Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool

Erwan Brottier[1], Franck Fleurey[2], Jim Steel[2], Benoit Baudry[2], Yves Le Traon[1]
[1]*France Télécom R&D*
2, av. Pierre Marzin - 22 307 Lannion Cedex – France
{erwan.brottier, yves.letraon}@francetelecom.com
[2]*IRISA- 35042 Rennes Cedex – France*
{ffleurey, jsteel, bbaudry}@irisa.fr

Abstract

In a Model-Driven Development context (MDE), model transformations allow memorizing and reusing design know-how, and thus automate parts of the design and refinement steps of a software development process. A model transformation program is a specific program, in the sense it manipulates models as main parameters. Each model must be an instance of a "metamodel", a metamodel being the specification of a set of models. Programming a model transformation is a difficult and error-prone task, since the manipulated data are clearly complex. In this paper, we focus on generating input test data (called test models) for model transformations. We present an algorithm to automatically build test models from a metamodel.

1 Introduction

MDE represents a significantly new approach to the development of software systems. However, while much work has been done on techniques for using MDE, there remain many challenges for the process of software validation, and in particular software testing, in a MDE context. Since model transformations are core mechanisms of MDE for building software from design to finally code, they impact strongly on the software development process. In that sense, making model transformations trustable is an obvious target in order to improve the reliability of automation during the development, what MDE advocates.

A model transformation is implemented as a regular program. As such, one argues that it may be designed, implemented and tested as any other program. We claim that in the same way testing techniques have been adapted to the OO paradigm, testing techniques must now be defined for the emergent Model-Oriented paradigm.

In fact, significant differences exist between the model-oriented paradigm and the previous ones. Indeed, model transformation programs (MTP) manipulate complex data structures that have no comparison with the data of traditional programming paradigm, such as string or integer. This key difference makes existing test techniques hard to use, especially for test data generation techniques.

Moreover, top-level specification formalisms can be reasonably used by functional test techniques: data structures are described as metamodels. A MTP is thus a good candidate for functional testing techniques (black box testing) and in particular automatic test data generation. Indeed, the input and output domains are described precisely by metamodels.

The *test data* that must be generated to test a transformation are input models. We also call them *test models*. The practical task of manually building and editing models for testing is especially tedious (the structure of the data is complex, models are difficult to manage, there is no tooling for testing, etc). Moreover, there are no precise stopping criteria to formally ensure that the transformation has been sufficiently tested. The generation of test models should then be automated to avoid the task of manually building them.

The contribution of this paper is about the automatic generation of test models, being given a metamodel describing the input domain of a model transformation. An algorithm is defined to automate test model generation. The algorithm takes a metamodel and fragments of models as an input and produces a set of test models. The model fragments are either provided by the tester or derived from the metamodel. They specify parts of the metamodel that should be instantiated with particular values that are interesting

17th International Symposium on Software Reliability Engineering (ISSRE'06)
0-7695-2684-5/06 $20.00 © 2006 IEEE

for testing. The algorithm then consists in combining model fragments and completing them to build valid instances of the metamodel. The various strategies used to combine and complete a model to make it conformant to its metamodel are presented as well as the limitations of this algorithm.

Section 2 presents the MDE approach. Section 3 introduces a 3-step process for test data generation for model transformation, details the adaptation of partition testing to this specific issue and explains the notion of model fragments. In section 4, an algorithm is defined that generates models using model fragments. At last, section 5 presents some related work.

2 Background on MDE

The goal of MDE [1] is to move away from traditional role of models (*e.g.*, UML diagrams) as blueprints for conversion into software by programmers, to a situation in which models are used as first-class development artefacts. In MDE, models are automatically transformed to other models and system code. In fact, MDE presents the more general view of systems which are described by models and relationships between them. In addition, these are managed automatically with mapping technologies such as code generators.

To perform this approach, MDE is based on the model oriented paradigm, which extends the object oriented paradigm and provides dedicated features for handling models correctly. It advocates above all the use of models, metamodels and specific programs called model transformations for easily manipulating them.

As it was the primary motivation for MDE, the domain of software development remains the main application of MDE. Its principles are represented in Figure 1, where the artifacts marked "M" are models, those marked "L" are languages, "MT" are model transformations, and "S" are specifications. After the developer has created a design in the form of a series of models, she then uses model transformations for successively refining these models, and eventually translating them into code. The model transformation programs are implemented in some transformation language which forms part of an overall MDE framework (e.g. QVT standard [2], Kermeta [3]). They are written by a transformation developer who may or may not be the same as the software developer.

The following sections present firstly a more thorough examination of the definitions of models, metamodels and model transformations. This includes the presentation of an example that will be used throughout the paper. Secondly, we introduce the task of model transformations test in the MDE context.

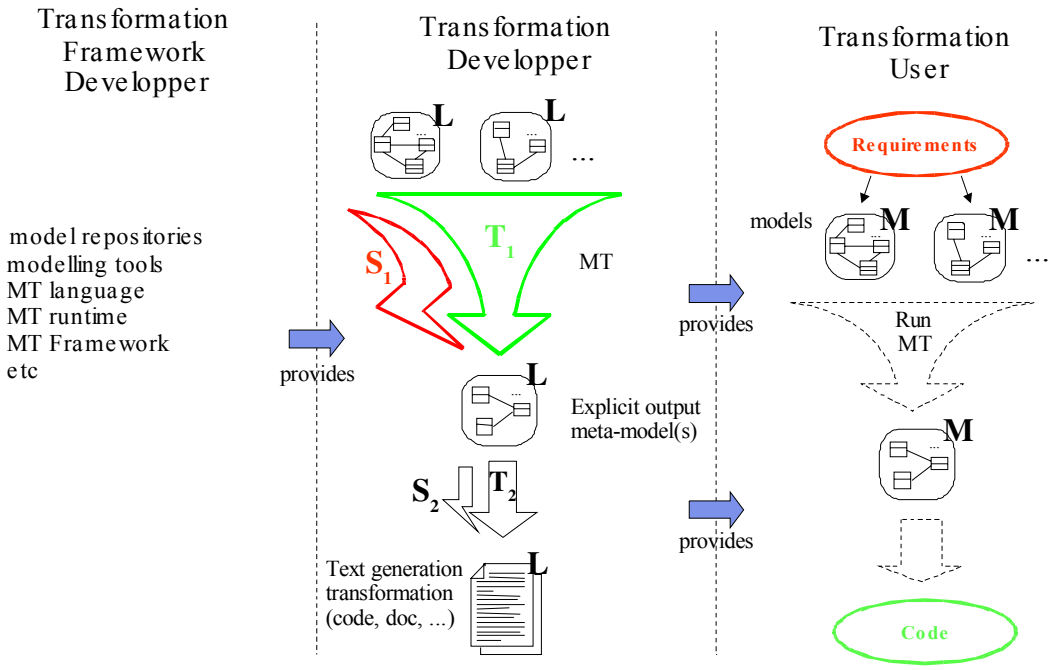

Figure 1 – Model-Driven Engineering Software Development Approach

0-7695-2684-5/06 $20.00

2.1 *Models and metamodels*

A model is a collection of objects and relationships between the objects that, together, provide a representation of some real part of a system. For example, a simple state machine diagram, such as the one in the Figure 2, might be represented in a human-friendly manner as labelled ellipses with oriented arrows between them.

Ellipses and arrows are a graphical representation of respectively states and transitions. A state can be composed of sub states.

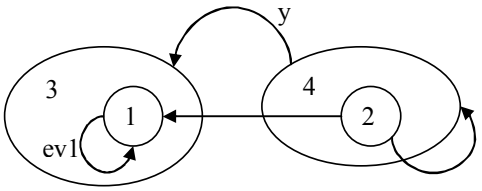

Figure 2 – A state machine diagram sample

For handling properly models and, by this way, automatically manipulating them, MDE has proposed the notion of metamodel, which defines a modeling language for a particular domain. It defines the concepts, called classes, and their properties that describe the domain (attributes and associations). A property can be either a class attribute or an association (relationship) between two classes.

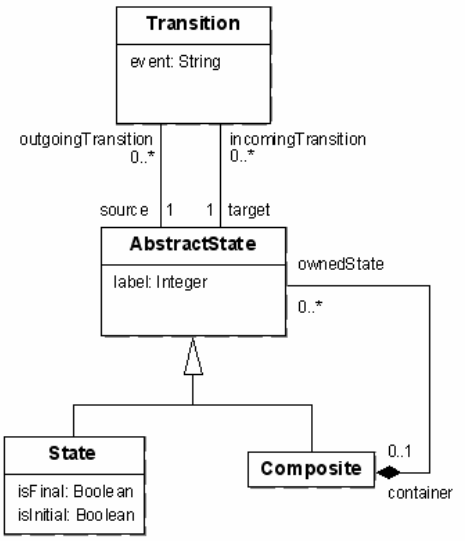

Figure 3 – Possible State Machine Metamodel

Figure 3 shows a possible metamodel that describes the concepts and relationships needed for expressing simple state machine diagrams, such as the one presented in Figure 2. Obviously, if we need more information about state machine like temporal constraints, we have to improve the metamodel for handling these well.

Figure 2 displays a human-friendly representation of an instance of our metamodel but it does not highlight the conformance to its metamodel. Figure 4 shows a representation of the same state machine, following the MDE representation under the form of an objects diagram. Each relation is an instance of a metamodel association and each object is an instance of a metamodel class. In addition, relations verify the constraints on the association cardinalities.

Metamodelling is thus used to describe a specific modelling "language", with classes for each of the concepts used in it. The metamodel of the Figure 3 is a simplified excerpt of the UML metamodel which allows describing other domains such as class or activity diagrams.

Metamodels are described with respect to a given "language". The OMG has proposed the Meta-Object Facility (MOF) [4] which is considered as a meta-metamodel. As a language, the MOF very closely resembles UML class modelling, with packages, classes, attributes and associations. A metamodel is thus a model itself. The metamodel of the Figure 3 can be described like the model of the Figure 4, according to the MOF. Like any language in MDE, MOF is described by a model, which is itself. This self description is analogous to defining an EBNF grammar for describing EBNF and prevents an endless progression.

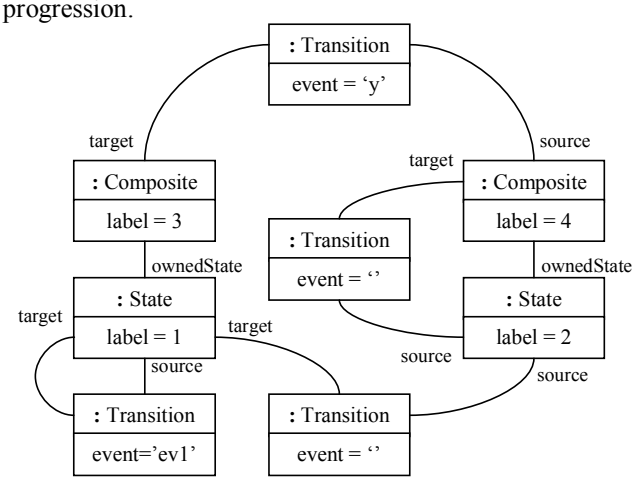

Figure 4 – A representation of our state machine model as an instance of the metamodel

2.2 *Model transformations*

Since we can properly define modelling languages such as our state machine metamodel and others, it becomes possible to define automatic processes between these. For example, we can imagine a process that transforms a state machine into a class diagram for documentation purposes or into code for

implementation purposes. A mapping must be done between the state machine metamodel and the UML metamodel in the first case, or an object oriented language metamodel in the second case (a simple pretty printer has to be defined for finally obtaining the code in a textual form). Such a mapping can be described by a *model transformation*.

A model transformation describes relationships between two or more models, by defining relations between elements in their metamodels. 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.

A MT is close to the specification of an operation. The first element of specification is a textual description of the effect the MT has to perform. This description is usually written in natural language and documents it. The second element is a set of parameters. Each of these parameters has a direction (in, out, in/out) and a type (simple type like object oriented primitive type or a metamodel). The two last elements of specification are pre and post conditions, which are expressed using the Object Constraint Language (OCL). The pre condition expresses constraints on the input parameters of the MT in order to specify the valid input data of the transformation. The post condition specifies the effect of the MT by linking input and output parameters.

The Figure 5 shows the specification of the SM2DOC model transformation that takes as an input a state machine model and produces as an output a UML class diagram model.

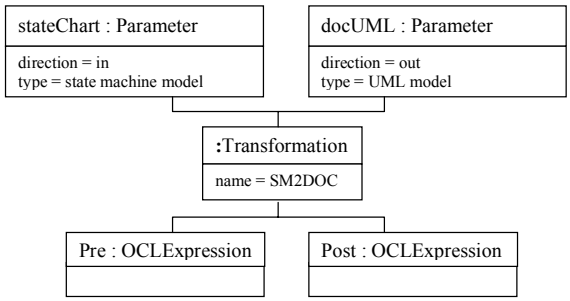

Figure 5 – The SM2DOC model transformation specification

In 2001, OMG issued a Request for Proposals [2] for a standardised language for defining MTs. A wide variety of languages have been proposed, from imperative languages to rule-based logic-like languages, and hybrids of the two like Kermeta [3].

In the following, we detail the process to generate test models as inputs for testing a MTP.

3 A process for automatic test data generation

In [5], different activities have been identified for the test data generation in the context of MT testing. They are displayed in Figure 6 in terms of a three-step process:

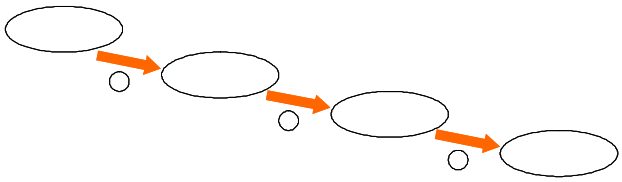

Figure 6 – A 3-step process for automatic test data generation for model transformation

Step 1: *from the effective metamodel to partition of simple types*. This step decomposes into equivalence classes the domains of all simply typed attributes and association cardinalities which appear in the actual input metamodel classes (called effective metamodel).

Step 2*: from partitions to model fragments*. Based on the list of partitions, this activity produces model fragments, which are "test objectives" on the input domain for the test model generation. The significance of the generated model fragments highly depends on the strategy that is used to combine values.

Step 3: *generation of models to exhibit model fragments.* The goal of this step is to generate models from the model fragments. This is the subject of this paper.

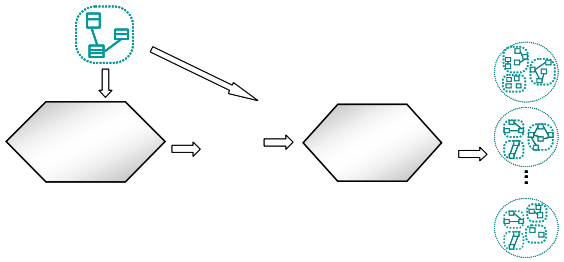

Figure 7 – The inputs of the test model generation algorithm

The contribution of this work is an algorithm for enabling the automation of this generation. Figure 7 illustrates the process for test model generation. It shows that the proposed model generator takes a set of model fragments and a metamodel as inputs in order to compute test models which are conformant to the input metamodel. Step 1 and 2 are related to the identification of test objectives, in the form of "model-fragments". Ideally, the model-fragments are deduced

automatically from the metamodel, based on the partitions and a given test criterion. In this paper, we assume they are provided by the tester and we do not present the test criteria to produce them automatically.

In the following of this section, we detail the notion of effective metamodel, partitions and model fragments.

3.1 Effective metamodel

Generating test data to cover the input domain of a program can only be achieved with precise and accurate description of this domain. In the context of MTs, the input metamodel provides such a description. However, the input metamodel for a transformation is usually larger than the actual metamodel used by the transformation. For example, a transformation on a UML class diagram takes the UML metamodel as an input metamodel, but it only manipulates the sub-part that describes the structure of class diagrams. We define the *effective metamodel* (EMM) as the exact input metamodel that is relevant to the MT. The effective metamodel can be seen as the input type for the transformation. In this paper, we assume that an effective metamodel is available. Obtaining this EMM is beyond the scope of this paper (in [5], we gave clues on the different strategies that can be applied to automatically identify the EMM).

The first step of the process (Figure 6) consists in building a *partition* for properties of the EMM whose type is primitive (boolean, integer, string): the attributes of the EMM classes and the multiplicities on associations.

3.2 Partitions for metamodel coverage

In the particular context of software testing, partitions have been used for defining category-partition testing [6]. This technique consists in defining equivalence classes on the input domain of the software and then testing the program with one value from each class. It has been adapted to test UML models in [7], and we also adapt it to test MTs. In this specific case, the input domain is defined by the input metamodel of the MT. A precise definition of a partition is given below.

Definition – Partition. *A partition of a set of elements is a collection of n subsets A_1, …, A_n such as A_1, …, A_n do not overlap and the union of all subsets forms the initial set. These subsets are called equivalence classes.*

Our adaptation of category-partition testing consists in defining partitions for those properties in the input metamodel whose type is primitive (e.g. attributes and multiplicity of associations). In [5] we propose two techniques for generating meaningful partitions: *default partitioning* and *knowledge-based partitioning*. The first technique 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}, {""}, {s / |s|>0}}, and for a [0..1] multiplicity it would be {{0}, {1}}. Knowledge-based partitioning consists of extracting representative values from the MT itself. These values can be provided by the tester or automatically extracted from the specification of the MT.

Since a partition is defined for a property of a class, it is noted Part(C::P), where C is a class and C::P a property P of C. Moreover, a partition defines a set of equivalence classes over the set of possible values for a property, so Part(C::P) = {EC_i | i∈ [1..#eqClasses]}, where EC_i is an equivalence class for C::P and #eqClasses is the number of equivalence classes.

To illustrate the proposed technique, Figure 8 displays the partitions that are obtained using a default partitioning policy for the metamodel of the Figure 3.

event	{{""}, {"evt1"}, {*}}
source	{1}
target	{1}
label	{{0}, {1}, {>1}}
container	{{0}, {1}}
incomingTransition	{{0}, {1}, {>1}}
outgoingTransition	{{0}, {1}, {>1}}
isFinal	{{true}, {false}}
isInitial	{{true}, {false}}
ownedState	{{0}, {1}, {>1}}

Figure 8 – Partitions for simple state machine metamodel

Once equivalence classes are defined, they can be combined to define *model fragments* (step 2).

3.3 Model fragments

An important notion for generating models for testing is to have criteria to qualify models. In this work, we consider that a model is relevant for testing if it contains interesting object structures. The quality of input models then directly depends on the definition of the "interesting object structures". We call such object structures *model fragments*. This paper does not focus on the selection of model fragments, which is considered as being provided by the tester to feed the test generation algorithm. A generated set of test models must cover every model fragment specified by the criterion.

The model fragments are specified using the partitions previously defined (step 2 of the Figure 6). All the proposed test criteria ensure that all the partitions for the metamodel properties are exhibited by model fragments. They differ from each other by the way they combine properties to specify object structures. Depending on the testing strategy which is adopted, it is possible to find a trade-off between the number and the efficiency of test data using either of the criteria.

The following concepts and notations are used:

- a model fragment MF is a set of object fragments: MF = {object-fragments}. It specifies that at least one of the input models must include all these object fragments
- an object fragment OF specifies a partial instance of a class C. It relates equivalence classes with properties. It is a set of property constraints, that associate a property P_j of a class C to an equivalence class EC_k: OF = $\{(C::P_j, EC_k)\}_{j,k}$.

Example:

A possible model fragment MF for the state machine is given by the following list of object fragments:

MF = {{ (AbstractSate ::outgoingTransition, {1}), (AbstractState ::label, {>1}), (AbstractState ::container, {0})}, {(Composite ::ownedState, {1})}, {(Transition ::event, {"ev1"})}}

Figure 9 displays an objects diagram that conforms to this model (1) and the corresponding graphical elements of a state machine (2). The goal of the test model generator is to complete this partial view of a model to make it conformant to the state machine metamodel. One possible resulting model is the one presented previously in Figure 2 and Figure 4. Depending on the test strategy, a test model may integrate one or several model fragments.

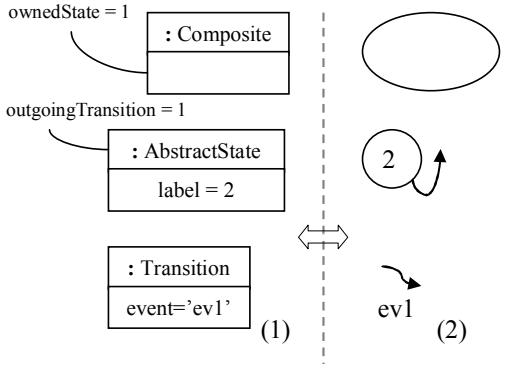

Figure 9 – Model fragment

The last step for test data generation consists in generating models that cover all the identified model fragments.

4 Test data generation: generation of input models

In this section we present an algorithm that generates a set of test models which include a set of model fragments. The algorithm is presented in Figure 10 in pseudo-code form. Besides this constant behaviour, the algorithm provides variation points that have strong effects on the resulting test models. They are presented in Table 1 and the references to them in the algorithm pseudo-code show when they are precisely used. We provide different strategies for each one (in the second column of the table).

The first variation point is the size of generated models (1 in Table 1). Each generated model covers at least one model fragment. At one extreme, one can imagine a single model that covers all object fragments, resulting in a single very large test model. At the other, one might have a separate model for each model fragment, resulting in a very large set of test data. Clearly, the former will provide problems for fault localization, while the latter will provide problems of scalability. For this reason, a test model generation algorithm should find an appropriate trade-off between these extremes. The "size of models" variation point allows the user to control the size of the input models generated by the algorithm. Each model is thus iteratively grown by including model fragments until it reaches the specified limit or can not become bigger (line 5 in Figure 10). The "max objects number" strategy considers the size of a model as the number of instances of classes it contains and allows expressing an upper limit. Similarly, the "min model fragments number" strategy allows expressing a lower limit about the covered model fragment.

After each object fragment is covered, a process is applied to the model to ensure its conformance to its metamodel. This completion process is also a variation point (2 in Table 1). It continues to grow the model until it is conformant to its metamodel. We have developed the "naïve" strategy without heuristics but it was not sufficient. For example, if two concepts in the input metamodel are linked by a bidirectional association with cardinality upper than one, the algorithm may not stop. In fact, issues concerning possible infinite cycle between instances of concepts are not correctly treated by this implementation. To deal with this issue, we have developed the "path" strategy which uses Tarjan's algorithm for avoiding cycle issues.

```
0   input : set of model fragments S, meta-model MM
1   output : set of models L conformed to MM
2   while there are uncovered model fragments in S do
3   {  create an empty model M
4      while the model size limit is not reached (1) and M still can grow do
5      {     choose an uncovered model fragment MF in S
6            for each object fragment OF in MF do
7            {     find an object O which is instance of the class partially specified by OF (3)
8                  for each constraint CT defined in OF on the property P do
9                  {     if P is an attribute  (value partition case) then
10                                choose a value and set it to P in O (5)
11                       else (multiplicity partition case)
12                       {           choose a cardinality N according to CT (5)
13                                   if the type of P is a class then
14                                         find N objects with a P type and set them to P in O (3)
15                                   else find N values in the partition of P and set them to P in O (5)
16                        }}
17                 add O to M
18                    completion of M until it is conformed to MM (2, 4)
19           }}
20        mark MF as covered
21        add M to L}
```

Figure 10 – Pseudo-code for model generation algorithm

When the algorithm is going to handle one constraint specified in an object fragment (line 9 to 18 in the algorithm), it has to choose values for the constrained property.

This choice is done in two steps:

1. Selection of an equivalence class in the partition.
2. Selection of a value in this equivalence class.

These choices follow the variation point 5 of Table 1. The "default" strategy selects randomly an equivalent class and a value in it. The "limits" strategy selects an equivalent class at random too, but the value is picked from the equivalent class limits. The "cycle" strategy selects one after the other each equivalent class, following a round trip, and then picks a value at random.

Besides, when a constraint is on the cardinality of an association, the algorithm has to select several instances of the class to assign to the concerned property. These instances can be either created or chosen among the ones contained in the model under construction. This selection is done by the "Object choice" points of variation (3, 4 in Table 1). We use it twice in the test generation process: it is first used during the model fragment coverage part of the algorithm (line 9 to 18 in the algorithm) and the other is used during the completion part (line 20 in the algorithm). For example, by running the algorithm with the set of model fragments used as an example in the section 3.3, the state machine of the Figure 2 can not be obtained with the "always create" strategy of this variation point.

To sum up, there are a number of points in this algorithm at which various strategies can be applied to control characteristics of the generated models.

Table 1 – Strategies for the algorithm

	Variation points	Provided strategies
1	Size of models	max objects number min model fragments number
2	Model completion	naive path
3	Object choice (during model fragment coverage)	always create reuse whenever possible new object for each OF
4	Object choice (during model completion)	always create reuse whenever possible
5	EC choice in partition and value choice in EC	default limits cycle

```
1. {{ label, 0 }} {{ ownedState, #0 }} {{ source, #1 }}
   {{ event, * }} {{ outgoing, #1 }}

2. {{ ownedState, #>1 }} {{ label, 1 }} {{ incoming, #>1 }}
   {{ event, "" }} {{ isFinal, true }}

3. {{ isFinal, true isInit, false container, 1
      outgoing, 0 incoming , 1 label, 1 }}
   {{ event, * source, 1 target, 1 }}
   {{ isFinal, false isInitial, true label, 1
      container, 1 outgoing, 1 incoming, >1 }}
   {{ event, "ev1" target, 1 source, 1 }}
   {{ label, * outgoing, 0 incoming, >1
      container, 0 ownedStates, 1 }}
```

Figure 11 – Sample object fragments

4.1 Example

To illustrate the algorithm, we use a simple example of MT for the flattening of composite state charts. As shown in Figure 3, the input metamodel for this transformation consists of four classes. The partitions for the properties of this metamodel are displayed in Figure 8. We propose in the Figure 11 three samples of model fragments.

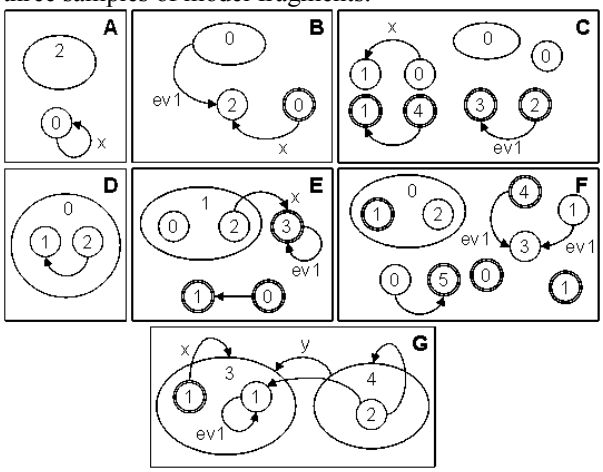

Figure 12 – Generated state machines

Figure 12 displays the models, in this case state machines, obtained by applying the algorithm with these model fragments. Each uses a size limit strategy of a minimum five model fragment per generated model. Models A, B and C correspond to the first list of model fragments using objects reuse strategies of "reuse whenever possible", "new object for each object fragment" and "always create" respectively. Models D, E and F show the models generated using the same strategies for the second set of model fragments. Model G shows the model generated for the third set of model fragments using a "reuse whenever possible" strategy.

4.2 Observations and limitations

It is clear from the generated models that the choice of strategies used in applying the algorithm has a strong effect on the nature of the resulting test data. These strategies can therefore serve as a way to tailor the generated models to the specific testing needs. For example, easy diagnosis is aided by smaller, more discrete, input models that might be generated by using an "always create" reuse strategy in combination with a small size limit. On the other hand, test suite minimisation may be achieved using a "reuse whenever possible" strategy, since this strategy will tend to generate the smallest number of models needed to cover all model fragments.

In model G we can see that a composite state contains two states with the same labels, and also a transition from an inner state to its containing state. These are valid patterns according to the metamodel but may violate a static well-formedness rule (e.g. expressed in OCL). The fact that our generator can not deal with static constraints associated to the input metamodel is an important issue for testing. Indeed, test models need to conform to the input metamodel but also satisfy the constraints to be valid input for testing.

Dealing with such constraints in the general case becomes a constraint logic programming (CLP) problem and is beyond the scope of this work. However, some simple constraints such as uniqueness of attribute values by providing extra information to the models generator. Alternatively, a crude approach of post-generation checking and acceptance/rejection, while inefficient, might prove effective. Work is currently underway to validate the algorithm and

qualify choices of strategies using empirical studies. We also plan adapting AI algorithms such as the bacteriologic algorithm [8] to take the constraints into account during the generation process.

4.3 The test generation tool

A prototype tool, called OMOGEN (autOmatic MOdel GENerator), has been implemented at France Télécom (Figure 13). It is now possible to automatically generate test data from a metamodel and a set of model fragments.

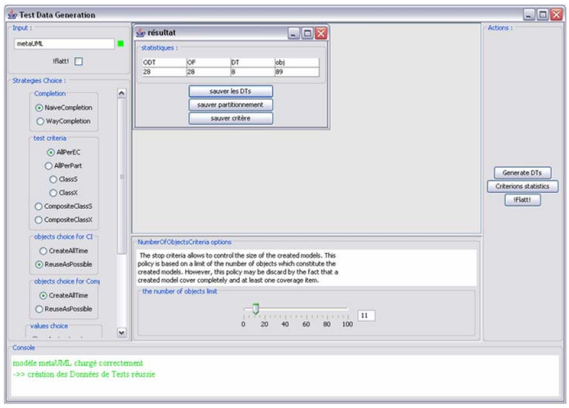

Figure 13 – OMOGEN, a test model generator for model transformations

To evaluate the efficiency of our approach, we will measure the quality of the generated models thanks to mutation techniques applied on MTPs [9]. To ease future case studies, we already have implemented such a system linked to OMOGEN to automatically produce statistics about the fault detection power of our approach. The Figure 14 presents the process which is performed by our statistics system.

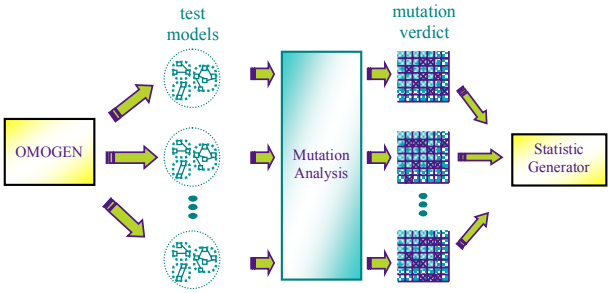

Figure 14 – automation of case study processes

5 Related work

Several works consider MTs as an essential feature in model-driven development (MDD) [1, 10, 11]. However, there are few works concerned with the validation of these particular programs. As such, this section summarizes works on the broader topic of testing in a MDD context. Among these works, three categories appear: validation for MTs [12, 13, 14], testing models [7, 15] and testing software which is developed following an MDD approach [16, 17]. The following identifies particular works in these three domains.

As stated earlier, the validation of MTs has not been studied much yet. In [14], the authors present the testing issues they have encountered when developing a MT engine, and what solutions they have adopted. They note the similarity between this task and that of testing transformations themselves, and address a number of mainly technical issues associated with using models as test data. The use of coverage criteria for the generation of test data is discussed as a possibility, although in their study the criteria are applied by hand, and not in a systematic, generalised way such as we present here.

In [13], Lin et al., identify all the core challenges for MT testing, and propose a framework that relates the different activities. The problem of test data generation is not addressed here. The authors focus more particularly on the problem of model comparison which is necessary for the oracle. They give a first algorithm inspired by graph matching algorithms. An example illustrates the different steps for testing. In [12], Küster considers rule-based transformations and addresses the problem of the validation of the rules that define the MT, i.e. syntactic correctness, convergence and termination of the set of rules.

An important testing activity in a MDD development cycle is the validation of models that drive the development of an application. In [7], Andrews et al. propose test criteria for executable UML design models. These criteria are based on the class and collaboration diagrams. The criteria on the class diagrams define different configurations which have to be covered on the model. These configurations are then instantiated to build the set of interacting objects used to test the model. In a complementary way, the criteria on the collaboration diagram define the scenarios to test the model. In [15] Goggola et al. adapt the UML validation tool USE to test UML and OCL models. The principle for the technique is to define properties that should be verified on the model. The tool then checks whether it is possible to generate snapshots from the model that verify the property.

The last activity related to this work is the adaptation of testing techniques for applications developed in a MDD context. In [16], Rutherford et al. report 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 reported benefits were that developing templates for test code enhanced the development process and allowed the developers to be more familiar with the code generated by MODEST. The costs are evaluated with an analysis of the complexity of templates for test-code generation.

In [17], the authors also explicitly address the problem of test generation in a MDE 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 models, 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. A case study based on model-driven development of web applications illustrates the approach.

6 Conclusion

With the emergence of model-driven development, model transformations appear as core assets for reuse. It thus becomes crucial to provide adequate techniques to test these model transformations. The generation of input test models has been addressed in this paper: anwell 4166-23(en)-a berth thtE beectudmodels arimeeus tn ll 41624Tw 0 -tsdel