RZ 3850 (# ZUR1307-046) 08/06/2013
Computer Science 30 pages

Research Report

A Method for Creating a Test Case Set to Achieve Maximum Specification Coverage in Model Transformation Testing

Dániel Kovács and Jochen M. Küster

IBM Research – Zurich
8803 Rüschlikon
Switzerland

A Method for Creating a Test Case Set to Achieve Maximum Specification Coverage in Model Transformation Testing

Dániel Kovács[1] and Jochen M. Küster[1]

IBM Research - Zurich, Säumerstr. 4
8803 Rüschlikon, Switzerland {dko,jku}@zurich.ibm.com

Abstract. When creating a model transformation in a test-driven approach the creation of a high-quality test suite is desirable. The quality definition usually incorporates the adequacy and the minimality of the test suite. When expanding a test suite, the overall quality of it should not be affected in a negative way. In order to calculate the quality of a test suite, coverage analysis can be used. If during the new test case creation the coverage analysis results are taken into consideration, lowering the negative impact on the quality of the test suite can be achieved more easily. However, creating models for the new test cases is not an easy task, especially not if the quality of the test suite has to be as high as possible. In this paper we present a method which can be used for creating models for new test cases. This method incorporates a periodical coverage analysis, so we can create new test cases while the low negative impact on the quality of the test suite is guaranteed.

1 Introduction

A crucial aspect when creating a model transformation in a test-driven approach is the quality of the test suite. This aspect can be measured in many ways, e.g. creating metrics on the language of the input models [4], creating a coverage analysis approach [5,6] or measuring redundancy of the test cases [27]. Creating test models for a model transformation is a time consuming task and needs a lot of effort if a software engineer wants to create a test suite with as high quality as possible.

If a software engineer wants to create an adequate test suite, he or she can define big test cases which cover as much of the input and output models as possible. However, creating many test cases is a slow process, and mistakes can be made at many places. Furthermore, if a test suite consists of big test cases the search for errors in the model transformation is complicated too. Creating a test suite which consists of small test models is also time and resource consuming. This problem can be addressed for example with model generation, but existing work on model generation (cf. [7,17,24,25]) shows that generating model instances based on a metamodel is a complicated and devious task.

In this paper, we define some constraints during the model generation process, with these the model instantiation can become a very useful tool for building high-quality test suites. We introduce our method for building test suites, which have a high quality regarding two aspects: the specification coverage achieved with the test cases is maximal (i.e. the test suite is adequate), and the test cases are as simple as possible. For guaranteeing the maximized specification coverage, the information which is gathered during the periodical coverage analysis [19] is used, while the simpleness of test cases is provided by the derivation process which yields the new test cases.

Model generation is usually done with the usage of grammars [26], which define the same models as the metamodels, or with the usage of model fragments [7], which are essentially building blocks of the possible models. We use a different approach and create a semi-automatic, iterative generation process for new test models. The process derives models for the new test cases from the models of the former test cases and the metamodels. After that they have to be verified by the software engineer for validness.

The paper is structured as follows. First we give some background concerning coverage analysis of model transformations, and describe a running example in Section 2. In Section 3 we describe algorithms which we will use for deriving new test cases. In Section 4 we explain how these algorithms can be used to create a simple test case set with maximum specification coverage. We discuss related work in Section 5 and conclude in Section 6.

2 Background

In this section we introduce a running example, which we will use in the following sections. After that we describe the theoretical foundations of the combined coverage approach [6], which we will use in our derivation algorithms, and provide a brief overview of the "Test Suite Analyzer for model transformations" tool, which we used to gather information about coverage while creating the test suite.

2.1 Running example

The simple case study, which we will use in the following sections as a running example to represent our method is a model transformation which transforms one input model into one output model. The metamodel for the input models of the transformation is a simplified subset of the BPMN metamodel [16] and can be seen in Figure 2-1.1. The metamodel for the output models describes a simple graph representation and can be seen in Figure 2-1.2.

In the case study we created a model transformation from this simplified BPMN metamodel to the simple graph representation in a straightforward way: we transformed `SequenceFlow`s into `Edge`s with an arbitrary weight, and `FlowNode`s into `Node`s with an adequate label.

2.2 The combined coverage approach

A model transformation transforms one or more input models into one ore more output models. The input and output models involved in the transformation are in the so-called implementation or concrete layer, while there is a so-called specification or definition layer above them.

The specification of the input and output models can be given in many ways e.g. with metamodels or with grammars. Although the expressive powers of these are essentially not equivalent [18], in this paper we focus on models defined in the context of a metamodel. A metamodel defines the relations of classes for a model, these relations can be e.g. inheritance, association, etc.

The specification of a model transformation is basically the transformation contract [9]. The notion of the transformation contract is based on the design-by-contract approach [23], applying it for the context of model transformations. A transformation contract contains three kinds of contract rules:

i) *precondition rules*, which are applied for the input models;
ii) *postcondition rules*, which are applied for the output models;
iii) *transformation rules*, which define the relations between input and output models.

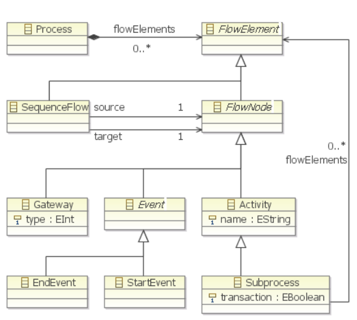
For measuring coverage the concept of test requirements is applied [2]. In the combined coverage approach [6] there are two kinds of test requirements: they can be *code based* or *specification based*.

A code based test requirement can for example be derived from every statement in the implementation. Such a test requirement is covered if that particular statement is executed. For code based coverage analysis numerous tools have been developed in the previous years (e.g. [8], [13], etc.).

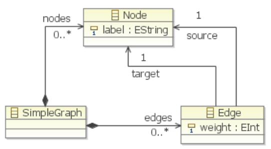
Specification based test requirements can be derived from different criteria. In our approach we use the *class coverage*, *attribute coverage* and *association coverage* criteria [11], and furthermore the *contract coverage* and *feature coverage* criteria [6].

In our running example we used a transformation contract which was created for testing and demonstrating purposes, containing the following rules:

- (precondition) a BPMN `Process` which contains at least one `FlowNode` must have at least one `StartEvent` and at least one `EndEvent`;
- (precondition) a `Gateway` must be “reasonable”, i.e. it must be connected to at least three `SequenceFlows`;
- (postcondition) a `SimpleGraph` which contains at least one `Node` must have at least one `Node` with label starting with “start event” and at least one `Node` with label starting with “end event”;

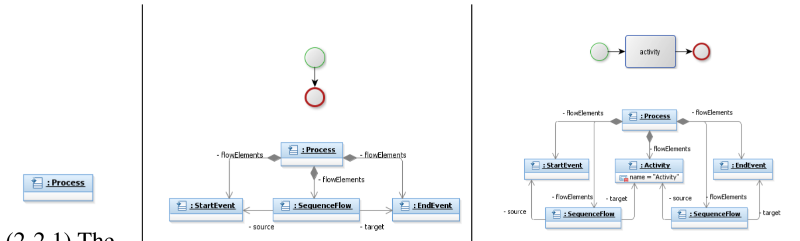


(2-1.1) A simplified subset of the BPMN metamodel



(2-1.2) A simple graph metamodel

Fig. 2-1: Metamodels for the simple case study



(2-2.1) The “empty” BPMN `Process`

(2-2.2) Test model for containing one `SequenceFlow`, with an abstract notation on top

(2-2.3) Test model for containing one `Activity`, with an abstract notation on top

Fig. 2-2: Example models for the running example

- (transformation) a `Gateway` must preserve the count of its incoming and outgoing edges during the transformation.

We created one feature for the input metamodel, which is the so-called nested subprocess feature (a model has the nested subprocess feature if it contains at least one subprocess which contains at least one subprocess, i.e. it can be defined with the OCL expression `self.flowElements->exists(x | x.oclIsTypeOf(Subprocess))` from the context of the class `Subprocess`).

All the defined test requirements are enumerated in Appendix B.

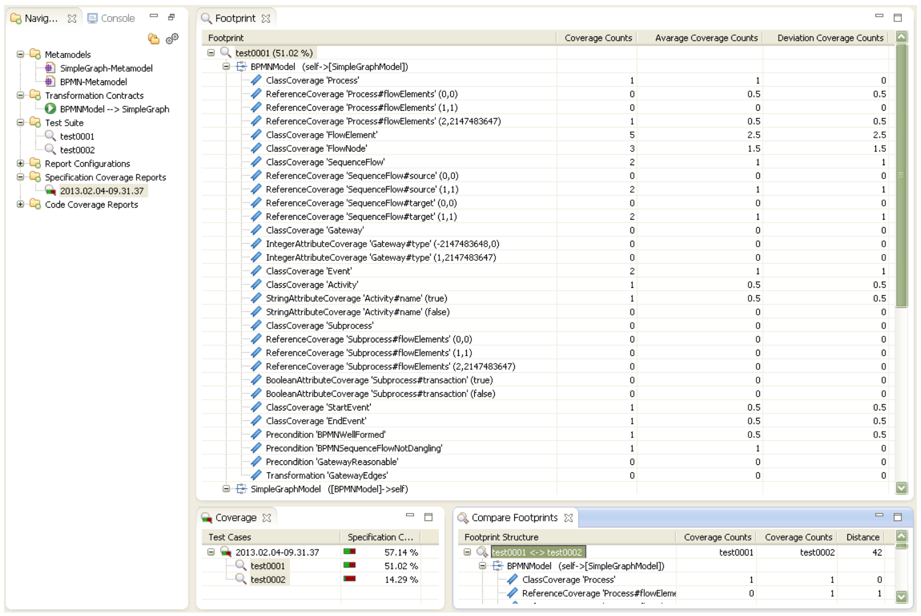
If we examine the example input models, shown in Figure 2-2, we can enumerate many missing (i.e. non-covered) test requirements: for example the class coverage requirement for `Gateway` or `Subprocess`, and the attribute and association test requirements which are connected with these classes.

2.3 Using the Test Suite Analyzer

In our recent work [19] we introduced a prototypical tool, named “Test Suite Analyzer for model transformations”, which can measure specification and code coverage of a JUnit Test Suite, created for testing a model transformation. In this section we give a brief overview of the tool, for details the reader is referred to [19].

The Test Suite Analyzer is a prototypical tool, which can create a so-called coverage report during the execution of a test suite. There are two phases in the usage of the tool: in the first phase the model transformation contract and the metamodel features are defined. These definition determine the contract and feature test requirements. After this, test requirements are defined for class, attribute and association coverage [11], which include defining equivalence partitions [2] for the latter two. In the second phase, along the execution of the test suite, the coverage report is generated and persisted.

The coverage report includes the code and specification coverage data of the test suite, regarding the metamodels and the model transformation. After that an analysis can be performed which can reveal some aspects of the test suite: its adequacy (the achieved coverage, regarding the defined test requirements) or its minimality (redundant test case detection). Uncovered test requirements can also be identified during the analysis.



Metamodels
- SimpleGraph-Metamodel
- BPMN-Metamodel

Transformation Contracts
- BPMNModel --> SimpleGraph

Test Suite
- test0001
- test0002

Report Configurations

Specification Coverage Reports
- 2013.02.04-09.31.37

Code Coverage Reports

Footprint	Coverage Counts	Avarage Coverage Counts	Deviation Coverage Counts
test0001 (51.02 %)			
BPMNModel (self->[SimpleGraphModel])			
ClassCoverage 'Process'	1	1	0
ReferenceCoverage 'Process#flowElements' (0,0)	0	0.5	0.5
ReferenceCoverage 'Process#flowElements' (1,1)	0	0	0
ReferenceCoverage 'Process#flowElements' (2,2147483647)	1	0.5	0.5
ClassCoverage 'FlowElement'	5	2.5	2.5
ClassCoverage 'FlowNode'	3	1.5	1.5
ClassCoverage 'SequenceFlow'	2	1	1
ReferenceCoverage 'SequenceFlow#source' (0,0)	0	0	0
ReferenceCoverage 'SequenceFlow#source' (1,1)	2	1	1
ReferenceCoverage 'SequenceFlow#target' (0,0)	0	0	0
ReferenceCoverage 'SequenceFlow#target' (1,1)	2	1	1
ClassCoverage 'Gateway'	0	0	0
IntegerAttributeCoverage 'Gateway#type' (-2147483648,0)	0	0	0
IntegerAttributeCoverage 'Gateway#type' (1,2147483647)	0	0	0
ClassCoverage 'Event'	2	1	1
ClassCoverage 'Activity'	1	0.5	0.5
StringAttributeCoverage 'Activity#name' (true)	1	0.5	0.5
StringAttributeCoverage 'Activity#name' (false)	0	0	0
ClassCoverage 'Subprocess'	0	0	0
ReferenceCoverage 'Subprocess#flowElements' (0,0)	0	0	0
ReferenceCoverage 'Subprocess#flowElements' (1,1)	0	0	0
ReferenceCoverage 'Subprocess#flowElements' (2,2147483647)	0	0	0
BooleanAttributeCoverage 'Subprocess#transaction' (true)	0	0	0
BooleanAttributeCoverage 'Subprocess#transaction' (false)	0	0	0
ClassCoverage 'StartEvent'	1	0.5	0.5
ClassCoverage 'EndEvent'	1	0.5	0.5
Precondition 'BPMNWellFormed'	1	0.5	0.5
Precondition 'BPMNSequenceFlowNotDangling'	1	1	0
Precondition 'GatewayReasonable'	0	0	0
Transformation 'GatewayEdges'	0	0	0
SimpleGraphModel ([BPMNModel]->self)			

Test Cases	Specification C...
2013.02.04-09.31.37	57.14 %
test0001	51.02 %
test0002	14.29 %

Footprint Structure	Coverage Counts	Coverage Counts	Distance
test0001 <-> test0002	test0001	test0002	42
BPMNModel (self->[SimpleGraphModel])			
ClassCoverage 'Process'	1	1	0
ReferenceCoverage 'Process#flowEleme	0	1	1

Fig. 2-3: Coverage report example, showed with the Test Suite Analyzer UI

In Figure 2-3 an example coverage report can be seen. In this example coverage report the test suite consists of two test cases for a model transformation which involves one input and one output model. On the left side an overview of the report can be seen, while on the right side the detailed footprint of the first test case is showed. The bottom shows an overview of the specification coverage achieved with the test cases and the comparison of two test cases. As an example, from this single view it can be seen that the class `Subprocess` or any of its attributes and associations are not covered with the test suite.

3 Deriving New Test Cases

In this section we discuss, how new test cases can be derived from existing ones, if the data containing the specification coverage of the current test suite is given. We provide here algorithms for deriving input models for the new test cases, expected output models can be created afterwards with using the model transformation on the derived input model. The algorithms can be used for deriving new output models too (based on the output models of the former test cases), but after this the input models have to be created manually, because the inverse of the transformation usually can not be done programmatically.

First, we introduce the notion of coverage count of test requirements, which is based on the work of Bauer et al. [6], here we use a variant of their definition and build new notions on it. We create the definitions for model transformations which include one input and one output model, but they can be easily adjusted to model transformations which involve more models.

We provide examples for every definition and algorithm, based on the test case set given in the previous Section, in Figure 2-2.

Definition 1 (Coverage count of test requirements [6]). Let *tc* denote an arbitrary test case from a test case set, in which the derived specification based test requirements are denoted with $tr_1, tr_2, \dots tr_n$. In this case $\forall\, k \in [1..n] : \texttt{CC}(tr_k)$ denotes the non-negative integer which states, how many times that particular test requirement is covered by the *tc* test case.

For example, two coverage counts from the given test case set:
$\texttt{CC}(tr_{ClassCoverage'StartEvent'}) = 2$;
$\texttt{CC}(tr_{ClassCoverage'Event'}) = 4$;
$\texttt{CC}(tr_{ClassCoverage'Gateway'}) = 0$ ◃

Definition 2 (Base class of test requirement). We call a metamodel class the base class of a specification test requirement, if that particular test requirement is derived from that class or an attribute of that class. We denote the base class of test requirement *tr* with $\textsf{BASE}(tr)$.

For example: $\textsf{BASE}(tr_{ClassCoverage'Gateway'}) =$ `Gateway`;
$\textsf{BASE}(tr_{AttributeCoverage'Subprocess\#transaction'\{true\}}) =$ `Subprocess` ◃

Definition 3 (Sub-hierarchy of metamodel). A part of a metamodel is a sub-hierarchy if it contains only a class and all classes which can be (directly or indirectly) derived from that class. We denote a sub-hierarchy of a metamodel as $\texttt{SUB}(metamodel, class)$.

For example the sub-hierarchy SUB(BPMN Metamodel, `FlowNode`) contains the following classes: `FlowNode`, `Gateway`, `Event`, `StartEvent`, `EndEvent`, `Activity`, `Subprocess` ◃

3.1 Deriving a test case for class coverage

If regarding the given test case set a test requirement derived from a metamodel class is not covered, then we can try to derive a new test case from the former test cases. For this, we define an ordering on the classes of a metamodel, and after that we provide an algorithm for deriving a model for a new test case.

Definition 4 (Root class). We call a distinguished metamodel class the root class if it is the top level container of every other class instances in a model.

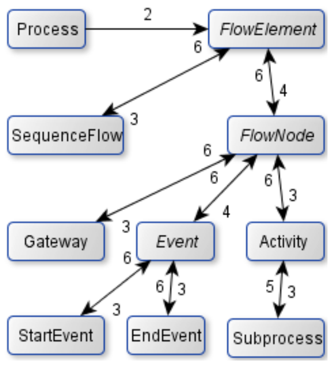
For example the root class of the BPMN Metamodel is the `Process` class. ◃

Definition 5 (Class ordering). Let $M_{part} = (M, C, R)$ be a partial metamodel, where C is the set of the classes from the metamodel M and R is the set of the composition and inheritance relations in the metamodel M. Let $G = (V, E, l, w)$ be a directed, labeled and weighted graph, the so-called ordering graph, where:

- $|V| = |C|$;
- $l : V \to \{\text{class names in } C\}$ is the labeling function, and it is a bijection (i.e. $v \in V$: if node v represents the class $c \in C$, then $l(v)$ is the name of class c);
- $E = \{r \mid r \in R \wedge r \text{ is composition}\} \cup \{r \mid r \in R \wedge r \text{ is inheritance}\} \cup \{\text{inverse of } r \mid r \in R \wedge r \text{ is inheritance}\}$;

- $w : E \to \{1;\ 2;\ 3;\ 4;\ 5;\ 6\}$ is the weight function, following the rules for $e \in E$:

$$w(e) = \begin{cases} 1 \text{ , if and only if it is derived from a composition relation and the} \\ \quad \text{target of the edge is representing a concrete class} \\ 2 \text{ , if and only if it is derived from a composition relation and the} \\ \quad \text{target of the edge is representing an abstract class} \\ 3 \text{ , if and only if it is derived from an inheritance relation, the direction} \\ \quad \text{of the edge is the opposite as in the metamodel and the target of the} \\ \quad \text{edge is representing a concrete class} \\ 4 \text{ , if and only if it is derived from an inheritance relation, the direction} \\ \quad \text{of the edge is the opposite as in the metamodel and the target of the} \\ \quad \text{edge is representing an abstract class} \\ 5 \text{ , if and only if it is derived from an inheritance relation, the direction} \\ \quad \text{of the edge is the same as in the metamodel and the target of the} \\ \quad \text{edge is representing a concrete class} \\ 6 \text{ , if and only if it is derived from an inheritance relation, the direction} \\ \quad \text{of the edge is the same as in the metamodel and the target of the} \\ \quad \text{edge is representing an abstract class} \end{cases}$$



h

Fig. 3-1: The graph derived from the BPMN metamodel

The ordering of the classes is defined by their minimum distance from the node of the root class in the ordering graph, which can be e.g. computed with Dijkstra's algorithm on the ordering graph, with the node representing the root class as a start node, and given the rule that if there are more possible next nodes, we order them into alphabetical order. Every node which is not reachable from the root node is added to the end of the ordering in alphabetical order (note: the non-connected classes are usually data or enumeration classes).

For example, the graph created from the BPMN metamodel (Figure 2-1.1) can be seen in Figure 3-1, hence the order of the classes is: `Process` < `FlowElement` < `SequenceFlow` < `FlowNode` < `Activity` < `Gateway` < `Event` < `Subprocess` < `EndEvent` < `StartEvent`.

We define the distance of two classes c_1 and c_2, noted with $d(c_1;\ c_2)$ similarly with Dijkstra's algorithm, taking the node representing c_1 as the start node, and calculating the distance from it to c_2 (note that this distance notion is not symmetric, e.g. $d(\texttt{Process}, \texttt{FlowElement}) = 2$, while $d(\texttt{FlowElement}, \texttt{Process}) = \infty$).

We say that the class c_1 is smaller than c_2, if it c_1 is earlier in the ordered sequence of classes than c_2 (e.g. `Activity` is smaller than `Event`). $\triangleleft$

Given this ordering, we can derive a new test case for a non-covered class from the available test cases with Algorithm 1.

In Algorithm 1 we assume that the test requirements derived from metamodel classes are ordered regarding the defined class ordering (Defintion 5). The phrase "closest BASE (`tr`) instance" at line 6 means a class instance whose type is either BASE (`tr`) or if it is abstract then a derived concrete class from it, which is the closest-smallest class to the non-covered class in the defined class ordering (where closest means their distance is minimal and smallest means the smallest of these classes regarding the ordering). A substitution is possible (line 5) if the new class instance can be a part of the model, regarding the composition relations in the metamodel. A class c_1 is container of an other class c_2 (line 12), if there is a directed edge from c_1 to c_2 in the ordering graph, with the weight of 1 or 2.

The subroutine described in Algorithm 2 creates a new class instance. The new class instance is the next class, from the context of a parameter class and regarding the defined class ordering. The "new valid instance" phrase means that if there are some necessary containments, then those classes have to be instantiated too. Note that because of the conditions in both parts of Algorithm 1, before invoking this subroutine, it can always return a new class instance.

An example for the first derivation strategy (lines 2–10) is a derivation from the third example model (Figure 2-2.3) for the class `Gateway`. In this case the algorithm finds the coverage count $\texttt{CC}(tr_{ClassCoverage'FlowElement'}) = 5$, and because $\texttt{Gateway} \in \textsc{sub}(\text{BPMN Metamodel}, \texttt{FlowElement})$, it copies this model. After this it checks whether a gateway can be the part of the model regarding the containment relations, and because it can, it substitutes the closest-smallest `FlowElement` instance, which is the `Activity` instance. Hence the derived model is the same as the base one, except that the `Activity` is substituted with a `Gateway`.

An example for the second derivation strategy (lines 11–17) is a derivation from the first example model (Figure 2-2.1) for the class `Subprocess`. In this case the algorithm finds the coverage count $\texttt{CC}(tr_{ClassCoverage'Process'}) = 1$, and because `Process` is a container for `Subprocess`, it copies this model. After this it adds a new association from the `Process` instance to a new `Subprocess` instance. Hence the derived model contains a `Process` and a `Subprocess` instance with a containment relation between them.

3.2 Deriving a test case for attribute coverage

If regarding the given test case set a test requirement derived from an attribute from a class from the metamodel is not covered, then we can try to derive a new test case from the former test cases. For this, we define an ordering on the attributes of a metamodel, and after that we provide an algorithm for deriving an input model for a new test case.

```
Input   : current test case set; not covered class class; metamodel M
Output: new input test model result, if it is derivable
1  foreach test case tc from current test case set do
2      foreach test requirement tr from tc do
3          if cc(tr) > 0 and class ∈ SUB (M, BASE (tr)) then
4              result ← copy of input model of tc;
5              if substitution is possible then
6                  substitute closest BASE(tr) instance with CreateClassInstance (M,
                   class) in result;
7                  return result;
8              end
9          end
10     end
11     foreach test requirement tr from tc do
12         if cc(tr) > 0 and BASE (tr) is a container of class then
13             result ← copy of input model of tc;
14             add association to CreateClassInstance (M, class) from BASE(tr) in
               result;
15             return result;
16         end
17     end
18 end
19 return can not derive new test case input model;
```

Algorithm 1: Derive a new test case for new class coverage

```
Input   : metamodel M; class class
Output: a new class instance
1  foreach class currentClass from SUB(M, class) do
2      if currentClass ≥ class and currentClass is not abstract then
3          return new valid instance of currentClass;
4      end
5  end
6  return can not create new class instance;
```

Algorithm 2: Create class instance function

Definition 6 (Attribute ordering). Let C denote the set of classes in a metamodel M. Let $c \in C : A_c$ denote the set of attributes of class c. We define an ordering on A_c in the following way for $attr_1, attr_2 \in A_c$:

$$attr_1 < attr_2 \overset{def.}{\Longleftrightarrow} \begin{cases} \text{typename of } attr_1 < \text{ typename of } attr_2 \\ \quad \text{, if typename of } attr_1 \neq \text{ typename of } attr_2 \\ \text{name of } attr_1 < \text{ name of } attr_2 \\ \quad \text{, if typename of } attr_1 = \text{ typename of } attr_2 \end{cases}$$

The equivalence partitions are ordered in their natural ordering.
If $c_1, c_2 \in C : c_1 < c_2$ (regarding Definition 5) then we say that $A_{c_1} < A_{c_2}$. Thus we defined an ordering on the attributes of a metamodel M.
We denote the class, containing attribute $attr$ with $\text{CONT}(attr)$ (i.e. $attr \in A_c \Leftrightarrow \text{CONT}(attr) = c$). $\triangleleft$

For example, the attributes of the BPMN metamodel (Figure 2-1.1) are ordered as: `Gateway.type`$\{(-\infty;\ 0]\}$ < `Gateway.type`$\{[1;\ \infty)\}$ < `Subprocess.transaction`$\{false\}$ < `Subprocess.transaction`$\{true\}$.

Given this ordering, we can derive a new test case for a not covered attribute from the available test cases with Algorithm 3. In Algorithm 3 we assume that the test requirements derived from metamodel attributes are ordered regarding the defined attribute ordering (Defintion 6).

If for example we have a model which contains a `Gateway` instance, with a value for the attribute `type` from the first equivalence partition, we can derive a new model based on this, adjusting the value of the attribute to fit into the second equivalence partition.

3.3 Deriving a test case for association coverage

If regarding the given test case set a test requirement derived from an association in a metamodel is not covered, then we can try to derive a new test case from the former test cases. For this, we define an ordering on the associations of a metamodel, and after that we provide an algorithm for deriving an input model for a new test case.

Definition 7 (Association ordering). Let C denote the set of classes in a metamodel M. Let $c \in C : AS_c$ denote the set of associations of class c. We define an ordering on AS_c in the following way for $assoc_1, assoc_2 \in AS_c$:

$$assoc_1 < assoc_2 \overset{def.}{\Longleftrightarrow} \text{name of } assoc_1 < \text{ name of } assoc_2$$

The equivalence partitions are ordered in their natural ordering.
If $c_1, c_2 \in C : c_1 < c_2$ (regarding Definition 5) then we say that $AS_{c_1} < AS_{c_2}$. $\triangleleft$

For example, the associations of the BPMN metamodel (Figure 2-1.1) are ordered as following:
`Process.flowElements`$\{0\}$ < `Process.flowElements`$\{[2;\ \infty)\}$ <
< `SequenceFlow.source`$\{1\}$ < `SequenceFlow.target`$\{1\}$ <
< `Subprocess.flowElements`$\{0\}$ < `Subprocess.flowElements`$\{[2;\ \infty)\}$.

We denote the class, which contains the associations *assoc* with FROM(*assoc*), and the class, which is associated with it with TO(*assoc*).

Given this ordering, we can derive a new test case for a not covered association from the available test cases with Algorithm 4. In Algorithm 4 we assume that the test requirements derived from metamodel associations are ordered regarding the defined association ordering (Defintion 7).

At line 6 we can simply modify the multiplicity of the association, because we have at least one. If the missing equivalence partition contains less associations we remove the necessary amount, if it contains more, we create as much copies as needed. Note that the subroutine described in Algorithm 2 can always return a new class instance, when invoked from Algorithm 4.

For example if we have a model which contains a `Subprocess` instance with zero `FlowElements` in it, then we create a new model with an association from the `Subprocess` instance to two new `SequenceFlow` instances (for the equivalence partition $\{[2; \infty)\}$).

On the other hand if we have a model which contains a `Subprocess` with at least two `FlowElements` in it, then we can create a new model with zero `FlowElements` in the `Subprocess` (for the equivalence partition $\{0\}$).

3.4 The limits of test case derivation

The contract rules and the features are defined in the Test Suite Analyzer for model transformations as Java classes, hence we can not create such simple algorithms for deriving test cases as in the case of class, attribute or association based test requirements. However, the new test cases for not covered contract rules and features should be made regarding similar guidelines which are noticeable in the derivation processes for other test requirements: the new test cases should be as small as possible and cover as few, yet uncovered test requirement as possible. As a rule of thumb the contract rule and feature test requirements should be covered in alphabetical order.

For code coverage, new test cases can not be derived, but given the code coverage report and the semantics of the model transformation, we can create new test cases, which use formerly unused parts of the transformation routine (or simplify the routine if it contains unnecessary parts).

3.5 Determining optimal derived test model

The formerly described derivation algorithms create a model for a new test case based on an arbitrary chosen former model, which from a new model for covering that particular test requirement can be derived. Usually there are more than one former test cases which can be the base of the derivation, and there can be big differences in the derived test models. In this section we provide a notion for the optimal derived test model, which is based on our empirical results.

For this, we define the test case derivation in a more formalized way with the notion of footprints. We use the footprint definition of Bauer et al. [6], and derive new notions from it. With the usage of ordered footprints, in theory, the derived test models with the algorithms described in Sections 3.1, 3.2 and 3.3 can be generated automatically.

```
Input  : current test case set; not covered attribute attr; missing equivalence partition
         eqp of attr
Output: new input test model result, if it is derivable
1  foreach test case tc from current test case set do
2      foreach test requirement tr from tc do
3          if cc(tr) > 0 and BASE(tr) = CONT(attr) then
4              result ← copy input model of tc;
5              change value of attr in result to fit into eqp;
6              return result;
7          end
8      end
9  end
10 return can not derive new test case input model;
```

Algorithm 3: Derive a new test case for new attribute coverage

```
Input  : current test case set; missing association assoc; missing equivalence
         partition eqp of assoc; metamodel M
Output: new input test model result, if it is derivable
1  foreach test case tc from current test case set do
2      foreach test requirement tr from tc do
3          if cc(tr) > 0 and BASE(tr) = FROM(assoc) then
4              result ← copy input model of tc;
5              if association multiplicity in FROM(assoc) ≠ 0 then
6                  modify cardinality of assoc in result to fit into eqp;
7              else
8                  add new association from FROM(assoc) in result to
                   CreateClassInstance (M, TO(assoc)) in the needed amount;
9              end
10             return result;
11         end
12     end
13 end
14 return can not derive new test case input model;
```

Algorithm 4: Derive a new test case for new association coverage

Definition 8 (Footprint [6]). Let *tc* denote an arbitrary test case from a test case set, in which the derived specification based test requirements are denoted with $tr_1, tr_2, \dots tr_n$. In this case the footprint of *tc* is defined as the following list:
$fp(tc) :=< \mathtt{cc}(tr_1), \mathtt{cc}(tr_2), \dots, \mathtt{cc}(tr_n) >$ ◁

Definition 9 (Restricted footprint, ordered restricted footprint). Let *tc* denote an arbitrary test case from a test case set, its footprint is $fp(tc)$. Furthermore, let $\textsc{t}$ denote a test requirement type. The possible test requirement types are:

$$\textsc{t} = \begin{cases} \textsc{class}_{\textsc{inp}} & \text{, if and only if it is derived from a class} \\ & \text{from the input metamodel} \\ \textsc{class}_{\textsc{out}} & \text{, if and only if it is derived from a class} \\ & \text{from the output metamodel} \\ \textsc{attr}_{\textsc{inp}} & \text{, if and only if it is derived from an attribute} \\ & \text{from the input metamodel} \\ \textsc{attr}_{\textsc{out}} & \text{, if and only if it is derived from an attribute} \\ & \text{from the output metamodel} \\ \textsc{assoc}_{\textsc{inp}} & \text{, if and only if it is derived from an association} \\ & \text{from the input metamodel} \\ \textsc{assoc}_{\textsc{out}} & \text{, if and only if it is derived from an association} \\ & \text{from the output metamodel} \\ \textsc{contract} & \text{, if and only if it is derived from the contract} \\ \textsc{feat}_{\textsc{inp}} & \text{, if and only if it is derived from a feature} \\ & \text{from the input metamodel} \\ \textsc{feat}_{\textsc{out}} & \text{, if and only if it is derived from a feature} \\ & \text{from the output metamodel} \end{cases}$$

We say that $fp(tc)$ is restricted to the test requirement type $\textsc{t}$ if it contains only the test requirements with type $\textsc{t}$. We denote this kind of footprint with $fp|_{\textsc{t}}(tc)$.

We say that $fp|_{\textsc{t}}(tc)$ is ordered, if it is ordered regarding the corresponding ordering (e.g. if $\textsc{t}$ is $\textsc{class}_{\textsc{inp}}$ or $\textsc{class}_{\textsc{out}}$ then we use Definition 5, etc.). We denote ordered restricted footprints with $ofp|_{\textsc{t}}(tc)$. ◁

Definition 10 (Ordered footprint). For a test case *tc* the ordered footprint, denoted with $ofp(tc)$ is the sequence of ordered restricted footprints in the following way:
$ofp(tc) :=< ofp|_{\textsc{class}_{\textsc{inp}}}(tc);\ ofp|_{\textsc{attr}_{\textsc{inp}}}(tc);\ ofp|_{\textsc{assoc}_{\textsc{inp}}}(tc);$
$ofp|_{\textsc{class}_{\textsc{out}}}(tc);\ ofp|_{\textsc{attr}_{\textsc{out}}}(tc);\ ofp|_{\textsc{assoc}_{\textsc{out}}}(tc);$
$ofp|_{\textsc{contract}}(tc);\ ofp|_{\textsc{feat}_{\textsc{inp}}}(tc);\ ofp|_{\textsc{feat}_{\textsc{out}}}(tc) >$ ◁

For every test case, we determine whether it can be a base of derivation or not. If it can be a base, then we derive the new input model. After we have derived all the possible new test cases we calculate a sub-footprint for each base and new test case. This sub-footprint can be defined as $sfp_{inp} :=< ofp|_{\textsc{class}_{\textsc{inp}}}(tc);\ ofp|_{\textsc{attr}_{\textsc{inp}}}(tc);\ ofp|_{\textsc{assoc}_{\textsc{inp}}}(tc) >$ (naturally if we are creating a new test case for the output model, the sub-footprint sfp_{out}, which consists of the ordered restricted footprints of the output model in the same order can be used).

We denote the length of a footprint as follows: $len(fp) := \|fp\|_2$ (we are using the euclidean length definition, which is defined, hence essentially $fp \in \mathbb{N}^n$).

We denote the difference of the derived and the base models as follows: $\mathit{diff} := \mathit{len}(\mathit{fp}_{\mathit{derived}}) - \mathit{len}(\mathit{fp}_{\mathit{base}})$.

We calculate then a so-called difference ratio for every derived-base footprint pair as follows:

$$\mathit{ratio}(\mathit{fp}_{\mathit{derived}}; \mathit{fp}_{\mathit{base}}) := \begin{cases} \infty & \text{, if } \mathit{diff} < 0 \\ 0 & \text{, if } \mathit{diff} = 0 \\ e^{\mathit{diff}} \cdot \ln \mathit{len}(\mathit{fp}_{\mathit{derived}}) & \text{, if } \mathit{diff} > 0 \end{cases}$$

The derived test model with the smallest ratio is the so-called optimal derived test model. If there is more than one, then we choose the one with the shortest base *sfp* of them (if there are more of those, then we choose the first in the order of the test cases).

4 Method for Building a Test Case Set

In this section we introduce a method for building a test case set for a model transformation, based on the data we gather during the periodical coverage analysis. We use the method in the context of a test-driven development approach introduced in our recent work [19], hence the method presented here can be easily integrated into it. We also use a similar case study we used in [19], which involves a model transformation from a simplified subset of the BPMN to a simple graph, these metamodels can be seen in Figure 2-1 on page 3.

First we introduce the method by using it to build up a test case set from scratch. In the second part we briefly discuss the usage of the method with a more realistic case study. We discuss the method for model transformations which include one input and one output model, but it can be easily adjusted to model transformations which involve more models.

4.1 Detailed description of method

In this subsection we describe our method in detail. The description of the two case studies we carried through can be found in Appendix A.

An overview of our method can be seen on Figure 4-1. The subprocess “Maximize ‘input model class’ coverage”, can be seen in Figure 4-2, the other subprocesses (“Maximize … coverage”) are similar to this, they differ only in their “subject”.

When starting to build a test case set from scratch, we start with an initial test case which involves the “empty model”, meaning that the input model is containing only the single instance of the root class (Definition 4). We assume that this is a valid input model and can be transformed into a valid (also “empty”) output model.

After this, we generate an initial coverage report, this activity involves executing the test suite which is composed from the test cases in the test case set. From the coverage report it should be determined what is the next missing test requirement, regarding the corresponding test requirement ordering (Definitions 5, 6 and 7). Using the corresponding derivation algorithm (desribed in Section 3), we can now derive a new test model. The validity of this new model should be verified by the software engineer. If the model is valid, then we can derive the next model, if it is invalid, it has to be “repaired” and a new coverage report should be generated afterwards.

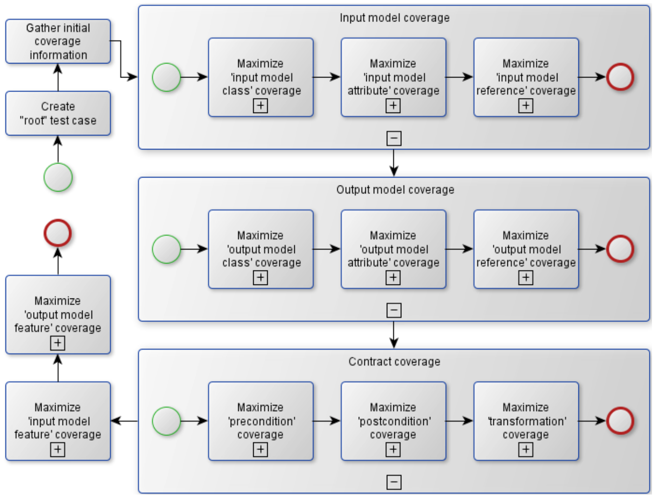

Gather initial coverage information
Create "root" test case
Input model coverage
Maximize 'input model class' coverage
Maximize 'input model attribute' coverage
Maximize 'input model reference' coverage
Output model coverage
Maximize 'output model class' coverage
Maximize 'output model attribute' coverage
Maximize 'output model reference' coverage
Contract coverage
Maximize 'precondition' coverage
Maximize 'postcondition' coverage
Maximize 'transformation' coverage
Maximize 'input model feature' coverage
Maximize 'output model feature' coverage

Fig. 4-1: Overview flow of expanding a test case set

4.2 Summary of the empirical results

With our method we can generate a test case set which provides maximum specification coverage. This generated test case set does not include any redundant test cases, because every new test case has at least one new component which is not covered with the former test cases. Our method can be seen as a generating a base in the linear state which is defined over the footprints. During this generation process we always add at a new model which has at least one orthogonal dimension to the test case set (orthogonal to the subspace defined with the footprints of the former test cases).

We do not state that the generated test case set is “optimal” or “minimal”, but it is composed of as simple and as small test cases as possible. It is build up in a well defined way and with some comments the newly covered specification test requirement can be tracked during the process, which improves the quality of the test suite even further.

In our method we start with one initial test case, containing only the so-called “empty” models. After this we incrementally build up our test case set, regarding the ordering defined for footprints (Definition 10). It can be seen that if we are strictly following this ordering, then starting from the initial test case, using the derivation algorithms (Algorithm 1, 3 and 4) we can always derive a new test case, but after this the validity of this new test case has to be verified by the software engineer.

It should be noted, that if an initial test case set is given, our method can be used to increase the specification coverage of the test suite. In this case the initial test case set is not composed only of the “empty” model, but many test cases, however the flow of our method is unaffected by this fact. The test case set should be ordered by some definition of complexity, e.g. it can be ordered in the increasing order of the euclidean length of the footprints (note that this is an arbitrary ordering and if we build up a test case set from scratch it is not ordered in this way).

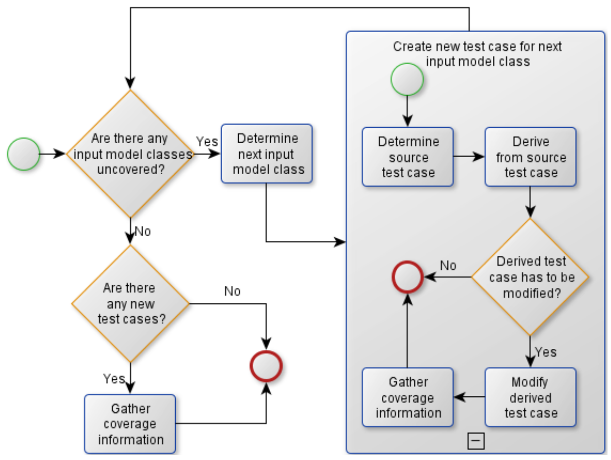

Are there any input model classes uncovered?
Yes
No
Determine next input model class
Are there any new test cases?
No
Yes
Gather coverage information
Create new test case for next input model class
Determine source test case
Derive from source test case
Derived test case has to be modified?
No
Yes
Modify derived test case
Gather coverage information

Fig. 4-2: Flow of creating new test cases in subprocess "Maximize 'input model class' coverage"

5 Related Work

The quality of a test suite can be measured in many ways. McQuillan et al. [22] introduce a code-based coverage criterion which define test requirements based on ATL. In contrast to their work we focus on specification based coverage. Andrews et al. [3] define coverage criteria for models composed in UML, while Fleurey et al. [11] adapt their approach to derive test requirement from the input metamodels of a model transformation. Bauer et al. [6] extend this approach further by introducing the so-called combined-coverage approach, which includes test requirements for the output models, the transformation contract and metamodel features too. We use this combined-coverage approach in our model derivation process.

For generating new instances of a metamodel usually string grammars like in the work of Alanen and Porres [1], or graph grammars like in the work of Ehrig et al. [10] are used. Taentzer [26] enhances this method after formalizing the metamodels as type graphs. When the goal is the generation of test models, usually model fragments are defined (as for example in [7,20,25]) or the generation process is defined over a language (as for example in [12, 17, 24]). In contrast to those approaches, we propose an iterative method for building a test case set, based on coverage information of the former iterations.

For determining the similarity between test cases usually a metric is defined as in the work of Yan et al. [27]. They use the euclidean distance between execution profiles for measuring the similarity, while we make use of the euclidean distance and the euclidean length of test case footprints for detecting an optimal extension for former test cases.

6 Conclusion

A good metric for measuring the quality of a test suite can be the specification and code based coverage which is achieved by the test cases in the test suite. In this paper we have provided a method for creating a test suite which generates maximum specification coverage at the end. This method is a direct enhancement to the test-driven approach we introduced in [19]. It is based on a so-called footprint of test cases which is a vectorial representation of the coverage achieved by a test case. This footprint allows a detailed analysis and is used in our method for the identification of missing test requirements, and from this information new test cases can be derived. In our semi-automatic approach we can derive test cases which are valid regarding the metamodel, but can be invalid regarding the full definition of the language. As described by Kleppe [18], usually a metamodel is not a full definition of a language, hence our future work includes incorporating other parts of the language definition (like semantic constraints described in OCL) into our method besides generating of derived test cases programatically. Our future work also includes the reduction of a complex test suite based on the analysis of the footprints while keeping the specification coverage on the same level.

References

1. M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical report, TUCS Turku Center for Computer Science, March 2003.
2. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.
3. 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.
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. Cobertura. http://cobertura.sourceforge.net/. Visited on 11/2/2010.
9. 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.
10. K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 2009. To appear.
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. Carlos Alberto González Pérez and Jordi Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In *ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems MODELS 2012*, 2012.
13. Google. CodePro Analytix 7.0. http://marketplace.eclipse.org/content/codepro-analytix. Visited on 14/3/2013.
14. Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4*, 2001.

15. Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructrue, Version 2.2*, 2009.
16. Object Management Group. *Business Process Model and Notation (BPMN) 2.0*, 2011.
17. 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.
18. A. G. Kleppe. A Language Description is More than a Metamodel. In *Fourth International Workshop on Software Language Engineering, Nashville, USA*, October 2007.
19. J. M. Küster, D. Kovács, E. Bauer, and C. Gerth. Integrating coverage analysis into test-driven development of model transformations. In *Technical Report*, volume RZ3846. IBM, 2013.
20. Maher Lamari. Towards An Automated Test Generation for the Verification of Model Transformations. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 998–1005, New York, NY, USA, 2007. ACM.
21. Rose Louis M., Kolovos Dimitrios S., Paige Richard F., and Polack Fiona A. C. Model Migration Case for TTC 2010, 2010.
22. J. McQuillan and J. Power. White-Box Coverage Criteria for Model Transformations. *Model Transformation with ATL*, page 63, 2009.
23. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
24. 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.
25. Sagar Sen, Benoit Baudry, and Jean-Marie 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.
26. Gabriele Taentzer. Instance Generation from Type Graphs with Arbitrary Multiplicities. *ECEASST*, 2012.
27. 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.

A Case studies

A.1 A simple case study

The input model from the case study for the initial test case can be seen in Appendix D in Figure D-1.1.

After creating `test0001` (containing the "empty" model), we generate an initial coverage report, this activity involves executing the test suite which is composed from the test cases in the test case set. From the coverage report it should be determined what is the next missing test requirement, regarding the corresponding test requirement ordering (Definitions 5, 6 and 7). In our case study the first missing test requirement is derived from the class of the input metamodel, called `SequenceFlow`. Using the second part of Algorithm 1, we derive a new test case from the initial one, as seen in Figure D-1.2. Taken the BPMN metamodel (and the BPMN 2.0 specification [16]) into account this model is invalid, because a `SequenceFlow` must have a `source` and a `target`, so we modify it to the model seen in Figure D-1.3, as this is the least complex model which contains a `SequenceFlow`. Because we had to modify a derived model, we gather the coverage information again.

The next missing test requirement is derived from the metamodel class `Activity`. We can derive a new test case from `test0001`, similarly as deriving `test0002` (we do not describe in detail, how we determine the optimal base model, the method is described in Section 3.5, the ratios for different derivations in our case study can be seen in Appendix C). This new input model can be seen in Figure D-1.4. This test case is also invalid, thus we have to modify it. We simply created the input model seen in Figure D-1.5 (In the following the models of test cases are going to be denoted in abstract syntax in the figures, because the concrete syntax is becoming too complex). Because we had to modify a derived model, we gather the coverage information again.

The next class in the input metamodel which is not covered yet is the `Gateway`. We derive the new test case from `test0003` (using the first part of Algorithm 1), as seen in Figure D-1.6. This derived model is invalid, because regarding our transformation contract, a `Gateway` must have at least three connected `SequenceFlow`s, hence we modify the derived model to a valid one, e.g. as it can be seen in Figure D-1.7. Because we had to modify a derived test case manually, we gather the coverage information.

There is still one class in the input metamodel which is not covered, the `Subprocess`. We can derive the new test case from `test0003` (using the first part of Algorithm 1), as it can be seen in Figure D-1.8. This test case is valid, but because we have now full class coverage on the input model, we gather the coverage information again.

After this we aim for the test requirements derived from the attributes in the input metamodel. There are two, which are not covered yet, one derived from the integer attribute from the equivalence partition $\{[1; +\infty)\}$ of `type` from the class `Gateway` and one other derived from the boolean attribute from the equivalence partition $\{true\}$ of `transaction` from the class `Subprocess`. For the first one, we derive a new test case from `test0004` (Figure D-2.1) and for the second one, we derive a new test case from `test0005` (Figure D-2.2). Both models are derived with Algorithm 3, and are valid, so we do not have to modify them. After this, the attribute coverage in the input metamodel is full, so we gather the coverage information again.

After this we aim for the test requirements derived from the associations in the input metamodel. There is only one, which is not covered yet, derived from the reference `flowElements` from the class `Subprocess`, from the equivalence partition $\{[2; +\infty)\}$. We can derive a test case from `test0005`, as seen in Figure D-3.1 using the second part of Algorithm 4, but it is invalid, so we modify it to the model which can be seen in Figure D-3.2. Because we modified a derived test case manually, we gather the coverage information.

After this we would have focused on the output model. However every test requirement is already covered by the test case set, so we did not have to create new ones.

After this, we would have aimed for the test requirements derived from the transformation contract, but in our case study they are all covered with the available test cases.

Next we focus on the feature coverage of the input model. There is one feature defined: the nested subprocess feature and this is not covered yet. The nested subprocess feature is fulfilled by a model if it contains at least one subprocess which contains at least one subprocess, hence we can create a test case as seen in Figure D-4.1, but this is invalid, so we modify it to the one seen in Figure D-4.2.

A.2 Realistic case study

In this section we describe a second case study we carried through using our method. It was based on the model migration test case for Transformation Tool Contest 2010 [21]. The case study involves the model migration of UML Activity Diagrams from the UML 1.4 Specification [14] to the UML 2.2 Specification [15]. We used the metamodels available in [21] with some slight modifications, they can be seen in Appendix E along with the detailed description of the case study. The figures for test cases can be seen in Appendix G, they are denoted in abstract syntax when it is possible, and in concrete syntax only if it is necessary. The initial, “empty” input model can be seen in Figure G-1.1.

The first uncovered test requirement is derived from the class `Partition`, hence a second test case can be derived from `test0001` as seen in Figure G-1.2, with the second part of Algorithm 1. However, it is invalid, because a `Partition` has to be associated with at least one `ModelElement`, regarding the UML 1.4 Specification. Thus we modify the model to get what can be seen in Figure G-1.3.

The next uncovered test requirement is derived from the class `Guard`, for this we can derive a new model from `test0002` as seen in Figure G-1.4. We can not derive a new test model from `test0001`, because we can not connect the `Guard` instance into the model (it is contained in a `Transition`, and `test0001` has zero `Transitions`). In `test0002` we connect the new `Guard` instance to the first `Transition` as described in the second part of Algorithm 1, and because a `Guard` contains exactly one `BooleanExpression`, we instantiate that too. We do not have to modify this derived model.

The next uncovered test requirement is derived from the class `ObjectFlowState`. We derive a new test case from `test0002` as seen in Figure G-1.5. We use the first part of Algorithm 1 and substitute the `ActionState` instance with an `ObjectFlowState` (we choose the `ActionState` because there are three classes which are the closest from the `ObjectFlowState` and the smallest of them is the `ActionState`). The derived model is invalid, because the initial state can not have an `ObjectFlowState` on the other side of its outgoing `Transition`, regarding one of the postconditions rules of our transformation contract. Thus we modify the model to the one seen in Figure G-1.6.

All the classes in the input model are covered with the fourth test case. The next uncovered test requirement is derived from the attribute and equivalence partition `ActionState.isDynamic`{*true*}. We derive `test0005` as seen in Figure G-2 from `test0002`.

The next uncovered test requirement is derived from the association and equivalence partition `ActivityGraph.partitions`{[2; ∞)}. For this we can derive a new model from `test0001` as seen in Figure G-3.1. As described earlier a `Partition` has to have at least one element, and regarding other semantic rules concerning the partitions, we modified the model to the one seen in Figure G-3.2.

The next uncovered test requirement is derived from the association and equivalence partition `StateMachine.transitions`{1}. For this we derive the invalid model seen in Figure G-3.3. We modify this to the simplest possible model with only one `Transition`, seen in Figure G-3.4.

The next uncovered test requirement in the input model is derived from the association and equivalence partition `ObjectFlowState.type`{1}. For this we derive a new test model from `test0004` as seen in Figure G-3.5.

The last uncovered test requirement is derived from the association and equivalence partition `StateVertex.incoming`{[2; ∞)}. For this we derived the model seen in Figure G-3.6 from `test0007`. Since only `join` and `junction` `PseudoState`s may have more then one incoming `Transitions`, we modify the model to the one seen in Figure G-3.7.

The sole test requirement not covered yet in the output model was the class `DecisionNode`. For this we derived a new test case from `test0001` as seen in Figure G-4.1. This is invalid, hence we have to modify it as seen in Figure G-4.2. Note that if we would have defined test requirements for the different values in the `PseudoStateKind` enumeration, then this test requirement would have been fulfilled with an input model created for the enumeration value `junction`.

Next we would have aimed for the contract coverage, but the contract was already fully covered with `test0008`. Because there are no features defined, we are finished with the test case set generation.

With these ten test cases the coverage was maximized. After this we added the test case given in [21] (see Figure G-5) as `test9999`. This test case provides a big specification coverage, which is around 86% (it covers 88 test requirements of the 102). The not covered test requirements can not be covered with big test cases, e.g. there is only one valid model which contains only one `Transition`, and that is a small model.

B Test Requirements for the Simple Case Study

We defined the following specification based test requirements with the tool for our running example:

- one class coverage test requirement for every metamodel class;
- two attribute coverage test requirements for `Gateway.type` with the equivalence partitions $\{\{(-\infty;\ 0]\};\ \{[1;\ \infty)\}\}$;
- two attribute coverage test requirements for `Subprocess.transaction` with the equivalence partitions $\{\{\mathit{false}\};\ \{\mathit{true}\}\}$;
- one attribute coverage test requirement for `Edge.weight` with the equivalence partition $\{(-\infty;\ +\infty)\}$;
- two association coverage test requirement for `Process.flowElements` with the equivalence partitions over the multiplicity of the association: $\{\{0\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirement for `Subprocess.flowElements` with the equivalence partitions over the multiplicity of the association: $\{\{0\};\ \{[2;\ \infty)\}\}$;
- three association coverage test requirement for `SimpleGraph.edges` with the equivalence partitions over the multiplicity of the association: $\{\{0\};\ \{1\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirement for `SimpleGraph.nodes` with the equivalence partitions over the multiplicity of the association: $\{\{0\};\ \{[2;\ \infty)\}\}$;
- four association coverage test requirements for `SequenceFlow.source`, `SequenceFlow.target`, `Edge.source` and `Edge.target` with the equivalence partitions over the multiplicity of the association: $\{1\}$;
- one contract rule coverage test requirement for every contract rule;
- one feature coverage test requirement for the nested subprocess feature.

C Ratios for Derived Test Cases in the Simple Case Study

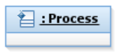
With **bold** the base test cases are typed. Non-derivable cases are noted with *ND*.

	test0001	**test0002**	**test0003**	**test0004**	**test0005**	**test0006**	**test0007**
test0002	**1.2452**						
test0003	**1.8305**	2.2182					
test0004	2.5227	2.4670	**2.1524**				
test0005	3.3338	3.3513	**2.4787**	3.0105			
test0006	*ND*	*ND*	*ND*	**0**	*ND*		
test0007	*ND*	*ND*	*ND*	*ND*	**2.0302**	*ND*	
test0008	*ND*	*ND*	*ND*	*ND*	**5.7716**	*ND*	5.7716

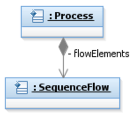
Table 1: Ratios for derived-base footprint pairs during the simple case study

In the last case the length of the sub-footprints of both base test cases was the same (7.6158), hence we choose `test0005` as a base test case.

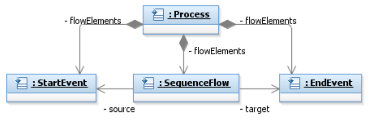
D Figures for Test Cases in the Simple Case Study



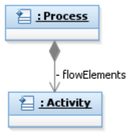
(D-1.1) The “empty” model (`test0001`)



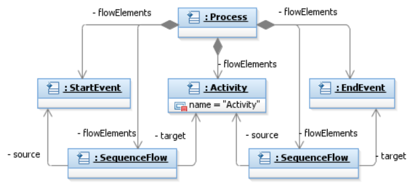
(D-1.2) Derived model for the second test case



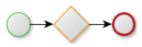
(D-1.3) Modified model for the second test case (`test0002`)



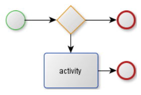
(D-1.4) Derived model for the third test case



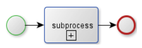
(D-1.5) Modified models for the third test case (`test0003`)



(D-1.6) Derived model for the fourth test case

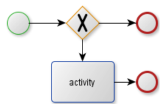


(D-1.7) Modified model for the fourth test case (`test0004`)

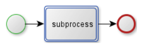


(D-1.8) Derived model for the fifth test case (`test0005`)

Fig. D-1: Test cases for maximized class coverage in the simple case study

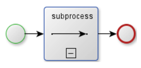


(D-2.1) First derived model for attribute coverage (`test0006`)

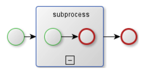


(D-2.2) Second derived model for attribute coverage (`test0007`)

Fig. D-2: Test cases for maximized attribute coverage in simple case study

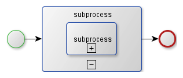


(D-3.1) Derived model for reference coverage

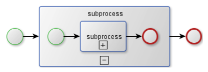


(D-3.2) Modified model for the ninth test case (`test0008`)

Fig. D-3: Test cases for maximized association coverage in the simple case study



(D-4.1) First attempt for covering the nested subprocess feature



(D-4.2) Test case for covering the nested subprocess feature (`test0009`)

Fig. D-4: Test case for maximized feature coverage in the simple case study

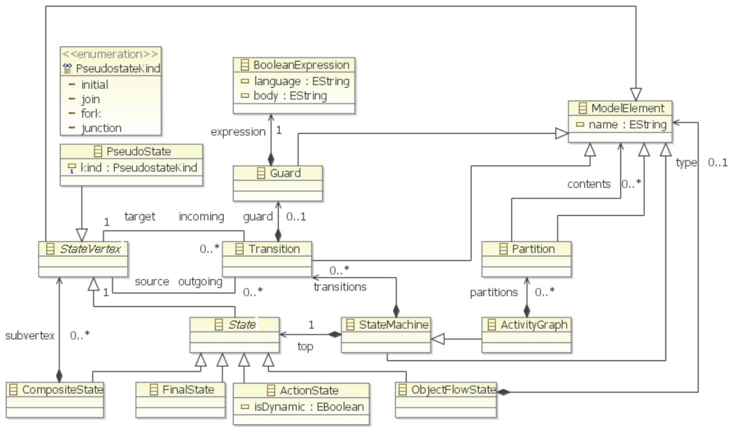


Fig. E-1: The metamodel of the simplified UML 1.4 Activity Diagram

E Details of the UML Case Study

The realistic case study involves migrating models representing Activity Diagrams in the UML 1.4 Specification [14] to models which corresponds to the UML 2.2 Specification [15]. The simplification of the metamodels is based on [21].

The transformation is as straightforward as it can be. The transformation contract consists of the relevant semantic rules of the two specifications, namely:

- precondition rules:
  - if an Activity Diagram contains at least one `State` in addition to the `StateMachine.top`, then there has to be exactly one `PseudoState` with `kind = initial` and exactly one `FinalState`;
  - a `PseudoState` with `kind = fork` must have exactly one incoming `Transition` and at least two outgoing `Transitions`, similarly a `PseudoState kind = join` must have at least two incoming and exactly one outgoing `Transitions`;
  - the `PseudoState kind = fork` and `PseudoState kind = join` `PseudoStates` must be well-nested;
  - the `PseudoState` with `kind = initial` must have zero incoming `Transitions`, similarly the `FinalState` must have zero outgoing `Transitions`;
  - the `PseudoState` with `kind = junction` must have one or two incoming `Transitions`;

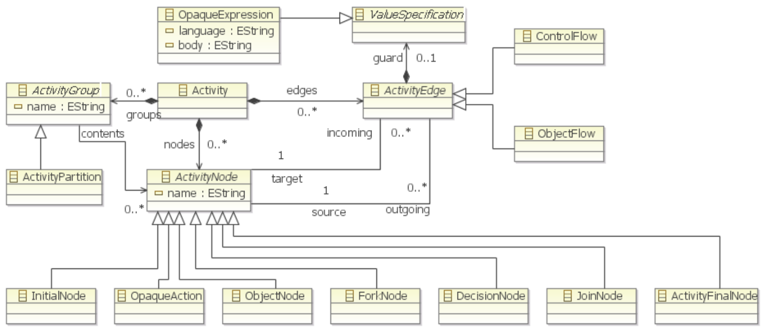


Fig. E-2: The metamodel of the simplified UML 2.2 Activity Diagram

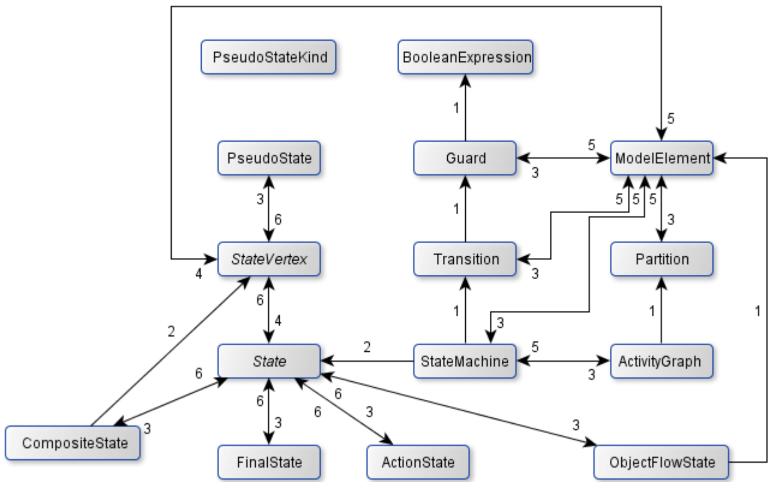


Fig. E-3: The ordering graph of the simplified UML 1.4 Activity Diagram

- any `State` if not stated otherwise must have exactly one incoming and exactly one outgoing `Transition`;
- `StateMachine.top` must be `CompositeState`;
- an Activity Diagram must be traversable from the `PseudoState` with `kind = initial`

– postcondition rules:

- `Decisions` must have one or two incoming `ActivityEdges` with the following restrictions:
  a) if there is one incoming `ActivityEdge`, and the type of it is `ControlFlow`, then all the outgoing `ActivityEdges` have to be `ControlFlows`;
  b) if there are two incoming `ActivityEdges`, and both of them are `ObjectFlows`, then the all of the outgoing `ActivityEdges` have to be `ObjectFlows`;

  - if an Activity Diagram contains at least one `ActivityNode`, then there has to be exactly one `InitialNode` and exactly one `ActivityFinalNode`;
  - if the `source` or `target` of an `ActivityEdge` is an `ObjectNode` then it has to be an `ObjectFlow` otherwise it has to be a `ControlFlow`;
  - a `ForkNode` must have exactly one incoming `ActivityEdge` and at least two outgoing `ActivityEdge`s, similarly a `JoinNode` must have at least two incoming and exactly one outgoing `ActivityEdge`s;
  - the `ForkNode`s and `JoinNode`s must be well-nested;
  - the `InitialNode` must have zero incoming `ActivityEdge`s, similarly the `ActivityFinalNode` must have zero outgoing `ActivityEdge`s;
  - the `InitialNode` must have a `ControlFlow` as outgoing `ActivityEdge`;
  - an `ObjectFlow` has to have at least one `ObjectNode` attached (as `source` or as `target`);
  - any `ActivityNode` if not stated otherwise must have exactly one incoming and exactly one outgoing `ActivityEdge`;
  - an Activity Diagram must be traversable from the `InitialNode`;
- transformation rules:
  - both models must have the same amount of `contents` in every `Partition`→`ActivityGroup`;
  - both models must have the same amount of `Partitions`→`ActivityGroups`;
  - both models must have the same amount of `StateVertexes`→`ActivityNodes`;
  - both models must have the same amount of `Transitions`→`ActivityEdges`.

The ordering of the metamodel classes, regarding Definition 5 is the following in the metamodels:

- for the UML 1.4 metamodel (Figure E-1):
  `ActivityGraph < Partition < StateMachine < ModelElement <`
  `< Transition < Guard < State < BooleanExpression <`
  `< ActionState < CompositeState < FinalState < ObjectFlowState <`
  `< StateVertex < PseudoState < PseudoStateKind`
- for the UML 2.2 metamodel (Figure E-2):
  `Activity < ActivityEdge < ActivityGroup <`
  `< ActivityNode < ValueSpecification < ActivityFinalNode <`
  `< ActivityPartition < ControlFlow < DecisionNode < ForkNode <`
  `< InitialNode < JoinNode < ObjectFlow < ObjectNode <`
  `< OpaqueAction < OpaqueExpression`

The ordering graph for the UML 2.2 metamodel is simple, the root class is the `Activity`. The ordering graph for the former UML 1.4 specification is complex, it can be seen in Figure E-3, the root class is the `ActivityGraph`. The ordering of the attributes and associations are trivial, we do not enumerate them.

We defined the following test requirements for the UML case study:

- one class coverage test requirement for every metamodel class;
- two attribute coverage test requirements for `ActionState.isDynamic` with the equivalence partitions {{*false*}; {*true*}};

- three association coverage test requirements for `ActivityGraph.partitions` with the equivalence partitions $\{\{0\};\ \{1\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirements for `Partition.contents` with the equivalence partitions $\{\{0\};\ \{[2;\ \infty)\}\}$;
- one association coverage test requirement for `Guard.expression` with the equivalence partition $\{1\}$;
- one association coverage test requirement for `StateMachine.top` with the equivalence partition $\{1\}$;
- one-one association coverage test requirement for `Transition.source` and `Transition.target` both with the equivalence partition $\{1\}$;
- three-three association coverage test requirements for `StateVertex.incoming` and `StateVertex.outgoing` with the equivalence partitions $\{\{0\};\ \{1\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirement for `CompositeState.subvertex` with the equivalence partitions $\{\{0\};\ \{[2;\ \infty)\}$;
- two association coverage test requirement for `ObjectFlowState.type` with the equivalence partitions $\{\{0\};\ \{1\}\}$;
- two times three association coverage test requirements for `Activity.edges` and `Activity.groups` with the equivalence partitions $\{\{0\};\ \{1\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirements for `Activity.nodes` with the equivalence partitions $\{\{0\};\ \{[2;\ \infty)\}\}$;
- two association coverage test requirements for `ActivityEdge.guard` with the equivalence partitions $\{\{0\};\ \{1\}\}$;
- one-one association coverage test requirement for `ActivityEdge.source` and `ActivityEdge.target` both with the equivalence partition $\{1\}$;
- two association coverage test requirements for `ActivityGroup.contents` with the equivalence partitions $\{\{0\};\ \{[2;\ \infty)\}\}$;
- three-three association coverage test requirements for `ActivityNode.incoming` and `ActivityNode.outgoing` with the equivalence partitions $\{\{0\};\ \{1\};\ \{[2;\ \infty)\}\}$;
- one contract rule coverage test requirement for every contract rule.

F Ratios for Derived Test Cases in the UML Case Study

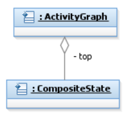
With **bold** the base test cases are typed. Non-derivable cases are noted with *ND*.

The decisions in the case of `test0004` and `test0005` are based on the lengths: $len(sfp_{inp}(\texttt{test0002})) = 11.5758$; $len(sfp_{inp}(\texttt{test0003})) = 12.3288$; $len(sfp_{inp}(\texttt{test0004})) = 14.7648$.

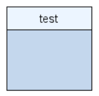
	test0001	**test0002**	**test0003**	**test0004**	**test0005**	**test0006**	**test0007**	**test0008**	**test0009**
test0002	**3.0950**								
test0003	*ND*	**5.3337**							
test0004	4.0439	**0**	0						
test0005	*ND*	**0**	0	0					
test0006	**12.2165**	587.1415	1277.0502	16779.9361	1062.5925				
test0007	**30.4602**	∞	∞	∞	∞	∞			
test0008	*ND*	*ND*	*ND*	**5.4855**	*ND*	6.1544	*ND*		
test0009	264.3699	9.8417	9.3191	10.4912	9.8417	11.6860	**8.5878**	10.7007	
test0010	**1.8558**	5.2886	5.2886	5.2578	5.2886	5.3453	3.4148	5.0856	5.2578

Table 2: Ratios for derived-base footprint pairs during the UML case study

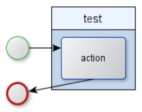
G Figures for Test Cases in the UML Case Study



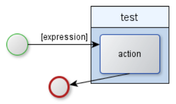
(G-1.1) The "empty" Activity Diagram in UML 1.4 (`test0001`)



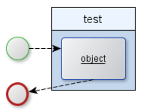
(G-1.2) The derived test model for the second test case



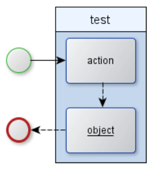
(G-1.3) The modified test model for the second test case (`test0002`)



(G-1.4) The derived test model for the third test case (`test0003`)



(G-1.5) The derived test model for the fourth test case



(G-1.6) The modified test model for the fourth test case (`test0004`)

Fig. G-1: Test cases for maximized input metamodel class coverage in the UML case study

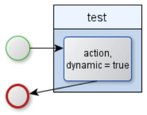
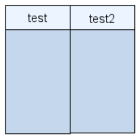
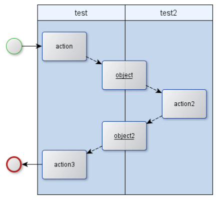


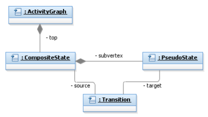
Fig. G-2: Test case for maximized attribute coverage in the UML case study (the derived test model for the fifth test case, `test0005`)



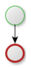
(G-3.1) The derived test model for the sixth test case



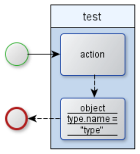
(G-3.2) The modified test model for the sixth test case (`test0006`)



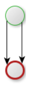
(G-3.3) The derived test model for the seventh test case



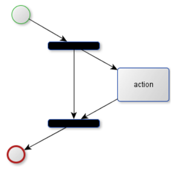
(G-3.4) The modified test model for the seventh test case (`test0007`)



(G-3.5) The derived test model for the eighth test case (`test0008`)



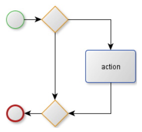
(G-3.6) The derived test model for the ninth test case



(G-3.7) The modified test model for the ninth test case (`test0009`)

Fig. G-3: Test cases for maximized association coverage in the UML case study

(G-4.1) The derived test model for the tenth test case



(G-4.2) The derived test model for the tenth test case (`test0010`)

Fig. G-4: Test cases for maximized output metamodel class coverage in the UML case study

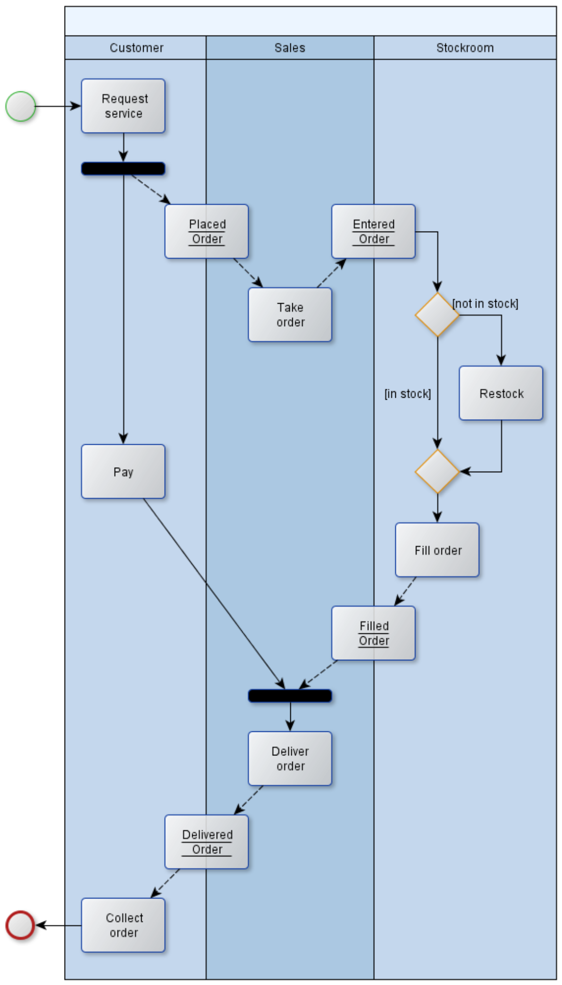


Fig. G-5: Test case for the UML case study given in [21]