Reusable MDA Components: A Testing-for-Trust Approach

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

[1] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{bbaudry, jmottu}@irisa.fr
[2] France Télécom R&D/MAPS/EXA, 2 avenue Pierre Marzin, 22307 Lannion Cedex, France
yves.letraon@francetelecom.com

Abstract. Making model transformations trustable is an obvious target for model-driven development since they impact on the design process reliability. Ideally, model transformations should be designed and tested so that they may be used and reused safely as MDA components. We present a method for building trustable MDA components. We first define the notion of MDA component as composed of its specification, one implementation and a set of associated test cases. The testing-for-trust approach checks the consistency between these three facets using the mutation analysis. It points out the lack of efficiency of the tests and the lack of precision of the specification. The mutation analysis thus gives a rate that evaluates: the level of consistency between the component's facets and the level of trust we can have in a component. Relying on this estimation of the component trustability, developers can consciously trade reliability for resources.

1 Introduction

MDA (Model Driven Architecture) is a very promising framework to promote high level reuse for software development. Instead of reusing code, the MDA proposes to reuse models for software design and to make these models first-class assets. In this context, models become more than illustrations for documentation, they are assets that can be manipulated, stored and modified by tools. Model transformations encapsulate specific techniques to manipulate and create models and they are used all along the development to introduce different design aspects. These transformations are important assets for reuse in MDA that are meant to be deployed in different software developments and at different times in the development. Thus, they must be analysed, designed and implemented using sound software engineering techniques to ensure that they are reused safely.

In this paper, we propose to encapsulate model transformations as components that are called MDA components. The contribution of this work is to propose a model for trustable components as well as a methodology to design and implement such components. Trustworthiness is a general notion which includes security, testability, maintainability and many other concerns. In this paper, we claim that trust is, in first, dependent on the quality of the tests in relation with the completeness of the specification, captured by executable contracts (as defined in design-by-contract [1]).

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 589 – 603, 2006.

While building MDA components, we consider it as an organic set composed of three facets: test cases, an implementation and contracts defining its specification. A trustable component is considered as being "vigilant" [2], in the sense it embeds contracts accurate enough to detect most of the erroneous states at runtime. Trust is evaluated using a testing-for-trust process and reflects the consistency between the specification and the implementation of the component.

This work details this three-facet model for a MDA component and explains how good test cases can be used to evaluate the trust associated to the component. Two major concerns are addressed for this study. First, we investigate the use of *mutation analysis* to check the consistency between the three facets of a component. This technique, originally designed to assess the efficiency of test cases, consists in systematically injecting faults in the program under test. It is then possible to measure how many faults the test cases or the contracts can detect. The rate of detected faults is called the mutation score. This technique was originally proposed for procedural programs and has been adapted to object-oriented programs. Here we study how mutation analysis has to be adapted to evaluate the trust in MDA components. Second, we discuss the expression of contracts for these components. Since there is no standard today for the specification or the implementation of model transformations (QVT, EMF, Kermeta…), this work proposes to establish a taxonomy of contracts.

Section 2 presents the notion of MDA components and the process we propose to build trustable components. Section 3 details the mutation analysis technique and its adaptation to the context of model transformation then section 4 details the taxonomy of contracts for MDA components. Section 5 illustrates the application of the process step by step on the UML2RDBMS example and provides first experimental results.

2 Trusting MDA Components

In this section, we propose a model for a MDA component that provides a basis on which a testing-for-trust process can be applied. Then we present this process which aims at improving the trust one may have in a MDA component by improving the efficiency of test cases and the accuracy of the specification.

2.1 Building MDA Components with the Triangle View

The proposed model to build trustable MDA components is based on an integrated design and test approach for software components. It is particularly adapted to a design-by-contract [1] approach, where the specification is systematically translated into executable contracts (invariant properties, pre/postconditions of methods). In this approach, test cases are defined as being an "organic" part of a component: a component is composed of its specification (documentation, methods signature, invariant properties, pre/postconditions), one implementation and the test cases needed for testing it (V&V: Validation and Verification). Fig. 1 illustrates this model of a component with a triangle representation.

From a methodological point of view, we argue that the trust we have in a component depends on the consistency between its three facets. The comparison between these three facets leads to the improvement of each one. The improvement of each

facet as well of the global consistency is an iterative process. First, a good set of test cases is generated. Then it is possible to improve the implementation: running the good test cases allows to detect faults and to fix them. At last the accuracy of contracts can be improved to make them effective as an oracle function for test cases.

Several difficulties arise in running this process. First, test generation since input data are models for a MDA component. Here, we consider that the test cases are provided by the tester. Second, the oracle function for a MDA component which can either be provided by assertions included in the test cases or by contracts. In a design-by-contract approach, our experience is that *most* of the decisions are provided by contracts. The last difficulty is the writing of contracts for a MDA component since there is no clear definition of what contracts for model transformations should be.

The trust in the component is thus related to the test cases effectiveness and the contracts "completeness". We can trust the implementation since we have tested it with a good test set, and we trust the specification because it is accurate enough to derive effective contracts as oracle functions.

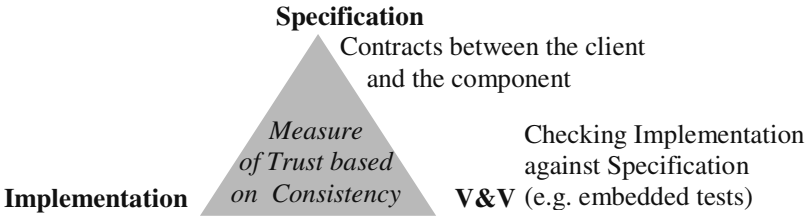

Fig. 1. Trust based on triangle consistency

2.2 The MDA Trusted Component Process

The process for building trust in a MDA-component consists in three steps as presented in Fig. 2. The process starts with an initial component that already has its three facets ready. The goal is then to improve each facet and to check the global consistency between facets. On the right side of the figure, the evolution of the trust we have in each facet of the triangle view of the MDA component is highlighted.

Step 1- Test cases improvement. It aims at improving the test set using a relevant test quality criterion. In this paper, we suggest to evaluate this quality using mutation analysis. We adapt it to evaluate the efficiency of test cases for model transformations programs, as detailed in section 3. This analysis produces a ratio, which estimates the *fault revealing power* of a test set (0 means that the test set does not detect any fault, and 1 means that all the seeded faults have been detected). We note Q_{td} this ratio and section 3 will go in the detail of its computation in the form of a *mutation score* (MS).

If this ratio is low when running a mutation analysis, it means that the test set has to be improved. This can be achieved by adding test cases by hand or by automatic optimization techniques. Step 1 ends when the quality of the test set reaches a satisfactory quality (good mutation score).

Step 2- Implementation improvement. This step leads to the correction of the model transformation program with the efficient test set. Indeed, test cases generated at

previous step have a high mutation score and are thus able to detect faults in the model transformation program. At this stage, the tester has to run these test cases on the model transformation, and, if some faults are detected, they have to be localized and fixed. At the end of step 2, the implementation facet is trustable, since its correctness has been validated with efficient test cases.

Step 3- Contracts improvement. It aims at embedding accurate contracts, considered as the executable part of the specification. The precondition for this step is that the MDA component has to possess a trustable implementation and efficient test cases. We propose using the mutation analysis again to estimate the accuracy of contracts in terms of *fault detection rate* (i.e. the proportion of faults the contracts actually detect knowing the test cases provoke a faulty output). It thus estimates the accuracy of contracts as embedded oracles.

The difference with step 1 is that the fault revealing power of the test data set is known with the Q_{td} value. Using contracts as embedded oracles, the new proportion of detected faults (%detected-faults) is computed. The quality of contracts is checked w.r.t Q_{td}. The quality estimated for contracts Q_{cont} is thus defined as:

$$Q_{cont} = \%\text{detected-faults} / \ Q_{td}.$$

This can be seen as an estimate of the accuracy of contracts to be a valid oracle for the test cases. So, when Q_{cont} equals 0, it means that no fault is detected by contracts and when Q_{cont} equals 1, it means that contracts are able to detect all the faulty outputs. So step 3 consists in improving contracts until the expected quality level is reached. At the end of the process, the contracts are embedded in the MDA component and it becomes 'vigilant', e.g. contracts have a good probability to dynamically detect faulty states [2].

When the three steps process ends, we obtain a MDA component with an estimate of the trust we may have in the test cases, the implementation and the contracts. This estimate has checked the consistency between test cases and implementation, and then between test cases, implementation and contracts. In section 3, we detail how the mutation analysis is adapted to the specific context of model transformation, and how contracts can be expressed in the particular context of model transformation.

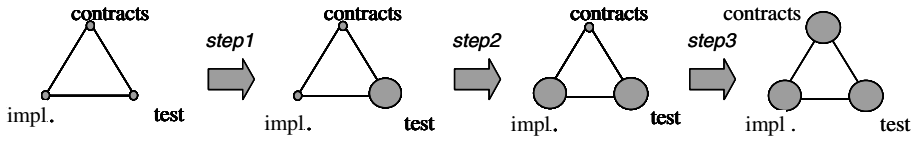

Fig. 2. Building a MDA trusted component

3 Mutation Analysis for Model Transformations

Mutation analysis is a testing technique that was first designed to evaluate the efficiency of a test set. It also allows to improve its effectiveness and fault revealing power. Originally proposed in 1978 [3], mutation analysis 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.

The process is presented in Fig. 3 with the execution of each test case against all the mutants of the program. 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. A mutation operator is applied to the original program to create each mutant.

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 one result differs, it means that one test case exhibits the fault; the mutant is *killed*. The mutant stays *alive* if no test case detects the injected fault. Sometimes, a mutant can never be killed, it is an *equivalent mutant* and it has to be suppressed from the set of mutants.

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

MutationScore = #KilledMutants / (#Mutants - #EquivalentMutants)

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

The value of mutation analysis is based on one assumption: if the test set can kill all the mutants, then this test set is able to detect real involuntary errors.

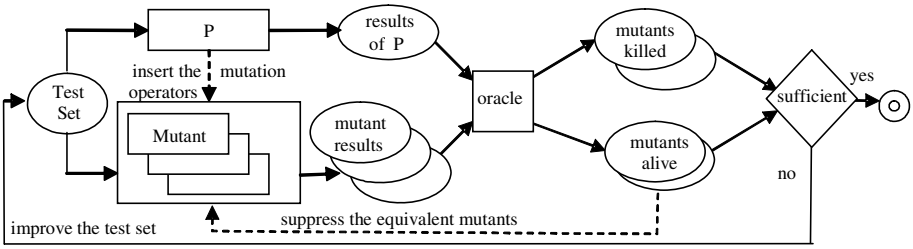

Fig. 3. Mutation process

The relevance of mutation analysis depends on the relevance of the mutants, which itself strongly depends on the relevance of the mutation operators. Classical mutation analysis 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. For example, classical mutation operators include arithmetic operator replacement (like replacing a '+' with '-'). 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 analysis with these operators, the faults are inserted systematically everywhere the pattern is found in the code.

In the next section, we explain why we do not want to transpose this classical mutation process to the model oriented development.

3.1 Mutation Analysis in a Model Development Context

All the classical and OO operators can be applied to model transformation programs, but their relevance to this 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. A transformation programmer will make not only classical programming faults but also specific faults related to the model transformation. 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. A wrong model transformation will differ from the correct one by complicated modifications in the transformation program.
- *Implementation language independency:* Today there are lots of model transformation languages with their own specificities and which are very heterogeneous (object oriented, declarative, functional, mixed). Thus we can not take advantage of a transformation language's syntax. 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. So we introduce semantic operators, they have to reflect the type of fault which may appear. That is studied next section (3.2).

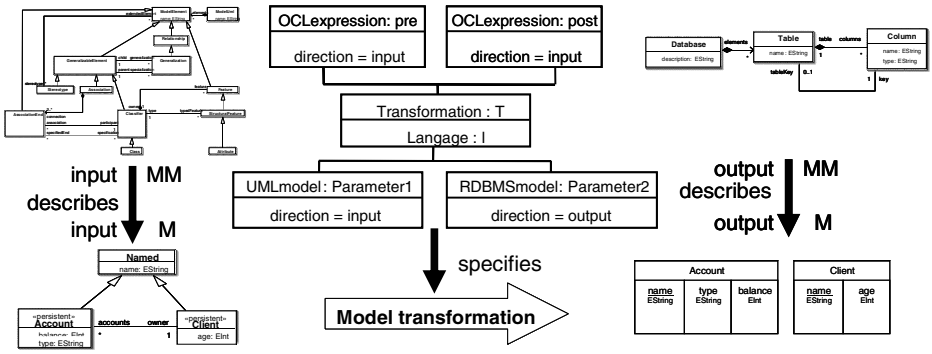

Fig. 4. Model transformation process

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 transformation. These new operators that we propose try to capture specific faults that take into account the semantics of a particular type of program: model transformation. Such mutation operators are called semantic operators.

We need to analyze the activities involved in the development of model transformations (Fig. 4) which may be fault-prone. The mutants produced by the mutation operator insertions has to preserve the conformity towards the metamodels involved in the transformation; they must be able to process the input models (depending on their metamodels) and must not create output models that do not conform to output metamodels. Thus mutation operators must be directly connected with the metamodel notion.

3.2 Semantic Faults for Model Transformations Activities

The operators introduced have to be defined based on an abstract view of the transformation program, by answering that question: which type of fault could be done during a model transformation implementation? For example, a transformation goes all over the input model to find the elements to be transformed, a fault can consist in the navigation of the wrong association in the metamodel, or in selecting the wrong elements in a collection. During a transformation, output model elements have to be created; a fault can consist in 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 elements.
- *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 model manipulations for transformations.

The decomposition of a model transformation with these fault-prone operations provides an abstract view useful to inject faults. We define mutation operators which are applied by injecting faulty navigation/filtering/creation/modification operations.

3.3 Example of Two Mutation Operators Dedicated to Model Transformation

For sake of conciseness, only two mutation operators are detailed. We present more operators in [5]. The example UML2RDBMS is a transformation which creates a set of tables (database) from the persistent classes of a UML diagram.

3.3.1 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, methods, relations). 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.

In our transformation UML2RDBMS, this operator generates a mutant which filters the non "persistent" classes instead of the "persistent" one. In both cases, the filtering acts on a collection of instances of the same type. Then it is viable because the rest of the transformation will not be influenced.

Filtering depending on the type of the classes could also be disturbed. A transformation could act on a collection of the generic class `E`. The instances in this collection are of type `E`, or its children classes (`F` and `G` for example). If a filtering on this collection selects only the instances of `F`, this operator creates two mutants: one selects the instances of `G` and the other the instances of `E`. All these classes share the same inherited properties. Then the fault injected by this operator will not be discovered. Even if a class redefines a property, the programmer should not detect the fault.

Collection filtering change with deletion (CFCD): This operator deletes a filter on a collection; the mutant returns the collection it was supposed to filter. This operator leads to the same cases than the CFCP operator, which justifies its relevance.

3.3.2 Operators 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:

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

Here, the filtering (on the collection `cls`) is implemented from line 2 to 5. If we apply the CFCP operator, a mutant is generated with the code:

```
ELists cls = getClasses(modelUse);
Iterator itCls = cls.iterator();
while (itCls.hasNext()){
  Class c = (Class)(itCls.next());
  if (c.is_persistent) cls.remove(c);    }//while //mutant without !
createTables(cls);
```

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

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

If the CFCP operator modifies only one statement, the CFCD affects a larger part of the program. This illustrates the fact that we need operators that are more than syntactic changes.

4 Contracts as Embedded Oracles

Embedded oracles – predicates for the fault detection decision – can either be provided by assertions included in the test cases or by executable contracts. Contracts are expressed at specification level and translated into executable properties which are checked at execution runs of the transformation. Today, there is no standard to express contracts for model transformations or to define the specification of a transformation. In the following, we define two levels of contracts w.r.t. the classification of [6]:

Basic contracts: The first level, basic, or syntactic, contracts, is required simply to make the system work. Interface definition languages, as well as typed object-based or object-oriented languages, let the component designer specifies the operations a component can perform, the input and output parameters each component requires, and the possible exceptions that might be raised during operation. Applied to MDA components, the input/output metamodels describe the "types" of the manipulated data. They are parts of the specification: the input/output models must be conformant to their respective metamodels. While for a classical procedure, a programmer can declare a variable as being an integer instead of a float, it is usual to consider that a model transformation takes a model as input not conformed to a whole metamodel but to one of its sub-part. For example, the UML2RDBMS transformation is restricted to the parts of the UML metamodel which describe what classes and their relations are. Thus contracts will check the conformance to this restricted metamodel that we called effective. Finally, we have contracts on the input models (*basic precondition contracts*) and on the resulting models (*basic postcondition contracts*).

Behavioral semantic contracts: The second level, behavioral and semantic contracts, improves the level of confidence in the execution context for the specific model transformation. Because the UML2RDBMS input metamodel does not define precisely that the input model must include at least a class with at least one attribute, the user can only guess this fact. So, specific properties can be attached to the input domain of the transformation, which play the role of preconditions specific to the transformation. In classical programming, a programmer can write that the character variable should only be equal to 'a' or 'b' for example. So, if the effective metamodel describes the "type" of an input/output model, the precondition can express properties which must be true only for the given transformation. Three categories of contracts can be expressed:

- *domain contracts (or precondition)* are properties specifying the input domain more precisely than a simple metamodel,
- *range contracts (or postconditions)* are specific properties on the output models,
- *domain/range contracts (or postconditions)* express properties linking the input and output models).

In a design-by-contract approach, our experience is that *most* of the oracle verdicts are provided by contracts derived from the specification. The fact that the contracts of

components are inaccurate to detect a fault exercised by the test case reveals a lack of precision in the specification. The specification should be refined and new contracts added. The trust in the component is thus related to the test set efficiency and the contracts "completeness". We can trust the implementation since we have tested it with a good test set, and we trust the specification because it is precise enough to derive accurate contracts as oracle functions.

5 The Testing for Trust of a MDA Component Illustrated

This section illustrates the three steps to improve the trustability of a MDA component using an example taken from the UML2RDBMS transformation implemented in Java. Mutation is applied on a particular method of the transformation (`createColumns`) and the improvement of the test set and the contracts to detect the mutants is illustrated. The `createColumns`, given below, takes a class and a RDBMS table as a parameter and adds one column in the table for each attribute of the class. The method is called recursively to add columns that correspond to inherited attribute.

```
private static Table createColumns(Class classUse, Table tableUse) {
   Iterator itClass = classUse.getFeature().iterator();
   while(itClass.hasNext()){
     metaUML.Attribute attributeUsed=(metaUML.Attribute)itClass.next();
     Column newColumn = MetaRDBMSFactory.eINSTANCE.createColumn();
     newColumn.setName(attributeUsed.getName());
     newColumn.setType(attributeUsed.getType().getName());
     tableUse.getColumns().add(newColumn);
   }//while
   Iterator itGeneralization = classUse.getGeneralization().iterator();
   if( itGeneralization.hasNext()){
     tableUse=createColumns((Class)
       ((Generalization)itGeneralization.next()).getParent(), tableUse);
   }//if
   return tableUse;
}//method createColumns
```

5.1 Test Set Improvement

The first step of the process consists in evaluating the quality of the test set. Mutation analysis allows computing a mutation score that evaluates the proportion of simple errors the test set can detect. If this score is not acceptable, meaning that the test cases do not detect enough injected errors, the test set must be improved.

For example, let us consider the following excerpt of the `createColumn` method:

```
private static Table createColumns(Class classUse, Table tableUse) {

   ……

   Iterator itGeneralization = classUse.getGeneralization().iterator();
   if( itGeneralization.hasNext()){
      tableUse=createColumns((Class)
        ((Generalization)itGeneralization.next()).getParent(),tableUse);
   }//if

   return tableUse;

}//method createColumns
```

The CFCP operator can modify the condition in the "if" statement to create the following mutant:

```
private static Table createColumns(Class classUse, Table tableUse) {
   ……
   Iterator itGeneralization = classUse.getGeneralization().iterator();
   if( ! itGeneralization.hasNext()){
      tableUse=createColumns((Class)
        ((Generalization)itGeneralization.next()).getParent(),tableUse);
   }//if
   return tableUse;
}//method createColumns
```

To kill this mutant, it is necessary to add a test case which produces different outputs between the mutant and original version. We need an input model with a class that has a super class with at least one attribute. Several models may be generated, with simple or multiple inheritance. For example let us consider a model called "Model 1" that contains a class A which inherits from two classes B and C (Fig. 5). So, applying this analysis to the whole set of mutants forces the creation of new test cases (input models) which exercise the model transformation program more efficiently than the simple test set provided by the tester.

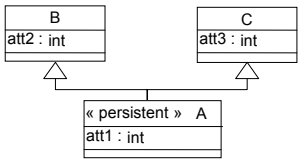

Fig. 5. Model 1

5.2 Implementation Improvement

When efficient test cases (according to the mutation analysis) have been produced, they are executed against the implementation to detect errors in the program. For example, when running the transformation with the model 1, an error is detected: the produced table contains only two columns named att1 and att2, it does not contain the column called att3. The error is due to the fact that, the transformation only considers one super class when it looks for attributes from super classes to add new columns in a table being build. The correct version of the program is given below (the "if" is replaced by a "while": all the super classes are navigated instead of only one):

```
private static Table createColumns(Class classUse, Table tableUse) {

   ……
   Iterator itGener = classUse.getGeneralization(). iterator();
   while( itGener.hasNext())
   {
      tableUse=createColumns((Class)
        ((Generalization)itGener.next()).getParent(),tableUse);}//while
   return tableUse;
}//method createColumns
```

If several errors are detected and fixed after running the test cases, a new mutation analysis is ran with the corrected program since the mutants will be slightly different from the ones generated in previous analysis. Once efficient test cases are available and the program has been fixed, it is possible to improve the contracts.

5.3 Contracts Improvements

While for the step 1 (test set improvement) the difference between the outputs of the original and the mutant programs was used to check the efficiency of the test set, step 3 is based only on the use of the mutants previously killed to evaluate the contracts. A mutant that is not killed at step 3 using contracts while at least one test case killed it at step 1 (using the difference of outputs as oracle) corresponds to a weakness of the contracts which should be improved.

For example, let us consider the following contract for the UML2RDBMS transformation. This contract is expressed in the context of the global metamodel for the transformation: the union of UML and RDBMS metamodels. It is necessary to link the input and output metamodels to express constraints between elements of the input (UML) and output (RDBMS) models.

Contract:
```
context MetaUmlRdbms inv:
self.modelUml.elements->select(e|e.oclIsTypeOf(Class)
and  e.stereotype->exists(s|s.name='persistent'))
->collect(ec|ec.oclAsType (Class))
  ->forAll(cp|self.database.elements
    ->one(t|t.name=cp.name and
             cp.feature->select(f|f.oclIsTypeOf(Attribute))
      ->collect(fa|fa.oclAsType (Attribute))
      ->forAll(a|t.columns->one(tc|tc.name=a.name))
             and t.columns.size()=cp.feature
             ->select(f|f.oclIsTypeOf(Attribute)).size()))
```

Let us also consider a possible mutant for the `createColumns` method where the initialization of the type of the created column has been deleted:

Mutant:
```
private static Table createColumns(Class classUse, Table tableUse) {
  Iterator itClass = classUse.getFeature().iterator();
    while(itClass.hasNext()){
      metaUML.Attribute attributeUsed=(metaUML.Attribute)itClass.next();
        Column newColumn = MetaRDBMSFactory.eINSTANCE.createColumn();
        newColumn.setName(attributeUsed.getName());
        tableUse.getColumns().add(newColumn);}//while
  ……
}//method createColumns
```

When running the test cases generated previously, the faulty part of the mutant program is executed (this mutant is killed with mutation analysis). However, when running the test cases using the considered contract as the oracle function, the mutant is not killed. Looking at this contract, it appears that a particular property has not been expressed that prevents it from killing this particular mutant. A more complete contract is given below (it adds a property that checks the type of the columns):

```
context MetaUmlRdbms inv:
self.modelUml.elements->select(e|e.oclIsTypeOf(Class) and
   e.stereotype->exists(s|s.name='persistent'))
 ->collect(ec|ec.oclAsType (Class))
  ->forAll(cp|self.database.elements
   ->one(t|t.name=cp.name and
            cp.feature->select(f|f.oclIsTypeOf(Attribute))
     ->collect(fa|fa.oclAsType (Attribute))
     ->forAll(a|t.columns->one(tc|tc.name=a.name
                                and tc.type=a.type.name))
            and t.columns.size()=cp.feature
            ->select(f|f.oclIsTypeOf(Attribute)).size()))
```

Table 1. Mutation scores with contracts and different mutation operators

	MS %	Range contract %	D/R 0 %	D/R 1 %	D/R 2 %	D/R 3 %	D/R %
classic	91,7	64,6	74,0	83,3	88,5	91,7	91,7
model	89,1	67,4	58,7	67,4	71,7	78,3	87,0
total	90,8	65,5	69,0	78,2	83,1	87,3	90,1

5.4 Results

Table 1 gives results for several mutation analyses on the UML2RDBMS example. The three lines in the table correspond to different types of mutation operators: "classic" corresponds to classical mutation operators (mutants were obtained using MuJava mutation tool [7]), "model" corresponds to mutation operators dedicated to model transformation and "total" is the combination of both types of operators. The analyses were run with 96 classical mutants and 46 model-specific mutants. The first column (MS= Q_{td}) corresponds to the mutation score using the behavior difference as an oracle (it estimates the quality of test set Q_{td}). The following columns present the quality Q_{cont} of contracts. The second column concerns only Range contracts (check that the result conforms to the output metamodel). Columns 3 to 6 give the results with Domain/Range (D/R) contracts at 4 successive levels of improvement when applying the step 3 of the design-for-trust process. Last column is the final score obtained with both Range and improved D/R contracts. This first experiment shows that the range contracts allow detecting 60-70% of the mutants killed by the test set. However, to reach a higher score, D/R contracts are needed in complement to Range contracts. The combination of both allows reaching a Q_{cont} of 90%. Roughly, it means that embedded oracles have a high probability to detect a faulty result: the MDA component is thus robust and 'vigilant'. These first results have to be validated with other experiments.

6 Related Works

The notion of MDA component has recently appeared as a necessary feature for a successful deployment of MDA. In [8], this type of component is defined as "a packaging unit for any artifact used or produced within an MDA-related process". This paper introduces several concepts and entities that are present in a MDA context and they give several examples of MDA components. If they present model transformations as the

most important component, they also consider metamodels, promoters, and consistency checkers as MDA components. This concept of MDA component may thus be used in a wider meaning than the restricted but also more precise definition we propose in this paper. In [9], Fondement and al. also propose to use MDE components to answer methodological needs. They analyze what are the different available technologies to improve reuse and define assets that can automate a model-driven methodology. They consider, package dependency, profiling, model transformation and metamodelling and show that they all have serious limitations to allow a real component oriented MDE. As for [8], this work uses a larger definition of MDA component as ours, however and they only give clues of what could be done to actually design MDA components. In this paper, we focus on a specific aspect of MDE, the model transformation to define MDA component and propose a trustability assessment.

Concerning the particular techniques necessary to build trustable components, there are few related works. These techniques are: specification of contracts for a transformation, test generation and mutation analysis. In [10], Cariou et al. study the applicability of OCL to express contracts to specify a model transformation. Other works propose to express rules that declare the behavior of the transformation, but most of these techniques are declarative implementation of the transformation and not a specification from which the implementation is derived.

There are also few works about model transformation testing. In [11], Lin et al., identify all the core challenges for model transformation testing, and propose a framework that relates the different activities. In [12], 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 [13], we looked at the problem of test data generation for model transformations and proposed to adapt partition testing to define test criteria to cover the input metamodel (that describes the input domain for a transformation).

At last, mutation technique has been widely studied to evaluate the test sets for imperative and object-oriented programming but, as far as we know, has not been studied to validate tests or contracts for model transformation, except in our work [5]. 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 transformation and not on the way models may be faulty.

7 Conclusion

The presented work detailed a process to help programmers/developers building trust in a model transformation encapsulated into a MDA component. This method, based on test qualification, also leads to contracts improvement. For a given MDA component, we propose to estimate the consistency between contracts, implementation and tests using mutation analysis as the main qualification technique. The process to improve the trustability of MDA components is incremental:

1. improving the test set by analyzing their efficiency using a mutation analysis,
2. improving the implementation, thanks to the previously evaluated test set,

3. improving the contracts by measuring their accuracy as embedded oracles, knowing that test cases are efficient to provoke a faulty execution of the model transformation program.

Test set, implementation and contracts improvements are guided by the fact that we know which faults are injected during mutation.

References

1. Meyer, B., Object-oriented software construction. 1992: Prentice Hall. 1254.
2. Le Traon, Y., B. Baudry, and J.-M. Jézéquel, Design by Contract to Improve Software Vigilance. IEEE Transactions on Software Enginnering, 2006.
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., Y.-R. Kwon, and A.J. Offutt. Inter-Class Mutation Operators for Java. in Proceedings of ISSRE'02 (Int. Symposium on Software Reliability Engineering). Annapolis, MD, USA: IEEE Computer Society Press, Los Alamitos, CA, USA. 2002.
5. Mottu, J.-M., B. Baudry, and Y. Le Traon. Mutation Analysis Testing for Model Transformations. in Proceedings of ECMDA-FA 2006. Bilbao, Spain. 2006.
6. Beugnard, A., J.-M. Jézéquel, N. Plouzeau, and D. Watkins, Making components contract aware. IEEE Computer, 1999. 13(7).
7. Ma, Y.-S., A.J. Offutt, and Y.-R. Kwon, MuJava : An Automated Class Mutation System. Software Testing, Verification and Reliability, 2005.
8. Bézivin, J., S. Gérard, P.-A. Muller, and L. Rioux. MDA Components: Challenges and Opportunities. in Proceedings of Metamodelling for MDA. York, England. 2003.
9. Fondement, F. and R. Silaghi. Defining Model Driven Engineering Processes. in Proceedings of WISME. Lisbon, Portugal. 2004.
10. Cariou, E., R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. in Proceedings of Workshop OCL and Model Driven Engineering. Lisbon, Portugal. 2004.
11. 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.
12. Küster, J.M. Systematic Validation of Model Transformations. in Proceedings of WiSME'04 (associated to UML'04). Lisbon, Portugal. 2004.
13. Fleurey, F., J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. in Proceedings of MoDeVa. Rennes, France. 2004.
14. Trung, D.-T., S. Ghosh, F. Robert, B. Baudry, and F. Fleurey. A Taxonomy of Faults for UML Designs. in Proceedings of 2nd MoDeVa workshop - Model design and Validation, in conjunction with MoDELS05. Montego Bay, Jamaica. 2005.