Validation in Model-Driven Engineering: Testing Model Transformations

Franck Fleurey, Jim Steel, Benoit Baudry
IRISA, Campus universitaire de Beaulieu 35042 Rennes cedex France
{ffleurey, jsteel, bbaudry}@irisa.fr

Abstract

The OMG's Model-Driven Architecture is quickly attracting attention as a method of constructing systems that offers advantages over traditional approaches in terms of reliability, consistency, and maintainability. The key concepts in the MDA are models that are related by model transformations. However, for the MDA to provide an adequate alternative to existing approaches, it must offer comparable support for software engineering processes such as requirements analysis, design and testing. This paper attempts to explore the application of the last of these processes, testing, to the most novel part of the MDA, that of model transformation. We present a general view of the roles of testing in the different stages of model-driven development, and a more detailed exploration of approaches to testing model transformations. Based on this, we highlight the particular issues for the different testing tasks, including adequacy criteria, test oracles and automatic test data generation. We also propose possible approaches for the testing tasks, and show how existing functional and structural testing techniques can be adapted for use in this new development context.

1 Introduction

In 2001 the Object Management Group (OMG) introduced the Model-Driven Architecture (MDA) [19] as the framework for its future standardisation efforts. Since this time, the idea has been widely discussed as an emerging technique for the development of large-scale software systems.

In essence, MDA proposes a move away from human interpretation of high-level models, such as design diagrams, into implementations, towards a more automated process where the models are used as first-class artifacts of the development process.

This potentially represents a significantly different approach to the development of software systems, a practice that has become increasingly well understood over time. However, while much work has been done on techniques for using MDA for software development, there remain many challenges for the process of software validation, and in particular software testing, in an MDA context.

In MDA, the most important behavioural artifact, and thus the most important artifact from a testing perspective, is that of the model transformation, which describes a relationship between two or more models. Therefore, model transformations are the key to automating the transition between models and from models into other forms, such as code. MDA makes many promises about providing a more efficient, more consistent, and more reliable approach to software development, but this can only be true if the model transformations behave like they are specified.

There are a number of limitations in using existing testing techniques to test model transformation programs. The data structures used by model transformations are complex, and generating test data using traditional techniques is unwieldy and inefficient. Thus, in the same way that testing techniques have been adapted to suit the emergence of object-oriented programming, the adaptation of techniques to better suit model-driven engineering will also allow for better and more appropriate methods of validation. For example, MDA has normalized the languages for defining models and meta-models, and this allows for the development of generalized tools and techniques for testing systems that previously had no commonly understandable definition.

This document is organized as follows. Firstly, section 2 briefly recalls some background on MDA and presents how model transformation programs are specified and implemented. Then, section 3 details the motivations of this work and the problems related to testing model transformations. Section 4 presents an adaptation of existing testing techniques to the model-oriented context and proposes a functional test adequacy criterion for the validation of model transformation programs. Section 5 then investigates a white-box refinement of this criterion. Section 6 details two automated test data generation techniques based on a systematic and an evolutionary algorithm to cover the previously proposed test criterion. Section 7 details some related works on MDA and testing and, finally, section 8 concludes this document.

2 Background on MDA

The goal of MDA is to move away from the traditional role of UML diagrams as blueprints for conversion into software by programmers, to a situation in which the models are used as first-class development artifacts that are automatically mapped to other models and to system code. Thus, MDA presents a more general view of systems composed of models, the relationships between them, in addition to technology mappings, such as code generators, where the connections between these elements are managed automatically.

As it was the primary motivation behind MDA, the domain of software development remains the principal application of MDA. This application is often referred to as MDE (Model Driven Engineering) [7], and its principles are represented in Figure 2, where the artifacts marked "M" are models, those marked "L" are languages, "T" are transformations, and "S" are specifications. So, after the developer has created a design in the form of a series of models, she then uses model transformations to successively refine these models, and eventually to translate them into code. The transformations are developed in some transformation language (which forms part of an overall MDA framework), and are written by a transformation developer, who may or may not be the same as the software developer. In this way, the development load has been split between the software developer and the transformation developer.

To validate the systems being developed, the software developer (transformation user) may use existing techniques, since in the end her software is implemented using existing technologies. However, she can gain even more confidence in her system by ensuring that the model transformations, which she has used in creating her system, are also validated. Thus, in the same way that the use of transformations can assist the process of developing the software, validating the transformations can assist the process of validating the software.

The following sections present a more thorough examination of the definitions of models, meta-models and model transformations. This includes the presentation of an example that will be used throughout the paper.

2.1 Models and meta-models

A model is a collection of objects and relationships between the objects that, together, provide a representation of some real system. For example, a simple relational database schema of tables and columns, such as the one in (1) of Figure 3, might be represented as objects representing the tables and objects representing the columns, with containment relationships between the table objects and the column objects. Such a model is shown in (2) of Figure 3.

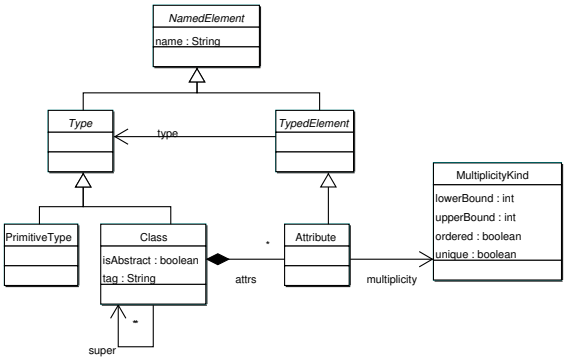

Figure 1. Extract of UML Meta-model

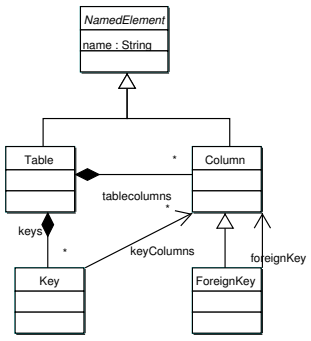

Figure 4. Meta-model for RDBMS Schemas

Of course, not all models are the same. For example, object-oriented models are expressed not using tables and columns, but using packages, classes and attributes. So an object-oriented system such as the one described by the UML diagram in (3) of Figure 3, can be represented by objects such as those shown in (4).

This represents a difference in modeling languages. The database schema is described using a modeling language with concepts of table and column, whereas the OO schema is described using a modeling language such as UML with concepts of class, attribute and package. Beginning with these terms, we can define a model to describe each modeling language, with classes for each of the terms used in the modeling language. These models of modeling languages are called meta-models. Figures 1 and 4 show the meta-models of the UML and RDBMS models presented earlier, simplified for demonstration purposes.

Of course, these meta-models must also be described by some language and, in MDA, this language is that of the Meta-Object Facility (MOF) [13]. As a language, the MOF very closely resembles UML class modeling, with packages, classes, attributes and associations. The relationship between the MOF, meta-models, and models, is com-

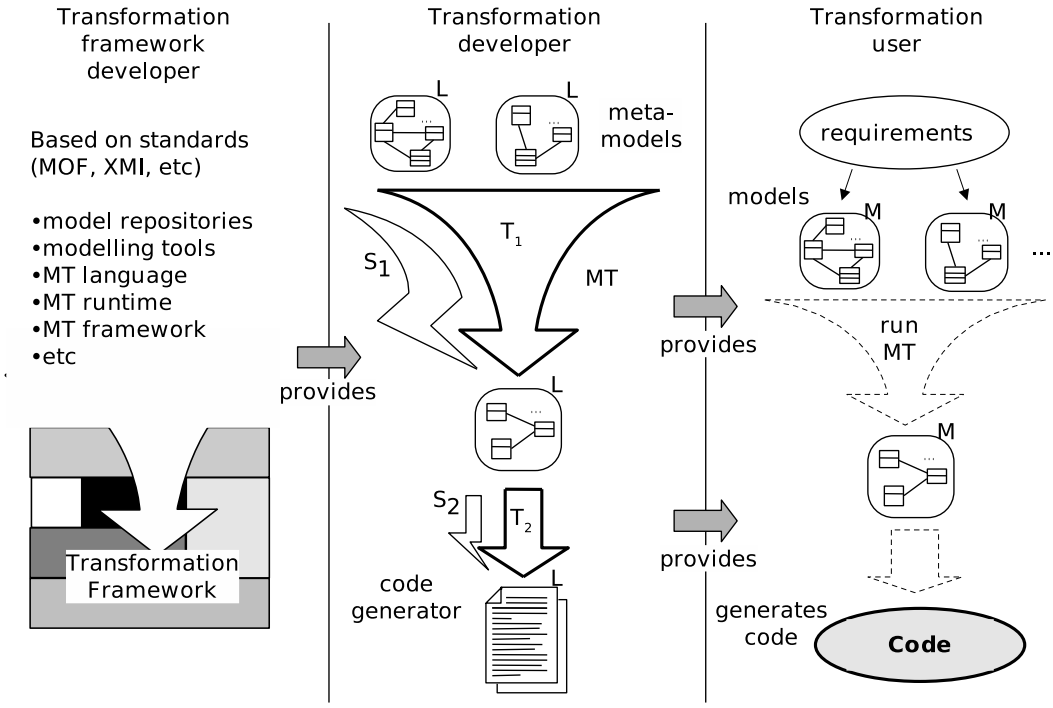

Figure 2. Separation of roles in Model-Driven Engineering

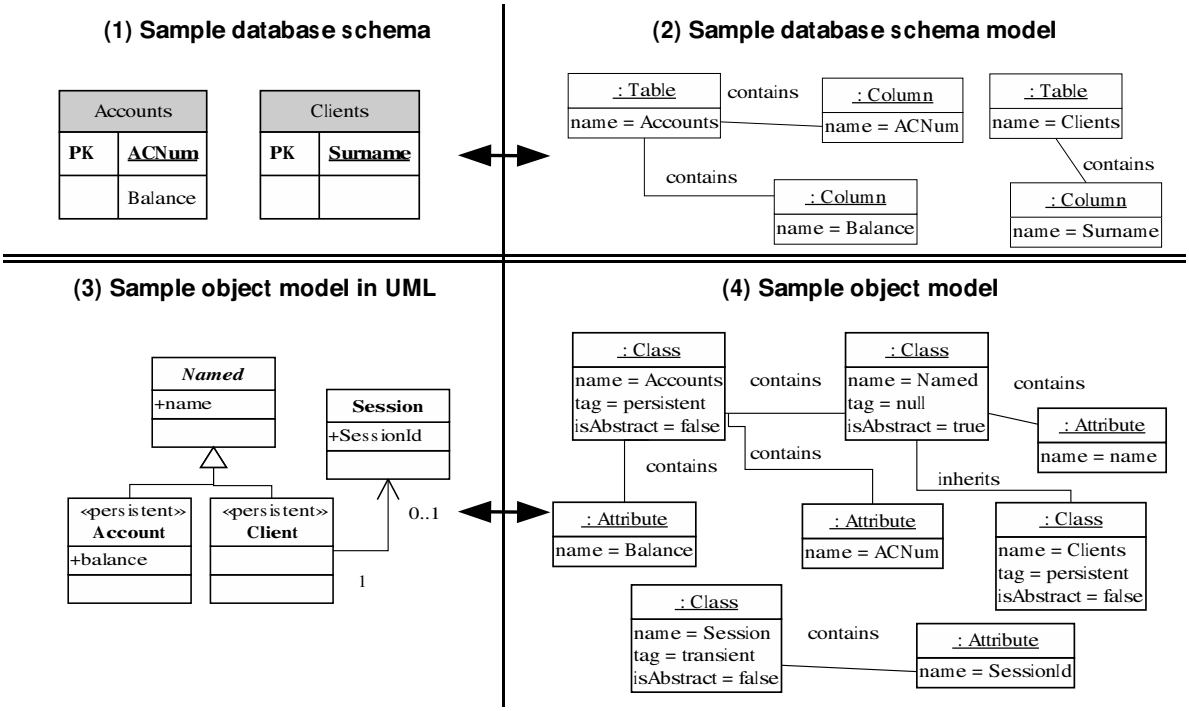

Figure 3. Sample RDBMS and O-O models

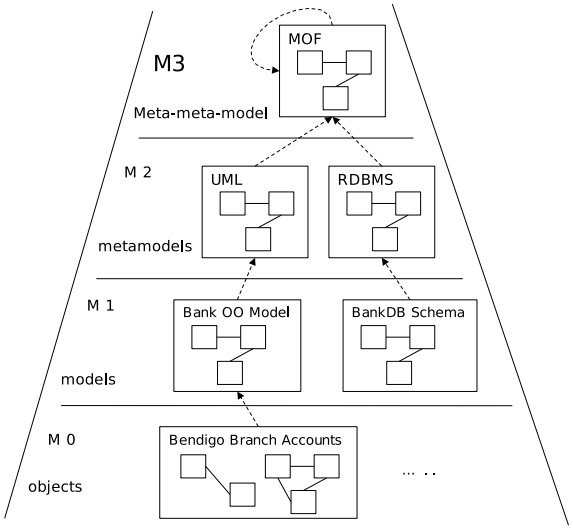

Figure 5. OMG Four-Layer Architecture

monly depicted using the meta-stack diagram shown in figure 5, showing the 3 "meta-levels", M3, M2 and M1. Like any language in MDA, MOF is described by a MOF metamodel, and thus it is self-describing. This self-description is analogous to defining an EBNF grammar for describing EBNF, and prevents an endless progression of meta-layers going up the meta-stack.

2.2 Model transformations

Obviously, the UML and RDBMS models presented in Figure 3 are related. In fact, it is easy to imagine that one has been generated from the other. Such a generation process can be described by a model transformation. Model transformations describe relationships between two or more models, by defining relations between elements in their meta-models. So, in this case, there is a relation between a UML Class and a RDBMS Table, where the names of the respective objects must be the same.

A model transformation (MT) is, essentially, a specification for a model transformation program. Model transformation programs (MTPs) take models and ensure that the elements in the models correspond to the relations defined by a certain model transformation. There are a number of languages used for defining model transformations and model transformation programs.

Model transformation programs are generally implemented either using general-purpose programming languages such as Java, using domain-specific programming languages such as extensions of OCL [17], or using a transformation engine based on rule evaluation. Generally, transformation programs are exposed using an operation interface, with typed parameters, and possibly return types and exceptions.

Languages for defining model transformations [10] are generally declarative, and include approaches based on relations [1], or those based on patterns of logical constraints [9]. In 2001, the OMG issued a Request for Proposals [14] for a standardized language for defining model transformations. A wide variety of languages have been proposed, from imperative languages to rule-based logic-like languages, and hybrids of the two. The process of settling on a single language is progressing slowly, but the form of such a standard is unknown at the time of writing.

At the minimum, it is expected that transformations will be specified as operations with OCL [15] pre- and post-conditions. We will use this form of specification as the basis for the specification of model transformations throughout this document. It is likely that a more structured language will eventually emerge as a standard, but we feel that techniques developed for testing against specifications written in this general language will apply equally, and probably more easily, to more structured specifications. The following section presents an example model transformation from UML class models to RDBMS database schemas, specified using this OCL-based language.

2.3 The UML to RDBMS transformation

This is a classical and simple example of a model transformation. From a UML model, it creates tables and columns in a relational database model to handle the persistence of the object model. In the UML model, some classes are tagged as persistent. For each of these classes, a table having the same name is created in the DB and, for each attribute of the class, a column is created. Figure 6 presents a specification for this transformation as it would be written according to the MDA paradigm.

In this example, the specification of the model transformation is close to the specification of an operation: a description, parameters and pre and post condition.

- Description: The first element of specification is a textual description of the model transformation. This description is usually written in natural language and documents the transformation.

- Parameters: The transformation, as any operation, has a set of parameters. Each of these parameters has a direction: in, out, or in/out. The types of these parameters can be either simple (e.g. string, integer) or complex: "MOF:Extent". In fact, in the context of the MOF, "extents" are the top level model container. Extents are not typed and an extent can contain model elements instantiated from various meta-models. In the specification presented Figure 6, the transformation has two parameters: "in UMLModel" and "in/out RDBMSModel" both defined as extents (i.e. models). The type of model elements that each parame-

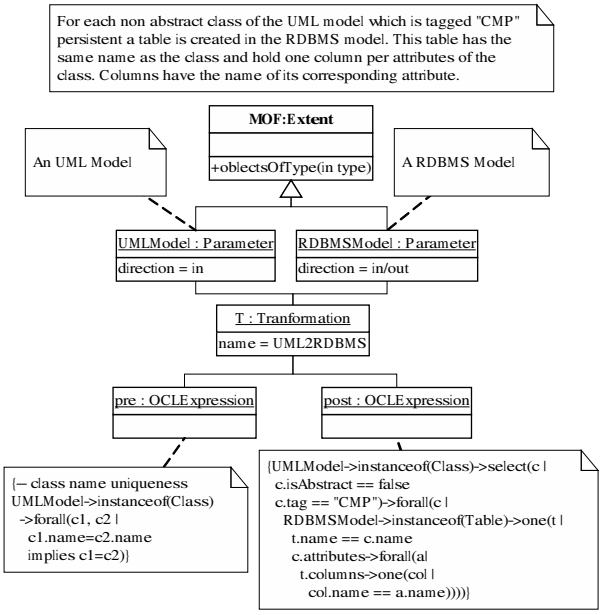

Figure 6. The OO2RDBMS transformation

ter may contain has been specified as documentation notes: The parameter UMLModel may be a UML model (i.e. instances of the UML meta-model) and the RDBMSModel parameter may be an RDBMS model.

- Pre-condition: In addition to parameters, the specification contains pre and post conditions expressed using OCL. The precondition expresses constraints on the input parameters of the transformation in order to specify the valid input data of the transformation. In the example presented in Figure 6, the precondition states that names of the classes of the UML model given as input must be unique.

- Post-condition: The post-condition specifies the effect of the transformation by linking input parameters and output parameters. For the UML2RDBMS transformation, it states that for each non-abstract class tagged "CMP" in the UML model, a corresponding table (same name) exists in the RDBMS model, and for each attribute of the class a corresponding column (same name) exists in the table.

3 Testing Model transformations

In the context of model-driven engineering (MDE), ensuring the correctness and reliability of MTPs is a key point. In the MDA paradigm model transformation programs (MTPs) are designed and implemented to be used and reused by a number of clients in various contexts. Thus, as with any software component, the implementation of a MT has to be consistent with respect to its specification/documentation. Today, the easiest and most common form of validation remains testing. In practice, the benefit of an adequate testing process, especially for components designed to be reused by a number of clients, is three-fold [12]:

1. Detect errors in the implementation. This is the primary goal of testing.

2. Complete the specification/documentation. Unspecified observed behaviour may be due to an incomplete specification.

3. Qualify the component. Testing allows the tester (and the user) to acquire a certain confidence in the implementation.

In practice, all of these aspects and the quality of the testing process strongly depend on the test adequacy criterion that is being used. Typically, two kinds of criteria are distinguished, on one hand *white-box* (or structural) criteria, which are based on the structure of the implementation, and on the other hand *black-box* (or functional) criteria, which are based on the specification. Once an appropriate test criterion is designed, it can be used as a basis to write or generate test data. The program can then be executed with these test data and an oracle function determines if the obtained results match the expected results.

As explained in section 2, a MT is implemented as a regular program. As such, one can argue that it may be designed, implemented and tested as any other program. However, there are a number of reasons to suggest that the testing of MTPs will benefit from the modification of existing techniques to better suit the problem. These reasons are the same as those that lead people to design specific languages for MTPs such as those mentions in section 2.

The first reason is that the increase in the level of abstraction makes existing techniques difficult to apply. For instance, a MTP manipulates complex data structures (models), which makes existing programming languages and test techniques hard to use. In the context of testing, in particular, many test data generation techniques focus on the generation of simple data and are consequently not directly applicable to the generation of models. Thus, in the same way that functional testing techniques have been adapted to the OO paradigm [8], testing techniques must now be defined for the emergent Model-Oriented paradigm.

The second reason is that MTPs share many common properties. This allows the design of specific languages and testing techniques for their definition and validation, respectively. For example, the models manipulated by a MTP are typically described by meta-models. As discussed in the following sections of this paper, the use of these meta-models can be a way of defining test criteria.

The following sections discuss, using the example of the OO2RDBMS transformation introduced in section 2, the limitations of some existing testing techniques, and present ideas to define test techniques better suited to the validation of MTP.

4 Functional Criteria

As functional testing techniques [6] consider the program under test as a black-box, they have the advantage of being independent from the implementation language. However, they strongly depend on the way the program under test is specified. Numerous testing techniques exist, for testing with formal specifications such as SDL and B, and for less formal languages such as UML use-case and sequence diagrams. As discussed in section 2.2, for MTPs, a top-level specification formalism can be reasonably used even if it has not yet been normalized : data structures are described as meta-models and transformations as operations with pre- and post- conditions (see Figure 6).

This section investigates the definition of a test criterion that takes advantage of this formalism to allow the selection of test data (i.e. input models) for MTPs, based on a coverage criterion on the input meta-model/s. This coverage criterion is inspired by existing testing techniques such as partition analysis [16] and UML-based testing [2]. Then, to improve the efficiency of this coverage criterion, the notion of an effective meta-model is introduced, being the part of a meta-model that is effectively used by MTP.

4.1 Meta-model coverage

Typically, the main difference between a MTP and a classical program is the complexity of its input data and the fact that the structure of that data is described by meta-models. In fact, a MT specification is a data-centric view of a program whereas most of the time, classical specification langages like SDL or UML state charts present a control-centric view of the program. The idea proposed in this section is to use MT specifications (meta-models and constraints) to define a data-centric test adequacy criterion based on the data-centric technique known as partition analysis [16] and on the adaptation of a UML-based test criteria defined in [2].

4.1.1 Partition analysis

Partition analysis was first proposed by Ostrand et al. in [16]. It consists of dividing the input domain of the program (or function) under test into several non-overlapping sub-domains (or classes). A test set can then be built by choosing a unique test datum from each class. This technique assumes that, if the program behaves correctly with one test datum from a class, then it should be valid for all data from this class. Thus, using such a technique, the partitioning of the input domain is crucial: classes must be well-chosen and the number of class must be reasonable. To define partitions, as shown on the following example, both the structure of the input data and constraints like pre- and post-conditions can be used.

Example: if a program takes an integer (x) as input, possible partition may be:
P={min, [min+1..-1], 0, [1..max-1], max }
However, if a post-condition of the program is:
(x>2) implies ...
Then a more adequate partition may be:
P={min, [min+1..1], 2, [3..max-1], max }

Inspired by this technique, the following sections propose a technique to identify subsets of data to test model transformation programs.

4.1.2 From UML models to MOF meta-models

The input scope of a model transformation program is defined by the meta-models of its input parameters. Thus, using this information, a set of criteria can be defined on those meta-models that allows partitioning the input space to select a set of relevant test data. Since meta-models using MOF are similar to UML class diagrams (classes, attributes, generalization, associations), we propose to reuse existing criteria defined for UML class diagrams. Andrews et al [2] define criteria to cover a UML model, and especially three criteria to ensure the coverage of a class diagram:

- Association-end multiplicities (**AEM**): for each association, each representative multiplicity pair must be covered.

- Class Attribute (**CA**) : for each attribute, each representative value must be covered.

- Generalization (**GN**) : For each generalizable element, each sub-type must be covered

The first two criteria (AEM and CA) are expressed in terms of representative values. The idea here is to reuse the techniques of partition testing to define a set of representative values for each attribute and each association end. These criteria are meaningful and easy to adapt to the meta-model level: the concepts of classes and associations in MOF and UML are very similar. In [2], the focus is on testing the behaviour of models, whereas for model transformations, we are interested only in their structures. As such, the GN constraint is not significant in this context because, although behaviours can be overridden in subtypes, structures cannot. Thus, we propose to adapt the two following criteria to achieve coverage of a meta-model:

- **AEM** (Association End Multiplicities): For each association end, each representative multiplicity must be covered. For instance, if an association end has the multiplicity [0..N], then it should be instantiated with the multiplicity 0, 1 and N. Representative multiplicity pairs can then be computed for an association by taking the Cartesian product of the possible multiplicities of each of its two ends.

- **CA** (Class Attribute): For each attribute, each representative value must be covered. If the attribute's type is simple (integer, string...), a set of representative values has to be computed. (If the attribute's type is complex, it has to

be processed as an association, according to the AEM criteria). In the same way as is done for associations, the representatives values of a class' simple attributes then need to be combined using a Cartesian product.

4.1.3 Representatives values

For the two criteria (AEM and CA), an appropriate set of partitions must be found in order to select representative values. As discussed in [2], two techniques can be applied for each criteria: *default partitioning* and *knowledge-based partitioning*.

The first, default partitioning, consists of defining, a priori, a partition based on the structure or the type of the data. For a string attribute this may be {null, "", "something"}, and for an [0..1] association end it would be {0, 1}. The advantage of such a partitioning technique is that it is fully automated and easy to implement as soon as a policy for each data type is provided. Most of the time, this policy can consist of choosing boundary values and possibly values outside the boundaries, if the goal is also to check the robustness of the transformation.

The second, knowledge-based partitioning, consists of extracting representative values from the model transformation itself. These values can be provided by the tester or automatically extracted from the specification of the model transformation. In particular, the pre- and post-conditions allow us to find relevant values for attributes and possibly multiplicities for association ends. This is illustrated by the example presented section 4.1.1, where the value 2 is extracted from a constraint expressed in the post condition.

Combining these two techniques allows each attribute and association end of the meta-model to be enumerated with sets of representative values: using knowledge-based partitioning if some information about this attribute or association is available in the specification, or default partitioning otherwise. As an example of this, Table 1 presents the partitioning for a few elements of the UML meta-model for the UMLModel input parameter of the OO2RDBMS transformation. One notices the presence of the "CMP" value for the attribute Class::tag, which have been extracted from the post-condition of the transformation. More generally, representative values can be extracted from the pre- and post-conditions of a transformation by parsing all constraints of the specification to select all literal values as representatives values for the model element to which they are compared.

4.1.4 Coverage items and test criterion

Given sets of representative values for each element of the meta-model, the idea of the test criterion is to ensure the coverage of these values by the tests. However, covering these particular values independently from each others is not sufficient. To illustrate this on the example of the OO2RDBMS transformation, consider the sets of values (shown in table 1) selected for the attributes Class::isAbstract and Class::tag. Selecting models to cover independently these values does not ensure that any of the selected test models contains a class both non-abstract and tagged "CMP", which is intuitively one of the most important situations with respect to the transformation. To deal with this, we define sets of combinations of representative values, called *coverage items*, that must be covered by the tests.

Table 1. Representative values

Meta-model element	Representative values
Class::name : String	Null, '', 'something'
Class::isAbstract : Bool	True, False
Class::tag : String	Null, '', 'something', 'CMP'
Class->attribute : [0..*]	[0], [1], [>1]
Attribute->owner : [1..1]	[1]
Attribute::name : String	Null, '', 'something'

Coverage item: A coverage item c_i is a constraint on input models of a MTP. This constraint ensures that a particular combination of representative values is instantiated by a set of input models.

In practice, for each class and association the representative values and multiplicities, respectively, must be combined in order to obtain a set of *coverage items.* Some combinations of values for attributes of a class (or multiplicities for an association) may be invalid (i.e. incompatible) with respect to constraints on the meta-model or pre-conditions of the model transformation, and can be eliminated. Finally, a set C of valid coverage items is computed.

Test criterion: Covering the meta-model involves covering all the valid coverage items computed for each class and association. More formally, a test set M satisfies the criterion if $\forall c_i \in C, \exists m \in M \mid c_i(m)$.

4.1.5 Summary and limitation

To sum up the discussion, the principle is to first find representative values and multiplicities for attributes and association ends of the input meta-model of the transformation. Then, for each class and association, compute the Cartesian product of those values. Lastly, constraints on the meta-model and the pre-conditions of the transformation need to be checked with these combinations of values in order to remove the invalid combinations. The remaining combinations represent a set of coverage items that can then be used to qualify a test set or to automatically generate test sets for model transformation programs.

As discussed previously, using this technique should allow the selection of test data that cover the behavior of the OO2RDBMS transformation. However, as the meta-model of the UMLModel parameter is the UML meta-model, the

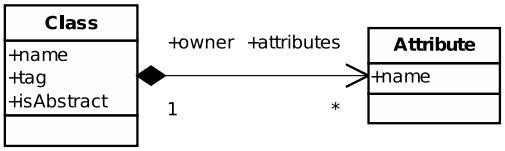

Figure 7. Effective meta-model

amount of non-relevant data selected may be significant. In fact, most of the test data selected would be irrelevant with respect to the transformation because the UML meta-model contains numerous classes that do not have any relevance in the context of the transformation. For instance, using this technique, test data would be selected to cover UML state machines and sequence diagrams, which is intuitively not necessary when testing the OO2RDBMS transformation. The following section proposes the notion of an effective meta-model in order to solve this problem.

4.2 Effective meta-model

In practice, it is very rare for a model transformation to use the entire meta-model of its input data. Since the testing technique proposed here is based on meta-model coverage, it may lead to numerous useless test cases.

To deal with this, the idea is to determine, before test generation, the actual subset of the meta-models that is relevant to the transformation. A pragmatic way to select such an effective meta-model is to use the OCL pre-/post- conditions from the specification of the transformation. The meta-model elements relevant for a transformation are at least the elements referred to by its specification. As an example of this, Figure 7 presents the sub-set of the UML meta-model suitable for the UMLModel parameter of the OO2RDBMS transformation. It can be automatically constructed by including every type, attribute and association of the UML meta-model referred to by the specification of the transformation presented in Figure 6. For associations, we must also include all subtypes of the referenced class.

Given this effective meta-model, the coverage criterion previously proposed can be applied with more efficiency. In the case of the OO2RDBMS transformation shown in Figure 7, only two classes, four attributes and one association remain. Yet intuitively, this small meta-model still seems sufficient for testing this transformation. The partitions for the attributes and the association ends of the reduced meta-model are presented in Table 1. A set of coverage items, as presented in Table 2, can then be computed by combining the partitions for each class and association. A total of 30 coverage items has been computed to ensure the coverage of the effective meta-model of the UMLModel parameter (in the table, the first line represents 8 coverage items). These 30 coverage items need to be filtered according to

Table 2. Coverage items

	Cri.	Element	Values
1	CA	Class	Null, *, *
9	CA	Class	'', True, Null
10	CA	Class	'', True, ''
11	CA	Class	'', True, 'something'
12	CA	Class	'', True, 'CMP'
13	CA	Class	'', False, Null
14	CA	Class	'', False, ''
15	CA	Class	'', False, 'something'
16	CA	Class	'', False, 'CMP'
17	CA	Class	'something', True, Null
18	CA	Class	'something', True, ''
19	CA	Class	'something', True, 'something'
20	CA	Class	'something', True, 'CMP'
21	CA	Class	'something', False, Null
22	CA	Class	'something', False, ''
23	CA	Class	'something', False, 'something'
24	CA	Class	'something', False, 'CMP'
25	CA	Attribute	Null
26	CA	Attribute	''
27	CA	Attribute	'something'
28	AEM	Class->Att.	[1-0]
29	AEM	Class->Att.	[1-1]
30	AEM	Class->Att.	[1-(>1)]

For the class "Class" the sequence of values {x,y,z} corresponds respectively to the values of the attributes Class::name, Class::isAbstract and Class::tag.

the constraints of the meta-model and the pre-condition of the transformation. In the UML meta-model, the name of a model-element cannot be null, so the coverage items numbered 1 to 8 and 25 are invalid. The precondition of the transformation states that the names of the classes within a model must be unique, which is the case for remaining coverage items. Following this, 21 items remain that cover the UMLModel parameter. We notice that the same thing would have to be done for the RDBMSModel in/out parameter of the transformation to cover the whole specification of the transformation.

The following sections present a white-box refinement of the test criterion that takes advantage of a static analysis of the code of the MTP and investigates techniques to automatically generate a set of input models that ensures the coverage of the coverage items proposed previously.

5 Static analysis

The idea of structural or white-box testing is to use the implementation of the program itself as a basis for the test criterion and the test data generation. This section proposes to enhance the criterion proposed previously by static analysis of the code of MTP. Firstly, we present how an effective meta-model can be extracted from the implementation of a transformation, and secondly how the implementation can also be used to compute representative values.

5.1 Effective meta-model

A static analysis of the MTP can be used to compute the effective meta-model. To do so, the idea is to collect the meta-model elements that are referred or used in the transformation. In practice, this can be automatically handled for most transformation languages as soon as they include a static type-checker. During the type-checking of the model transformation program, all meta-type referenced by the program can be collected to construct the effective meta-models that are relevant for each parameter. The obtained meta-model has the advantage of being complete and specific to the actual implementation under test.

This effective meta-model can then be used for several purposes. Firstly, it is a reliable meta-model for test generation with the test criterion described previously. Secondly, it can be used for verification: compared to the effective meta-model extracted from the specification to check the consistency between the specification and the implementation. In particular, it may allow improving the specification or the documentation of the MT or detecting unexpected behavior of the MT implementation. Lastly, apart from the validation considerations, the knowledge of the effective meta-model of a MT implementation allows us to check if an implementation can be used with a particular meta-model (a new version of a meta-model referred to by the specification, for example).

5.2 Representative values

In the same way as was done with the specification, we can also perform a static analysis of the code of the MTP to collect representative values for each attribute of the meta-model. For this purpose a simple algorithm that browses the code of the MTP can be designed. Initially, it associates an empty set of representative values to each attribute of the effective meta-model. Then, the algorithm analyzes the code and collects references to the attributes of the meta-model. The exact information that can be extracted from the code depends on the MT language that has been used to write the transformation, but in most cases the references collected should be either comparisons or assignments. Intuitively, the idea is that if a particular value is compared or assigned to an attribute in the MTP then this value may be an "interesting" value for this attribute. In practice, the value of the attribute may be:

- a literal value. This literal value must be added to the set of representative values of the attribute.

- another attribute. The sets of representative values for the two attributes need to be merged into a single set.

- an expression of the language. In most cases, no information on its possible values can be statically computed and the set of representative values of the attribute is left unchanged.

When all the code of the MTP has been processed, a set of representative values is associated with each attribute. These values can then be used to apply the test criterion described previously or for automatic model generation as discussed in the next section.

6 Automatically generating models

This section investigates techniques to automatically generate a set of models that covers coverage items. Automatically generating models for a particular meta-model is not a trivial problem. In fact, several issues have to be resolved, including the selection of classes to instantiate and the selection of appropriate values for attributes. In the current context, the idea is to guide the model generation using the coverage items and to use the partitioning previously presented to select concrete and representative values for the attributes.

When generating test models, two factors must be taken into account: the size of the tests set and the size of test cases. It is important to have small test cases to facilitate the understanding of each test case and to allow an efficient diagnosis when a test case detects an error. On the other hand, the set of test cases must be kept at a reasonable size to reduce execution time and the effort for oracle definition (if it is not fully automated). In the particular case of generation of models, the number of coverage items is an upper bound for the size of the tests set, as a test case usually covers several items.

6.1 Systematic approach

This section discusses a systematic algorithm to build a set of models that covers the coverage items for a transformation program. This algorithm builds instances of the effective meta-model and instantiates the attributes using the representative values and the coverage items. The process is iterative: it tries to build a model that covers as many items as possible. Once a valid model has been generated, the algorithm adds this model to the solution set and builds another one to cover the remaining coverage items. The following sections describe how models can be generated to cover as many coverage items as possible, and how the size of the solution set and of the generated models can be kept reasonable.

6.1.1 Building models

The automatic generation of meta-model instances starts by instantiating a class of the effective meta-model that corresponds to a coverage item (this assures that at least this item is covered by the model). To do so, values have to be

assigned to the attributes of the class. For each attribute, the value is either defined in the coverage item, or chosen among the set of representative values attached to this attribute. Now, if the class that has just been instantiated depends on other classes in the meta-model, or, if other classes depend on this class, all these other classes have to be instantiated. This is necessary to build a valid input model for the transformation program. The instantiation of a complete valid model from a partial model is iterative. First, the set of meta-model elements that have to be instantiated is computed. Then, the ones that allow covering an uncovered coverage item are selected in order of priority. The selected candidate is instantiated and added to the model. The set of candidate meta-model elements is then re-computed and the process iterates until there are no more meta-model elements that need to be instantiated.

6.1.2 Building the solution set

If some coverage items remain uncovered after the construction of a valid input model, two solutions exist: generate another model, or add elements to this model to cover other items. In the first case, the size of the test models obtained may be small but the number of generated test cases will be high. On the contrary, using the second policy, the algorithm will try to generate a single model to cover all the coverage items. Obviously neither of these solutions is acceptable and a trade-off must be found to ensure both the reasonable size of the test cases and the reasonable size of the test set. The generation algorithm must then be parameterized by a limit for the size of the generated models, i.e., the maximum number of model elements they can contain. Now, when a valid model is generated, the size limit fixes the policy to follow: if the model is smaller than the limit, new elements can be added to satisfy the other coverage items, otherwise it is memorized and a new model is built. The algorithm stops when all coverage items are covered by at least one model.

6.1.3 Discussion

In practice, this algorithm allows the automatic generation of a set of models that satisfies the test adequacy criterion of section 4.1.4. However, the generated set of models strongly depends on the order in which the coverage items are selected. Depending on this order, the obtained results may be very different and certainly not optimal, i.e. the set of test models may not be minimal. The next section investigates the adaptation of an optimization algorithm that has been especially designed in previous works for the generation and optimization of complex test data.

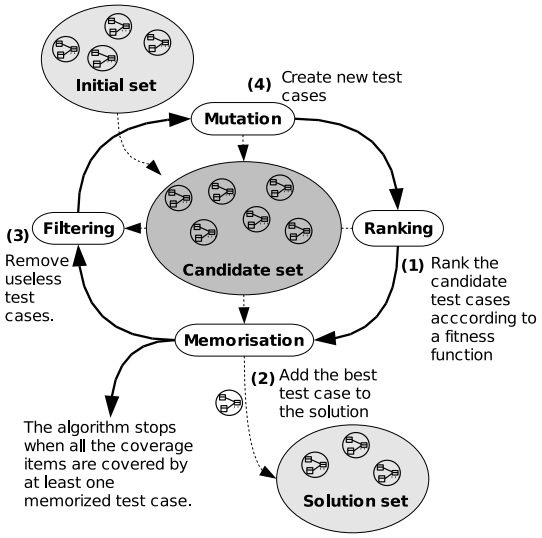

Figure 8. Bacteriologic algorithm

6.2 Bacteriologic approach

In the literature, many techniques based on optimization algorithms have been proposed for the generation of test data. This section presents an adaptation, to generate models, of the bacteriologic algorithm proposed in [5, 4]. The bacteriologic algorithm is an original adaptation of genetic algorithms that has been designed especially for the problem of generating a set of complex test data that globally covers a set of items (the test criterion). This section details how the algorithm can be adapted to generate a set of models that covers a set of coverage items.

Figure 8 shows the principle of a bacteriologic algorithm. A starting set of test cases (models) must be provided to initialize the candidate set. This set can be an existing set of test cases or can be randomly generated. During the execution of the algorithm, a solution set of test cases is incrementally constructed. Each iteration of the algorithm consists of four steps:

1. Ranking: The candidate test cases (models) are ranked according to a fitness function that estimates their potential contribution to the overall coverage of the solution set. The fitness function for generating models to test a MTP might be the number of unfulfilled coverage items that are covered by a candidate test model.

2. Memorization: During the memorization step, the best test case is possibly added to the solution set according to a selection policy. This policy allows adjusting the trade-off between the size of the final solution set and the rate of convergence of the algorithm, e.g. through the use of a fitness threshold for memorization [4].

3. Filtering: The filtering step involves removing the useless test cases from the candidate set in order to keep

the size of the candidate set reasonable. The easiest way to achieve the filtering is to remove all test cases whose fitness value is null, i.e. that will not contribute anything to the solution set.

4. Mutation: Last, but not least, the mutation step is the one that actually creates new candidate test cases. During this step, the best test cases (according to the fitness function) are selected and a mutation operator is applied to them in order to create new test cases. Thus, to apply the algorithm to generate tests for model transformations, an adapted mutation operator needs to be defined. Such a mutation operator would take advantage of the partitions that have previously been defined for each element of the metamodel. For example, it could randomly select an element in an existing model and create a new model by selecting a value from a different partition.

The advantage of using such an algorithm instead of a systematic algorithm for the automatic generation of test models is better optimization (minimization) of the generated tests. However, the bacteriologic algorithm requires an initial set of test cases, which might be generated by a systematic algorithm, and then optimized and completed using a bacteriologic algorithm. In future work, we plan to implement the two techniques and compare, or perhaps combine, them in order to design an efficient test generator for model transformation.

7 Related Work

The emergence of MDA has led to a lot of research work on design issues for effective model-based software development. However, there has been little work yet about testing in this particular design paradigm. The authors in [11, 18] address specific issues when testing in the MDA context. In [18] the authors propose to automatically generate test harnesses in a generative programming environment and [11] propose guidelines for model-driven testing. However, none of these works looks at the problem of testing transformation programs. In section 4 we detailed the work by Andrews et al. [2] that is closely related to testing in the MDA. This section discusses another related work presented by Atkinson et al. in [3].

M. Rutherford is interested in test-code generation while developing an application in a model-driven context. In [18], he reports an experiment to generate test code in parallel with the system. The experiment is done in the context of a generative programming tool called MODEST. The authors distinguish between the automatically generated code and the domain specific code that is hand-written. The idea is that the generated code constrains the possibility to write domain-specific code. Then, in the same way, there should be automatically generated test code to constrain the writing of test code for domain specific code. 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 reported benefits were that developing templates for test code enhanced the development process and allowed the developers to be more familiar with the code produced by MODEST. The costs are evaluated with an analysis of the complexity of templates for test-code generation.

In [11], the authors also explicitly address the problem of test generation in a MDA context and propose to develop model-driven testing. In particular, this work focuses on the separation between platform-independent models and platform-specific models for testing. The generation of test cases from the model, as well as the generation of the oracle are considered to be platform-independent. The execution of the test cases in the test environment is platform-specific. The two platform-independent issues are not discussed in detail. The authors refer to previous work and highlight the use of model simulation for the oracle. Then, they show how to employ design patterns to run test cases in a specific environment. The case study is based on model-driven development of web applications.

In [3], the authors propose a specific technique for testing the integration of components in component-based development. They propose to embed test cases in the component in order to offer the component a "built-in" ability to check the correctness for the environment in which it is integrated. This means that the tests aim to verify the client/server interactions. This helps integration by automating part of the testing, by distributing the test instead of exclusively testing at the system level, and the components are more robust since they can raise a flag if they are plugged to an illicit environment. This technique conforms to MDA in the sense that the contracts which express the rules to use a component correctly, and that are used to build the embedded test cases, are independent from a particular component technology.

8 Conclusion and future work

In this paper, we present an initial exploration of techniques for software validation in a model-driven environment. As the primary behavioural artifact in the OMG's model-driven architecture, model transformation programs must be rigorously tested. Although this can be achieved using traditional testing techniques, we present a number of strategies that adapt these to better suit model transformations, which differ significantly in the data structures that are used. Since the languages used to specify model transformations are still in the process of being normalized, we use a lowest-common denominator language based on operations and OCL pre- and post-conditions.

We present a test adequacy criterion for testing model

transformations. Based on the techniques of partition testing and UML class diagram coverage, the criterion consists of the derivation of coverage items for the source meta-models. To avoid the common problem of larger-than-necessary meta-models, we also discuss a technique for deriving an effective meta-model, as the useful subset of the actual meta-model. Using this coverage criterion, we discuss two approaches to the generation of test data, based on systematic and bacteriologic algorithms.

There remains a lot of research to be done into the testing of model-driven architectures. We now intend to clarify and empirically evaluate the techniques presented here to assess their effectiveness in first small- and later large-scale model transformations. Once the language for model transformations has been normalized, it will be possible to expand on the techniques discussed here for white-box or structural testing, including the definition of mutation operators for the use of mutation testing to qualify and improve the quality of the test set. Also, while we examine the testing of model transformations, there is also much work to be done in the testing of systems developed using model-driven techniques.

References

[1] D. H. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *UML 2002 - The Unified Modeling Language, 5th International Conference, Proceedings*, pages 243–258, 2002.

[2] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Journal of Software Testing, Verification and Reliability*, 13(2):95–127, april-june 2003.

[3] C. Atkinson and G. Hans-Gerhard. Built-in contract testing in model-driven, component-based development. In *1st International Working Conference on Component Deployment*, Austin, TX, USA, 2002.

[4] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, 2002.

[5] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. Genes and bacteria for automatic test cases optimization in the .NET environment. In *procedings of the Thirteenth International Symposium on Software Reliability engineering (ISSRE)*, november 2002.

[6] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition edition, 1990.

[7] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In *UML 2003 - The Unified Modeling Language, 6th International Conference, Proceedings*, pages 175–189, 2003.

[8] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999.

[9] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. Model transformation: A declarative, reusable patterns approach. In *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003*, pages 174–195, Sept. 2003.

[10] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation, ICGT'02*, volume 2505 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[11] R. Heckel and M. Lohmann. Towards model-driven testing. In *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[12] J.-M. Jézéquel, D. Deveaux, and Y. Le Traon. Reliable Objects: Lightweight Testing for OO Languages. *IEEE-Software*, 18(4):76–83, jul-aug 2001.

[13] O. M. G. (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, Sept. 1997.

[14] O. M. G. (OMG). MOF 2.0 Query/Views/Transformations RFP. OMG Document ad/2002-04-10, Oct. 2002.

[15] O. M. G. (OMG). The object constraint language (OCL), 2003. http://www.omg.org/docs/ptc/03-08-08.pdf.

[16] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676 – 686, 1988.

[17] D. Pollet, D. Vojtisek, and J.-M. Jézéquel. OCL as a core UML transformation language. WiTUML 2002 Position paper, Malaga, Spain, June 2002.

[18] 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*, pages 377–396, 2003.

[19] R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document, Nov. 2000.