrated into the overall development process of a model transformation. In this paper, we explain, how our prototypical tool, the Test Suite Analyzer [6] can be used for measuring coverage achieved by test model transformations. It measures coverage achieved by test cases in a model transformation by computing a so-called footprint which contains the main characteristics of each test case execution. We describe a test-driven development process for model transformations and we explain how a software engineer can make use of our tool in such a process, in order to systematically develop model transformation test suites with high coverage values.

The paper is structured as follows. We first give some fundamentals concerning model transformations and coverage analysis in Section 2. Section 3 introduces the Test Suite Analyzer and describes the use cases supported. In Section 4 we explain how the tool can be used to develop model transformations in a test-driven approach. We discuss related work in Section 5 and conclude in Section 6.

2 Fundamentals

2.1 Theoretical foundations

A model transformation transforms one or more input models into one or more output models. For a model transformation one can distinguish between the specification and the implementation (source code) of the transformation. The input and output domains, i.e. the possible input and output models of a model transformation are specified by metamodels, see Figure 1. Metamodels can be used by relying on an input/output language that is defined by a standard (i.e. the UML definition) or can be specified by the software engineer on his/her own. The logic for transforming the input models to output models is specified declaratively by a transformation contract [10], which is inspired by the design-by-contract approach [24], adapted for model transformations. A transformation contract consist of three types of contract rules: *precondition rules*, which have to be fulfilled by the input models; *postcondition rules*, which have to be fulfilled by the output models; and *transformation rules*, which describe the relation between the input and output models. Transformation contracts are also part of the specification of a model transformation, as illustrated in Figure 1.

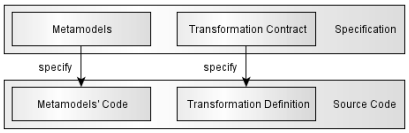


Fig. 1: Specification and concretization of a model transformation

The logic specified by a transformation contract is implemented by the so-called transformation definitions [9]. Transformation definitions can be expressed using one

of the numerous model transformation languages (e.g. Query View Transformation (QVT) [13] or ATLAS Transformation Language (ATL) [17]) or using general purpose programming languages (such as Java). For transforming input models to output models, some transformation definitions use code to represent the models by objects and references. Such code, called metamodel code, is usually automatically derived from the metamodels (e.g. using EMF). The transformation definition and the metamodel code belong to the implementation of the model transformation, as shown in Figure 1. Besides these two fraction of code, glue code (implementing functionalities such as logging, exception handling, initializers, controllers, etc.) might exist. In the following we do not pay attention to these latter pieces of code.

A model transformation has to be thoroughly tested in order to achieve high quality. Software testing is usually done by creating a number of test cases, consisting of input values and expected results, which test the correct behavior of a System Under Test (SUT) on given input [1]. Thus, for a model transformation, a test case consists of input models and expected output models. For example, in Figure 2 two transformations from BPMN to a simple graph representation can be seen. A test case for such a transformation consists of the BPMN as input model and the simple graph as expected output model, and the test would check whether the transformed output and the expected models are the same or not.

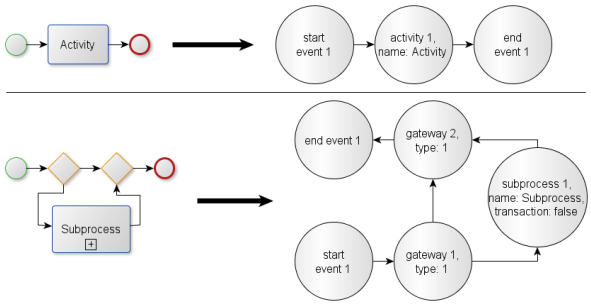
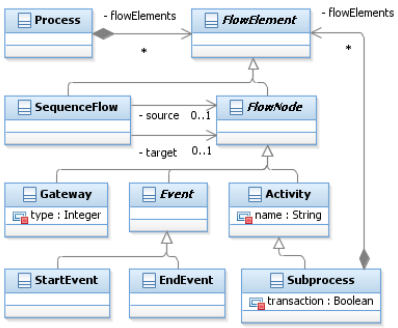


Fig. 2: Two example test cases (transforming a BPMN into a simple graph)

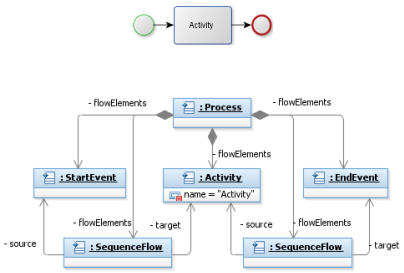
To determine the quality of a test suite, coverage analysis is applied. In this context, the concept of a test requirement [1] is used in order to represent a particular element of the SUT that has to be tested for coverage. The elements that are used to derive test requirements from the SUT are defined by coverage criteria. One very common coverage criterion is statement coverage, which derives a test requirement from each statement of the SUT. As this coverage criterion derives test requirements from the source code it is a so-called code-based coverage criterion. Besides this there is the so-called specification-based coverage criterion, which derives test requirements from the specification of model transformations (e.g. the metamodels). For example the class coverage derives test requirements from classes in the metamodels.

A test requirement is covered, if the according element is executed/used by a test case during execution. Hence a test requirement derived by statement coverage is covered, if that specific statement is executed; and a test requirement derived from a class of a metamodel is covered if that class is instantiated in the model. The data about covered and non-covered test requirements are collectively called coverage information. For model transformation, specification-based coverage criteria as well as code-based coverage criteria can be used, which both yield different kinds of coverage information. In this paper we focus on specification-based coverage.

2.2 The usage of specification-based coverage



(a) The simplified BPMN metamodel



(b) An exemplary BPMN model and a possible illustration above it

Fig. 3: The used metamodel and a concrete model instance of it

The combined coverage approach [6] is an approach that derives test requirements by different specification-based coverage criteria from metamodels and the transformation contract of a model transformation. In the following, we briefly summarize the combined coverage approach, for a detailed introduction, the reader is referred to [6]. The examples used for illustration make use of a simplified subset of the Business Process Model and Notation (BPMN) metamodel shown in Figure 3a. A simple instance in concrete and abstract syntax is illustrated in Figure 3b.

The coverage criterion *class coverage* [11] uses the classes of a metamodel to derive test requirements. For each class in the metamodel, a test requirement is derived. For the metamodel shown in Figure 3a, one test requirement is, for example, derived from the class `StartEvent`. This test requirement is covered by the `StartEvent` instance in the model shown in Figure 3b.

The coverage criterion *attribute coverage* [11] derives test requirements from the attributes of the classes of a metamodel. For this, the common software testing technique, equivalence partitioning [1] is used to partition the domains of the specific attributes into blocks. For example a straightforward equivalence partitioning consists of two test requirements for boolean values (for `true` and `false`), two for string values (for empty and not empty strings) and two for integers (for negative and positive values). A test requirement is e.g. derived from the block $\{$*not empty string*$\}$ for the attribute `Activity.name`. This test requirement is covered by the instance of the class `Activity` in Figure 3b, which has a not empty string as name.

The coverage criterion *association coverage* [11] (also called *reference coverage*) uses associations of a metamodel to derive test requirements. For this, equivalence partitioning is used on the multiplicity of the associations. For example a straightforward equivalence partitioning consists of the partitions $\{\{0\};\ \{1\};\ \{[2,+\infty)\}\}$ on the multiplicity of the association. A test requirement is e.g. derived from the block $\{[2,+\infty)\}$ for the association `Process.flowElements`, which is covered by the instance of the class `Process` as it does contain five `FlowElements`.

In addition to these coverage criteria, the combined coverage approach uses features to derive test requirements. A feature can be seen as a particular characteristic of a concrete model instance. Since models consists of model elements, a feature is a particular combination of several model elements of the corresponding metamodel. Features are defined by the tester of a model transformation. The coverage criterion *feature coverage* uses the features defined in the context of a metamodel to derive test requirements, thus a test requirement is created that requires instances of the metamodel to have the particular combination of model elements defined by the feature. An exemplary feature for the BPMN metamodel is defined by the Object Constraint Language (OCL) expression `self.flowElements->exists(x | x.oclIsTypeOf(Subprocess))` (defined in the context of the class `Subprocess`). This feature describes nested subprocesses. The resulting test requirement is not covered by the exemplary BPMN model as it does not contain a `Subprocess` that itself contains another `Subprocess`. Currently, features in the combined coverage approach have to be defined manually.

The combined coverage approach derives test requirements from the transformation contract too. As described earlier in Section 2.1, a transformation contract consists of contract rules. The coverage criterion *transformation contract coverage* uses the contract rules of a transformation contract to derive test requirements. For each contract rule of a transformation contract, a separate test requirement is created. An exemplary precondition rule for the BPMN metamodel is that every `SequenceFlow` must have both a `source` and a `target` or none of them. The result of the evaluation of a transformation contract for a model transformation is called contract result. The contract result contains the evaluation of each contract rule which are called contract rule results. A contract rule result is a pair of integers, the first counts how often the condition stated by the contract rule is fulfilled, and the second counts how often is it neglected (an additional error message can be attached to every failure). The derived test requirement is satisfied if the contract rule result is a $(x, 0)$ pair of integers where $x > 0$ holds. As the exemplary BPMN model contains two `SequenceFlow`s that have exactly as many sources as targets, the contract rule result for the former mentioned exemplary rule is $(2, 0)$ and thus the derived test requirement is covered.

A straightforward way to represent a coverage report is the concept of footprints. A footprint is basically a mapping from the different test requirements and their coverage count for the different test cases – where the coverage count states how many times a particular test requirement is covered by the test case. For example if a class is instantiated in a test case, then the coverage count for the corresponding class coverage test requirement for that test case is 1. If there are for example two class instances with attribute values from the same equivalence partition, than the test requirement for that equivalence partition of that attribute of that class is 2.

In this section we described shortly the theory behind coverage analysis, which is extensively used by our tool. In the next section we proceed and describe the architecture and the use cases of it.

3 The Test Suite Analyzer for Model Transformations

In order to validate the combined coverage approach (described in Section 2), we created a prototypical tool for analyzing test suites of model transformations, we call this prototype the Test Suite Analyzer for Model Transformations. It supports different mechanisms for analyzing coverage achieved by the test suites of model transformations. Figure 4 shows a screenshot and illustrates the obtained specification coverage information and comparison of two particular test cases (bottom of figure) and the corresponding footprint (top of figure). On the left you can see the metamodels, contracts, etc. involved in the model transformation.

We distinguish between different phases of using the tool: setup and usage. The setup is the first phase, within it the data to be generated needs to be defined, similarly the transformation has to be designed and implemented. Currently this has to be done by the software engineer by:

- creating the transformation contract rules as plain Java classes;
- assembling the transformation contract from the defined contract rules;
- creating the transformation (currently transformation defined with a Java class are supported);
- creating the metamodel features as plain Java classes (optional);
- creating the so-called test case creator and the coverage controller creator, which are classes responsible for gathering the coverage data.

In the second, usage phase, the gathered data can be interpreted by the software engineer. In the following, we describe the use cases supported by our tool for these two phases.

The supported use cases in the first phase are shown in Table 1. The three use cases 'Create equivalence partitioning', 'Specify and implement transformation contract' and 'Specify features for the metamodels', determine the specification coverage information to be obtained. The fourth use case 'Produce coverage report' measures the coverage of a certain test suite. The first three use cases correspond directly to the theory behind our approach: in order to measure coverage with regard to the transformation contract, this contract has to be defined. Similarly, in order to measure coverage with regard to features, such features have to be defined. In addition, usually equivalence partitions also have to be defined.

The supported use cases in the second phase are shown in Table 2. In order to investigate different test cases with regard to their coverage, the three 'Inspect' use cases

Use Case	Description
Preparations	
Create equivalence partitioning	The user is provided with functionality to express equivalence partitions for attributes and associations.
Specify and implement transformation contract	Transformation contract for transformation definitions is specified.
Specify features for the metamodels	Features can be defined in the context of applied metamodels.
Produce coverage report	The actual calculation of the coverage information in terms of footprints for test cases is created and persisted.

Table 1: Summary of use cases for preparing combined coverage approach

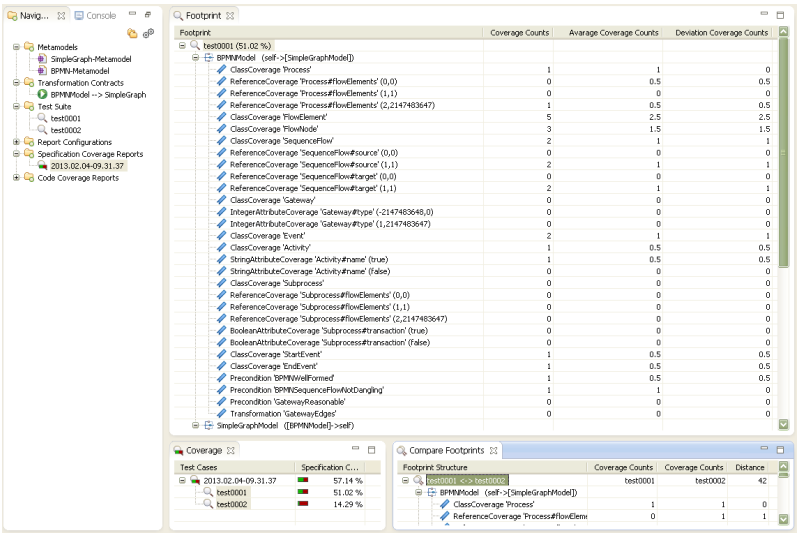


Fig. 4: A screenshot from the Test Suite Analyzer

can be used. In order to identify unsatisfied test requirements, the software engineer can use ‘Inspect contract coverage’ and after ‘Compute missing coverage’ he or she can use ‘Inspect specification coverage information’. Our tool also supports computation of redundant test cases through the use case ‘Compute redundant test cases’. After this the software engineer can ‘Inspect redundancy information’.

The architecture of our current environment for using the prototype is shown in Figure 5. It requires that during test case execution, suitable data is generated which is then stored in the form of a coverage report. This coverage report can then be used by the tool to find out various information about the test cases. Based on this, the software engineer can decide to act in various forms, which is described in detail in Section 4. In our environment, the Test Case Execution Component is based on the JUnit framework and is used to execute test cases using a Model Transformation Execution Component. In principle, the Model Transformation Execution Component can be any of the well-known model transformation engines (such as ATL [17]) or general purpose programming languages (such as Java). The Model Transformation Execution Component executes the model transformation on a set of input models. The Test Case Execution Component

Use Case	Description
Analyze coverage report	
Inspect contract information	A view is provided for inspecting contract fulfillments information.
Compute missing coverage	A computation is provided for calculating missing test requirements.
Inspect specification coverage information	Different views are provided for the analysis of separate footprints as well as for the comparison of different footprints.
Compute redundant test cases	Computation of pairs of redundant test cases.
Inspect redundancy information	The redundant test cases can be identified through different views.

Table 2: Summary of use cases for analyzing coverage information

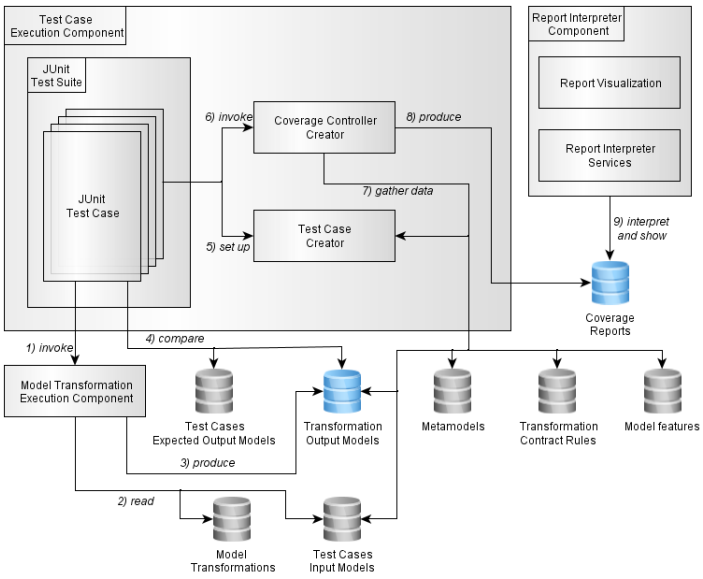


Fig. 5: The architecture of the Test Suite Analyzer

then evaluates the result of each execution. During this phase, the metamodels of the models as well as model transformation contract rules and model features are taken into account to produce the data in the form of a coverage report.

To be more precise, Figure 5 illustrates the architecture of the Test Case Execution Component too. Inside this component, two components from the Test Suite Analyzer are responsible for generating suitable data for the coverage report: the Coverage Controller Creator and the Test Case Creator. The former is responsible for deciding against which metrics coverage is measured, and the latter is responsible for adjusting these various metrics to the concrete test cases.

4 Using the Test Suite Analyzer for Test-driven Development

In this section, we describe how our prototype is used in a test-driven development process of model transformations. We first give an overview of the development approach and then we explain the details of the approach along a case study.

4.1 The development approach

Existing software development processes such as the Rational Unified Process [18] are incremental and iterative. For model transformations, recent work by Siikarla et al. [27] argues that model transformations must also be developed in several iterations. Given our prototypical tool, the question arises, how such a tool can be integrated into the development approach. In the following, we present our approach which focusses on

design, implementation and testing activities of model transformation development, integrating the Test Suite Analyzer for measuring coverage and creating test cases. Figure 6 provides an overview of our approach. Note that our approach does not include activities for requirements specification, analysis and detailed roles involved in the process. For these activities, we refer to the body of existing work on model transformation design (e.g. [20, 29]). In the following, we describe the main steps of the approach.

- The environment for the tool is set up.
- An initial test case set is created for the model transformation.
- After requirements specification and analysis of the model transformation, development starts with the design and implementation of a first version of the model transformation. This activity also includes implementing transformation contracts and features.
- The model transformation is tested using the test case set and the coverage is measured using the tool.
- Potential defects are categorized according to their severity and then treated in an improvement activity of the model transformation.
- The coverage report is analyzed and the test suite is adapted by adding test cases or potentially removing redundant test cases.
- If the test case set is to be extended, the tool is used in order to determine missing coverage. Using the information about missing coverage, new test cases are created and added to the test case set.
- Once a model transformation passes all test cases, new functionality can be designed and implemented if there are requirements not addressed in the previous iteration.
- If new functionality is added to the transformation, the tool is used in order to determine relevant test cases. These test cases can be used by the software engineer in order to implement the new functionality.

When following this approach, the model transformation will always pass all test cases in the test case set. As such, the quality of the model transformation will gradually increase over time if the activities (increasing set of test cases and adding functionality) are aimed at increasing the quality of the model transformation under development. In the following section, we elaborate each of the activities, illustrating them with a case study.

4.2 Detailed description and case study

In this subsection we describe in detail, how our prototype can be used for development of model transformations. For a practical insight we incorporate the simple case study we carried through, which was based on a simplified BPMN metamodel and a very simple graph metamodel. They can be seen in Figure 7.

Our development cycle is based on the approach described in [19], and it can be seen in Figure 6. We provide here a walkthrough for the approach and also give some guidelines how it can be used in real-life situations.

Let us assume that a project needs a simple model transformation with one input model and one output model, and for that the two metamodels are given. From now

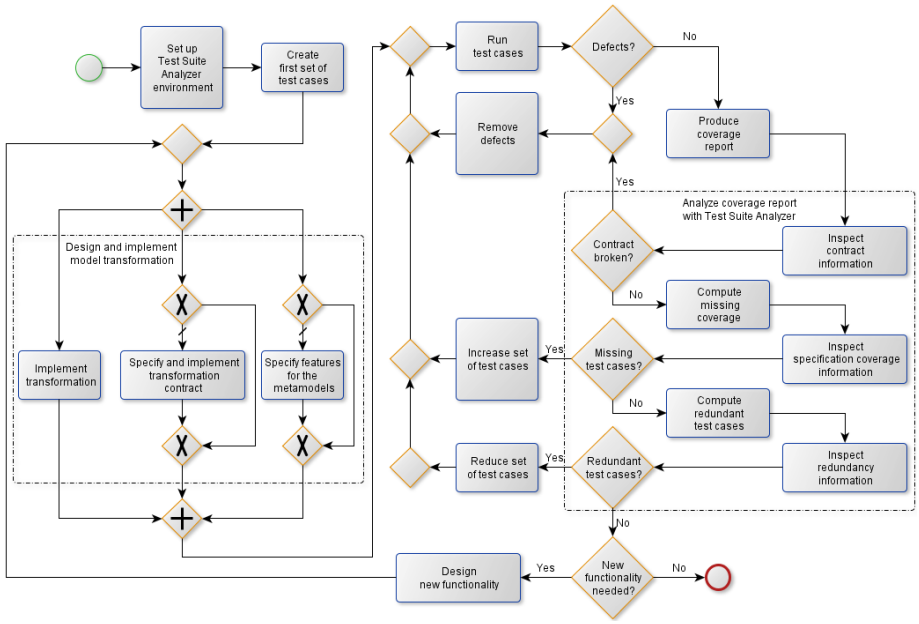
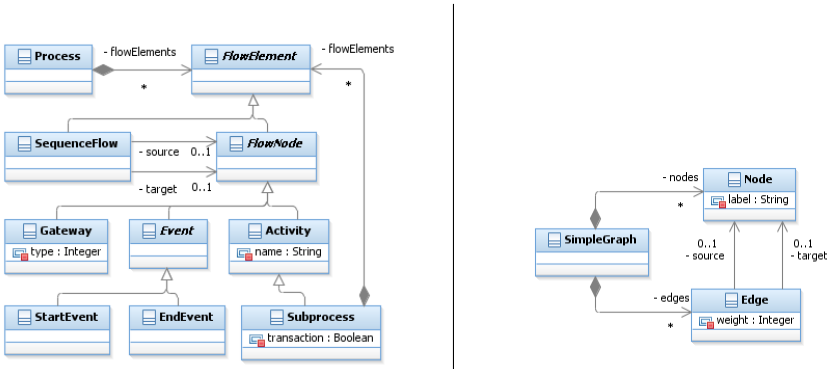


Fig. 6: Agile test-driven development approach for model transformations

on we will refer to them as input (Figure 7a) and output (Figure 7b) metamodels. The described steps can be expanded for situations which involve more input and/or output metamodels in one transformation.

Set up Test Suite Analyzer environment. The first activity is the set up phase. First, a specific Coverage Controller Creator and a Test Case Creator have to be implemented – a basic implementation of these classes can be found in our prototype, however the customization of them is very project specific. After this, the JUnit Test Suite has to be created in a way that after every execution of the test cases the coverage report is generated and persisted. In our case study, we have created a `check` method which is called from every test case with the different models (the input and the expected output). This method is responsible for verifying the transformation, i.e. the transformation generates the expected output or not, and if it does, then it calls the specific Coverage Controller Creator and Test Case Creator to gather and store the coverage report into a directory on the file system.

Create first set of test cases. In the next activity we create the initial test cases. Our guideline is that a test case set should be composed of small test cases, which are "as orthogonal as possible", meaning that the intersection of the test cases' coverage is as minimal as possible. In our experience, such a test case set is the most flexible and maintainable on the long run. The first set of test cases should contain the trivial, most basic and not extreme models and behaviors. Hence it is *small* and it should be *quickly implemented*, without any complicated or extreme cases taken into account. The reason behind this is an arguable observation: if in the first run complex test cases are created, it is likely that the coverage of the metamodel is higher than with the simple



(a) (b)

Fig. 7: Metamodels for the case study

ones; however when the test set is improved later, the probability of redundant test case creation is much higher (by "redundant test cases" we mean test cases which have the exact same footprint or the footprint of the first is the part of the footprint of the second one – partial footprint redundancy is unavoidable and even desirable, cf. [16]). In our case study the first set of test cases was minimal too: it contained the test case presented on the upper part of Figure 2 on page 3, and the model containing only an empty `Process`.

Design and implement model transformation. In this group of activities we create the concrete transformation definition for the two metamodels. It contains three activities, from which two are optional but recommended.

Implement transformation. In this activity we create the concrete transformation, either in a transformation language or in a programming language. In our case study we used a plain Java class. For two simple transformation examples, see Figure 2 on page 3. As a first approach our transformation was as simple as possible, transforming `SequenceFlow`s into `Edge`s and `FlowNode`s into `Node`s (it was simple because for example each `Subprocess` had been transformed into a subgraph which was not connected to the rest of the graph's `Node`s).

(Optional) Specify and implement transformation contract. In this optional but strongly recommended activity a transformation contract is created, which consists of several transformation rules. The creation of the transformation contract is independent from the creation of the transformation, hence a contract can be implemented before or after the transformation, however as a rule of thumb it should be implemented before the transformation (we should create the specification *before* the implementation). The contract and its rules are used in the verification process of a transformation. The rules are defined as Java classes. In our case study our contract contained the following rules:

- (precondition) a not empty BPMN `Process` must have at least one `StartEvent` and at least one `EndEvent`;
- (precondition) a BPMN `SequenceFlow` must have both a source and a target or none of them;

- (precondition) a `Gateway` must be “reasonable”, i.e. it must be connected to at least three `SequenceFlows`;
- (postcondition) a not empty `SimpleGraph` must have at least one `Node` with label starting with “start event” and at least `Node` with label starting with “end event”;
- (postcondition) an `Edge` must have both a source and a target or none of them;
- (transformation) a `Gateway` must preserve the number of its incoming and outgoing edges during the transformation.

It should be noted that this contract is created for testing and demonstrating purposes, we did not intend to create a full nor an optimal contract.

(Optional) Specify features for the metamodels. Features are particular characteristics of a metamodel (see Section 2.2 for details). Creating them is optional but recommended, because a well defined feature set can help in improving the quality of a test suite. A good feature set includes the most common features for the possible models. In our case study we did not specify any features in the beginning.

Run test cases. After the first test case set is complete, the JUnit Test Suite should be executed.

Defects? and **Remove defects.** If there are any defects discovered, they have to be removed. A defect could be at many places:

- in a metamodel: if the metamodels were created during the development process, they can be flawed, e.g. in our case study we detected during the first test run that we forgot to set the default value for a string attribute to the empty string;
- in a test case or in a model: the test cases can be broken, they can be created in the wrong way, e.g. in our case study we could have created a test case where the edges would not be transformed to the expected output;
- in the transformation: the transformation can be implemented incorrectly.

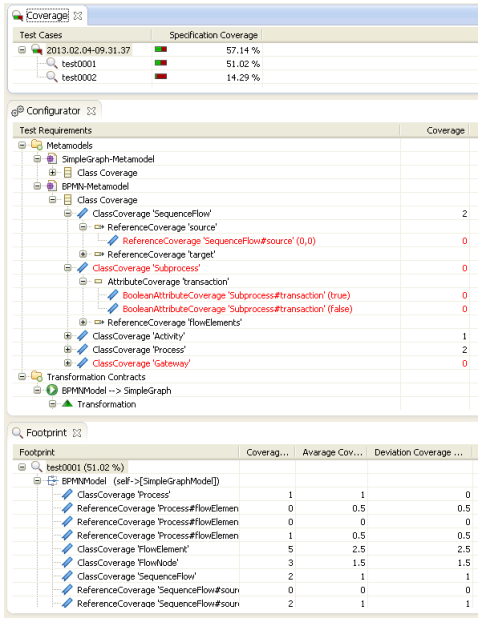


Fig. 8: Coverage report example

Produce coverage report. The coverage report is automatically generated and persisted by the tool after each successful execution of every test case.

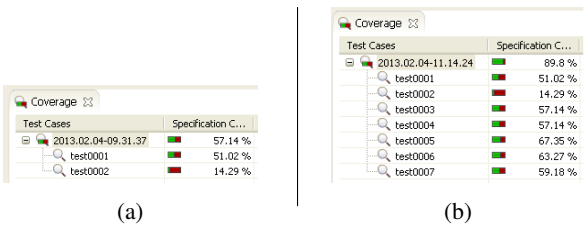
Analyze coverage report with Test Suite Analyzer . When there are no defects left, the coverage report should be analyzed. For an example report see Figure 8: on the top

the overall coverage regarding the test cases can be seen, in the middle the so far not covered test requirements are listed and in the bottom the complete footprint for one test case can be seen.

Inspect contract information and **Contract broken?** First, the fulfillment of the contract has to be checked. If the contract is broken, it means that one or more of its rules are neglected, however the cause of this is unknown. This has to be considered also as a defect (an "in logic" defect), and should be repaired. Our tool shows which of the rules are broken, but locating the concrete defect can be hard, depending on the complexity of the metamodels, the models, the rules and the transformation routine. In our case study on the first run the contract was not broken.

Compute missing test cases, **Inspect specification coverage information** and **Missing test cases?** Continuing the analysis of the coverage report, the next thing which is to be considered is the completeness of the test case set with regard to the defined coverage criteria. Our tool can show which parts of the metamodels, contracts and features are not covered with the test cases so far, regarding the defined equivalence partitions.

Increase set of test cases. From the missing coverage overview, new test cases can be derived. During the expansion of the test case set, the guidelines are similar as they were during the creation of the first set of test cases. The test cases should intersect each other as little as possible – however there is a legitimate need for some bigger tests, they should be added only after the desired coverage is achieved with smaller test cases. During this step first the coverage of the input model, than the coverage of the output model and after that the coverage of the contract and features should be addressed. In our case study, the overall coverage with the simple test cases was almost 60% (Figure 9a), which can be seen as small coverage, but it should be considered that this was a very simple test case set, still it covered more than a half of the metamodels, and contract. Regarding the coverage report analysis we created five more test cases, after that the coverage was almost 90% (Figure 9b). During the coverage report analysis we could determine, that this is our "semantic maximum" (the missing test cases included only the semantically invalid cases: `Node` without label, a `Process` with only one `FlowElement`, and `Gateway`s with negative type values).



(a) (b)

Fig. 9: Comparing two coverage reports

Compute redundant test cases, **Inspect redundancy information**, **Redundant test cases?** and **Reduce set of test cases.** After the coverage is sufficient, the reduction of the test case set should be considered. Two or more test cases are redundant, if they cover the same part of the transformation, and everything is covered in the same amount by them. In this case, all but one of them can be ignored, the coverage will not be affected. If the test case set provides the desired coverage and there are no redundant test cases in it, then we call it a reduced test case set. In our case study we followed the guidelines described throughout this section, and so we could create a test case set which was not redundant.

New functionality needed? If during the development cycle the need for one or more new functionalities emerges, they should be added only after the transformation is tested properly. Hence this should be the last step, after creating the reduced test case set.

Design and implement new functionality. If there is a need for a new functionality in the system, it should be designed and implemented in this activity. It should be done in small steps, creating a new test case if the functionality in question needs it. If the implementation of the new functionality needs a change in the system (e.g. modifying the transformation contract), the coverage analysis should be performed after every change too. This is important, because changes in the system could affect the coverage in unforeseen ways. In our case study we enhanced the transformation of `Subprocesses` so that the transformation produced a connected graph – we also implemented a feature, which described the nested subprocesses. After we finished the implementation, we executed the test suite and discovered that we have to adjust two of our test cases to the new transformation definition but did not need to add any new test cases, because the only new test requirement (the nested subprocess feature) was already covered by one of our test cases.

With this short description and case study we showed how the Test Suite Analyzer can be used in an agile test-driven development process of model transformations. However it should be noted that with such little metamodels a sufficient coverage could be relatively easily achieved, in real-life situations it is much harder.

5 Related Work

In the domain of software testing, several coverage criteria exist to determine the adequacy of test suites. McQuillan et al. [23] introduces a code-based coverage criterion for model transformations to derive test requirements from transformation definitions composed in ATL. In contrast to their work, we focus on specification-based coverage analysis as this facilitates the derivation of test cases for increasing coverage.

Specification-based coverage analysis has been studied by Andrews et al. [2]. They define coverage criteria for models composed in UML, including coverage criteria for UML class diagrams. Andrews et al. define three coverage criteria, which are called association-end multiplicity criterion, generalization criterion, and class attribute criterion. Fleurey et al. [11] adapt the approach of Andrews et al. for deriving test requirements from the input metamodel of a model transformation.

For determining the similarity between test cases, the domain of software profiling makes use of execution profiles for test cases. Leon et al. [21] as well as Yan et al. [30] detect similar test cases by defining metrics for computing the distance between execution profiles. Yan et al. use the euclidean distance between execution profiles for this computation. In contrast to using euclidean distance, we make use of the Manhattan distance, although both possibilities yield the same results for finding redundant test cases.

A test-driven approach for semantics specification of behavioral modeling languages based on graph transformations is described in [28]. Soltenborn and Engels evaluate the quality of model transformations using test cases that consist of example

behavioral models and their supposed traces of execution events. In [3], the approach is extended by a coverage analysis.

Another aspect of model-driven development of model transformations is the issue of automatic model generation. This issue has been studied for example by Bottier et al. [7]. They created a framework which derives input model based on predefined model fragments (which are based on specification based test requirements). More complex coverage topics were studied by Guerra [14] and Sen et al. [26]. Model generation based on code coverage test requirements was studied too (e.g. [25]). Integration of these solutions with our prototype is going to be taken into consideration.

In contrast to the existing approaches to coverage analysis (e.g. [12, 22]), we describe an integrated approach to coverage analysis realized in a prototypical tool. Regarding this, we explain how our tool can be used for developing high-quality model transformations. Concerning agile development of model transformations, we described an initial approach in [19] but we significantly extended this process in this paper and we also explained how the results of coverage analysis can be integrated into the development process.

6 Conclusion

Coverage analysis achieved by test cases for a model transformation is important for ensuring a high-quality test suite. In this paper, we have introduced a prototype tool which allows the software engineer to discover missing and redundant test cases in a test suite for a model transformation. It is based on a combined coverage approach which is independent from any specific model transformation language and computes a so-called footprint of a test case while being executed. This footprint allows a detailed analysis and is used for identification of missing and redundant test cases. We have explained how our tool can be used in agile test-driven development of model transformations and we have illustrated this using a model transformation from a simplified BPMN model to a simple graph. Future work includes the automatic generation of proposing missing test cases based on the result of coverage analysis in order to further improve the usability of the tool.

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
2. A. Andrews, R. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.
3. S. Arifulina, C. Soltenborn, and G. Engels. Coverage Criteria for Testing DMM Specifications. *ECEASST*, 47, 2012.
4. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *Proceedings of IMDT workshop in conjunction with ECMDA'06*, Bilbao, Spain, 2006.
5. E. Bauer and J. Küster. Combining Specification-Based and Code-Based Coverage for Model Transformation Chains. In *ICMT*, volume 6707 of *LNCS*, pages 78–92. Springer, 2011.
6. E. Bauer, J. Küster, and G. Engels. Test suite quality for model transformation chains. In *TOOLS 2011*, volume 6705 of *LNCS*, pages 3–19. Springer, 2011.

7. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based Test Generation for Model Transformations: An Algorithm and a Tool. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2006. IEEE Computer Society.
8. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE '02*, pages 267–270. IEEE Computer Society, 2002.
9. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
10. L. Seinturier E. Cariou, R. Marvie and L. Duchien. OCL for the Specification of Model Transformation Contracts. In *Workshop OCL and Model Driven Engineering, UML 2004*, 2004.
11. F. Fleurey, B. Baudry, P. Muller, and Y. Le Traon. Qualifying Input Test Data for Model Transformations. *Software and Systems Modeling*, 8(2):185–203, 2009.
12. P. Giner and V. Pelechano. Test-Driven Development of Model Transformations. In Andy Schrr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin / Heidelberg, 2009.
13. Object Management Group. *MOF 2.0 Query / Views / Transformations RFP*, December 2009.
14. Esther Guerra. Specification-Driven Test Generation for Model Transformations. In Zhenjiang Hu and Juan Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg, 2012.
15. M. Harder, B. Morse, and M.D. Ernst. Specification Coverage as a Measure of Test Suite Quality. *Septermber*, 25:452, 2001.
16. M. Heimdahl and D. George. Test-suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 176–185. IEEE Computer Society, 2004.
17. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
18. P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1999.
19. J. Küster, T. Gschwind, and O. Zimmermann. Incremental Development of Model Transformation Chains Using Automated Testing. In *MODELS 2009*, volume 5795 of *LNCS*, pages 733–747. Springer, 2009.
20. J. M. Küster, K. Ryndina, and R. Hauser. A Systematic Approach to Designing Model Transformations. Technical report, IBM Research, Research Report RZ 3621, July 2005.
21. D. Leon, A. Podgurski, and L. White. Multivariate Visualization in Observation-based Testing. In *ICSE '00*, pages 116–125, New York, NY, USA, 2000. ACM.
22. Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations. *Model-driven Software Development*, 1:219–236, 2005.
23. J. McQuillan and J. Power. White-Box Coverage Criteria for Model Transformations. *Model Transformation with ATL*, page 63, 2009.
24. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
25. J.-M. Mottu, S. Sen, M. Tisi, and J. Cabor. Static Analysis of Model Transformations for Effective Test Generation. In *ISSRE - 23rd IEEE International Symposium on Software Reliability Engineering*, 2012.
26. S. Sen, B. Baudry, and J.-M. Mottu. Automatic Model Generation Strategies for Model Transformation Testing. In Richard Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 148–164. Springer Berlin / Heidelberg, 2009.

27. M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä. Transformations Have to be Developed ReST Assured. In *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.
28. C. Soltenborn and G. Engels. Towards Test-Driven Semantics Specification. In *MODELS 2009*, volume 5795 of *LNCS*, pages 378–392. Springer, 2009.
29. B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, and Y. Berbers. Towards a Transformation Chain Modeling Language. In *SAMOS'06*, volume 4017 of *LNCS*, pages 39–48. Springer, 2006.
30. S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou. A Dynamic Test Cluster Sampling Strategy by Leveraging Execution Spectra Information. In *ICST '10*, pages 147–154. IEEE Computer Society, 2010.