Towards Dependable Model Transformations: Qualifying Input Test Data

Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, Yves Le Traon

Qualifying Input Test Data for Model Transformations

FRANCK FLEUREY[1], BENOIT BAUDRY[1], PIERRE-ALAIN MULLER[1], YVES LE TRAON[2]

1 IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{ffleurey, bbaudry, pierre-alain.muller}@irisa.fr

2 ENST Bretagne, 2 rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné Cedex
yves.letraon@enst-bretagne.fr

Abstract. Model transformation is a core mechanism for model-driven engineering (MDE). Writing complex model transformations is error-prone, and efficient testing techniques are required as for any complex program development. Testing a model transformation is typically performed by checking the results of the transformation applied to a set of input models. While it is fairly easy to provide some input models, it is difficult to qualify the relevance of these models for testing. In this paper, we propose a set of rules and a framework to assess the quality of given input models for testing a given transformation. Furthermore, the framework identifies missing model elements in input models and assists the user in improving these models.

1. Introduction

Model-driven engineering (MDE) proposes a move away from human interpretation of high-level models, such as design diagrams, towards a more automated process where models are used as first-class artefacts of a development process. The core mechanism for this automation is model transformation. A model transformation typically implements process-related steps including refactoring, model composition, aspect weaving, code generation or refinement. Writing complex model transformations is error-prone, and efficient testing techniques are required as for any complex program development and is an important challenge if MDE is to succeed

[1]. The need for reliable model transformations is even more critical when they are to be reused. Indeed, a single faulty transformation can make a whole model-based development process vulnerable.

To test a model transformation, a tester will usually provides a set of *test models* that conform to the input meta-model of the transformation, run the transformation with these models and check the correctness of the result. While it is fairly easy to provide some input models, qualifying the relevance of these models for testing is an important challenge in the context of model transformations [2]. As for any testing task, it is important to have precise adequacy criteria that can qualify a set of test data. For example, a classical criterion to evaluate the quality of the test data regarding a program is code coverage: a set of test data is adequate if, when running the program with these data, all statements in the program are executed at least once. This is a "white-box" criterion since it requires the knowledge of internal logic or code structure of the program. Other criteria are functional or "black-box" [3]. They rely only on a specification of the system (input domain or behavior) under test and do not take the internal structure of the program into account.

In this paper, we propose a framework for selecting and qualifying test models for the validation of model transformations. We propose "black-box" test adequacy criteria for this selection framework. We chose black-box criteria for two reasons: to have criteria which are independent of any specific model transformation language and to leverage the complete description of the input domain provided by the input metamodel of the transformation. It is important that the proposed approach is generic and compatible with any model transformation language because currently there are many languages for transformation and none of them has emerged as the best or the most popular. The proposed criteria can be used to validate model transformations implemented with a general purpose language such as Java, the specific model transformation language QVT [4] proposed by the OMG, a meta-

modelling language such as Kermeta [5], a rule-based language such as Tefkat [6], or a graph transformation language such as ATOM3 [7]. The second reason why we choose black box criteria is to leverage the fact that the input domain for a transformation is defined by a meta-model. Indeed, the input meta-model of a transformation completely specifies the set of possible input models for a transformation. In this context, the idea is to evaluate the adequacy of test models with respect to their coverage of the input meta-model. For instance, test models should instantiate each class and each relation of the input meta-model at least once.

Models are complex graphs of objects. To select useful models we first have to determine relevant values for the properties of objects (attributes and multiplicities) and next to identify pertinent structures of objects. For the qualification of values of properties we propose to adapt a classical testing technique called category-partition [8] testing. The idea is to decompose an input domain into a finite number of sub-domains and to choose a test datum from each of these sub-domains. For the definition of object structures, we propose several criteria that define structures that should be covered by the test models.

An important contribution of this work consists in defining a meta-model that formally captures all the important notions necessary for the evaluation of test models (partitions and object structures). This meta-model hence provides a convenient formal environment to experiment different strategies for test selection, and a framework that checks if test models are adequate for testing. The framework automatically analyses a set of test models and provides the testers with valuable feedback concerning missing information in their test models. This information can then be used to iteratively complete a set of test models.

The paper is organized as follows. Section 2 discusses a motivating example and provides an informal description of the technique. Section 3 proposes a meta-model that captures the different concepts needed to define test criteria and evaluate the

efficiency of test data. Section 4 proposes several test criteria. Section 5 employs a simple case study to show how the proposed technique can be applied to improve test models. Finally, section 6 discusses related works and section 7 draws conclusions.

2. Motivating example

To discuss and illustrate the techniques we propose, we use a simple model transformation which flattens hierarchical state machines (we call this transformation *SMFlatten*). The transformation takes a hierarchical state machine as input and produces an equivalent flattened state machine as output. Figure 1 presents the application of this transformation to a simple example. Both the input model and the output model of the transformation are state machines. Figure 2 displays the state machine meta-model we use. According to this meta-model, a state machine is composed of a set of states, composite states and transitions. Each state is labeled by an integer (property *label*) and an event is associated with each transition (property *event*). Properties *isInitial* and *isFinal* on class STATE respectively specify initial and final states.

The validation of the *SMFlatten* transformation consists in running it with a well-chosen set of hierarchical state machines and checking that the obtained flattened state machines semantically correspond to their sources. As it is obviously impossible to test the transformation with every possible input state machine, the first issue is to select a set of input state machines that is likely to reveal as many errors as possible in the transformation. In the following sections we call such a set of input models a set of *test models*.

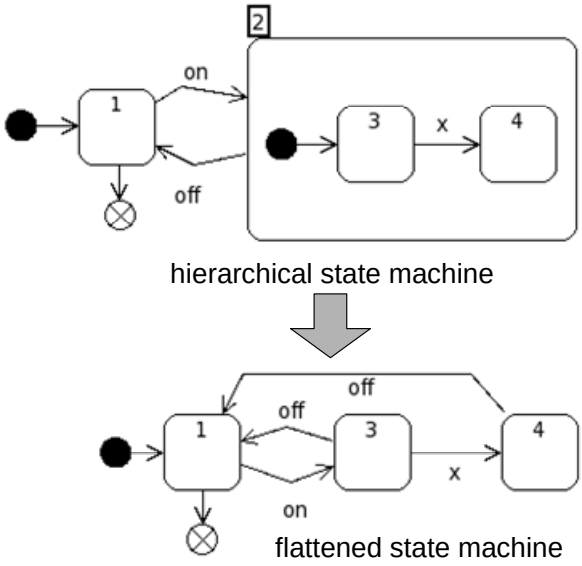

Figure 1 – An example of hierarchical state machine flattening

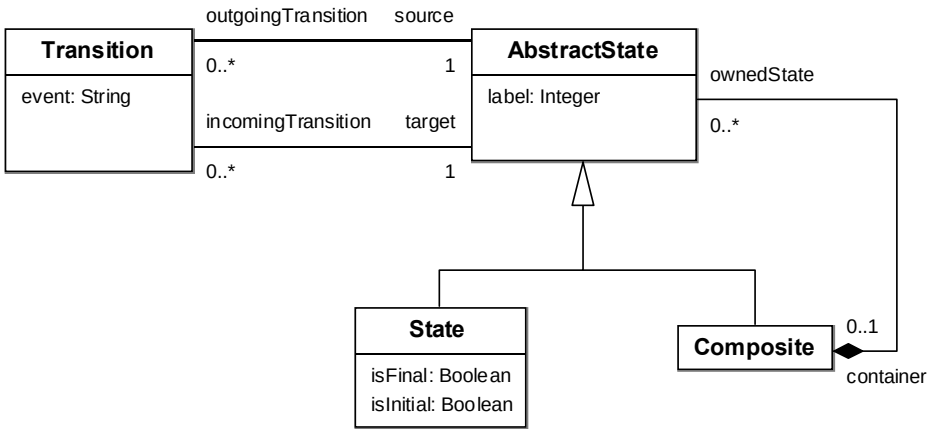

Figure 2 – Simple Composite State Machine Meta-model

A reasonable set of test models for the *SMFlatten* transformation should, at least, fulfill the three following coverage requirements:

- **Class coverage.** Each concrete class of the state meta-model should be instantiated in at least one test model.

- **Attribute coverage.** Each attribute in the meta-model should be instantiated with a set of representative values. For example, in the test models there should be both some final states and some non-final states in order to cover the values of property *isFinal* of class *State*.
- **Association coverage.** Each association in the meta-model should be instantiated with a set of representative multiplicities. The state machine meta-model specifies that a composite state can contain from none to several states. To cover these possibilities, the test models should contain, at least, composites states with no inner states, with only one inner state and with several inner states.

The first requirement (class coverage) is simple and can be applied directly. However, in order to take advantage of the two remaining properties, the way "representative" values and multiplicities are defined must be expressed more formally. In the following sections we propose to adapt category-partition testing to select relevant ranges of values for properties and their multiplicities.

Covering individually each attribute or association of the meta-model with a set of representative values is not sufficient. In addition to the above coverage requirements the representative values and multiplicities should be combined to build relevant test models. For instance, the *SMFlatten* transformation should be tested with models which contain states that are both initial and that have several outgoing transitions. There should also be several states with all possible combinations of values for isFinal and isInitial. In the following sections, we propose 10 systematic strategies (defined as test criteria) to combine values for properties.

3. A framework for selecting test models

This section introduces the framework we use to define test criteria for model transformations (that are detailed in the next section). First, we explain how we can define generic test criteria for any source metamodel. Then, we introduce the notions

of partition and model fragment that are necessary to specify the instances of the metamodel that are relevant for testing. Finally, we present the metamodel that captures all these notions. It is the core for the definition of test criteria and for the tool that checks that test models satisfy a test criterion.

3.1. Generic approach

The goal of this work is to propose criteria to evaluate the coverage of test models with respect to the structure of their corresponding meta-models. In practice, meta-models like the state machine meta-model of Figure 2 are specified using a meta-modelling language. Today, several meta-modelling languages exist: the Meta-Object Facilities (MOF [9]), ECore, CMOF, EMOF, etc. The work presented in this paper is based on the EMOF (Essential Meta-Object Facilities [10]). This means that the metamodels we manipulate to define test criteria are modelled using EMOF. We choose EMOF because it is a standardized language that is well supported by tools such as Eclipse Modelling Framework (EMF [11]). Although it is based on EMOF, the ideas and techniques presented in this paper can be adapted to any object-oriented meta-modelling language.

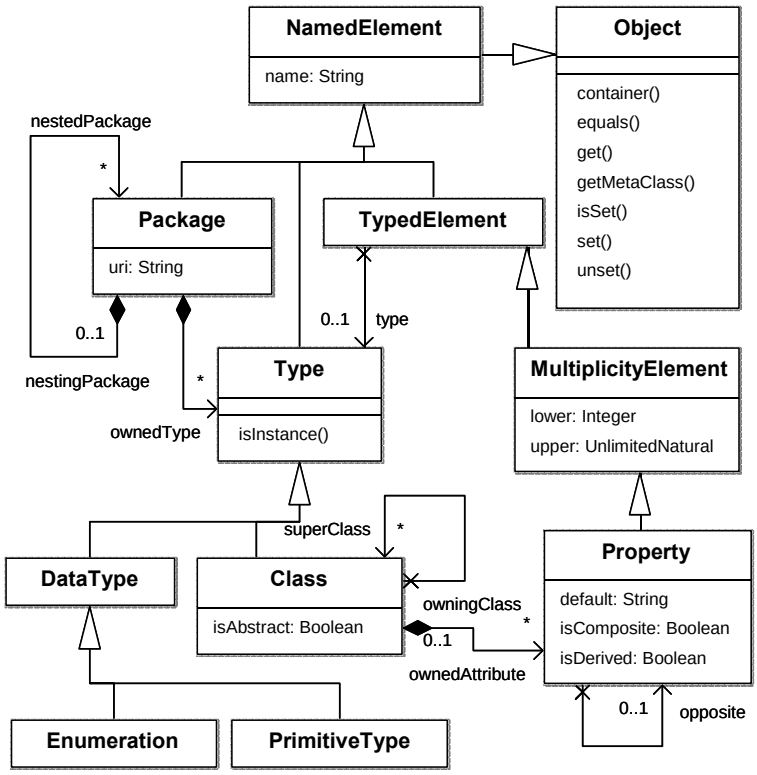

Figure 3 – Main classes of EMOF

EMOF was standardized by the OMG (Object Management Group) as a compact meta-modelling language. It contains a minimal set of concepts which are necessary for meta-modelling. Figure 3 presents the main classes of EMOF which are relevant to the work presented in this paper (overall, EMOF contains 21 classes). According to EMOF, a meta-model is composed of a set of packages (class PACKAGE). Each package contains a set of types which can be either data types (classes PRIMITIVETYPE and ENUMERATION) or classes (class CLASS). Each class is composed of a set of *properties* (class PROPERTY). The notion of property is central to EMOF because they are a compact representation for both attributes in classes and associations between classes. If the type of a property is a data type then it corresponds to an attribute. For example, in the state machine meta-model (Figure 2)

the property *label* of class ABSTRACTSTATE corresponds to an attribute of type integer. If the type of a property is a class then it corresponds to an association. In that case, properties of two classes involved in an association can be defined as opposites. Such correspondence pairs of properties are used to represent the ends of a single association. In the state machine meta-model the associations between ABSTRACTSTATE and TRANSITION have been defined in this way. The first association corresponds to a property *outgoingTransition* of type TRANSITION in class ABSTRACTSTATE and a property source of type ABSTRACTSTATE in class TRANSITION. To have a generic approach, in the following we will not distinguish between attributes and associations, but deal only with classes and their properties in the metamodel to be covered.

3.2. Partitioning values and multiplicities of properties

The basic idea of category-partition testing strategies [8] is to divide the input domain into sub-domains or ranges and then to select test data from each of these ranges. The ranges for an input domain define a partition of the input domain and thus should not overlap. Partition testing has been adapted to test UML models in [12], and here we adapt it to test model transformations. In this specific case, the input domain is modeled by the input meta-model of the transformation. The idea is to define partitions for each property of this meta-model. A precise definition of a partition is recalled below.

Definition – Partition*. A partition of a set of elements is a collection of n ranges* $A_1, \ldots, A_n$ *such that* $A_1, \ldots, A_n$ *do not overlap and the union of all subsets forms the initial set. These subsets are called ranges.*

Notation – Partition. *In this paper, the partitions are noted as follows:*

- *Boolean partitions are noted as a set of sets of Boolean values. For example {{true},{false}} designates a partition with two ranges: a range which contains the value true and a range which contains the value false*
- *Integer partitions are noted as a set of sets of Integer values. For example, {{0}, {1}, {x | x ≥ 2}} designates a partition with three ranges: 0, 1, greater or equal to 2.*
- *String partitions are noted as a set of sets of String values. A set of string values is specified by a regular expression. For example {{'evt1'}, {""}, {".+"}} designates a partition with three ranges: a range which contains the string 'evt1', a range which contains the empty string and a range which contains all strings with one or more character. In the regular expression language, "." designates any character and "+" specifies that the preceding symbol has to be repeated one or more time.*

To apply this idea for the selection of test models, we propose to define partitions for each property of the input meta-model of a transformation. These partitions provide a practical way to select what we called the "representative" values introduced in the previous section: for a property *p* and for each range *R* in the partition associated with *p*, the test models must contain at least one object *o* such that the value *o.p* is taken in *R*. For instance, the partitions {{'evt1'}, {""}, {".+"}} for the property *event* of class *Transition* in the state machine meta-model, formalize that the test models should contain transitions with a particular event called 'evt1', transitions with an empty event and transitions with a random non-empty event. The same kind of strategy is used for multiplicities of properties: if a property has a multiplicity of *0..**, a partition such as {{0}, {1}, {x | x ≥ 2}} is defined to ensure that the test models contain instances of this property with zero, one and more than one object.

The effectiveness of category-partition testing strategies relies on the quality of the partitions that are used. The approach for the generation of meaningful ranges is

usually based on the topology of the domain to be partitioned. The idea is to isolate boundaries and singular values in specific ranges in order to ensure that these special values will be used for testing. Figure 4 shows the partitions obtained for all properties of the state machine meta-model used by the *SMFlatten* transformation (partitions on the multiplicity of a property are denoted with a #). The default partitions based on the types of properties can be automatically generated. On this example, they seem sufficient. Yet, if other values have a special meaning in the context of the transformation under test, the tester can enrich the partitions to ensure that some additional value is used in the test models.

Transition::**event**	**{''}, {'evt1'}, {'.+'}**
Transition::**#source**	**{1}**
Transition::#**target**	**{1}**
AbstractState::**label**	**{0}, {1}, {x | x>1}**
AbstractState::#**container**	**{0}, {1}**
AbstractState::#**incomingTransition**	**{0}, {1}, {x | x>1}**
AbstractState::#**outgoingTransition**	**{0}, {1}, {x | x>1}**
State:**isInitial**	**{true}, {false}**
State::**isFinal**	**{true}, {false}**
Composite::**#ownedState**	**{0}, {1}, {x | x>1}**

Figure 4 – Partitions for the state machine meta-model

The next section introduces the meta-model that captures the concept of partitions associated with a meta-model, as well as concepts needed to combine ranges in order to define test criteria. An instance of this meta-model can then be used to check that a set of test models covers the desired values and combinations.

3.3. Model and object fragments

As introduced previously, independently covering the values and multiplicity of each property of the input meta-model is not sufficient to ensure the relevance of test models. As an example, let us consider the property *ownedState* of class *Composite* in the state machine meta-model. The partitioning step has defined three ranges for the

multiplicity of this property to ensure that the test models contain empty composite state, composite states with only one inner state and composite states with several inner states. However, none of these constraints requires that any composite state has both incoming and/or outgoing transitions and inner states. The selection criterion should be expanded to ensure the combined coverage of the ranges for the properties *ownedState* combined with ranges of properties, *incomingTransition,* and *outgoingTransition*. This way the test models would include composite states with both a variable number of inner states and a variable number of outgoing transitions.

A naive approach to combine partitions of various properties would be to generate the combinatorial product of all partitions for all properties of the meta-model. However, this approach is not practically viable:

1. **Combinatorial explosion**: combining ranges of all properties quickly results in unmanageable number of combinations. For the state machines example, which is fairly simple, , the number of combinations reaches 1944 (3*1*1*3*2*3*3*2*2*3) for the partitions defined in Figure 4.
2. **Unconsidered relevance**. Among the 1944 combination, some are more relevant than others for testing: for instance, combining property *ownedState* and property *outgoingTransition* is interesting for the testing of composite states while combining the property *label* of class *State* and the property *event* of class *Transitions* is not.
3. **Missing relavant combinations**: combining ranges of properties is generally not sufficient to ensure the relevance of test models. We often found it necessary to include test models that cover combinations of ranges for a single property for several objects. For example, the 1944 combinations obtained for the state machine meta-model do not ensure the existence of a single test model that includes more than one composite state. This is clearly not sufficient to test a transformation that flattens composite states.

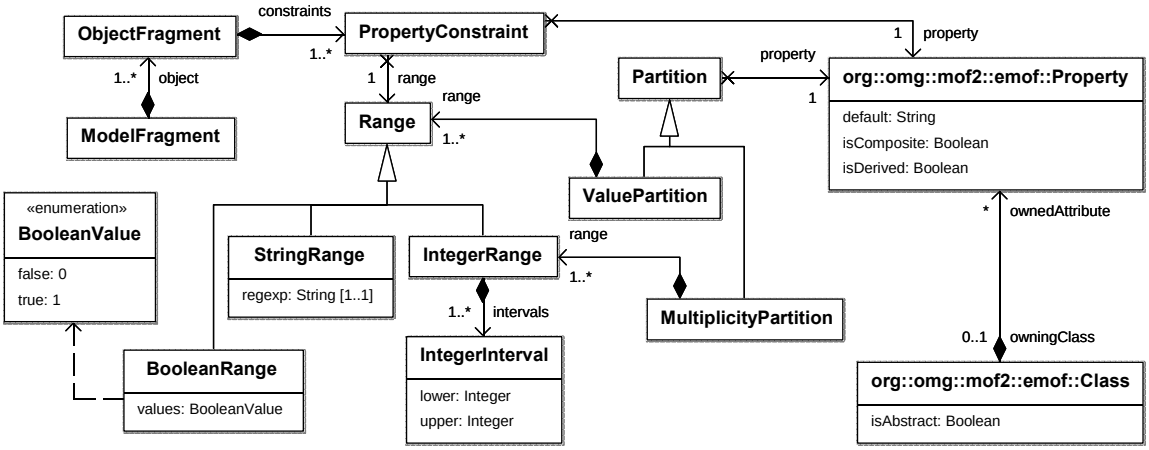

Figure 5 – Meta-model for test criteria definition

As naive strategies are not sufficient to provide satisfactory support for the selection of relevant test models, we propose the notions of object and model fragments to define specific combinations of ranges for properties that should be covered by test models. The meta-model in Figure 5 captures the notions of partition associated to properties as well as model and object fragments. We distinguish two types of partitions modelled by the classes VALUEPARTITION and MULTIPLICITYPARTITION that respectively correspond to partitions for the value and the multiplicity of a property. For a MULTIPLICITYPARTITION, each range is an integer range (class INTEGERRANGE). For a VALUEPARTITION, the type of ranges depends on the type of the property. Here we consider the three primitive types that are defined in EMOF for the value of a property. Thus we model three different types of ranges (STRINGRANGE, BOOLEANRANGE, INTEGERRANGE). Figure 6 shows how the partition part of the meta-model is instantiated to represent the partitions associated with two properties of the state machine meta-model.

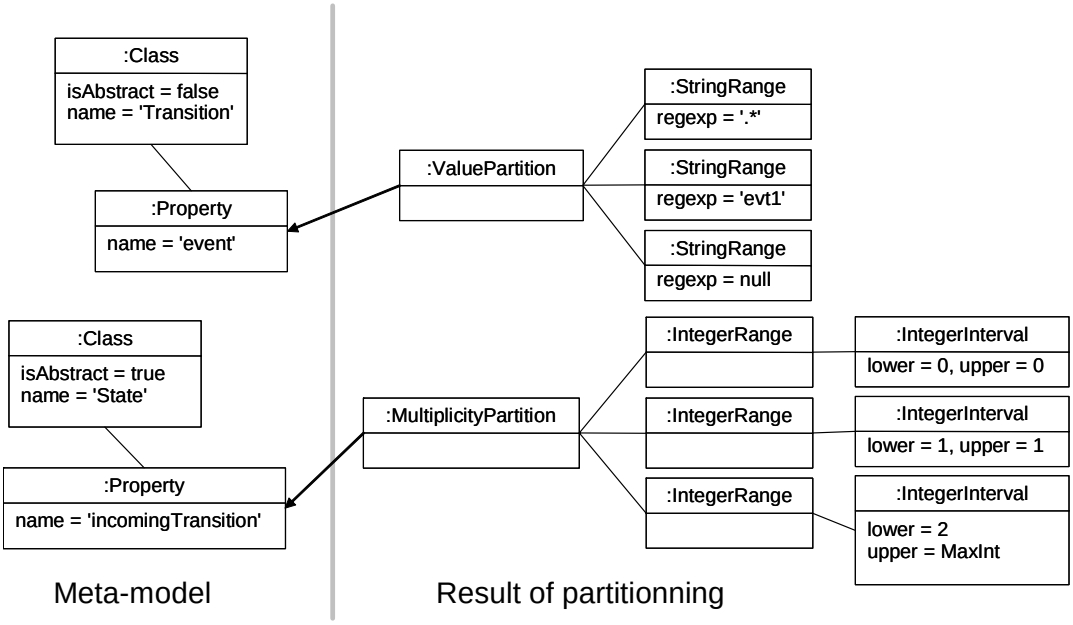

Figure 6 – Partitions and ranges

To represent combinations of partition ranges, the meta-model of Figure 5 proposes the notions of model fragments (MODELFRAGMENT), object fragments (OBJECTFRAGMENT) and property constraints (PROPERTYCONSTRAINT). A model fragment is composed of a set of object fragments. An object fragment is composed of a set of property constraints which specify the ranges from which the values of the properties of the object should be taken from. It is important to note that an object fragment does not necessarily define constraints for all the properties of a class, but can partially constrain the properties (like a template).

The model and object fragments are defined in order to check that the set of test models covers the input meta-model of a transformation. A model fragment is said to be covered by a test model if, for each object fragment in the model fragment, there exists one object in the test model that matches the object fragment. An object in a test model is said to match an object fragment if it satisfies every property constraint of the object fragment. A property constraint is said to be satisfied by an object if the

value of the property for that object is included in the range associated with the property constraint.

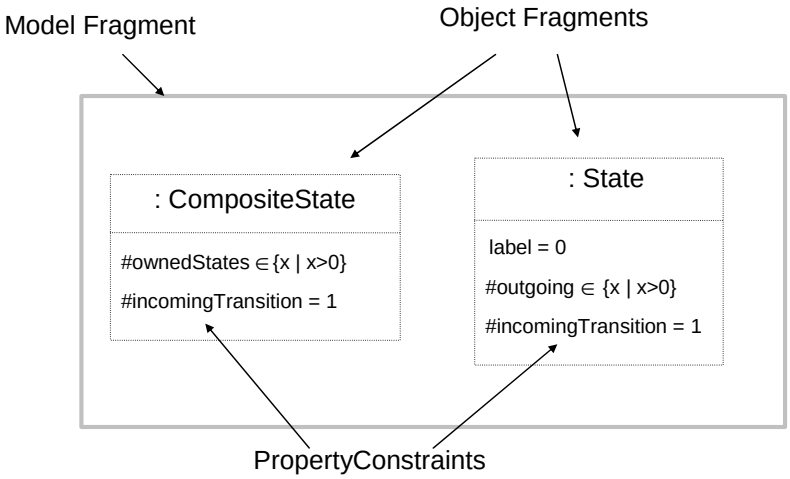

Figure 7 – A model fragment

Figure 7 presents a simple model fragment that combines ranges for properties of the state machine meta-model. The model fragment is composed of two object fragments which specify that a test model must contain:

- a composite state with several inner states and an incoming transition.
- a state labeled 0 with one incoming and several outgoing transitions.

In the next sections we use a more compact textual notation for model fragments. Model fragments are represented by $MF\{of_1, of_2, ..., of_n\}$ where object fragments (of_i) are represented by $<ClassName>(c_1, c_2, ..., c_n)$. Using this notation the model fragment of Figure 7 is represented by :

```
MF{CompositeState(#ownedStates > 0, #inTrans. = 1),
State(label = 0, #outTrans. > 0, #inTrans. = 1)}
```

Using the model fragment representation, the particular combinations that should be covered by the test models can be easily represented. Yet, the selection of these combinations is still an issue. In section 4, we propose a set of strategies to automate the generation of sets of model fragments. The idea is to use these model fragments

as a test adequacy criterion: adequate test models must cover every model fragment. The next section discusses this adequacy validation algorithm and the associated test engineering process.

3.4. Qualification and selection of test models

Based on the concepts defined in the meta-model in the previous section, it is possible to define an iterative engineering process for selecting a set of input models intended to test a model transformation. This process, described in Figure 8 takes two inputs (white ovals): the input meta-model of the transformation under test and a set of test models. From the input meta-model, the first step generates the default partitions for all features contained in the meta-model. The second step combines these partitions to build a set of model fragments. This step takes a test criterion as its parameter that defines how fragments are to be composed. Test criteria will be detailed in section 4. During both partitioning and combination the tester may enrich the generated models to take domain specificities of the transformation under test into account.

When the model fragments are generated from the input meta-model, step three checks that there is at least one test model that covers each model fragment. If there are fragments not covered by the test models, the tester should improve the set of test models by adding new models to cover the identified remaining fragments (step 4). This process does not only allow for an estimate of the quality of a set of test models but also provides the testers with valuable information to improve the test set. As illustrated in Figure 8, steps 1, 2 and 3 are implemented with the Kermeta language. The implementation of the tool, MMCC, is discussed in section 5.

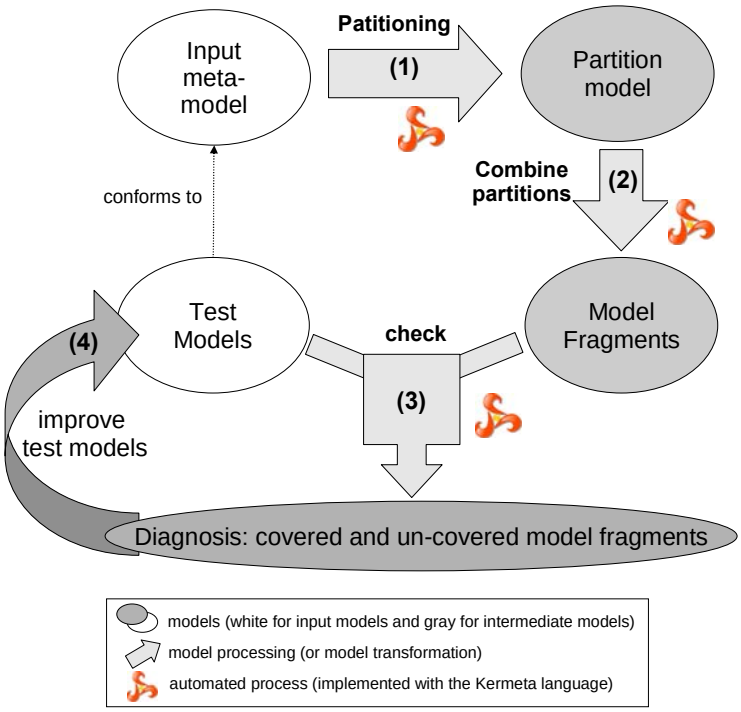

Figure 8 – Check and improve the quality of test models

4. Black-box test criteria

Class coverage	`Class.allInstances()->forAll( C |`
	` ModelFragment.allInstances()->exists( MF |`
	`  MF.object->exists( OF |`
	`  OF.constraint->exists( PC |`
	`  PC.property.owningClass == C`
	`))))`

Range coverage	`Partition.allInstances()->forAll( P |`
	`  P.range->forAll( R |`
	`  ModelFragment.allInstances()->exists( MF |`
	`  MF.object->exists( OF |`
	`  OF.constraint->exists( PC |`
	`  PC.range == R`
	`)))))`

Figure 9 – Constraints for class and range coverage

As discussed previously, the issue of building relevant model fragments cannot be resolved with a naive strategy, such as creating all combinations of ranges for all properties of the input meta-model. In general, finding the appropriate combinations of values without a sound knowledge of the transformation under test will be difficult. However, the input meta-model of a transformation itself provides information about the relationships between the properties it contains. In this section we use structural information of the input metamodel to propose a set of adequacy criteria for the construction of model fragments to be covered by the test models.

Test criterion for meta-model coverage *: A test criterion specifies a set of model fragments for a particular input meta-model. These model fragments are built to guarantee class and range coverage as defined in the following rules (formally specified in OCL in Figure 9).*

Rule 1*- Class coverage: Each concrete class must be instantiated in at least one model fragment.*

Rule 2 *-Range coverage: Each range of each partition for all properties of the meta-model must be used in at least one model fragment.*

Test criterion satisfaction for a set of test models*: A set of test models satisfies a test criterion if, for each model fragment MF, there exists a test model M such that all object fragments defined in MF are covered by an object in M. An object O corresponds to an object fragment OF if, for each property constraint in OF, the value for the property in O is included in the range specified by OF.*

The following sections propose several ways of combining constraints in object and model fragments. These criteria all ensure at least the coverage of these two requirements.

4.1. Simple coverage criteria

As a start, this section defines two criteria which both ensure range coverage by combining property constraints in two different manners. The first criterion, *AllRanges* does not add any constraints to the two rules defined in the previous section. The second criterion, *AllPartitions* is a little stronger, as it requires values from all ranges of a property to be used simultaneously in a single test model. Figure 10 formalizes these two criteria in pseudo-code constraints.

***AllRanges* criterion**	`Partition.allIstances()->forAll( P |`
	`  P.range->forAll( R |`
	`  ModelFragment.allInstances->.exists( MF |`
	`  MF.object->size == 1`
	`  MF.object->one.constraint->size == 1`
	`  MF.object->one.constraint->one.range == R`
	`)))`

***AllPartitions* criterion**	`Partition.allInstances()->forAll( P |`
	`  ModelFragment->exists( MF |`
	`  P.range->forAll( R |`
	`  MF.object->exists( OF |`
	`  OF.constraint->size == 1`
	`  OF.constraint->one.range == R`
	`))))`

Figure 10 – Ranges and partitions coverage

In order to illustrate these criteria Figure 11 and Figure 12 present a set of model fragments that would be obtained from the state machine meta-model. The partitions used to create these model fragments are the ones shown in Figure 4. For both criteria the model fragments presented correspond to properties of TRANSITION and ABSTRACTSTATE. In the case of the *AllRanges* criterion (Figure 11), each model fragment is made up of only one object fragment which contains a single constraint. In the case of the *AllPartitions* criterion (Figure 12), a model fragment is created for each property of the meta-model. This model fragment contains one object fragment per range of the partition associated with the property it corresponds to.

```
MF{Transition(event = "")},
MF{Transition(event = "evt1")},
MF{Transition(event ∈ .+)},
MF{Transition(#source = 1)},
MF{Transition(#target = 1)},
MF{AbstractState(label = 0)},
MF{AbstractState(label = 1)},
MF{AbstractState(label ≥ 2)},
...
```

Figure 11 – Model fragments for *AllRanges* criterion

```
MF{Transition(event = ""), Transition(event = "evt1"),
   Transition(event ∈ .+)},
MF{Transition(#source = 1)},
MF{Transition(#target = 1)},
MF{AbstractState(label = 0), State(label = 1), State(label ≥2)},
...
```

Figure 12 – Model fragments for *AllPartitions* criterion

In practice these criteria can be used to create an initial set of model fragments, but in most cases this set of model fragments should be completed either by the tester or by using a stronger criterion.

4.2. Class by class combination criteria

In a meta-model, properties are encapsulated into classes. Based on this structure and on the way meta-models are designed, it is natural that properties of a single class usually have a stronger semantic relationship with each other than with properties of other classes. To take advantage of this, we propose four criteria that combine ranges class by class. These criteria differ on the one hand by the number of ranges combinations they require and on the other hand by the way combinations are grouped into model fragments.

Figure 13 proposes two strategies for combining the ranges of the properties of a class. The first one is quite weak as it only ensures that each range of each property is covered at least once. The second is substantially stronger as it requires one object

fragment for each possible combination of ranges for all the properties of a class. The operation *getCombinations* used for the definition of this strategy simply computes the Cartesian product for ranges of a set of partitions. Its signature is the following: **getCombination** (Set<Partition>) : Set<Set<Range>>

OneRangeCombination each range for each property of a class needs to be used in at least one object fragment	`Range.allInstances()->forAll( R | ObjectFragment.allInstances()->exist( OF | OF.constraint->exists( PC | PC.range = R )))`
AllRangeCombination all possible combinations of ranges for the properties of a class must appear in one object fragment	`Class.allInstances()->forAll( C | getCombinations( Partition.allInstances()->select{ P | C.ownedAttribute->contains(P.property)} )->forAll( RSet | ObjectFragment.allInstances()->exists( OF | RSet->forAll( R | OF.constraint->exists( PC | PC.range == R and PC.property == R.partition.property ))))))`

Figure 13 – Two strategies for ranges combination

OneMFPerClass a single model fragment contains all combinations of ranges for a class	`Class.allInstances()->forAll( C | ModelFragment.allInstances()->select( MF | MF.object->forAll( OF | C.ownedAttribute->size == OF.constraint->size and C.ownedAttribute->forAll( P | OF.constraint->exists( PC | PC.property == P ))))->size == 1)`
OneMFPerCombination each model fragment contains a single combination of ranges for a class	`ModelFragment.allInstances()->forAll( MF | MF.object->size == 1 and Class.allInstances()->exists( C | MF.object->forAll( OF | C.ownedAttribute->size == OF.constraint->size and C.ownedAttribute->forAll( P | OF.constraint->exists( PC | PC.property == P )))))`

Figure 14 – Two strategies to create model fragments

Figure 14 presents the two strategies that we propose to group combinations of ranges class by class. In both cases the idea is to create object fragments that contain constraints related to every property of a class. The two strategies differ in the way these object fragments are organized in model fragments. The first strategy (*OneMFPerClass*) forces grouping of all object fragments related to a class into a single model fragment whereas the second one requires a model fragment for every object fragment.

Based on the strategies for combinations of ranges and for building model fragments, we propose the four test criteria displayed on Figure 15.

Test criteria	Definition
Comb∑	`OneRangeCombination` **and** `OneMFPerCombination`
Comb∏	`AllRangesCombination` **and** `OneMFPerCombination`
Class∑	`OneRangeCombination` **and** `OneMFPerClass`
Class∏	`AllRangesCombination` **and** `OneMFPerClass`

Figure 15 – Four test criteria based on class by class combinations

To illustrate the differences between these four criteria, Figure 16, Figure 17 and Figure 18 present examples of model fragments obtained respectively using the CombΣ criterion, the ClassΣ criterion and the CombΠ criterion. Again, the input meta-model considered is the state machine meta-model. The model fragments represented only corresponds to class ABSTRACTSTATE. This class contains three properties: *label*, *incomingTransition* (inTrans.) and *outgoingTransition* (outTrans.). For each of these properties a partition made of three ranges has been defined on Figure 4. For both CombΣ and ClassΣ criteria the expected value combinations are the same: each range has to be covered once. As shown on Figure 16 and Figure 17 three object fragments have been defined to fulfill this requirement. The difference between the two criteria is the way the object fragments are encapsulated into model fragments. In the case of the CombΣ criterion there is one model fragment per object

fragment and in the case of the ClassΣ criterion there is only one model fragment per class.

```
MF{AbstractState(label=0, #inTrans.=0, #outTrans.=0)},
MF{AbstractState(label=1, #inTrans.=1, #outTrans.=1)},
MF{AbstractState(label≥2, #inTrans.≥2, #outTrans.≥2)},
...
```

Figure 16 – Model fragments for CombΣ criterion

```
MF{
  AbstractState(label=0, #inTrans.=0, #outTrans.=0),
  AbstractState(label=1, #inTrans.=1, #outTrans.=1),
  AbstractState(label≥2, #inTrans.≥2, #outTrans.≥2)
}
...
```

Figure 17 – Model fragments for ClassΣ criterion

CombΠ and ClassΠ criteria differ in the way object fragments are arranged into model fragments, just like the CombΣ and ClassΣ criteria. Figure 18 presents the model fragments obtained for the CombΠ criterion. The 27 model fragments correspond to the 27 combinations of ranges obtained from the three ranges associated to each of the three properties of *AbstractState*.

```
MF{AbstractState(label=0, #inTrans.=0, #outTrans.=0)},
MF{AbstractState(label=1, #inTrans.=0, #outTrans.=0)},
MF{AbstractState(label ≥2, #inTrans.=0, #outTrans.=0)},
MF{AbstractState(label=0, #inTrans.=1, #outTrans.=0)},
MF{AbstractState(label=1, #inTrans.=1, #outTrans.=0)},
MF{AbstractState(label ≥2, #inTrans.=1, #outTrans.=0)},
MF{AbstractState(label=0, #inTrans.≥2, #outTrans.=0)},
MF{AbstractState(label=1, #inTrans.≥2, #outTrans.=0)},
MF{AbstractState(label ≥2, #inTrans.≥2, #outTrans.=0)},

MF{AbstractState(label=0, #inTrans.=0, #outTrans.=1)},
MF{AbstractState(label=1, #inTrans.=0, #outTrans.=1)},
MF{AbstractState(label ≥2, #inTrans.=0, #outTrans.=1)},
MF{AbstractState(label=0, #inTrans.=1, #outTrans.=1)},
MF{AbstractState(label=1, #inTrans.=1, #outTrans.=1)},
MF{AbstractState(label ≥2, #inTrans.=1, #outTrans.=1)},
MF{AbstractState(label=0, #inTrans.≥2, #outTrans.=1)},
MF{AbstractState(label=1, #inTrans.≥2, #outTrans.=1)},
```

```
MF{AbstractState(label ≥2, #inTrans.≥2, #outTrans.=1)},

MF{AbstractState(label=0, #inTrans.=0, #outTrans.≥2)},
MF{AbstractState(label=1, #inTrans.=0, #outTrans.≥2)},
MF{AbstractState(label ≥2, #inTrans.=0, #outTrans.≥2)},
MF{AbstractState(label=0, #inTrans.=1, #outTrans.≥2)},
MF{AbstractState(label=1, #inTrans.=1, #outTrans.≥2)},
MF{AbstractState(label ≥2, #inTrans.=1, #outTrans.≥2)},
MF{AbstractState(label=0, #inTrans.≥2, #outTrans.≥2)},
MF{AbstractState(label=1, #inTrans.≥2, #outTrans.≥2)},
MF{AbstractState(label ≥2, #inTrans.≥2, #outTrans.≥2)},

...
```

Figure 18 – Model fragments for CombΠ criterion

4.3. Criteria and inheritance

The criteria presented in the previous section combine the properties of a single class. However, as we have seen for the state machine meta-model in section 3, it might be necessary to consider inherited properties. For instance, in order to test the transformation that flattens state machines properly it is necessary to ensure that some input models have a pair of composite states that have various numbers of incoming and outgoing transitions. In the meta-model (Figure 2) the only property of COMPOSITE is *ownedState*. The properties *incomingTransition* and *outgoingTransition* are inherited from STATE. This section proposes four criteria based on the same combination principles as the ones defined in the previous section but taking inherited properties into account.

These criteria not only combine ranges for properties owned by each class but also ranges for inherited properties. With this new strategy, we obtain four additional criteria: IF-CombΣ, IF-CombΠ, IF-ClassΣ and IF-ClassΠ defined analogously to the four criteria of previous section (The IF prefix stands for Inheritance Flattening).

Note that using these criteria, it is unnecessary to create model fragments corresponding to abstract classes. Using the previous set of criteria this step was

mandatory to ensure the coverage of the partitions associated to the properties of abstract classes. With the IF criteria, these partitions are implicitly covered for each concrete sub-class of an abstract class.

```
MF{Composite(label=0, #inTrans.=0, #outTrans.=0, #ownedState=0)},
MF{Composite(label=0, #inTrans.=0, #outTrans.=0, #ownedState=1)},
MF{Composite(label=0, #inTrans.=0, #outTrans.=0, #ownedState ≥2)},
MF{Composite(label=0, #inTrans.=0, #outTrans.=1, #ownedState=0)},
MF{Composite(label=0, #inTrans.=0, #outTrans.=1, #ownedState=1)},
MF{Composite(label=0, #inTrans.=0, #outTrans.=1, #ownedState ≥2)},
MF{Composite(label=0, #inTrans.=1, #outTrans.=1, #ownedState=0)},
MF{Composite(label=0, #inTrans.=1, #outTrans.=1, #ownedState=1)},
MF{Composite(label=0, #inTrans.=1, #outTrans.=1, #ownedState ≥2)},
...
```

Figure 19 – Model fragments for IF-CombΠ criterion

Figure 19 presents some model fragments obtained for COMPOSITE using the IF-CombΠ criterion. Only a sub-set of the model fragments is presented. COMPOSITE owns one property and inherits three properties from STATE. For each of the four properties, the associated partition contains three ranges. This leads to a total of 81 (3^4) combinations for COMPOSITE.

4.4. Comparison of the test criteria and discussion

This section concludes the definition of test criteria by comparing the criteria defined in the three previous sections. Figure 20 presents the subsumption relationship between the criteria. A criterion is said to subsume another one if any set of models that satisfies the first criterion satisfies the second criterion. By this definition, the subsume relationship is transitive. The relation allows comparing the strength of the criteria with respect to one another. On Figure 20, criteria *AllRange* and *AllPartition* appear as the weakest ones. Above them the remaining criteria are represented from bottom to top as they consider the encapsulation and inheritance.

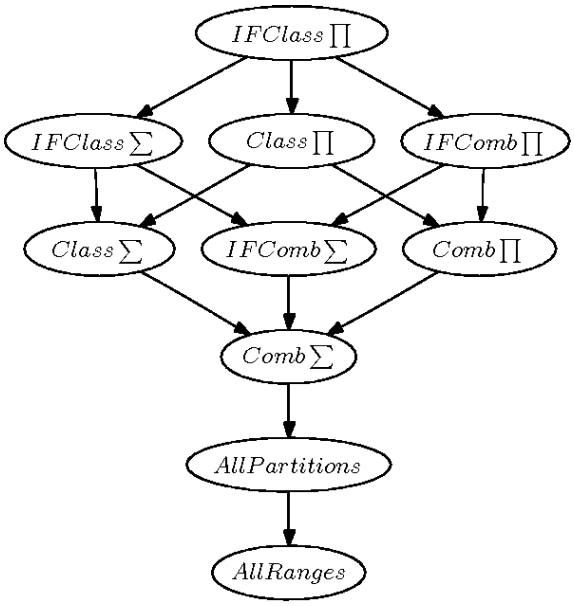

Figure 20 – Topology of the subsumption relationship

To illustrate and compare the criteria presented in the previous sections, Figure 21 gives the minimal number of model fragments, object fragments and property constraints required to verify each criterion with the state machine meta-model based on the partitions shown in Figure 4. The number of model fragments gives an indication of the number of test models and the number of object fragments gives an indication of the size of the test models. As a comparison, the last line of the table corresponds to the naïve Cartesian product strategy (AllCombinations) discussed in section 3.

It is interesting to notice that all the proposed criteria significantly reduce the number of fragments compared to the naive strategy. However, for all criteria that require the Cartesian product on ranges, the number of fragments is still quite high. Future work should investigate and compare the fault detection power of these criteria and may propose more efficient criteria that not only use the information provided by the meta-model but also some knowledge of the model transformation under test.

Test criteria	#Property Constraints	#Object Fragment	#Model Fragment
AllRanges	23	23	23
AllPartitions	23	10	10
Comb∑	28	11	11
CombX	236	64	64
Class∑	28	11	4
ClassX	236	64	4
HF-Comb∑	42	9	9
HF-CombX	2115	381	381
HF-Class∑	42	9	3
HF-ClassX	2115	381	3
AllCombinations	19440	1944	.

Figure 21 – Comparison of test criteria on the state machine example

A general issue with test criteria is that they define some objectives that cannot be satisfied by any test case. For example, structural test criteria for programs specify infeasible paths [13], or mutation analysis produces equivalent mutants [14]. In the same way, the criteria we have defined here may specify uncoverable model fragments (e.g., fragments that violate well-formedness rules). A further investigation would consist of detecting such fragments to remove them from the set of fragments to be covered. In practice, such fragments will have to be identified by the tester. To limit the search effort, the tester can look for the uncoverable fragments in the set of uncovered fragments detected after step three of the process shown in Figure 8.

5. Tool and Experiments

This section introduces MMCC (Metamodel Coverage Checker) the tool that we have implemented to check that a set of models satisfies a test criterion. The implementation uses the Kermeta language and manipulates models stored in an EMF repository. We subsequently present two examples to illustrate how MMCC is used to improve a set of test models.

5.1. Tool

In order to validate the feasibility of the process shown in Figure 8 and to experiment with the test criteria, we have implemented the Metamodel Coverage Checker (MMCC [15]). MMCC automates (1) the generation of partitions from a source metamodel, (2) the generation of model fragments according to a particular test criterion and (3) the qualification of test models with respect to these model fragments. MMCC is implemented using EMF, and Kermeta [16]. Kermeta is well suited as the implementation language for two reasons. First, Kermeta is designed for the manipulation of models and metamodels, and is thus well suited to implement model transformations (steps 1 and 2 are model transformations). Second, and most importantly, Kermeta is an extension of EMOF with an action language, and as such it allows the user to add operations in metamodels that are modeled with EMOF. This feature of Kermeta was very useful in the implementation of MMCC. The Figure 22 displays the metamodel of MMCC. It is very similar to the one shown in Figure 5, extending it with three classes, and a number of operations. This substantially simplifies the implementation of MMCC. For example it is easy to check that the value of a property is contained in a range by invoking the operations of RANGE. The additional classes PARTITIONMODEL and FRAGMENTS were added as top-level 'containers' for the model, as customary in EMF. PROPERTYPARTITION was added to decouple the MMCC metamodel from EMOF. This class contains the names for the feature and class for which a partition is defined. This prevents us from keeping a direct relationship to the actual feature in the input metamodel.

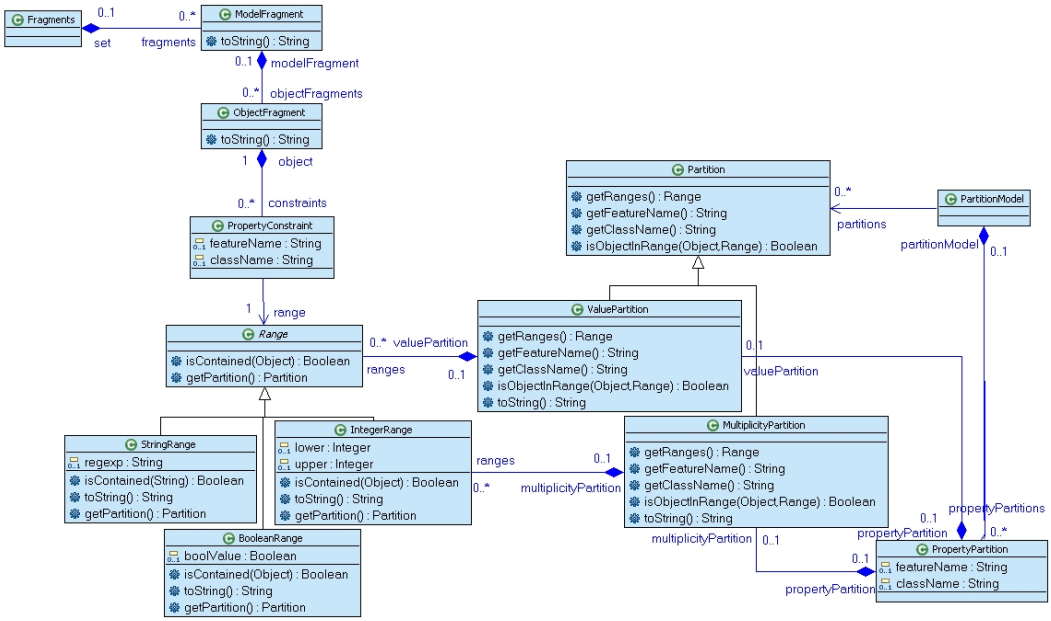

Figure 22 - Metamodel for the Metamodel Coverage Checker tool

As described in Figure 23, MMCC is composed of two separate programs. The first one generates the partitions and model fragments and the second one checks if a set of models satisfies a test criterion.

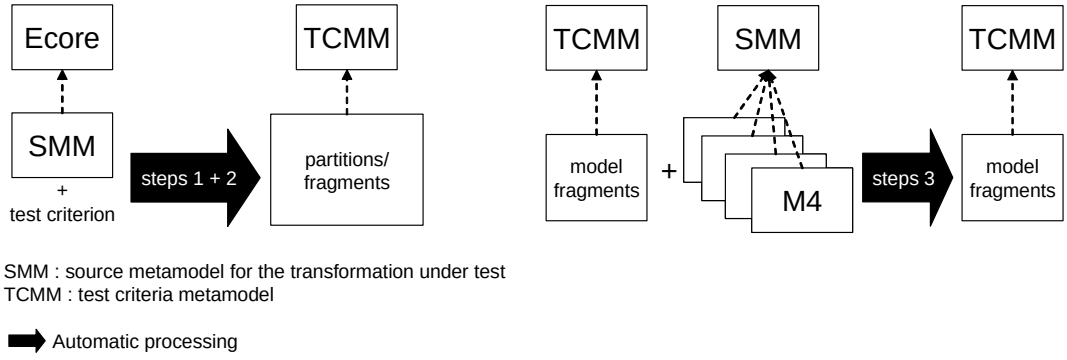

Figure 23 - Tools chain for the MMCC

The first part of MMCC realizes steps 1 and 2 of the process of Figure 8. It is implemented as a model transformation that takes two input parameters: the source metamodel (SM) of the transformation under test, and a test criterion. The test criteria

have been defined using a hierarchy of classes in the implementation of the tool. Because the tool is implemented in EMF, the source metamodel must conform to Ecore, the meta-meta-model of EMF. The output is a model containing a set of partitions and fragments that conform to the test criteria metamodel. In practice this transformation is divided into two model transformations that are executed sequentially. The first one processes the source metamodel SM and generates a set of default partitions and ranges. The second one generates a set of model fragments using the partitions and ranges according to the test criterion that has been selected.

The second part of MMCC implements step 3 of the process to check if a set of models satisfies the test criterion. It takes two input parameters: a set of models that conform to the source metamodel SM and the set of model fragments produced in steps 1 and 2. The output is the set of model fragments that are not covered by the set of test models. If it is empty, the set of models satisfies the test criterion. Otherwise, it is necessary to manually analyze the remaining fragments to understand why they are not covered.

5.2. Discussion with the state machine metamodel

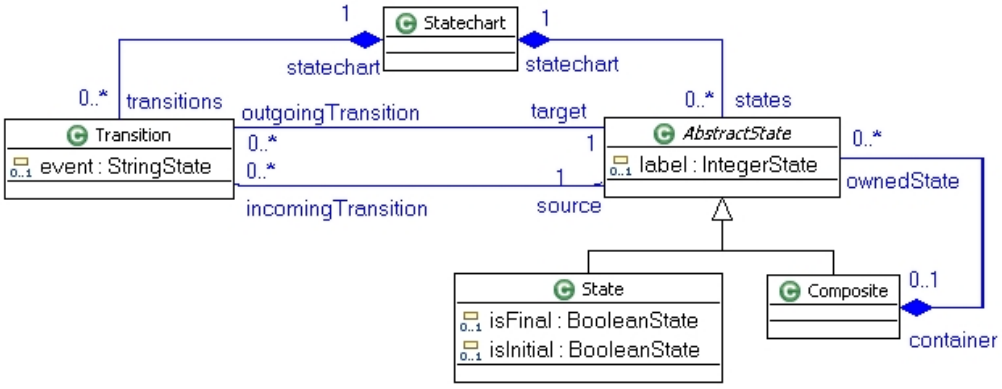

Figure 24 - State machine metamodel in EMF

This section illustrates the use of the tool for the improvement of a set of test data. We illustrate this discussion with the SMFlatten transformation. To test this transformation, it is necessary to build a set of state machines according to the metamodel of Figure 24. Since MMCC is implemented in EMF, all classes of the metamodel must be contained in a "root" class. This explains the presence of STATECHART that is not in the metamodel of Figure 2, and the presence of PARTITIONMODEL and FRAGMENTS in Figure 22. Figure 25 displays a possible set of test models as an example. It contains three state machines, each having specific characteristics: the first one has only one transition, the second one has two transitions and the third one has a composite state. Now, let us use MMCC to evaluate the quality of this set of models according to the *AllRanges* and *AllPartitions* test criteria.

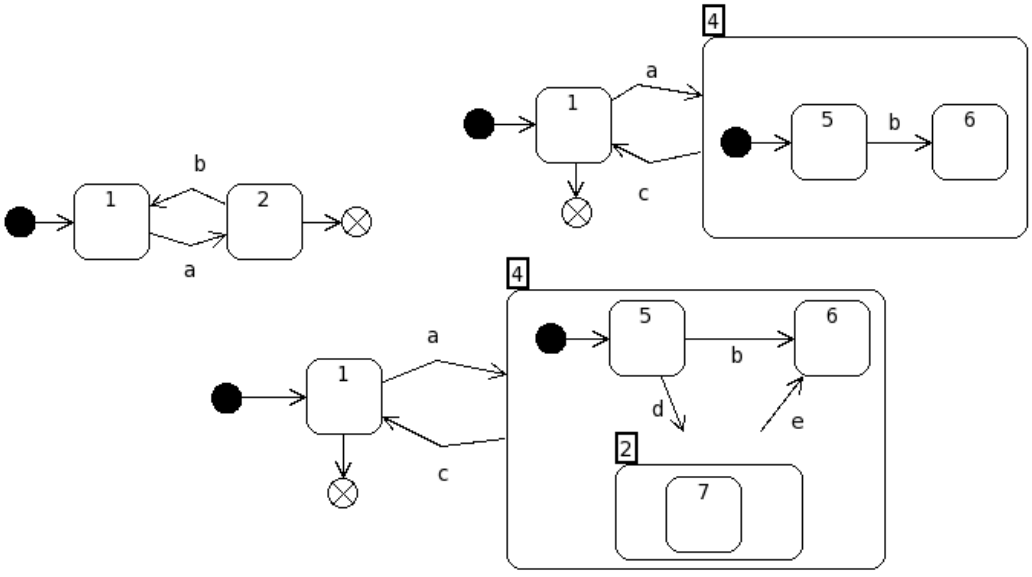

Figure 25 - Test models for the SMFlatten transformation

We start with the *AllRanges* test criterion. When MMCC runs with the metamodel of Figure 24 and the *AllRanges* criterion, it generates 30 model fragments. When we run the second stage of MMCC with the three test models on these 30 model fragments, it computes that 25 fragments are covered. MMCC also provides the 5 exposed model fragments:

```
MF{AbstractState(label = 0)}
MF{Composite(ownedState = 0)}
MF{Statechart(transitions = 0)}
MF{Statechart(states = 0)}
MF{Statechart(states = 1)}
```

When looking at the set of exposed model fragments, it appears that the initial set of test models misses some boudary cases. In order to satisfy the *AllRanges* criterion, it is necessary to add corresponding test models in the set. When adding the two test models shown in Figure 26, the criterion is completely satisfied. All model fragments are covered. The first model is a state machine with only an empty composite state and the second one is an empty state machine (no state, transition or composite state).

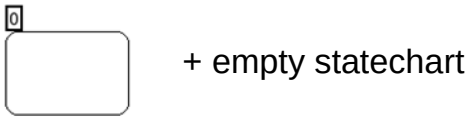

Figure 26 - Two models added to satisfy the AllRanges criterion

Now, we look at the *AllPartitions* test criterion. When MMCC runs with the metamodel of Figure 24 and the *AllPartitions* criterion, it generates 14 model fragments. When running the MMCC's second stage with the three test models on these 14 model fragments, the result of the analysis is that 9 fragments are covered. MMCC provides the 5 exposed model fragments:

```
MF{AbstractState(label = 0), AbstractState (label = 1),
AbstractState(label ≥2)}
MF{Composite(ownedState = 0), Composite(ownedState = 1),
Composite(ownedState ≥2)},
MF{Transition(event = ""), Transition(event ∈ .+)}
MF{Statechart(transitions = 0), State(transitions = 1),
State(transitions ≥2)}
MF{Statechart(states = 0), State(states = 1), State(states ≥2)}
```

In order to improve the initial set of test models according to the *AllPartitions* criterion, we add the model displayed in Figure 27. Again, these data miss boudary cases: a composite with no `ownedState` and a transition with no event label. When running MMCC again with this additional model in the set of test models, 12 fragments are covered. However, the following fragments are still exposed:

```
MF{Statechart(transitions = 0), State(transitions = 1),
State(transitions ≥2)}
MF{Statechart(states = 0), State(states = 1), State(states ≥2)}
```

When looking at the missing model fragments, it appears that they can not be covered by any test model. It is impossible to build a state machine that has 0 transition AND 1 transition, or a state machine that has exactly 1 state AND more than 2 states. This illustrates a limitation of almost all test adequacy criteria: they generate constraints that can not be satisfied. The criteria have the same limitation, but they are still useful to improve a set of test data. The model in Figure 27 is an interesting case that was not present in the initial set.

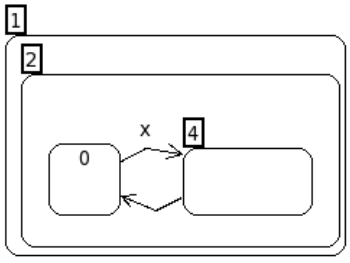

Figure 27 - Additional test model to satisfy the AllPartitions criterion

When looking at these two results we notice that in both cases the missing model fragments concern boundary cases that are usually forgotten when generating test data. We also notice that the model that we added to satisfy the *AllPartitions* criterion is more complex than the models added to satisfy the *AllRanges* criterion. This

confirms that the *AllPartitions* criterion which is stronger than the other one (according to the subsume relationship) also leads to the identification of more complex data and thus should improve the quality of testing. Indeed, checking that the transformation runs correctly with complex input data improves our confidence in the transformation more than if it runs with small simple test models.

5.3. Case Study

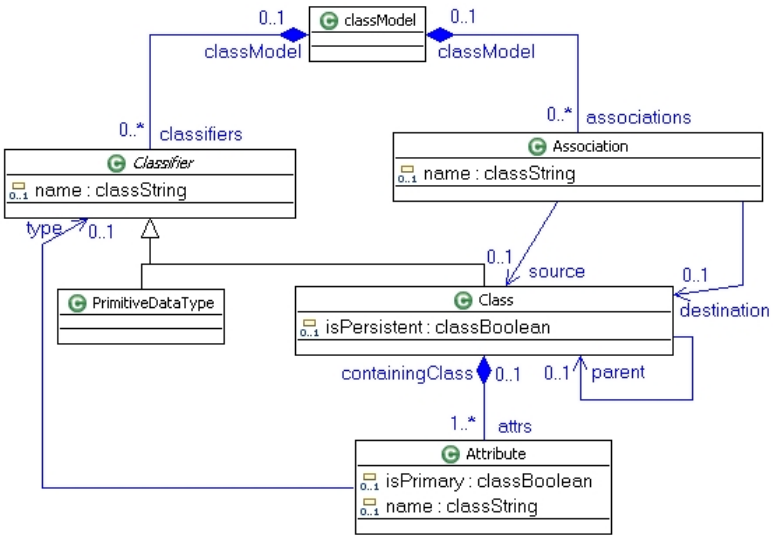

Figure 28 – Meta-model for classes

In this section we illustrate another application of MMCC with another model transformation example that was proposed at the Model Transformation in Practice (MTIP) workshop at the Models'05 conference [17]. As a benchmark for transformation techniques, the organizers of the workshop provided a precise specification of several model transformations. The first one consisted in transforming class diagrams into database tables. The input meta-model is given in Figure 28. The organizers of the workshop also provided an example for this transformation that gave a class model in the form of a graph of objects and the

expected output database. For illustration purposes, we consider this class model (Figure 29) as a test model for the transformation.

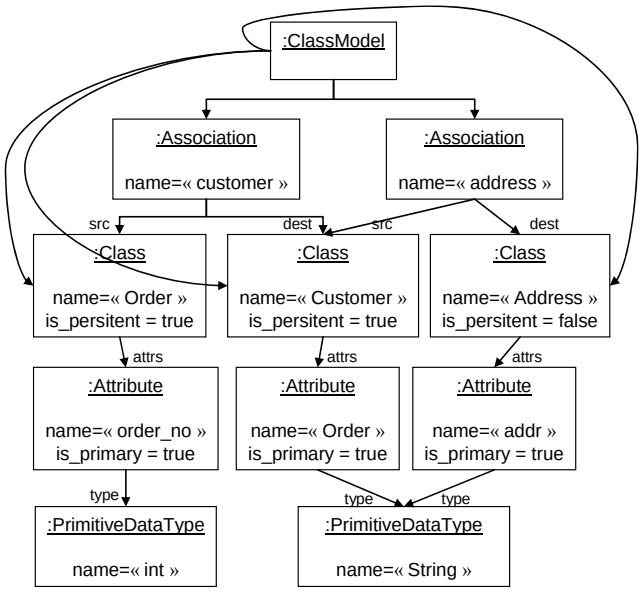

Figure 29 – One source model

It appears that the model does not satisfy any of the coverage criteria defined in the previous sections, not even the simplest *AllRanges*. The test model misses 15 ranges for several properties of the model:

```
Classifier(name = “”),
Class(isPersistent = false), Class(parent = 1), Class(attrs = 0),
Class(attrs ≥ 2)
Attribute(isPrimary = false), Attribute(name = “”), Attribute(type =
0),
Association(name = “”), Association(destination = 0),
Association(source = 0)
classModel(classifiers = 0), classModel(classifiers = 1),
classModel(associations = 0), classModel(associations = 1)
```

This means that for any test criterion, there will be uncovered model fragments. When running MMCC with the *AllRanges* criterion, 33 model fragments are generated (each one containing one range) and 15 fragments are exposed. The five models of Figure 30 can be added to the set of test models to satisfy the *AllRanges* criterion.

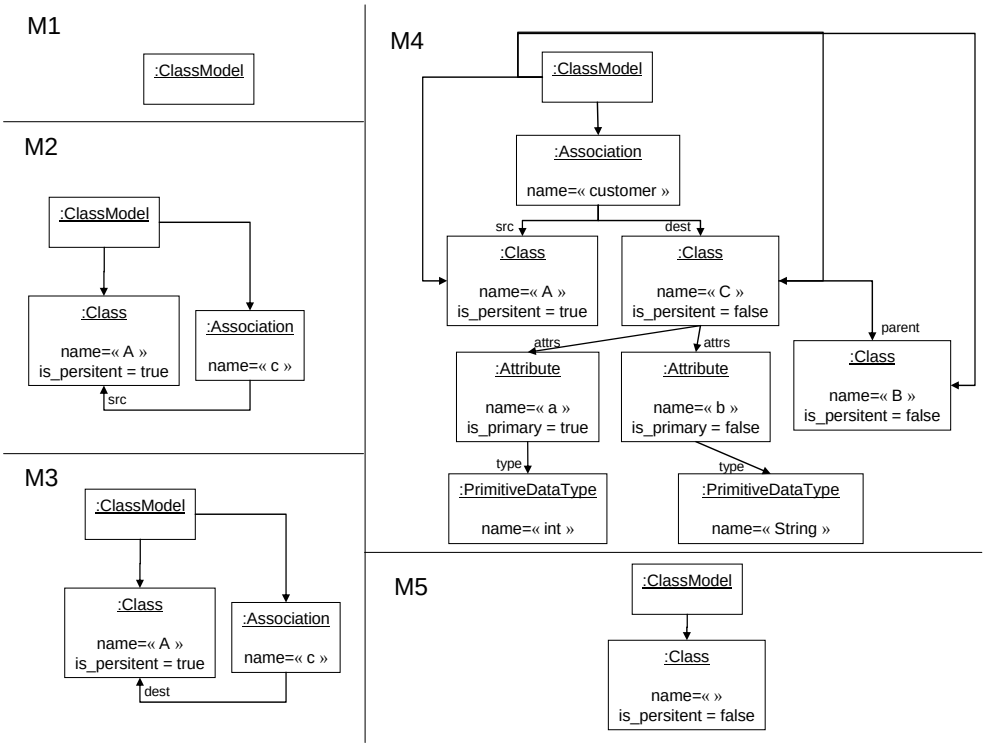

Figure 30 - Fives models to cover the Allranges cirterion

When running MMCC with the *AllPartitions* criterion, none of the 15 model fragments is found to be covered by the test model. The information on exposed ranges and model fragments helps testers to improve the test models, while providing them with flexibility in the design of the test suite: the tester may opt to add all possible cases to a single model or to create several models focusing on particular fragments. It should be noted that the framework currently does not enforce the requirement that the *type* of the *Attribute* is never a *Class*.

6. Related work

Model transformations are the essential feature in model-driven development (MDD). However, works concerned with the validation of these pivotal programs are just

beginning to emerge. This section presents the state of the art on model transformation validation and then broadens the scope to look at other works that deal with validation issues in MDD.

Steel et al. [18] present testing issues encountered when developing a model transformation engine, and the solutions adopted to cope with them. They note the similarity between the task of testing the transformation engine and the testing of transformations themselves, and address a number of important technical issues associated with using models as test data. In particular, they discuss the use of coverage criteria based on metamodels for the generation of test data. In their study, the criteria are applied by hand, and not in the systematic, generalized way presented in this work.

Küster [19], addresses the problem of model transformation validation in a way that is very specific to graph transformation. He focuses on the validation of the rules that define the model transformation with respect to termination and confluence. His approach aims at ensuring that a graph transformation will always produce a unique result. Küster's work is an important step for model transformation validation, but contrary to the approach presented here, it does not aim at validating the functionality of a transformation (*i.e.*, it does not aim at running a transformation to check if it produces a correct result). Küster et al. [20] also consider graph transformation rules, but in the paper they leverage the specificities of the implementation to propose a white-box testing approach. First, they propose a template language which they use to define generic rules that can be used to automatically generate a set of rules that serve as input data. The second contribution of this work consists in identifying model elements that are transformed and that are also manipulated by constraints on the model. In this way, they identify constraints that might be violated after the transformation and they build test data that aims at validating that these constraints are not violated. Darabos et. al [21] also investigate the testing of graph

transformations. They consider graph transformation rules as the specification of the transformation and propose to generate test data from this specification. Their testing technique focuses on testing pattern matching activity that is considered the most critical of a graph transformation process. They propose several fault models that can occur when computing the pattern match as well as a test generation technique that targets those particular faults.

Baldan et al. [22] propose a technique for generating test cases for code generators. The criterion they propose is based on the coverage of graph transformation rules. Their approach allows the generation of test cases for the coverage of both individual rules and rule interactions but it requires the code generator under test to be fully specified with graph transformation rules. Heckel et al. [23] apply the same ideas for automatically generating conformance tests for web services. One of their contributions is to apply partition testing on the WSDL specification of the inputs of the web services under test in order to select the test data.

All these approaches to model transformation validation and testing consider a particular technique for model transformation and leverage the specificities of this technique to validate the transformation. This has the advantage of having validation techniques that are well-suited to the specific faults that can occur in each of these techniques. The results of these approaches are difficult to adapt to other transformation techniques (that are not rule-based). None of these approaches has proposed a clear and precise model for the definition of test criteria of test data generation and qualification. In this paper, we have considered a very generic approach for model transformation and have proposed a framework to express test criteria to test any transformation, based on its source meta-model.

Mottu et al. [24], propose a methodology to evaluate the trust in a model transformation by qualifying the efficiency of test cases and contracts that specify the transformation. The idea of this approach is that the quality of a transformation can

be evaluated by checking the consistency between the test data, the implementation of the transformation and the specification of the transformation refined as executable contracts. The consistency between those aspects is measured using mutation analysis [25]. This technique consists of seeding faults in a program and checking whether the test data or the contracts can detect them. In these study, the faults that were injected were specific faults that typically occur when developing a model transformation. Mottu et al. [26] analyzed the process of model transformation to define generic fault models and showed how these generic faults mapped to actual faults in different languages.

Giese et al [27] focuses on the formal verification of model transformation critical properties. The authors use triple graph grammars (TGG) to represent both systems and transformations in order to formalize transformations and allow critical properties to be verified through the use of a theorem prover. Such a verification technique could be used in combination with testing when a formal specification is available.

Although there are few works that focus on model transformation testing or verification, a number of other topics are connected to this filed of research. In particular, we discuss testing of compilers and model validation in the following.

The problem of model transformation can be connected to existing work on testing compilers. Compilers are specific transformations which translate programs written in a particular programming language into another programming language. The input domain of a compiler is usually specified using grammars and test data (programs fed as input to the compiler) are represented as trees. Like for model transformations the correctness of compilers is critical to the reliability of programs that are developed using them. Borjarwah et al. [28] present a survey of existing compiler test case generation methods. Most of these testing techniques propose to use the grammar for selecting test programs. In previous work [29], we have used such techniques for generating tests for parsers. There are very few recent works on the generation of test

data for compilers. Currently, compilers are verified and tested using ad-hoc and very specific techniques and with large benchmarks developed on a long period of time. This is a very costly approach for validation but it is acceptable for compilers for two reasons: first, compilers and programming languages have a slow evolution rate and second they are used by a large community of developers.

Such an approach can not be used for model transformation testing and validation because they typically target a smaller community and require constant evolutions and adaptations.

Another important issue related to model transformation validation is model validation. The works in [12, 30] develop approaches to support model testing. These works focus on testing executable UML design models. The models they consider for testing consist of a class diagram, OCL pre and post conditions [31] for methods and activity diagrams to specify the behaviour of each method. From this model, the authors generate an executable form of the model, which can be tested. In [30], they propose to model test cases using UML2.0 sequence diagrams. From these test cases specifications and the class diagram, they generate a graph that corresponds to all possible execution paths defined in the different scenarios. The authors then use test criteria defined in [12] and automatically generate test data and an initial system configuration to cover each execution path.

Gogolla et al. [32] propose an approach that detects errors in the early development stages of UML model development. The authors present the USE tool which aims at animating and testing UML class diagrams and their associated OCL constraints. In a USE specification, OCL constraints specify invariants on the structure of the system as well as the behaviour of the methods. The authors define the notion of a snapshot for testing UML designs. A snapshot is an object diagram that represents system states at any time with objects, attribute values and links. Snapshots can be declaratively defined using a language called ASSL. In USE, a test consists of

defining a snapshot that represents an expected object configuration and then checking that it can actually be constructed without violating any model-immanent OCL constraints in the process.

Rutherford et al. [33] report on an experiment to generate test code in parallel with a system whose development is model-driven. The experiment uses a generative programming tool called MODEST. The paper reports the costs and benefits of developing additional templates for test code for the MODEST tool, so it can generate as much test code as possible. The study reported that developing templates for test code enhanced the development process and benefited the developers by increasing their familiarity with the code generation approach used by MODEST. The costs are evaluated with an analysis of the complexity of templates for test-code generation.

Heckel et al [34] also explicitly address the problem of test generation in a MDA context and propose to develop model-driven testing. In particular, their work focuses on the separation of platform-independent models and platform-specific models for testing. The generation of test cases from the model, as well as the generation of an oracle are considered to be platform-independent. The execution of the test cases in the test environment is platform-specific. A case study based on model-driven development of web applications illustrates their approach.

The last important research field that is very much related to our work is the area of model-based testing [35, 36]. Although it is very difficult to assimilate model-based testing to a single homogeneous set of works, we can compare some important trends in model-based testing with the approach proposed in this work. The book by Utting and Legeard [36] identifies four main approaches known as model-based testing. The first one is the “generation of test input data from a domain model”. In that case, the approach presented in this paper is clearly model-based testing. Our approach for model transformation testing uses a model i) to generate objectives for test data

(model and object fragments), ii) to evaluate the quality of test data and iii) it should be possible to use the same model to automatically generate test data. The model used to drive all these testing activities is the input metamodel for the transformation.

As pointed out by Utting and Legeard, this approach is only one specific approach to model-based testing. They cite three other well-known approaches, among which "generation of test cases and oracles form a behaviour model". Their book focuses mainly on this approach. We believe that this is also the most commonly accepted definition of model-based definition. In that context, the approach proposed in this paper is not model-based testing since the model we consider is not behavioural and does not model the system under test, but specifies the input domain of the transformation.

7. Conclusion and future work

At the heart of model-driven engineering are model transformations. Transformations are complex programs; they must be made reliable to have model-driven engineering deliver its promises. Therefore, as for any complex program, thorough testing is required to gain confidence in model transformations.

Testing transformations is typically performed by applying a transformation to a set of input models, and then by comparing actual results with expected results. Defining the right set of input models is a non-trivial task.

In this paper we defined test adequacy criteria to qualify input test models for model transformation testing. These criteria are based on the partitioning of metamodel properties domains. Each criterion defines a set of model fragments that has to be covered by input test models. The notions of partition for properties and model fragments are formally captured in a metamodel. Since these criteria only rely on the structure of the input metamodel of the transformation under test, they can be applied to validate test data for transformations implemented in any language.

We have developed a prototype tool that computes a set of model fragments for any metamodel, according to a particular criterion. For a set of input models, the tool can identify the fragments that remain to be covered. This is valuable information that allows the tester to incrementally improve the initial set of test models. We used a transformation from the MTIP workshop at MoDELS'05 as an illustration of the test improvement process.

There is still a lot of research work ahead us, most importantly in the area of criteria definition and tuning. First, it is important to validate the proposed criteria with respect to their ability to produce test models that can detect faults in a model transformation. This is immediate future work that will consist in generating test models that satisfy the criteria and run a mutation analysis to evaluate their fault detection capabilities. The mutation analysis could be done using the operators defined by Mottu [26] for imperative transformation languages or by Darabos [21] for rule-based transformation languages.

An important perspective of this work consists in automating the generation of test models that satisfy the criteria. A first experiment is proposed in [37]. This work proposes an algorithm that processes a set of model fragments, generated with a particular criterion, to generate a set of test models. The algorithm proposes several strategies to build objects from model fragments and combine those objects into a complete model. This algorithm shows that it is possible to automatically generate models that satisfy a test criterion. However, there are still some limitations: the major difficulty is to generate models that satisfy all constraints that can be placed on the input metamodel, that is well-formedness rules of the metamodel itself, plus the pre conditions of the transformation. A possible solution to this problem is to adapt constraint-solving techniques and evolutionary techniques to take all these constraints (metamodel, constraints, pre-condition and test criterion) into account for efficient model synthesis. A first step in that direction is presented in [38] in which Sen et. al

define mutation operators to build models incrementally by applying small mutations on models to create new models.

Another important future work consists in having more precise criteria to capture more information that is necessary for automatic generation. This work proposes black-box criteria to evaluate the adequacy of test models. This has the benefit of having a solution which is independent of any transformation technology and that leverages the input metamodel as a description of the input domain. However, the limitation of these criteria is that, alone, they will not enable the automatic generation of test models. This is because a large amount of information is not present in the metamodel alone. To design a fully automated test model generator, it is necessary to analyze information from the specification of the transformation and to use white-box criteria that will ensure the coverage of the transformation. This raises two important future work: i) investigate what could be a good language for specification of requirements for transformation and how theses requirements could be analyzed for test generation; ii) propose white-box criteria for specific model transformation techniques as proposed in the work of Küster [20].

8. References

1. R. France and B. Rumpe, *Model-Driven Development of Complex Software: A Research Roadmap*, in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Editors. 2007, IEEE - CS Press.
2. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. *Challenges for Model Transformation Testing*. In Proceedings of *IMDT workshop in conjunction with ECMDA'06*. Bilbao, Spain, 2006.
3. B. Beizer, *Black-Box Testing*. John Wiley & Sons ed. 1995: Wiley.
4. OMG. *MOF 2.0 Q/V/T OMG Revised submission*. 2005. Available from: http://www.omg.org/cgi-bin/doc?ad/05-03-02.
5. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. *Weaving executability into object-oriented meta-languages*. In Proceedings of *MoDELS'05*, p. 264 - 278. Montego Bay, Jamaica, October 2005.
6. K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. *Model Transformation: A declarative, reusable patterns approach*. In Proceedings of *EDOC'03 (Entreprise Distributed Object Computing Conference)*, p. 174 - 185. Brisbane, Australia, September 2003.
7. J. de Lara and H. Vangheluwe. *AToM3: A Tool for Multi-formalism and Meta-modelling*. In Proceedings of *FASE '02 (International Conference on Fundamental Approaches to Software Engineering)*, p. 174--188, 2002.

8. T.J. Ostrand and M.J. Balcer, *The category-partition method for specifying and generating functional tests*. Communications of the ACM, 1988. **31**(6): p. 676 - 686.
9. OMG. *Meta-Object Facilities*. 2007. Available from: http://www.omg.org/mof/.
10. OMG. *MOF 2.0 Core Final Adopted Specification*. 2004. Available from: http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
11. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose, *Eclipse Modeling Framework*. 2003: Addison Wesley Professional.
12. A. Andrews, R. France, S. Ghosh, and G. Craig, *Test adequacy criteria for UML design models*. Software Testing, Verification and Reliability, 2003. **13**(2): p. 95 -127.
13. W.R. Adrio, M.A. Branstad, and J.C. Cherniavsky, *Validation, Verification, and Testing of Computer Software*. ACM Computing Surveys, 1982. **14**(2): p. 159 - 192.
14. A.J. Offutt and J. Pan, *Automatically Detecting Equivalent Mutants and Infeasible Paths*. Software Testing, Verification and Reliability, 1997. **7**(3): p. 165 - 192.
15. MMCC. *Metamodel coverage checker*. 2007. Available from: http://www.irisa.fr/triskell/Softwares/protos/MMCC/.
16. Kermeta. *The KerMeta Project Home Page*. 2005. Available from: http://www.kermeta.org.
17. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. *MTIP workshop*. 2005. Available from: http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf.
18. J. Steel and M. Lawley. *Model-Based Test Driven Development of the Tefkat Model-Transformation Engine*. In Proceedings of *ISSRE'04 (Int. Symposium on Software Reliability Engineering)*. Saint-Malo, France, November 2004.
19. J.M. Küster, *Definition and Validation of Model Transformations*. Software and Systems Modeling, 2006. **5**(3): p. 233 - 259.
20. J.M. Küster and M. Abd-El-Razik. *Validation of Model Transformations - First Experiences using a White Box Approach*. In Proceedings of *MoDeVa'06 (Model Design and Validation Workshop associated to MoDELS'06)*. Genova, Italy, October 2006.
21. A. Darabos, A. Pataricza, and D. Varro. *Towards Testing the Implementation of Graph Transformations*. In Proceedings of *GT-VMT workshop associated to ETAPS'06*, p. 69 - 80. Vienna, Austria, April 2006.
22. P. Baldan, B. König, and I. Stürmer. *Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems*. In Proceedings of *ICGT 2004*, p. 194-209, 2004.
23. R. Heckel and L. Mariani. *Automatic Conformance Testing of Web Services*. In Proceedings of *FASE 2005*, p. 34-48, 2005.
24. J.-M. Mottu, B. Baudry, and Y. Le Traon. *Reusable MDA Components: A Testing-for-Trust Approach*. In Proceedings of *MoDELS'06*. Genova, Italy, October 2006.
25. R. DeMillo, R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer*. IEEE Computer, 1978. **11**(4): p. 34 - 41.
26. J.-M. Mottu, B. Baudry, and Y. Le Traon. *Mutation Analysis Testing for Model Transformations*. In Proceedings of *ECMDA'06 (European Conference on Model Driven Architecture)*. Bilbao, Spain, July 2006.
27. H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. *Towards Verified Model Transformations*. In Proceedings of *MoDeVa workshop associated to MoDELS'06*, p. 78--93, October 2006.
28. A.S. Boujarwah and K. Saleh, *Compiler test case generation methods: a survey and assessment*. Information and Software Technology, 1997. **39**(9): p. 617-625.
29. B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, *From Genetic to Bacteriological Algorithms for Mutation-Based Testing*. Software Testing, Verification and Reliability, 2005. **15**(1): p. 73-96.
30. T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. *A Tool-Supported Approach to Testing UML Design Models*. In Proceedings of *ICECCS'05*. Shanghai, China, June 2005.
31. OMG. *UML 2.0 Object Constraint Language (OCL) Final Adopted specification*. 2003. Available from: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.
32. M. Gogolla, J. Bohling, and M. Richters, *Validating UML and OCL Models in USE by Automatic Snapshot Generation*. Software and Systems Modeling, 2005. **4**(4): p. 386-398.
33. M.J. Rutherford and A.L. Wolf. *A case for test-code generation in model-driven systems*. In Proceedings of *The second international conference on Generative programming and component engineering*, p. 377 - 396. Erfurt, Germany, September 2003.

34. R. Heckel and M. Lohmann, *Towards Model-Driven Testing.* Electronic Notes in Theoretical Computer Science, 2003. **82**(6).
35. AGEDIS. *Automated Generation and Execution of Test Suites for DIstributed Component-based Software*. 2000. Available from: http://www.agedis.de/index.shtml.
36. M. Utting and B. Legeard, *Practical Model-Based Testing*. 2007: Morgan Kaufmann.
37. 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 Proceedings of *ISSRE'06 (Int. Symposium on Software Reliability Engineering)*. Raleigh, NC, USA, 2006.
38. S. Sen and B. Baudry. *Mutation-based Model Synthesis in Model Driven Engineering*. In Proceedings of *Mutation'06 workshop associated to ISSRE'06*. Raleigh, NC, USA, November 2006.