Mutation Analysis Testing for Model Transformations

Jean-Marie Mottu[1], Benoit Baudry[1], and Yves Le Traon[2]

[1] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{jean-marie.mottu, bbaudry}@irisa.fr
[2] France Télécom R&D, 2 av. Pierre Marzin 22307 Lannion Cedex, France
yves.letraon@francetelecom.com

Abstract. In MDE, model transformations should be efficiently tested so that it may be used and reused safely. Mutation analysis is an efficient technique to evaluate the quality of test data, and has been extensively studied both for procedural and object-oriented languages. In this paper, we study how it can be adapted to model oriented programming. Since no model transformation language has been widely accepted today, we propose generic fault models that are related to the model transformation process. First, we identify abstract operations that constitute this process: model navigation, model's elements filtering, output model creation and input model modification. Then, we propose a set of specific mutation operators which are directly inspired from these operations. We believe that these operators are meaningful since a large part of the errors in a transformation are due to the manipulation of complex models regardless of the concrete implementation language.

1 Introduction

Validation refers to a process that aims at increasing our confidence that software meets its requirements. It usually relies on a combination of reasoning and testing, and encompasses unit, integration, and acceptance testing. Testing is thus a key aspect of software development, because of its cost and impact on final product reliability.

In the case of model-driven development, classical views on testing and their associated testing models are not well-suited to the significant changes this software paradigm has induced to the development process. The standardization of a model transformation language (QVT) reveals the need of a systematic way for specifying and implementing the model transformations. However, as for any other program, faults may occur in a model transformation program which must be detected through testing. Programming a model transformation is a very specific task which implies operations a classical programmer does not usually manipulate, such as navigating the input/output metamodels or filtering model elements in collections. If a skilled programmer of a model transformation can still introduce classical faults in the program, specific faults appear. These specific faults are more at a semantic level than classical programming faults. For instance, the programmer may have navigated a wrong association from class A to class B, thus manipulating class incorrect instances of the expected type. He may also be wrong in the criteria used to select some class instances (e.g. selecting all classes while he should have selected only persistent ones). Such faults are related to new fault categories we introduce in this paper.

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 376 – 390, 2006.
© Springer-Verlag Berlin Heidelberg 2006

A fundamental step in the elaboration of a test environment for a given software programming paradigm consists of defining criteria to estimate the quality of a test dataset. Structural coverage criteria are a classical way to have such an estimate. However, they are not directly related to the capacity of a test dataset to reveal faults (e.g. a faulty statement can be executed and covered several times without provoking the error).

In this paper, we focus on mutation analysis as a convincing way to check the efficiency of a test dataset for detecting faults in the program under test. Mutation analysis consists of systematically creating faulty versions of a program (called mutants) and of checking the efficiency of a test dataset to reveal the faults in these erroneous programs. The main interest of mutation analysis is to provide an estimate of the quality of a test dataset with the proportion of faulty programs it detects. Instead of structural test adequacy criteria, this estimate really reflects the "fault revealing power" [1] of the test dataset. To be effective, the mutation analysis must create mutant programs which correspond to realistic faults. In this paper, we first study the limitations of classical mutation operators for seeding model transformation programs. Then, we analyze the main activities involved in a model transformation, independently of a specific model transformation language. This analysis of model transformation leads to its decomposition in four main activities, which are fault-prone. Mutation operators are then proposed at this generic level of decomposition, which correspond to faults a programmer may (realistically) introduce into his code. The contribution of this paper is thus two-fold:

- a study of the specific faults a programmer may do in a model transformation,
- a definition of specific mutation operators for applying mutation analysis to model transformations and asses the quality of a test dataset.

The paper is structured as follows. Section 2 recalls the general process of mutation analysis. Section 3 explains the limitations of classical fault categories (classical mutation operators); studies the fault-prone activities involved in model transformations and introduce the notion of semantic mutation operators. Section 4 details the mutation operators dedicated to model transformations while Section 5 illustrates the application of two operators on several implementations of a same model transformation.

2 Mutation Testing

Mutation analysis is a testing technique that was first designed to evaluate a test dataset. It also allows to improve their effectiveness and fault revealing power [1, 2]. It has been originally proposed in 1978 [3], and consists in creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. A mutant is the program modified by the injection of a single fault. In practice, faults are modelled as a set of *mutation operators* where each operator represents a class of software faults. To create a mutant, it is sufficient to apply its associated mutation operator to the original program.

A test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a *mutation score* is associated with the test set to measure its effectiveness in terms of percentage of the revealed non-equivalent mutants.

It is to be noted that a mutant is considered *equivalent* to the original program if it does not exist any input data on which the mutant and the original program produce a different output. A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program test quality. It can be viewed as a kind of reliability assessment for the tested software. The value of the mutation analysis is based on one assumption: if the test dataset can detect that all the mutants contain fault, then this set is able to detect real involuntary errors.

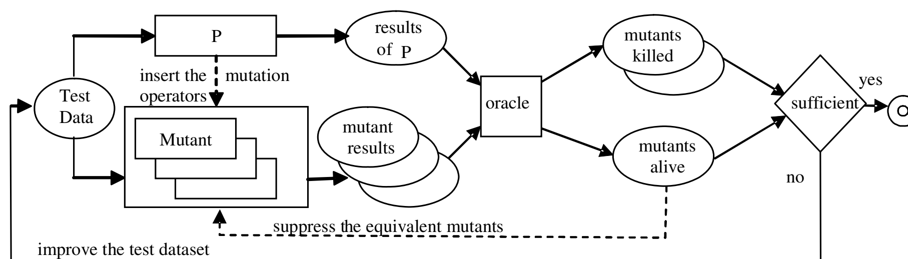

Fig. 1. Mutation process

The process of mutation analysis is presented in Figure 1 with the execution of each test datum against all the mutants of the program. An oracle function is used to determine if the failure is detected. This function compares each mutant's result with the result of the program P; the latter being considered as correct. If the results differ, it means that the test data exhibit the fault; the test data kill the mutant. During the test selection process, a mutant program is said to be *killed* if at least one test data detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test data detects the injected fault. If a mutant is alive, there are two possibilities:

- the mutant is equivalent,
- actual test data are not able to highlight this fault: the mutant is still alive and the test dataset must be involved. New test data may be generated or existing ones can be modified.

The equivalent mutants are suppressed from the set of mutants and a list of killed mutants is obtained. The mutation score for the test dataset is computed, which is the proportion of killed mutants compared to the total number of non-equivalent mutants; this score quantifies the quality of the test dataset.

$$MutationScore = \frac{\#KilledMutants}{\#Mutants - \#EquivalentMutants}$$

If the score is insufficient, we have to improve the test set, which could be done with new test data or actual data involving.

The relevance of mutation analysis is based on the mutants' relevance, which itself strongly depends on the mutation operators relevance. Classical mutation testing is related to a set of faults specified by mutation operators which define syntactic patterns which are identified in the program in order to inject a fault. Classical mutation operators include relational and arithmetic operator replacement (for example replacing a '+' with a '-'), variable and method calls replacements, statement deletion. Some operators dedicated to OO programs, and especially to Java, have been introduced by Ma et al. in [4], (method redefinition, inherited attributes etc.). These faults are related to the notions of classes, generalization, and polymorphism. These operators take into consideration specificities related to the semantics of OO languages, but remain simple faults which can be introduced by a syntactic analysis of the program. To execute mutation testing with these operators, the faults are inserted systematically everywhere the pattern is found in the code.

In the next part 3, we explain why we don't want to transpose this classical mutation process to the model oriented development.

3 Adapting Mutation Analysis to Model Driven Development

This section studies the application of mutation to model transformation programs, and shows that the classical mutation operators are not suitable for these specific programs. Faults at this level are related to the main activities involved in a model transformation.

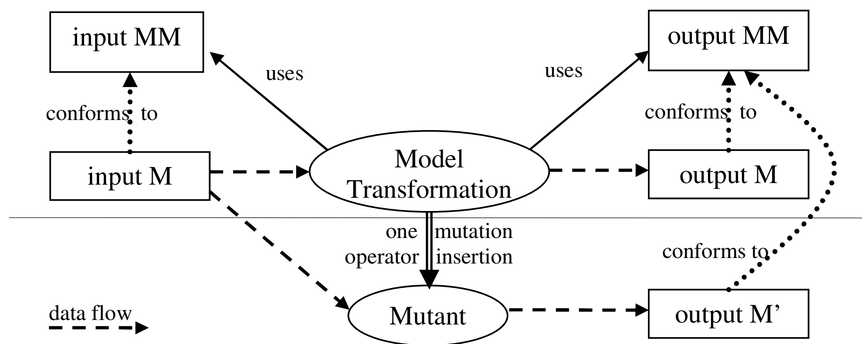

Fig. 2. Model transformation process and a model transformation mutant creation

3.1 Mutation Analysis in a Model Development Context

Before proposing fault models specific to the model transformations, we need to analyze the activities involved in development which may be fault-prone. The Figure 2 illustrates the general model transformation process. An input model is transformed and an output model is returned. Each one has its own metamodel, the transformation uses these metamodels to know which action it can process on the

input and the output models. The mutants produced by the mutation operator insertions has to preserve this conformity with the metamodels, they must be able to process the input models and must not create output models that do not conform to output metamodels. Thus mutation operators must be directly connected to the metamodel notion.

3.2 Limitations of Classical Mutation Operators

All the classical and OO operators can be applied to model transformations programs, but their relevance to this very particular context is limited due to the following reasons:

- *Mutant significance*: seeded faults are far from the specific faults a transformation programmer may do if he is competent. Indeed, the transformation programmer will make an incorrect model transformation, which seems correct from his point of view, and not only a classical programming fault. He may forget some particular cases (e.g. forget to deal with the case of multiple inheritances in an input model), manipulate the wrong model elements etc. An incorrect model transformation will differ from the correct one by complicated modification in the transformation program, and not necessarily by a single faulty statement, as in classical approaches to mutation. A semantic fault, defined to represent fault in the transformation, produces viable mutants, which are not detected simply at compilation, execution or during the rest of the programming.
- *Mutant viability*: a simple fault has a high probability to generate a non viable mutant program (that does not compile or run correctly). Since a transformation program navigates both the input and the output metamodels, most simple faults will disturb this navigation in a non-consistent way (e.g. trying to navigate non-existing association due to a syntactic replacement). Thus, these faults will be detected either during programming, at compilation or at runtime. This approach generates many mutants which are not viable candidates for mutation analysis.
- *Implementation language independency*: The semantic operators have to reflect the type of fault which may appear during the implementation of a transformation. The first constraint to define these operators is that they can not take advantage of a transformation language's syntax. Indeed, today there are lots of model transformation languages which all have their specificities and which are very heterogeneous (object oriented, declarative, functional, mixed). To be independent from a given implementation language is an important issue. That leads us to choose to focus on the semantic part of the transformation instead of the syntactic one imposed by a language, that's studied next part (3.3).

Classical mutation operators (object oriented or not) are still useful, to check code or predicate coverage, for example. However they depend on the language which is used in the implementation, thus they have to be completed by injecting faults which make sense, in terms of erroneous model transformations. These new operators that we propose in the section 4 try to capture specific faults that take into account the semantics of a particular type of program: model transformations. Such mutation operators are called semantic operators.

3.3 Semantic Faults for Model Transformations Activities

The operators introduced have to be defined based on an abstract view of the transformation program, by answering the question: which type of fault could be done during a model transformation implementation? For example, if a transformation traverses the input model to find the elements to be transformed then a fault can consist of the navigation of the wrong association in the metamodel, or of selecting the incorrect elements in a collection. During a transformation, output model elements have to be created; a fault can consist of creating elements with the wrong type or wrong initialization. The analysis of these possible faults for a model transformation leads to distinguish 4 abstract operations linked to the main treatments composing a model transformation:

- **navigation:** the model is navigated thanks to the relations defined on its input/output metamodels, and a set of elements is obtained.
- **filtering:** after a navigation, a set of elements is available, but a treatment may be applied only on a subset of this set. The selection of this subset is done according to a filtering property.
- **output model creation:** output model elements are created from extracted element(s).
- **input model modification:** when the output model is a modification of the input model, elements are created, deleted or modified.

These operations define a very abstract specification of transformations, which highlights the fault-prone steps of programming a model transformation. However, we believe they explore the most frequent important model manipulations for transformations. Operators defined at this level will have to be wrapped to real languages.

Any model transformation combines and mixes these 4 operations of navigation/ filtering (read mode) and output/input model modification (write mode). Let us consider the transformation UML to RDBMS to illustrate this decomposition (see Fig. 3). In the class diagram of the input model, the persistent classes are selected and correspondent tables are created with columns corresponding to the attributes. First the input model (a) is navigated to find the classes (b) which are filtered to keep the persistent ones (c). A table is created for each one (d). Then the navigation covers the persistent classes to find their attributes (e) (inherited attributes too) and corresponding columns are created (f). Finally columns are filtered (g) to find an appropriate key which is created (h).

Navigation, filtering, creation, modification are fault-prone operations which are sequentially dependent: while the navigation returns elements, these elements are often filtered before being used for model creation (or modification). We obtain a basic cycle which is repeated to compose a complete model transformation. The decomposition of a model transformation into such basic cycles provides an abstract view useful to inject faults. So, we define mutation operators which are applied on these basic cycles, by injecting faulty navigation/filtering/creation/modification operations.

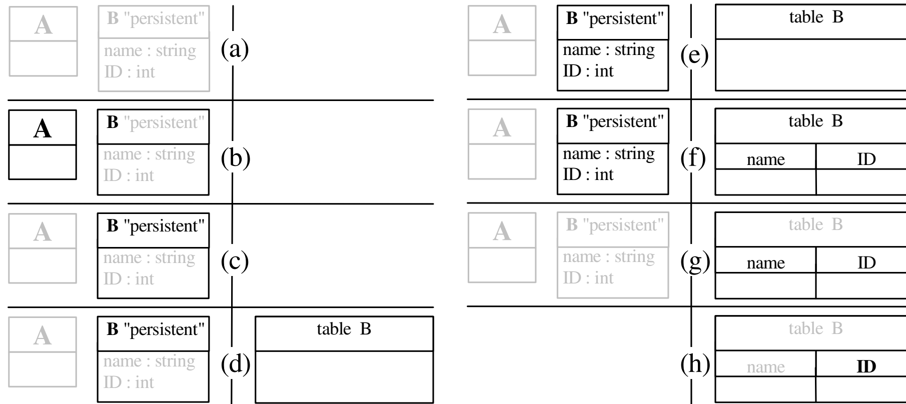

Fig. 3. Process of the transformation studied

4 Mutation Operators Dedicated to Model Transformations

Based on the analysis of the fault-prone operations which constitute a model transformation, we propose several new mutation operators. They act on the navigation and the filtering processed by the transformation on input or output models and the creation of the output model.

We present our operators starting with there names and abbreviations. Then we give a concise explanation of their functionality. The UML to RDBMS transformation and the Figure 4 (which represents a simple metamodel) will help us to illustrate our operators in a third time. Finally, we explain when these operators are the most pertinent.

4.1 Mutation Operators Related to the *Navigation*

Relation to the same class change (RSCC): This operator replaces the navigation of one association towards a class with the navigation of another association to the same class (when the metamodel allows it).

Example: the class `A` has three relations `b1`, `b2`, and `b3` to the class `B`: if the original transformation navigates `A.b1` then the operator replaces `b1` with `b2` and `b3`, making two mutants.

Different cases could occur when a wrong relation is navigated towards the same class, depending on the cardinality. To replace `A.b1` with `A.b2` leads to a cardinality difference. The results are respectively a variable and a collection, so the compilation will fail. On another hand, replacing `b2` with `b3` leads to a relevant fault (complex to detect): both returning a collection of instances of `B`. Thus the rest of the transformation being not affected, the fault is harder to detect.

Relation to another class change (ROCC): This operator replaces the navigation of an association towards a class with the navigation of another association to another class.

Example: the class `A` has several outgoing relations: `b1`, `b2`, `b3` (to `B`) and `c` (to `C`), if the transformation navigates `A.b1` then the operator creates one mutant, replacing `b1` with `c`.

This operator is really relevant when the expected class and the unwanted one have same properties used in the rest of the transformation (same attribute, method or outgoing relation). Due to inheritance, this case is very common because it directly makes direct use of the generalization process. If the wrong navigation leads to a class which has a common parent with the wanted class, the parent's attributes, methods and relations are inherited by both classes. As a consequence the rest of the transformation should not be affected by this fault. In our example, the classes `B` and `C` have a common parent `E`, so they inherited of the same attribute `name`.

Relation sequence modification with deletion (RSMD): During the navigation, the transformation can navigate many relations successively. This operator removes the last step off from the composed navigation.

Example: from an instance of `A`, we can obtain a collection of instances of `F` with the composed navigation `A.b1.f`. In the generated mutant, the navigation becomes `A.b1` and the result is not a collection of instances of `F` but of instances of `B`.

This operator leads to the same cases than the ROCC operator, which justifies its relevance.

Relation sequence modification with addition (RSMA): This operator does the opposite of RSMD. The number of mutants created depends on the number of outgoing relations of the class obtained with the original transformation.

Example: a relation is added: `A.c` becomes `A.c.d` for example. Only one mutant is created because the class `C` has a single outgoing relation.

This operator leads to the same cases than the ROCC operator, which justifies its relevance.

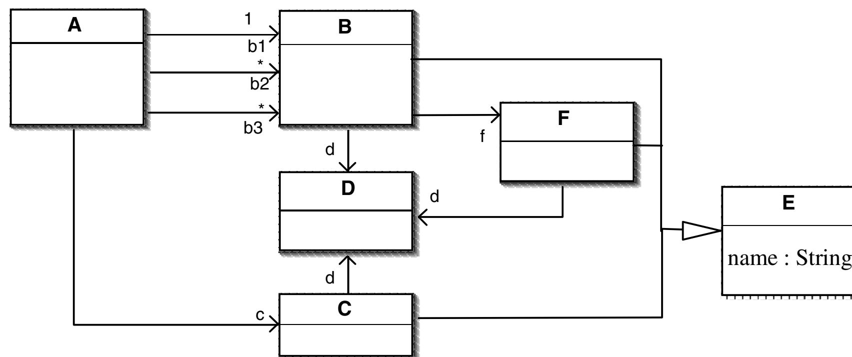

Fig. 4. Metamodel example

Mutation operators related to the *filtering*

Filtering manipulates collections to select only the elements useful for the transformation. In a general way, a filter may be considered as a guard on a collection, depending on specific criteria. Two types of filtering are considered. First, instances of a given class may be selected in function of their properties (attributes…). That's the property filtering. The second one can select some instances among a collection of instances of generic classes. That's the type filtering.

Collection filtering change with perturbation (CFCP): This operator aims at modifying an existing filtering, by influencing its parameters. One criterion could be a property of a class or the type of a class; this operator will disturb this criterion.

Example: in our transformation UML to RDBMS, this operator generates a mutant which filters depending on a wrong stereotype instead of the "persistent" one. Filtering depending on the type of the classes could also be disturbed in the Figure 4 example. The transformation could act on a collection of the generic class `E`. The instances in this collection are of type `E`, `B`, `C` or `F` (the classes of its children). If a filtering on this collection selects only the instances of `B`, this operator creates three mutants: one selects the instances of `C`, another the instances of `E` and the last the instances of `F`.

Two kinds of filtering are considered. In the simplest case, the filtering acts on a collection of instances of the same class, and depends on one of its properties. Then it is viable because the rest of the transformation won't be influenced. Indeed, the expected erroneous collections just have a different size. Secondly, we can consider the filtering depending on the type. To filter a collection of a generic class depending on this class or any of the classes of its children makes no difference. All these classes share the same inherited properties (attributes, methods, relations), so the fault injected by this operator will not be discovered.

Collection filtering change with deletion (CFCD): This operator deletes a filter on a collection; the mutant returns the collection it was supposed to filter.

Example: in the UML to RDBMS example, only persistent classes are used. The navigation provides all the classes and the filtering selects the persistent ones to create correspondent tables. The operator suppresses the filtering and the transformations will create tables for all classes, even the ones that are not persistent. It could be a filtering depending on the class types: the filtering of a collection of classes of type `E` and its specialization classes could be useful to select only the `A` instances. The mutant will kept all the instances without type consideration.

This operator leads to the same cases than the CFCP operator, which justifies its relevance.

Collection filtering change with addition (CFCA): This operator does the opposite of CFCD. It uses a collection and processes a useless filtering on it. This operator could return an infinite number of mutants, we have to restrict it. We choose to take a collection and to return a single element arbitrarily chosen.

Example: we do not need to illustrate this one because of its clearness.

This operator leads to the same cases than the CFCP operator, which justifies its relevance.

Mutation operators related to the *creation*

These operators are based on two abstract operations: the creation of the output model elements and the creation part of the modification operation.

Class' compatible creation replacement (CCCR): This operator replaces the creation of an object by the creation of an object of a compatible type. It could be an instance of a child class, of a parent class or of a class with a common parent.

Example: if the transformation creates an instance of B (one child of E), then the operator creates three mutants: one creating an instance of C, one an instance of F, the last an instance of E.

This operator is really interesting because the wrong created class and the right one have common inherited properties (relations, methods and attributes), then the rest of the transformation could be not affected and the fault is not detected.

Classes' association creation deletion (CACD): This operator deletes the creation of an association between two instances.

Example: when the transformation creates the relation b1 between an instance of A and an instance of B, the corresponding mutant does not create this relation.

If the relation not created has the cardinality 1 in the metamodel (or a bigger one fixed) then its absence can affect the rest of the transformation. But if the cardinality is n, then the transformation will not much be affected. The model can have an instance of A connected towards several instances of B with relations b1. If one relation is not created, the navigation A.b1 will just return a collection deprived of one element, without detectable consequence.

Classes' association creation addition (CACA): This operator adds a useless creation of a relation between two class instances of the output model, when the metamodel allows it.

In our example, the metamodel (Figure 4) allows three different relations between A and B. Then if the transformation manipulates instances of A and B then the operator generates three mutants, each one creating one of the relations b1, b2, b3 between A and B, even if one of them already exists (then overwritten relation could point to a different instance that the original one).

The errors injected by this operator are not easily detectable: collections will just have one item added and relations of cardinality 1 will just be created or overwritten.

These faults are directly linked to the way we design model transformations which are divided following the four abstract operations (navigation, filtering, creation, modification). These operators aim to generate viable mutants with pertinent faults to improve the value of the mutation analysis. This allows improving test dataset capacity to detect the faults of the programmer.

5 Examples

In this section, we present how two of our mutation operators are implemented in different languages. This reveals that the distance between the specification of a semantic mutation operator and its implementation varies with the model transformation language. This also emphasizes the need to present a semantic vision of the mutation analysis in model oriented programming instead of a syntactic one directly associated to specific languages.

We use samples of a model transformation written for the workshop MTIP (part of the MoDELS 2005 Conference). We selected three different languages to illustrate the different implementations of two mutation operators (CFCP and CFCD). The model transformation studied is quite similar to the UML to RDBMS transformation

used in this paper: we study the filtering operation which selects the persistent classes to create correspondent tables.

The mutation operator CFCP will perturb the filter. In this example, the mutation operator takes the negation of the filtering condition: the non persistent classes are selected. More complex perturbations may be implemented (e.g. replacing is_ persistent with any other Boolean attribute of the class). The mutation operator CFCD will delete the filtering, tables are created for all the classes. Due to the fact the example is very simple, the seeded faults would be easily detected with a simple oracle function.

5.1 Operator Implementation with an Imperative Model Transformation Written in *Kermeta*

The transformation written in *Kermeta* is published in [5]. The excerpt is:

```
getAllClasses(inputModel)
     .select{c|c.is_persistent}
     .each { c | var table:Table init Table.new
                 Table.name:=c.name
                 Class2table.storeTrace(c,table)
                 Result.table.add(table)
            }
```

In this example, the filtering (on the collection elts) is just a conditional expression, line 2. If we apply the CFCP operator, a mutant is generated with the code:

```
getAllClasses(inputModel)
     .select{c|not c.is_persistent}
     .each { c | var table:Table init Table.new
                 Table.name:=c.name
                 Class2table.storeTrace(c,table)
                 Result.table.add(table)
            }
```

If we apply the CFCD operator, a mutant is generated with the code:

```
getAllClasses(inputModel)
     .each { c | var table:Table init Table.new
                 Table.name:=c.name
                 Class2table.storeTrace(c,table)
                 Result.table.add(table)
            }
```

In *Kermeta*, filtering a collection is really simple because the operation `select` has this role. So mutation operators related to the filtering are quite simple to implement.

5.2 Operator Implementation with a Declarative Model Transformation Written in *Tefkat*

The transformation written in *Tefkat* is published in [6]. The sample is:

```
RULE ClassAndTable(C, T)
  FORALL Class C {
    is_persistent: true;
```

```
    name: N;
  }
  MAKE Table T {
    name: N;
  }
  LINKING ClsToTbl WITH class = C, table = T
;
```

In this example the filtering (on the collection elts) is just a conditional expression, line 3. If we apply the CFCP operator, a mutant is generated with the code:

```
RULE ClassAndTable(C, T)
  FORALL Class C {
    is_persistent: false;
    name: N;
  }
  MAKE Table T {
    name: N;
  }
  LINKING ClsToTbl WITH class = C, table = T
;
```

If we apply the CFCD operator, a mutant is generated with the code:

```
RULE ClassAndTable(C, T)
  FORALL Class C {
    name: N;
  }
  MAKE Table T {
    name: N;
  }
  LINKING ClsToTbl WITH class = C, table = T
;
```

With this declarative language, code modifications are also relatively simple. The rules are written close to the transformation design. Mutation operator implementation does not affect many statements of code.

5.3 Operator Implementation with a Language Not Devoted to the MDE, *Java*

We wrote the entire transformation using *Eclipse Modeling Framework* (EMF). The sample we are interested in is:

```
Vector cls = getClasses(modelUse);
Iterator itCls = cls.iterator();
while (itCls.hasNext()){
  Class c = (Class)(itCls.next());
  if (not c.is_persistent){
      cls.remove(c);
  }
}
createTables(cls);
```

In this example the filtering (on the collection `cls`) is implemented from line 2 to 9.

If we apply the CFCP operator, a mutant is generated with the code:

```
Vector cls = getClasses(modelUse);
Iterator itCls = cls.iterator();
while (itCls.hasNext()){
  Class c = (Class)(itCls.next());
  if (c.is_persistent){
      cls.remove(c);
  }
}
createTables(cls);
```

If we apply the CFCD operator, a mutant is generated with the code:

```
Vector cls = getClasses(modelUse);
createTables(cls);
```

If the CFCP operator modifies only one statement, the CFCD one affects a larger part of the program. In Java, the distance between the specification of the mutation operator and their implementation can be higher.

5.4 Implementing Semantic Mutation Operators

The feasibility of the implementation depends on the operators and the language. To have a comparison basis with existing mutation tools (*MuJava* [4]), we applied manually the operators on the *Java* implementation and we get the following results:

	RSCC	ROCC	RSMD	RSMA	CFCD	CFCP	CFCA	CACD	CACA	CCCR	Total
#mutants	5	13	2	7	2	6	7	3	1	0	46
implementation difficulty	A	A	A	A	A	B	B	A	C	B	

The first line presents the number of generated mutants for each operator (46 mutants were generated), and the second line gives a qualitative estimate of the difficulty for seeding the program per operator. An “A” means that the operator is easy to implement, a “B” that it is difficult and a “C” that it forces a very careful analysis of the code. Depending on the target language, this difficulty may be different, but we believe that it still corresponds to the difficulty to automate the fault injection with a dedicated tool.

With *MuJava*, 96 mutants were generated and 19 were not viable (detected at compile or runtime). With the specific mutants, 2 were not viable among the 46. Combining classical and dedicated mutants would allow a more complete verification of the quality of a test dataset. Classical mutant operators capture simple programming faults which still exists in any programming language, but the semantic operators go further and improve the level of confidence in the dataset by capturing fault related to the MDD.

6 Related Work

Several works consider model transformations as an essential feature in model driven development (MDD) [7, 8]. However, there are few works concerned with the validation of these particular programs.

As stated earlier, the validation of model transformations has not been studied much yet. In [9], the authors present the testing issues they have encountered when developing a model transformation 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. In [10], Lin et al., identify all the core challenges for model transformation testing, and propose a framework that relates the different activities. The authors focus more particularly on the problem of model comparison which is necessary for the comparison of models produced by the original program and the mutants. They give a first algorithm inspired by graph matching algorithms. In [11], Küster considers rule-based transformations and addresses the problem of the validation of the rules that define the model transformation, i.e. syntactic correctness and termination of the set of rules. In [12], we looked at the problem of test data generation for model transformation and proposed to adapt partition testing to define test criteria to cover the input metamodel (that describes the input domain for a transformation).

Mutation analysis has often been studied in classical and object oriented programs, like *Java* [2, 4]. In [13], authors studied how to apply mutation to components' interfaces. They do not base their analysis at the code implementation level of a component, but at its interface level. In [14], mutation analysis is studied in a UML context. The idea is to propose a taxonomy of faults when designing UML class diagrams. The work presented in this paper focuses on the specific faults related to model transformations and not on the way models may be faulty.

7 Conclusion and Future Work

The approach presented in this paper aims at adapting mutation analysis for building trust into model transformation programs using this technique. By measuring the quality of a test dataset with mutation, we seek to build trust in a model transformation passing those tests. To adapt mutation analysis, we first studied the main activities involved in a model transformation and deduced some categories of faults a programmer may do. Mutation operators have been proposed for the specific paradigm of model transformation and illustrated on several implementations of the same model transformation.

Further work will consist in addressing the issue of the operators implementation in a tool (for a well-chosen model transformation language) and in conducing experi-mental studies for validating the relevance of mutation operators to the MDD context.

References

1. Voas, J.M. and K. Miller, *The Revealing Power of a Test Case.* Software Testing, Verification and Reliability, 1992. **2**(1): p. 25 - 42.
2. Offutt, A.J., J. Pan, K. Tewary, and T. Zhang, *An experimental evaluation of data flow and mutation testing.* Software Practice and Experience, 1996. **26**(2).
3. DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
4. Ma, Y.-S., J. Offutt, and Y.R. Kwon, *MuJava : An Automated Class Mutation System.* Software Testing, Verification and Reliability, 2005. **15(2)**: p. 97-133.
5. Muller, P.-A., F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. *On Executable Meta-Languages applied to Model Transformations*. in *Model Transformation in Practice Workshop, part of the MoDELS 2005 Conference*. 2005. Montego Bay, Jamaica.
6. Lawley, M. and J. Steel. *Practical Declarative Model Transformation With Tefkat*. in *Model Transformation in Practice Workshop, part of the MoDELS 2005 Conference*. 2005. Montego Bay, Jamaica.
7. Bézivin, J., N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. *Reflective model driven engineering*. in *UML'03*. 2003. San Francisco, CA, USA.
8. Judson, S.R., R. France, and D.L. Carver. *Model Transformations at the Metamodel Level*. in *Workshop in Software Model Engineering (in conjunction with UML'03)*. 2003. San Francisco, CA, USA.
9. Steel, J. and M. Lawley. *Model-Based Test Driven Development of the Tefkat Model-Transformation Engine*. in *ISSRE'04 (Int. Symposium on Software Reliability Engineering)*. 2004. Saint-Malo, France.
10. Lin, Y., J. Zhang, and J. Gray, *A Testing Framework for Model Transformations*, in *Model-Driven Software Development - Research and Practice in Software Engineering*. 2005, Springer.
11. Küster, J.M. *Systematic Validation of Model Transformations*. in *WiSME'04(associated to UML'04)*. 2004. Lisbon, Portugal.
12. Fleurey, F., J. Steel, and B. Baudry. *Validation in Model-Driven Engineering: Testing Model Transformations*. in *MoDeVa*. 2004. Rennes, France.
13. Ghosh, S. and A. Mathur, *Interface mutation.* Software Testing, Verification and Reliability, 2001. **11**(4): p. 227-247.
14. Trung, D.-T., S. Ghosh, F. Robert, B. Baudry, and F. Fleurey. *A Taxonomy of Faults for UML Designs*. in *2nd MoDeVa workshop - Model design and Validation, in conjunction with MoDELS05*. 2005. Montego Bay, Jamaica.