Towards an Automation of the Mutation Analysis Dedicated to Model Transformation

Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, Jean-Luc Dekeyser

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Towards an Automation of the Mutation Analysis Dedicated to Model Transformation

Vincent Aranega[1], Jean-Marie Mottu[2], Anne Etien[1*],
Thomas Degueule[2], Benoit Baudry[3], Jean-Luc Dekeyser[1]

[1]*LIFL - University of Lille 1, Lille, France*
[2]*LINA - University of Nantes, Nantes, France*
[3]*INRIA - IRISA, Rennes, France*

SUMMARY

A major benefit of Model Driven Engineering (MDE) relies on the automatic generation of artefacts from high-level models through intermediary levels using model transformations. In such a process, the input must be well-designed and the model transformations should be trustworthy.
Due to the specificities of models and transformations, classical software test techniques have to be adapted. Among these techniques, mutation analysis has been ported and a set of mutation operators has been defined. However, mutation analysis currently requires a considerable manual work and suffers from the test data set improvement activity. This activity is seen by testers as a difficult and time-consuming job, and reduces the benefits of the mutation analysis. This paper addresses the test data set improvement activity. Model transformation traceability in conjunction with a model of mutation operators, and a dedicated algorithm allow to automatically or semi-automatically produce test models that detect new faults. The proposed approach is validated and illustrated in a case study written in Kermeta. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: MDE; Model Transformation; Mutation Analysis; Traceability; Mutation Operator

1. INTRODUCTION

Model Driven Engineering (MDE) relies on models (*i.e.* high level abstractions) to represent the system design. Model transformations are critical assets in MDE, which automate essential steps in the construction of complex software systems (*i.e.* they can transform artifacts from an abstraction layer to a lower one). For example, in the Gaspard2 project [1], model transformations automatically generate source code for different languages such as OpenMP (in case of scientific computing applications) or VHDL (in case of embedded applications) from UML models. Model transformations are used many times to justify the efforts relative to their development. So if they are faulty, they can spread faults to models several times. Moreover, since transformations are black boxes for the *end users*, they have to be trustworthy. So, for all these reasons, model transformations have to be tested during development and thoroughly validated.

Among all the existing testing techniques, this paper focuses on mutation analysis [2] as a way to systematically qualify and improve a set of test data for detecting faults in a program under test. For this purpose, faulty versions of this program (called *mutants*) are systematically created by injecting one single fault per version. Each injected fault depends on a mutation operator that represents a

*Correspondence to: E-mail: anne.etien@lifl.fr

kind of fault that could be introduced by programmers. The efficiency of a given test data set is then measured by its ability to highlight the fault injected in each mutated version (killing these mutants). If the proportion of killed mutants [3] is considered too low, it is necessary to improve the test data set [4].

This activity corresponds to the modification of existing test data or the generation of new ones, and is called *test data set improvement*. It is usually seen as the most time-consuming step. Experiments measure that the test data set initially provided by the tester often already detect 50 to 70% of the mutants as faulty [5]. However, several works state that improving the test set to highlight errors in 95% of mutants is difficult in most of the cases [6, 7]. Indeed, each non-killed (*i.e. alive*) mutant must be analysed in order to understand why no test data reveals its injected fault and consequently the test data set has to be improved.

This paper focuses on the test data set improvement of the mutation analysis process. It is dedicated to the test of model transformation. In this context, test data are models.

Due to their intrinsic nature, model transformations rely on specific operations (*e.g.* data collection in a typed graph or collection filtering) that rarely occur in traditional programming. In addition, many different dedicated languages exist to implement model transformation. Thus, the mutation analysis techniques used for traditional programming cannot be directly applied to model transformations; new challenges to model transformation testing are arising [8]. A set of mutation operators dedicated to model transformation has been previously introduced [9]. This paper tackles the problematic of the test model set improvement by automatically considering mutation operators.

Tools and heuristics are provided to assist the creation of new test models. The approach proposed in this paper relies on a high level representation of the mutation operators and a traceability mechanism establishing, for each transformation, links between the input and the output of the transformation. The first original contribution consists in precisely modeling the mutation operators dedicated to model transformation. Thus, all the results of the mutation testing process (*i.e.* which model kills which mutant created and by which mutation operator) are gathered in a unique model. Based on this model, relevant elements of some input test models are selected among the initial test set. The second original contribution is the generation of new test models from those elements of input models. For this purpose, each mutation operator is studied to identify a set of cases that could let a mutant *alive*. Patterns and heuristics are associated to each of these cases. The *patterns* specify, in terms of the test model, cases where the input model lets the mutant alive. The *heuristics* provide recommendations to generate new test models that should highlight errors in the mutants. It has been observed that in most cases, the patterns are automatically detected, and models automatically generated using those heuristics, reducing the efforts needed to increase the mutation score.

The approach is illustrated with the transformation from UML to a database schema showing for some mutation operators how patterns are detected and how new test models are produced. An experiment is run on a second case study and measure the effort necessary to get a 100% mutation score. Whereas a tester should entirely analyse existing test models to create new models, with a dedicated assistant only 33% of the models and 27% of the elements they contain should be analysed.

This paper is composed as follows. Section 2 briefly introduces the MDE concepts used in the paper. Section 3 presents existing works for mutation analysis, test model qualification and test set improvement approaches. Section 4 presents the mutation analysis adapted to model transformations and highlights challenges. Section 5 reminds previous works about test model improvement, which are used as the basis of the current contribution. Section 6 introduces the modeling of the mutation operators. Section 7 details how new test models are created based on the identification of a problematic configuration. Section 8, illustrates the approach proposed in this paper on the example of the class2rdbms transformation where the test model set is improved until 100%. Finally, conclusions are drawn and perspectives are proposed in Section 9.

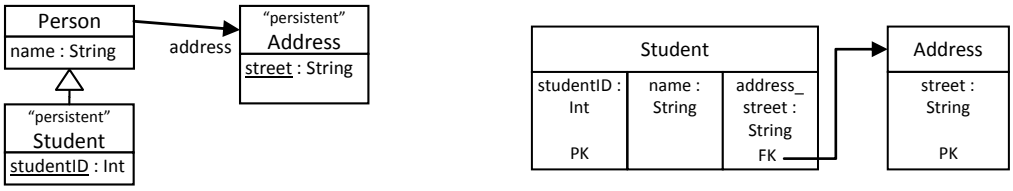
DOI: 10.1002/stvr

2. MODEL TRANSFORMATION TESTING: CONCEPTS AND MOTIVATING EXAMPLE

This section introduces a motivating example of model transformation; its concepts that require adaptation of testing techniques are then detailed. The model transformation, called class2rdbms, has been chosen. It creates a Relational Database Management Systems (RDBMS) model from a Simple Class Diagram (simpleCD). This transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [10]. It was designed to experiment and validate model transformation language features, and then it has been used in several works.

2.1. Modeling

In order to work at an abstract and higher level than the one proposed by classical programming, the *model* paradigm has been proposed. A model *represents* a system. It is an abstraction because it synthesizes a part of a system and avoids some details [11]. A model is restricted to a given goal, thus it only gathers information relevant to it. It represents a system with *elements* interconnected by *relations* (*e.g.* UML diagrams in Object-Oriented programming). These relations are used to access elements from others. For example, Figure 1 presents a simple class diagram (underlined attributes are primary keys) and the corresponding relational database (PK means primary key and FK foreign key).

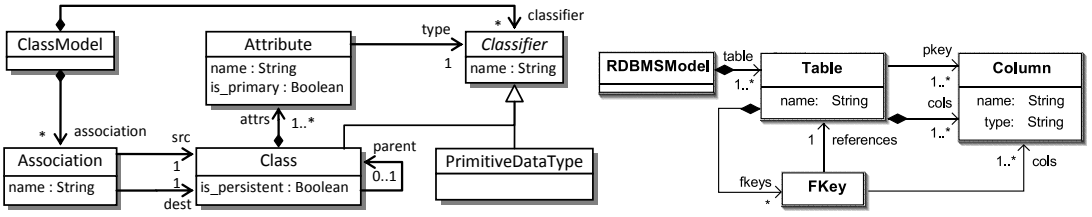


(a) Simple Class Diagram Model Sample (b) RDBMS Model Sample

Figure 1. One Input Model and its Output Model produced by class2rdbms transformation

2.2. Metamodeling

The models are defined using dedicated languages: the *metamodels*. A metamodel precisely defines the model elements, their structure as well as their semantic. It could be viewed as a language grammar; thus, many different models may be associated to a single metamodel [12, 13]. Figure 2(a) represents a simplified version of the class diagram metamodel. It defines the concepts of $ClassModel$ that contains $Classifiers$ and $Associations$. A $Classifier$ is either a $PrimitiveDataType$ or a $Class$. A $Class$ may be persistent or not, it may inherit from another class (through the $parent$ link), and may contain one or several $Attribute$. An $Association$ links two classes, one being the source (src), the other the destination ($dest$).



(a) Simple Class Diagram Metamodel (b) RDBMS Metamodel

Figure 2. The Input and Output *Metamodels* of class2rdbms transformation

The strong link between a metamodel and the associated models is called a "conformance" link. A model *conforms* to a metamodel if all its features are defined by the metamodel.

The class diagram example is represented in Figure 1(a) using a graphical syntax close to UML. In MDE, this model could be represented with an abstract syntax where each element is defined

Prepared using stvrauth.cls
DOI: 10.1002/stvr

as an instance of a metamodel class. This level of detail is close to the way model transformation manipulates a model and will be useful in the proposed contributions. In Figure 3, the class diagram of Figure 1(a) is represented with this abstract syntax.

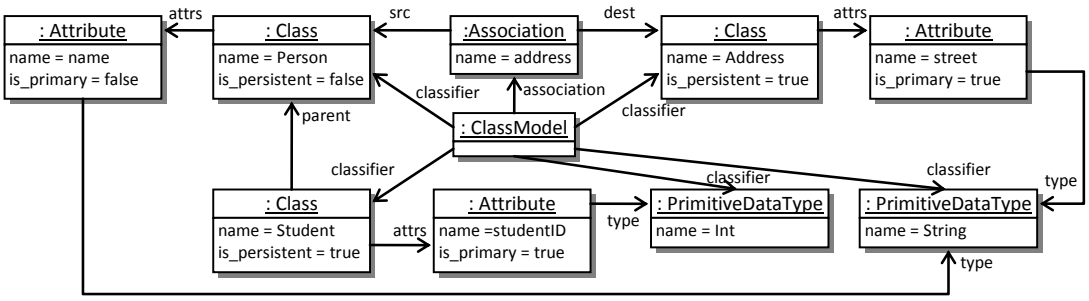


Figure 3. The Input Model Example as an Instance Diagram of the Input *Metamodel* of class2rdbms

2.3. Model Transformation

Model transformations allow the automatic production or modification of models and present thus a major interest for MDE [14]. A *model transformation* is based on its input and output metamodels specifying respectively its input and output domains. Basically, the transformation establishes a set of relationships between input model elements and output model elements.

class2rdbms transforms an input simple class diagram model into an output RDBMS model. Such a simple class diagram model is illustrated in Figure 1(a). A person has a name and an address, and she could be a student with a studentID. class2rdbms transforms this model into the RDBMS model of the Figure 1(b). Briefly, the persistent classes are changed into tables. The attributes and associations become columns. Primary attributes (underlined) become primary keys PK and the associations become foreign keys FK. Figure 2(b) presents the RDBMS metamodel, defining the output model elements.

Concretely, a model transformation is expressed in a *model transformation language*. In order to describe a model transformation, many approaches and languages can be used. A classification of these approaches is proposed by Czarnecki *et al.* [15]. Most of the model transformation languages decompose transformation into smaller parts called *rule* such as a program is decomposed into functions. Practically, a rule focuses on the transformation of a specific input element. In this paper, the rule concept is used to express the way input elements are transformed. Moreover, even if the transformation approaches are different, they usually work in the same way. Indeed, most transformations aim to create a new model[†], and can be considered as a set of three kinds of operations: $navigations$ between model elements in order to reach some specific elements; $filtering$ of model element collections in order to express some conditions and keep only some subparts of the initial collections; and $creations/modifications$ of new model elements. Listing 1 is an extract of the class2rdbms transformation written in QVTo [16]. Examples of navigation can be found all over the listing and are expressed using dot notation. Line 5 and the expression between brackets line 10 illustrate filtering. Creations are expressed with the mapping of object key words. In other languages like Kermeta (see Listing 2, page 23), these operations are differently specified.

Listing 1: Extract of the class2rdbms transformation in QVTo

```
1  transformation class2rdbms (in srcModel:UML,out dest:RDBMS);
2   [...]
3    -- maps a class to a table, with a column
4    mapping Class::class2table () : Table
5      when {self.kind='persistent';}
6     {
```

[†]There also exist *in place* transformations that modify an existing model, like a refactoring.

DOI: 10.1002/stvr

```
7       name := 't_' + self.name;
8       column := self.attrs->map attr2OrdinaryColumn();
9       key_ := object Key { -- nested population section for a 'Key'
10          name := 'k_'+self.name; column := result.column[kind='primary']; };
11    }
12    -- Mapping that creates an ordinary column from a leaf attribute
13    mapping Attribute::attr2OrdinaryColumn (): Column {
14      name := prefix+self.name;
15      kind := self.kind;
16      type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif;
17    }
```

2.4. Model Transformation Testing

Obviously, model transformation may be considered as a program and consequently tested. However, existing approaches do not take into account the specific features of model transformations, *i.e.* (i) the three fundamental operations composing them and (ii) models as input data. Traditional testing approaches have thus to be adapted to model transformations. Such adaptations may be performed for each specific transformation language / approach or in the opposite may take into account their heterogeneity. The approach proposed in this paper adopts the second alternative relying on the common features of the transformations.

As seen before, the definition of model transformations relies on their input and output metamodels. Thus to test the transformations, the elements of the input and the output models have to be considered as instances of the metamodel elements leading to large and complex graphs. Figure 3 illustrates the complexity of an input model example, despite the simplicity of that Class diagram model represented in Figure 1(a). Consequently, the generation and the evaluation of the test data, which are test input models, are complex. Moreover, this complexity increases by considering that the test data set (i) should cover the input domain of the model transformation, which may be very large, and (ii) should be able to detect faults in transformation. In addition, MDE development environments lack reliable support to analyze and transform models. Therefore, it is more efficient to propose techniques to automate or assist in the generation of test models, rather than to evaluate the efficiency of a test model set and leave the tester to manage to improve it manually.

In this paper contrary as in other works [17, 18], the test oracle challenge is not concerned.

3. IMPROVING A TEST DATA SET: STATE OF THE ART

Among the various proposed approaches, test model qualification provides information useful to generate efficient test data. Few works study test model qualification and generation, considering efficiency in terms of input domain coverage, model transformation rules coverage, specification coverage and potential fault coverage. First these different ways to obtain qualified test model sets are presented, then the focus is put on the last one with mutation analysis.

3.1. Test Model Qualification and Generation Approaches

Model transformation domains are specified with metamodel and several works consider that characteristic to qualify test model sets.

Fleurey *et al.* [19] qualify a set of test models regarding its coverage of the input domain. This approach is based on the partitioning of the metamodels. Sen *et al.* [20] developed the Pramana tool (formerly named Cartier) to generate test models based on the proposal of Fleurey et al. However, both approaches produce more models than necessary because they aim to provide tests covering the whole input domain, even if only a subpart of the input domain is used by the transformation. In the case of UML, it often occurs that transformations only deal with a subpart (*e.g.* the class diagram or the state diagram). To avoid producing test models relative to metamodel parts not involved in

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the transformation, Sen *et al.* [21] prune the metamodel to extract only the subparts involved in the transformation before providing the tests.

Mottu *et al.* [22] propose a white-box approach that uses a static analysis to automatically generate test inputs for transformations. This static analysis uncovers knowledge about how the input model elements are accessed by transformation rules. On the other hand, because these approaches rely on static analysis of the transformation, if there is some dead code, tests are generated for these parts, even if they are never executed.

Guerra [23] tackles the test model generation challenge by deriving, from the transformation specification, a set of test models ensuring a certain level of coverage of the properties in the specification. These input models are calculated using constraint solving techniques.

Those approaches study transformations only statically, *i.e.* without executing them and without considering the potential errors. As a result, a 100% mutation score has not yet been reached in the experiments of those papers (70% [23], 89.9% avg. [20], 97.62% avg. [22]). Moreover, these methods help to generate qualified test models without providing information and methods to improve test models when highest quality is mandatory (for example, in case of the generation of highly critical applications). In the opposite, the contributions proposed in this paper help the tester to improve the quality of test models set thanks to mutation analysis enhanced with mutation operator metamodeling and traceability.

3.2. Mutation Testing Approaches to Measure Test Data Set Efficiency to Detect Faults

Numerous papers study mutation testing in a general context. Jia *et al.* propose a survey on mutation testing development [24]. They observe an increase of mutation testing publications from 1978 (mutation testing emergence) to 2009 making the mutation testing a mature technique. The main problem tackled in the literature is the creation of mutation operators. In this subsection, works related to the design of mutation operators are discussed and, in the next subsection, the improvement of the test data quality with mutation testing.

Several works advise to use mutation testing [25, 26, 27]. They analyse its ability to qualify test set with a high fault power detection and with different properties (for instance, test sets with fewer test cases). Those papers compare mutation testing with edge-Pair [28], All-uses [29] and Prime Path Coverage criteria (line coverage or statement coverage) [30] techniques, concluding that mutation testing provides better results.

Most of the works consider the mutation operators directly based on the syntax of a programming language: C [31], ADA [32], Java [33]. These mutation operator sets take into consideration the programming paradigm of the language (e.g. procedural [34], object oriented [33]). However, they are defined using the syntax of a programming language.

To fit with model paradigm and model transformations, Mottu *et al.* proposed an adaptation of mutation testing [9]. The originality of that work was not to use a programming syntax to define mutation operators, but to consider abstract operations applied by the transformation on models. Based on this work, Tisi *et al.* propose to formalize these mutation operators as a set of higher-order transformations (HOT) (*i.e.* transformations whose input or output model is itself a transformation [35]). For instance, Fraternali *et al.* propose an implementation of Mottu *et al.* operators as ATL HOTs [36] (ATL is a specific model transformation language). Some transformation languages do not use a model as an internal representation of the transformations. The approach proposed by Tisi *et al.* [35] can thus only be used for transformation languages supporting such a representation.

Few works consider mutation operators independently from the syntax of a language. Mutation operators proposed by Ferrari *et al.* are designed considering aspect oriented programming characteristics [37]. Recently, Simão *et al.* [38] proposes MuDeL, a language used to precisely define mutation operators independently from the used language. They provide generic definitions of each operator that may be reused with several languages. They also use a compilation from MuDeL operators to the language syntax. However, the proposed generic definitions of operators do not consider the specificity of new programming paradigms (as model transformation in model-driven engineering) and new kinds of analysis and transformation of specific data (as models). Moreover,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

MuDeL deals with context-free mutations (when $x = 2$ becomes $x+ = 3$ it is context free but not when it becomes $y = 2$) whereas in the approach proposed in this paper, several mutation operators are not context free (for instance, some operators require to know the elements of the metamodel to perform the mutation of the transformation rules).

3.3. Mutation Analysis to Improve Test Data

Several works deal with test set improvement to increase mutation score. Fleurey *et al.* propose an adaptation of a bacteriologic algorithm to model transformation testing [39]. The bacteriologic algorithm [40] is designed to automatically improve the quality of a test data set. It measures the power of each data to highlight errors to (1) reject useless test data, (2) keep the best test data, (3) combine the latter to create new test data. Their adaptation consists of creating new test models by covering part of the input domain still not covered.

MuTest is a project that aims to use mutation analysis to drive test case generation [41]. In order to produce new test data, MuTest generates multiples assertions to kill a selected mutant. Once the new test data is found, the test set is minimized in order to keep the test set as small as possible.

EvoSuite is a tool which is able to improve automatically a test set for the Java language [42]. It relies on mutation testing to produce a reduced set of assertions that maximizes the mutation score. In order to produce the new test data, EvoSuite tries to find test cases that violate the oracles. However, this tool directly handles Java byte code, and it is so close to the Java language that it makes its adaptation particularly difficult.

Some other works use evolutionary and genetic algorithms for automatic test input data generation in the context of mutation testing. A fully automated ant colony optimization algorithm is provided by Ayari *et al.* [43]. From an initial test set, the algorithm progressively promotes best test cases and combines them to generate new test data in order to kill mutants. The authors obtain a maximum mutation score of 89% on a small Java program.

However, all of these works only consider traditional programming, even if some propose bacteriologic algorithm for component based programming [39]. They do not take into account the particularities of model transformations (such as models as input data) ,and prevent their use for model transformations, as explained in the introduction.

4. MUTATION ANALYSIS AND MODEL TRANSFORMATIONS

Mutation analysis relies on the following assumption: if a test data set can reveal the faults voluntary and systematically injected in various versions of the program under test, then this set is able to detect involuntary faults. The efficiency of the test data set to detect the injected faults is evaluated by calculating the mutation score, *i.e.* the proportion of detected faulty versions [3]. Next subsections present the mutation analysis process adapted to model transformations and the challenges considered in this paper.

4.1. Mutation Analysis Dedicated to Model Transformations

Model transformations have their own specificities such as the manipulated data structures (*i.e.* models which conform to their metamodels) or characteristic operations that may lead to semantic faults. The mutation analysis process for traditional programs [44] has been adapted to model transformations [9]. It is presented in Figure 4. The four main activities are the following.

Preliminary step (activity (a)): this activity is divided into two parts: the creation of mutants and an initial test data set creation. Mutants are defined by modifying an instruction in a transformation rule from the original model transformation. This rule which owns the modified instruction is called *mutated rule* in the remainder of the paper. The modification of the instruction is performed using one of the mutation operators dedicated to model transformations [9]. These operators have been defined independently from any transformation language. They are based on the three basic operations performed by a model

DOI: 10.1002/stvr

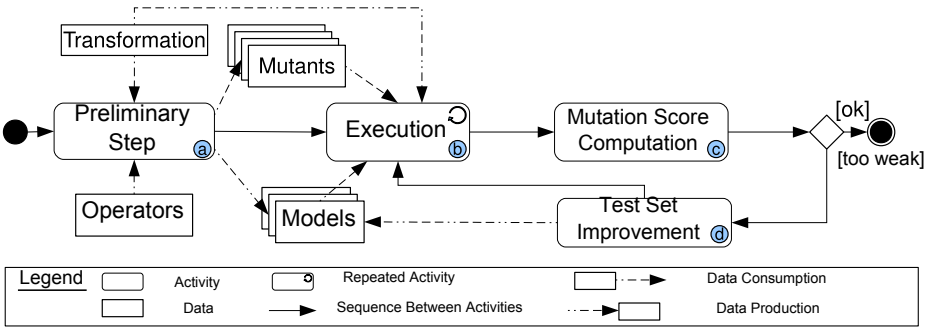


Figure 4. Mutation Testing Process Dedicated to Models

transformation: the *navigation* through references between objects in models, *filtering* of objects collections and the *creation* (or the *modification*) of model elements. An initial test data set containing several test models is possibly automatically built, using approaches such as the ones discussed in section 3.1.

The mutant and original program execution (activity (b)): each created mutant T_1, $T_2, \ldots, T_k$, as well as the original non-mutated transformation T is executed on each input test model.

The mutation score computation (activity (c)): the mutation score is the proportion of *killed* mutants compared to the overall number of mutants. A mutant is marked as killed when one (or several) test model highlights the injected error. A test model highlights an error if the result of its execution by the mutant differs from the execution by the original transformation. Thus, if for a given test model m, $T_i(m) \neq T(m)$ then T_i is $killed$ else T_i is $alive$. Output model comparison can be performed using adequate tools adapted to models or large graphs such as EMFCompare [45]. The mutation score computation is thus automatically performed.

The test data set improvement (activity (d)): if the mutation score is considered too low, new test models are produced in order to kill *live* mutants; and equivalent mutants are rejected [24] (these last have the same behaviour than the transformation, so they are not considered being faulty).

Although, once again as the classical mutation testing process, the whole process starts again until the mutation score reaches a beforehand fixed threshold, 100% is ideal [4].

4.2. A Process Remaining Mainly Manual

Although some tasks of the mutation testing process are automatic, this process remains a complex and long work for the tester. For any program under test (including model transformation), among the mutation testing tasks:

- *creation of mutants* can be automated. However, mutation operators are applied to the code of the program under test. Thus, for each new language, mutation operators must be designed in its syntax (or ported from non syntactic ones [36]), and implemented,
- *execution* of the program and the mutants is an automated task,
- *comparison of the output data* is also an automated task.

The *test data set improvement* task (requiring *analysis of mutants*), is considered as time-consuming [26] and is currently performed manually. Indeed, injected faults which are not highlighted by test data must be analysed in both ways: statically (*i.e.* using a static analysis of the code) and dynamically (*i.e.* an execution of the program is required) in order to create new data which could kill a live mutant. The result of activity (a) is considered in the remainder of the paper as a prerequisite.

This paper focuses on the *test data set improvement* task (activity (d)) for the mutation analysis process dedicated to model transformations.

DOI: 10.1002/stvr

4.3. Contribution to Model Transformation Mutation Analysis

This paper proposal is based on the following hypothesis: building a new test model from scratch can be extremely complex, while taking advantage of existing test models could help to construct new ones. Consequently, an approach helping to create new test models from modifications of relevant existing models has been developed. Thus, first, a relevant test model is selected to be modified. As the work deals with a large amount of information, the induced problems could be resumed in three questions:

- Among all the existing pairs $(test\ model, mutant)$, which ones are the most relevant to be studied? Moreover, in those models, which parts are relevant to improve the models?
- Why a mutant has not been killed by a specific test model?
- How to modify a selected test model to produce a different output model and thus kill the studied mutant?

In this paper, answers to these three questions are provided and thus the automation of the test data set improvement activity of the mutation analysis process is improved. The approach is composed of two major steps: (i) the selection of a relevant pair $(test\ model, mutant)$ and (ii) the creation of a new test model by adequately modifying the one identified. The first step relies on an intensive use of the model transformation traceability [46]. It is briefly reminded in Section 5. The second step corresponding to the heart of this contribution is presented in Section 6 and 7. Both steps are illustrated in Section 8, and are experimented in Section 8.9.

5. MODEL TRANSFORMATION TRACEABILITY: A WAY TO COLLECT INFORMATION

This Section reminds previous works [46] proposing a systematic procedure to answer the first question: "*Among all the existing pairs (test model, mutant), which ones are the most relevant to be studied? And in those models, which parts are relevant to improve the model?*" The principles of *model transformation traceability* are summarized, the *mutation matrix* is introduced and the relationships between these two concepts are detailed.

5.1. Model Transformation Traceability

Regarding MDE and more specifically model transformations, the traceability mechanism links input and output models elements [47]. It specifies input model elements used by the transformation to generate output model elements.

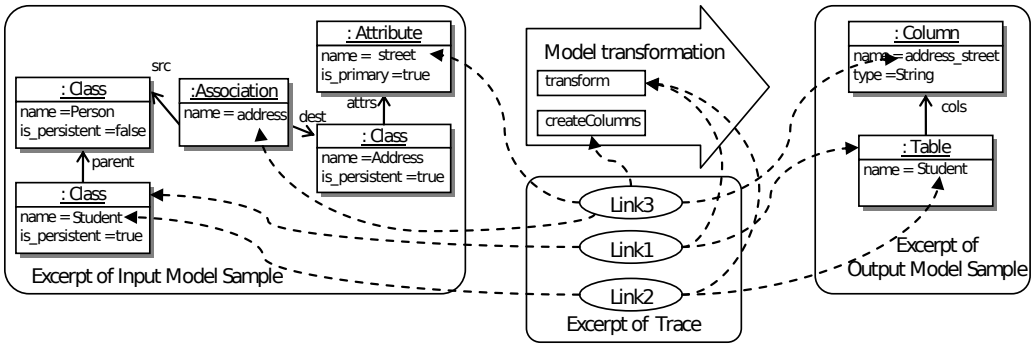


Figure 5. Trace Example

Various traceability approaches have been developed, but they are dedicated to a specific transformation language [48], [49], [50] or they take into account only classes and not attributes [47]. A traceability approach has been developed [51] independently from any transformation language. Each creation/modification of an element by a *rule* leads to the creation of a unique link. Each

DOI: 10.1002/stvr

link refers to (i) a set of source elements (attribute or class instances), (ii) a set of target elements and (iii) the transformation rule that leads to this creation/modification. Each link corresponds to an execution of a rule, that may be executed several times on different input elements. Figure 5 illustrates an example. $Link1$ indicates that the output instance of $Table$ has been created from one input instance of $Class$. Moreover, $Link1$ specifies that the instances it binds have been read and created by the $transform$ rule.

Each trace is formally modeled. It captures formal relations between models and the transformation. Moreover, it can be analyzed independently from any transformation language. Nevertheless, the automatic generation of traces must be tied to a specific transformation language. In this paper, the trace generation is adapted for the *Kermeta*[‡] language in order to pursue the works initiated by Mottu *et al.* [9, 20]. Usually traces can be automatically generated without altering the transformation by adding plugins to the transformation engine [51]. However, these solutions rely on an intermediary internal representation of the transformations used by other transformation languages such as *QVTo* [16]. Kermeta does not use any intermediary representation. Two alternatives remain: drastically modifying the engine or manually inserting instructions in the transformation [48]. This latter solution has been adopted in the experimentations since it allows quick prototyping and requires a small development effort[§].

5.2. Model Transformation Traces and Mutation Matrix Generation

The execution step named (b) in Figure 4 is more complex than just executing the mutants for each test model. A trace model is generated for each execution and a *mutation matrix* [9] is built to gather all the results. Figure 6 sketches this step that has been detailed in [46].

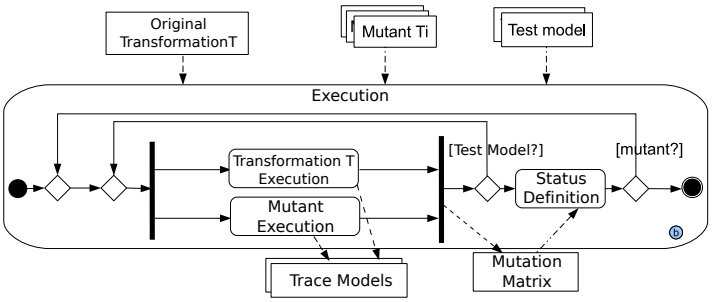


Figure 6. Mutation Matrix and Trace Generation

The execution step requires the following parameters: the transformation under test T, its mutants T_i and the test models. For each transformation execution (original or mutant), a trace (*Trace Model*) is produced and associated to the corresponding pair $(test\ model, mutant)$. Furthermore, the state (*i.e. killed* or *alive*) for this pair is stored in a cell of the mutation matrix.

More precisely, each cell of the mutation matrix corresponds to the execution of a mutant T_i with a test model m_j. The mutation matrix is presented in Figure 7. It is organized through three main concepts [46]:

- *Mutant* $(T_0 \dots T_n)$ which represents the executions of a mutant in a column,
- *Model* $(m_0 \dots m_m)$ which represents the transformations of a test model m_j in a row,
- *Cell* $(C_{00} \dots C_{mn})$ which represents the execution of one mutant with one single test model. It specifies if the test model lets the mutant alive or not (with a $boolean$). Furthermore, each cell is associated to a *Trace* $(lt_{00} \dots lt_{mn})$.

Thus, from a *Mutant* or a *Model*, it is possible to access to its *Cells* and so to its *Traces*. The mutation matrix acts as a pivot between the mutants, the test models and the traces. It is then used

[‡]http://www.kermeta.org
[§]Experimental material [52]

DOI: 10.1002/stvr

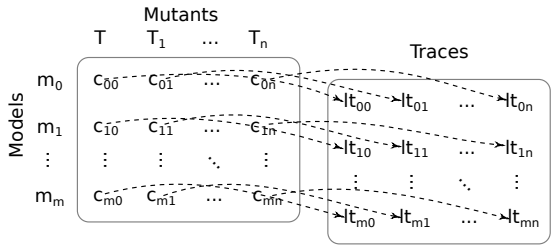


Figure 7. Mutation Matrix

to define final status of the mutant. Indeed, if a mutant has no cell marked as *killed*, it is considered as *alive*.

5.3. Identification of Relevant Pairs (Model, Mutant)

The proposed approach relies on the assumption that test models owning elements which are used by the *mutated rule* (and so by the mutated instruction) are good candidates to be improved to kill the mutant. Indeed, this rule has been executed on elements of these test models, but the resulting models do not differ from the ones of the original transformation executions possibly because of neutralization by the remainder of the transformation. The mutant is the exact copy of the original transformation except the mutated instruction. The difference of the outputs can only result from the execution of this mutated instruction and thus of the mutated rule. The traceability mechanism allows the tester to identify these candidate models and, for each one, to highlight the elements consumed and produced by the mutated rule. Thus, the test model improvement activity (Figure 4, activity (d)) is decomposed into three steps as illustrated in Figure 8.

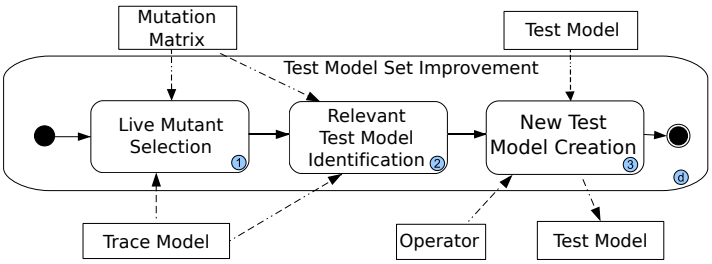


Figure 8. Test Model Set Improvement Process

Live Mutant Selection. The selection of a relevant pair begins by the choice of a live mutant (activity (1)). This is automatically performed by exploring the boolean attribute of the mutation matrix cells.

Relevant Test Model Identification. For the selected mutant, the corresponding cells of the mutation matrix are explored, and each trace model is navigated to identify if the mutated rule has been executed, *i.e.* if, in the trace, there exists a $link$ pointing to it. Two situations can occur: (i) If the rule has never been executed, a new test model must be created, potentially from scratch, taking care that conditions for the mutated rule to be executed are fulfilled. (ii) If the mutated rule has been called at least once the algorithm selects some candidate test models. Moreover, for each identified test model, two other sets are provided representing respectively the input and the output elements handled by the mutated rule. Among these models, the one with the smallest sets of elements is selected: using the two sets associated to the chosen model, the tester can focus on the smallest set of input model elements and its small counterpart in the output model to understand why these elements do not kill the mutant being considered and consequently to modify the identified model. Using the input model with the smallest set of elements reduces the space search and, consequently, the effort to analyse why its elements do not kill the mutant.

DOI: 10.1002/stvr

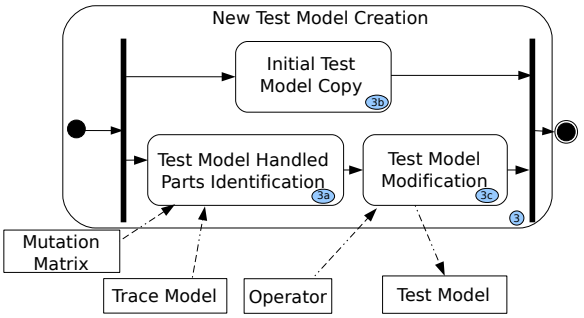


Figure 9. New Test Model Creation Process

New Test Model Creation. In order to increase the test model set without regression (*i.e.* without making alive some mutants killed by the identified test model), this identified test model is preliminary copied as shown in Figure 9, activity $3(b)$ and then modified. The activity 2 has provided the input and output elements handled by the mutated rule. According to the mutant and the applied operators, the input or the output elements are identified in the corresponding model (Figure 9, activity $3(a)$). Section 7 focuses on the modification of the test model (activity $3(c)$) to automate and assist this activity which is manual until now. The mutation matrix and the trace may be used to automatically identify relevant pairs $(test\ model, mutant)$ and the input and output elements involved in the execution of the mutated rule. In the next section, the mutation operators automatically use them as input of the $3(c)$ activity (Figure 9).

6. MODELLING MUTATION OPERATORS

In order to automate the treatment of the live mutants, mutation made in them should be more precise than an informal description (as it was proposed by Mottu *et al.* [9]). The mutation operators are designed to automatically detect where they could be applied. The heterogeneity of model transformation languages issue is prevented by using a language independent approach. Moreover, only the modification and creation of new test models that are not dependent on the transformation language are addressed.

The *original contribution* consists in modeling mutation operators based on their effects upon the data manipulated by the transformation under test instead of based on their implementation in the transformation language being used. The mutation operators proposed by Mottu *et al.* [9] are thus designed based on the metamodels of the models manipulated by the transformation.

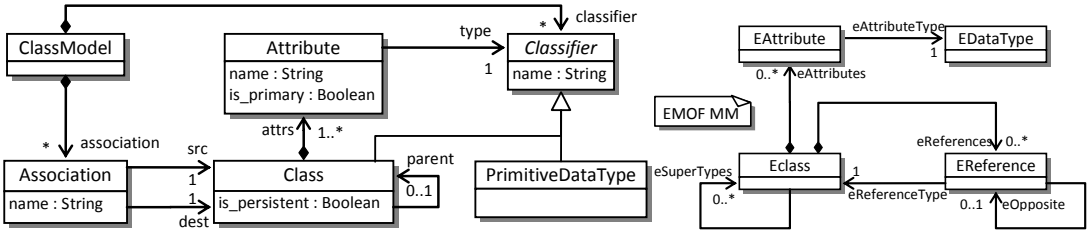
6.1. Principle

The mutation operators for model transformation are based on the three classical operations that occur in model transformations: *navigation*, *filtering*, and *creation* (or *modification*) of input and output model elements [9]. These three operations are sequentially applied. They form a basic cycle which is repeated to compose a whole model transformation. Such a decomposition into elementary operations provides an abstract view which is useful for fault injection.

Each mutation operator is designed as a metamodel expressing how the operator may be applied on any transformation. The 10 mutation operators defined by Mottu *et al.* [9] lead to the creation of 10 mutation operator metamodels. Those metamodels are independent from any transformation and any transformation language. Their instantiations depend on the transformation under test and express how mutants operate on its input/output metamodels. A single instance can be used to represent a kind of mutation applied to several code areas leading to the creation of several mutants.

The application of one mutation operator on one transformation returns *mutation models* which conform to the corresponding mutation operator metamodel. Whereas a mutation metamodel is generic, its models are dedicated to one model transformation, but still language independent.

DOI: 10.1002/stvr



(a) Simple Class Diagram Metamodel (b) EMOF Meta-Metamodel

Figure 10. Class diagram metamodel and its own metamodel: EMOF meta-metamodel

Those mutation models are based on the input and output metamodels of the transformation. They define how input/output model elements could be treated by the original transformation and how the mutants would treat them. For instance, Figure 10(a) reminds the metamodel of the class2rdbms transformation. One mutation model would describe that the src link is navigated instead of the $dest$ link. Another mutation model would describe that the $dest$ link is navigated instead of the src link. Another one would select a subset of $Class$ instances in a set collected through the $classifier$ link.

The application of one mutation operator model on one transformation implementation returns the mutants. This step is out of the scope of this paper since it depends on the transformation language.

To understand how those mutation operator metamodels are created and then instantiated in mutation operator models, the concept of metamodel needs to be further explained.

A metamodel is itself a model, and thus it conforms to one metamodel which defines its concepts. This metamodel called meta-metamodel is the EMOF meta-metamodel and is illustrated in Figure 10(b). The class diagram metamodel (Figure 10(a)) conforms to the EMOF meta-metamodel. Thus, for example, $Class$ and $PrimitiveDataType$ are instances of $EClass$ whose $eSuperType$ is $Classifier$, another instance of $EClass$. $is\_persistent$, $is\_primary$, or $name$ are instances of $EAttribute$ and their $EDatatype$ are $Boolean$, $Boolean$, and $String$, respectively. $dest$ and src are $EReference$ without $eOpposite$ $EReference$.

6.2. Three Examples of Mutation Operator Models

The following subsections present the metamodels of three mutation operators with definition extracted from Mottu *et al.* work [9], an application example extracted from the class2rdbms transformation, and a description of the used concepts and relations. For the sake of conciseness, the 7 other mutation operators are detailed in an annex [52]. In order to create the mutants, the effective operators applied to the transformation under test (*i.e.* the mutation operator models) have to be effectively defined. This is automatically performed with a model transformation [52].

6.2.1. Navigation Mutation: Relation to the Same Class Change Operator (RSCC)

Definition "The RSCC operator replaces the navigation of one reference towards a class with the navigation of another reference to the same class."

The RSCC operator can be applied on the input or the output metamodel of the transformation but only if it exists, in the metamodel, at least two EReferences between the two same EClasses. One EReference is originally navigated, the other is navigated by a mutant. Thus applied on a model transformation, RSCC operator replaces the original navigation by another to the same EClass.

The RSCC operator metamodel is presented in Figure 11. In order to ensure its independence from any transformation and transformation language, it is expressed on generic concepts that can appear in any transformation whatever the used language. Indeed, the RSCC operator metamodel uses the EMOF metamodel (on the left of Figure 11) to specify the input or output elements of the transformation the operator is applied on. Note that for each operator metamodel, several abstract

DOI: 10.1002/stvr

classes have been introduced (on the right) refactoring some references and increasing reusability between operator metamodels as shown in the Annex [52].

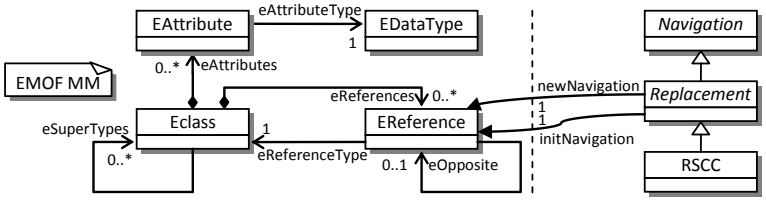


Figure 11. *RSCC* operator metamodel

Metamodel Description Table I gathers the classes and the relations dedicated to the *RSCC* operator. Additional constraints are necessary to ensure the viability of the mutant created. The first constraint prevents the mutants to be equivalent. The second constraint requires the two EReferences being to the same EClass.

CLASS/RELATION	DESCRIPTION
RSCC	The mutation operator
initNavigation	EReference initially navigated by the transformation
newNavigation	EReference navigated after the mutation
additional constraint	newNavigation != initNavigation
additional constraint	newNavigation.eReferenceType = initNavigation.eReferenceType

Table I. Description of the RSCC Operator Metamodel

Example The mutation model illustrated in Figure 12 is an example of the application of the RSCC mutation operator on the class2rdbms transformation. The grey part instantiates the operator RSCC. This mutation model would be applied in the transformation each time a rule navigates the $dest$ EReference (replacing it with the src EReference), whatever its place in a navigation chain, returning each time a different mutant. Note that the RSCC operator metamodel would be instantiated a second time by inverting $initNavigation$ and $newNavigation$.

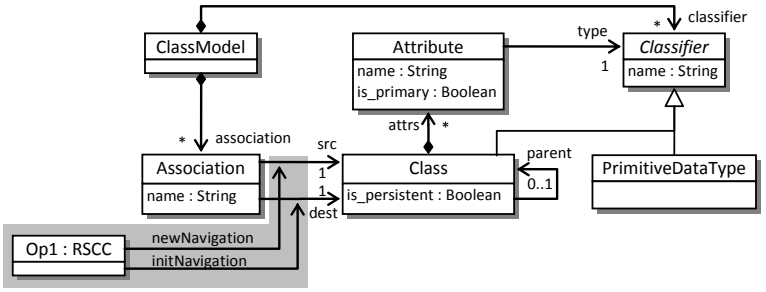


Figure 12. One example of the *RSCC* mutation models for the class2rdbms transformation

Such a mutation operator model could then be applied to the implementation of the transformation. For instance, the following mutant has been produced in the Listing 3, page 23, written in Kermeta with its mutated navigation Line 3.

6.2.2. Filtering Mutation: Collection Filtering Change with Addition (CFCA)

The filtering operations handle collections and select a subset of elements useful for the transformation; based on specific criteria. The collection may be collected by an eAttribute or through an EReference, or it may be an EClass collection.

DOI: 10.1002/stvr

Definition "This operator [...] uses a collection and processes [an extra] filtering on it. This operator could return an infinite number of mutants and need to be restricted. It has been chosen to take a collection and to return a single element arbitrarily chosen."

Figure 13 and Table II describe the metamodel of the *CFCA* mutation operator.

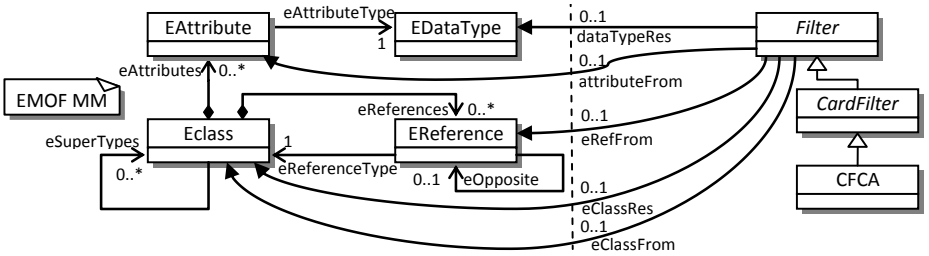


Figure 13. *CFCA* operator metamodel

Metamodel Description Table II gathers the classes and the relations dedicated to the *CFCA* operator. The table is divided into several parts because the three last ones are mutually exclusive. In fact, the filter may concern either an EReference, an EAttribute (both pointing collections of element), or a collection of EClasses and that also appears in the cardinalities associated to the $eRefFrom$, $attributeFrom$, and $eClassFrom$ EReferences. The fault is not directly described in this metamodel (as it was with $newNavigation$ in RSCC), since the operator modifies the original filter, arbitrarily, selecting only one element, for instance.

CLASS/RELATION	DESCRIPTION
CFCA	The mutation operator
eRefFrom	Define the EReference on which the filtering is applied
eClassRes	Type of the elements returned by the filtering through the EReference
attributeFrom	Define the EAttribute on which the filtering is applied
dataTypesRes	Type of the EAttributes returned by the filtering
eClassFrom	Define the collection of EClasses on which the filtering is applied
eClassRes	Type of the elements returned by the filtering

Table II. Description of the CFCA Operator Metamodel

Example Figure 14 illustrates one mutation model of CFCA applied to class2rdbms transformation. The grey part instantiates the operator CFCA whereas the remainder is the input metamodel of the transformation. This mutation model would be applied to the transformation each time a rule filters a collection of $Class$ instances returned by the $classifier$ reference. i.e. if a $ClassModel$ has several $Classes$ (such as in Figure 1a), CFCA operator modifies the filter to select only one of the $Class$ in the model.

6.2.3. Creation Mutation: Classes' Association Creation Addition (CACA) Operator

The *creation* mutation operators are relative to the last phase of the transformation process (*i.e.* they concern the creation or modification of output metamodel elements).

Definition "This operator adds an extra relation between two class instances of the output model, when the metamodel allows it."

Prepared using stvrauth.cls
DOI: 10.1002/stvr

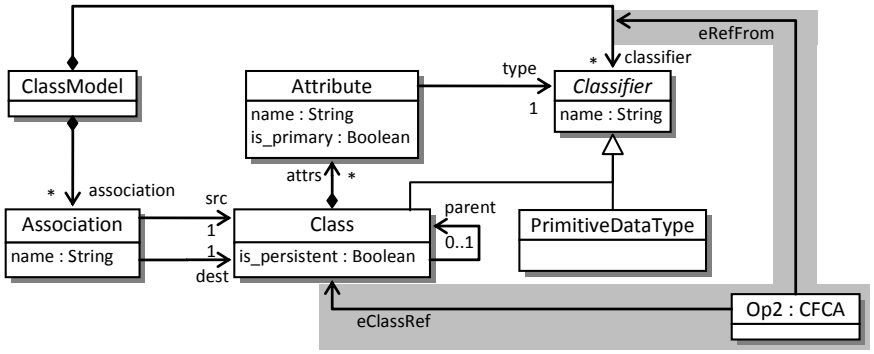


Figure 14. One example of the *CFCA* mutation models for class2rdbms transformation

The creation operators (*e.g.* CACA) are defined on the output metamodel of the transformation. The output model resulting from the execution of the original transformation or from a mutant must always conform the output metamodel. Thus, the mutant may add in the output model only references that are instances of $ERe f erences$ in the output metamodel. Moreover, according to [9], if the EReference has an upper cardinality equals to 1 and if an instance already exists, then adding an extra has no consequence, the EReference is overridden. The *CACA* operator metamodel is illustrated in Figure 15.

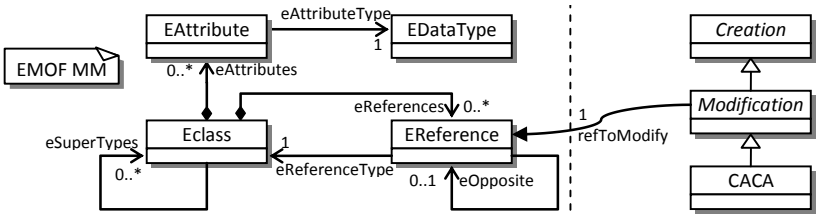


Figure 15. *CACA* Operator Metamodel

Metamodel Description Table III sums up the description of the *CACA* metamodel by describing its specific classes and relation. For this mutation operator, the *refToModify* EReference corresponds to the reference which is added in the output model.

CLASS/RELATION	DESCRIPTION
CACA	The mutation operator
refToModify	EReference added by the mutant

Table III. Description of the *CACA* Operator Metamodel

Example Figure 16 illustrates one mutation model of CACA applied to class2rdbms transformation. An EReference *references* from a $FKey$ instance to a $Table$ instance is wrongly added in an output model. Since the $re f erences$ EReference has a cardinality of 1 and may have already been initialized, the EReference is simply overridden.

The other operators are described using the same technique in the Annex [52]. In order to use these models, it is necessary to bind them with the other models corresponding to the tested transformation or the mutants, the trace and the test models. The mutation matrix already plays a pivot role between these latter; it is thus modified to also handle the mutation models.

DOI: 10.1002/stvr

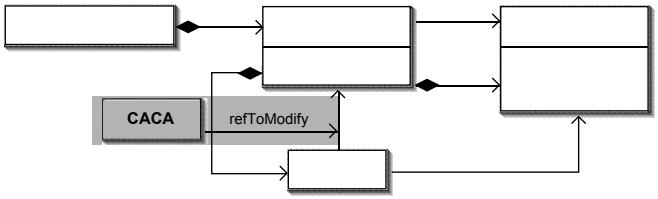


Figure 16. *CACA* Operator Application Example

6.3. Mutation Matrix Binding

Each mutant is returned by a single application of one mutation operator on the original transformation. In order to enable a direct access from a given mutant T_j to its associated operator Op_k, a link is added to each mutant in the mutation matrix, as shown in Figure 17. Two mutants T_0 and T_1 may be linked to the same mutation operator model. Indeed, as an instruction can be used many times at different places in a model transformation (as in traditional programming), a same mutation operator model Op_k can fit for several mutants.

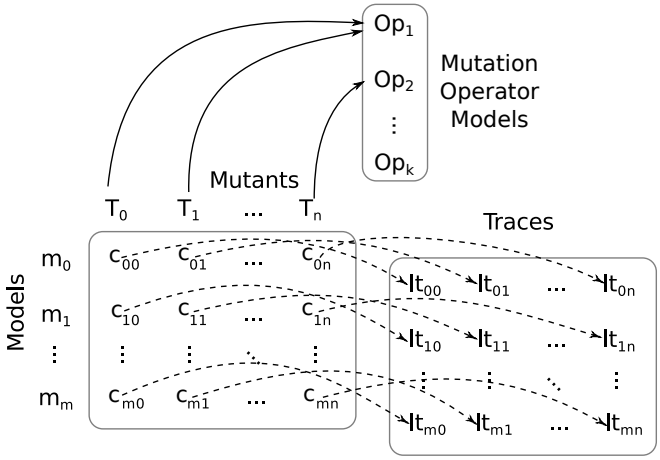


Figure 17. Mutation Matrix and Mutation Operator

7. CREATION OF A NEW TEST MODEL BY MODIFYING AN EXISTING ONE BASED ON PATTERN IDENTIFICATION

Based on the abstract representations of the operators and their definition on the input or output metamodel of the transformation, it is possible to identify why a mutant remains alive, and give some recommendations to modify existing test models that in their new versions should kill the mutant. For each mutation operator, few test model *patterns* (*i.e.* specific configurations in the model) leaving a mutant alive are identified. For each pattern, modifications that should kill the mutant are identified. This solves the two unresolved issues: "*Why a mutant has not been killed by a specific test model?*" and "*How a selected test model could be modified to produce the expected output model and thus kill the studied mutant?*". It has to be noticed that the following works are specified at a meta level based only on the abstract representation of the operators. It is not possible to be absolutely sure that the recommended modifications applied to a test model will kill the mutant. In other terms, the approach proposed in this paper provides an automatic analysis of the situation and advises some first modifications to be performed; in a lot of cases, they will be enough to kill the mutant.

DOI: 10.1002/stvr

7.1. Presentation of the Patterns Notion on the RSCC operator

In this subsection, some cases (also called *patterns*) that may let a mutant alive are illustrated. For example, the *RSCC- Relation to the Same Class Change* operator is considered. This operator replaces the navigation of one reference towards a class with the navigation of another reference to the same class (*cf.* 6.2.1). Thus, for an original transformation navigating the $self.a.ba.c$ sequence, one mutant may navigate the $self.a.bb.c$ sequence. The dot notation is used as in object oriented languages to navigate from one class to another: $self$ refers to a class, a, ba and bb to references, and c either to another reference or an attribute (see Figure 18 for an example of possible input models for this transformation). For this operator, three patterns letting the mutant alive are identified:

- **Pattern 1:** the original sequence and the mutated one finally point to the same instance,
- **Pattern 2:** the original sequence and the mutated one finally point to *null*,
- **Pattern 3:** the value of the element property pointed by the original sequence and the mutated one are the same.

These three patterns are represented in Figure 18 on three simple models and described below. Those models would have been selected because they execute the mutated rule without killing the mutant (following the process of the Section 5). The way to modify the test model is different according to the *pattern*.

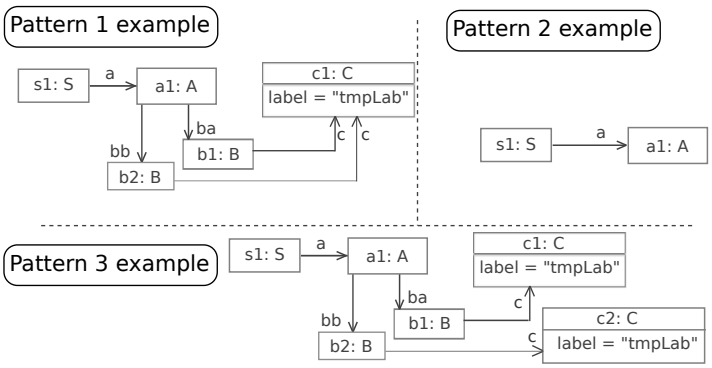


Figure 18. Patterns Examples

Pattern 1 The first pattern occurs when either the mutated navigation points to the same element as the original navigation (ba and bb point to the same instance of B) (not represented in the Figure) or the original sequence and the mutated one point to the same instance (as in the Figure).

The mutant can be killed if the mutated and the original navigation sequences point to two different instances of the same EClass. To produce the new test model, a new instance of this EClass, with different attribute values, is added to the model and the EReferences are updated.

Recommendation for Pattern 1: the idea is to add a new instance completely different from the one recovered by navigating the original EReference. This recommendation, albeit simple, may not *kill* a live mutant. It may therefore be more appropriate to add an element in the collection handled by the mutant:

- If at least one of the two EReferences (original or mutated) has its upper bound strictly greater than 1 ($1-n$ and $1-*$ EReferences, *i.e.* collections), a new instance is added into the collection. Indeed, adding a new instance into a collection handled by the mutant or the original transformation can greatly influence the navigation results.
- If at least one of the two EReferences has its lower bound equal to 0, it is possible to delete that instance. Thus, the original transformation or the mutant could not recover the instance and could return a different result (according to the deleted reference).

DOI: 10.1002/stvr

Pattern 2 The second pattern occurs when an intermediate reference cannot be navigated. In this example, if the a reference (or c reference) points to *null*, neither the original sequence $self.a.ba.c$ nor the mutated one $self.a.bb.c$ can be fully navigated. The element recovered by the two sequences is *null*.

The selected test model can then be modified by updating empty references, *i.e.* by updating the *null* reference to an element with the correct type.

Recommendation for Pattern 2: once again, the idea is to add a new instance. As for the previous pattern, a new instance is created and the reference navigated by the original transformation is updated in order to point to this new object.

Pattern 3 The third pattern occurs when the elements recovered by the original sequence $self.a.ba.c$ sequence and by the mutated one $self.a.bb.c$ have the same value for a given attribute (*e.g.* an attribute named "label" with value $tmpLab$).

To solve this problem, the tester can modify one of the attributes by changing its value.

Recommendation for Pattern 3: this time, it is the value of the properties that must be modified. Thus, once the elements handled by the faulty rule are identified, the value of their attributes is changed to create a new test model.

Automatic Pattern Detection: for each element retrieved by the algorithm described in Section 5.3 (elements handled by the mutated rule), references pointed by *initNavigation* and *newNavigation* are navigated. These instances are then compared with each other in order to detect if they are the same (Pattern 1) or $null$ (Pattern 2). If the instances are different, their attributes are compared in order to detect common values (Pattern 3). According to the identified pattern, the model can be modified.

7.2. Patterns for the CFCA Operator

The mutants created by the application of the *CFCA* operator select a single item in a collection. The mutant remains alive, because the original filter did not return any element or returned a single element. Indeed, if no item is returned by the original filter, the mutated filter will also return an empty set. Similarly, if one element is returned by the original filter, the mutation will have no effect. In both cases, the mutant and the original transformation will behave the same way. In the opposite, if the filtered collection contains several elements, a different behaviour should be highlighted between the original transformation and the mutant.

Without a complete specification of the filtering condition, it is difficult to provide a seamless solution to produce a new model. Nevertheless, it is possible to consider two patterns which let a mutant alive:

Pattern 1 the collection does not contain any element,

Pattern 2 the collection contains only one element.

Automatic Pattern Detection: to automatically determine the number of items in the collection, the elements that are handled by the mutated rule and that are in the collection pointed by the *eRefFrom* EReference (respectively the collection of *attributeFrom* in case of collection of attributes) are selected (Figure 13). The number of items in the collection is then computed.

Recommendation for Pattern 1: if no element is present in the collection, two different instances satisfying the filtering condition are created and added to the collection. Adding only one element resolves the Pattern 1 but Pattern 2 would occur.

Recommendation for Pattern 2: if one element is present in the collection, an instance which satisfies the filtering condition is created and added to the filtered collection.

Depending on the complexity of the filtering condition, modifying an existing model can be tedious. In the cases not covered by the two proposed patterns, the recommendations will not be adapted; the s will have to study in details the input test models to create a new one. Nevertheless, the testers' work is eased thanks to the traceability mechanism, by identifying the test model elements handled by the mutated rule.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

7.3. Pattern for the CACA Operator

Creation operators inject faults that add, delete, or replace elements in the output models. Consequently, the probability to observe a difference between the mutated transformation output model and the original transformation output is higher than navigation or filtering operators. However, since the creation operators imply changes in the output model, patterns are not based on specific configuration in a test model.

The description of the operator [9] states that when the upper bound of the mutated EReference is 1, whatever the number of times an element is linked to others, only the last one is taken into account, all the previous ones are overridden. For EReferences whose upper bound is $*$, the mutated instruction execution involves the addition of one extra element in the collection. So normally, the number of elements in the collection should be different in the output model resulting from the original transformation and the one generated by the mutant. If it is not the case, it may mean that this mutated instruction has not been executed.

Since the CACA operator concerns the output model, it is difficult to provide a seamless solution to produce a new model. Nevertheless, it is possible to consider three patterns which let a mutant alive. They are pretty vague since they are expressed directly on the output metamodel and not the input model. The associated recommendations are very generic; they concern the way to modify the input model.

Pattern 1 the cardinality of the mutated EReference in the output metamodel is $0..1$,

Pattern 2 the cardinality of the mutated EReference in the output metamodel is 1,

Pattern 3 the cardinality of the mutated EReference in the output metamodel is $0..*$, $1..*$ or $n..m$ and this part of the rule has never been executed.

Automatic Pattern Detection: the model of the CACA operator applied to the mutant is automatically analysed in order to identify the cardinality of the output metamodel EReference pointed by the $refToModify$ reference of the operator. The pattern 3 is identified if the upper bound of the cardinality differs from 1 and if there is no instance of the EReference pointed by the *refToModify* reference in the output model.

Recommendation for Pattern 1: It is possible to apply an ad hoc modification; the idea here is to avoid the overriding of the original EReference. The output model element pointed by the original EReference must thus be removed. So, in the output model returned by the original transformation, the reference will point to no object, whereas the reference in the mutant output model returned by the mutated transformation should point to an element. To achieve that, the *refToModify* reference is navigated and the instance it points to is collected. The input model elements creating this instance are automatically identified using the trace. Finally, the choice of the input model elements to delete or modify is the only task left to the tester.

Recommendation for Pattern 2: As previously explained, only the last time the element is linked to others is taken into account. If a mutant whose CACA operator is applied to an EReference with cardinality 1 is alive, this means that the added reference and the original one lead to the same result. The way this element is obtained from the input elements has to be studied and consequently modified.

Recommendation for Pattern 3: The CACA operator adds an element in a collection. Since the mutant remains alive, this means that the part of the mutated rule has not been executed. The test model has to be modified in order to force the execution of this part of the mutant. Of course, it is possible that the mutation is applied to a dead code zone and in that case no test model will lead to its execution.

7.4. Synthesis

In this section, 8 patterns, 2 or 3 for each mutation operator have been identified. For the sake of conciseness, the patterns for the other mutation operators are explained in the Annex [52]. These patterns undoubtedly leave mutants alive. For each of these patterns, a modification of an existing input model has been proposed to create a new test model that should kill a mutant. However, for some operators, it may occur that the identified patterns are not enough and that a more detailed

DOI: 10.1002/stvr

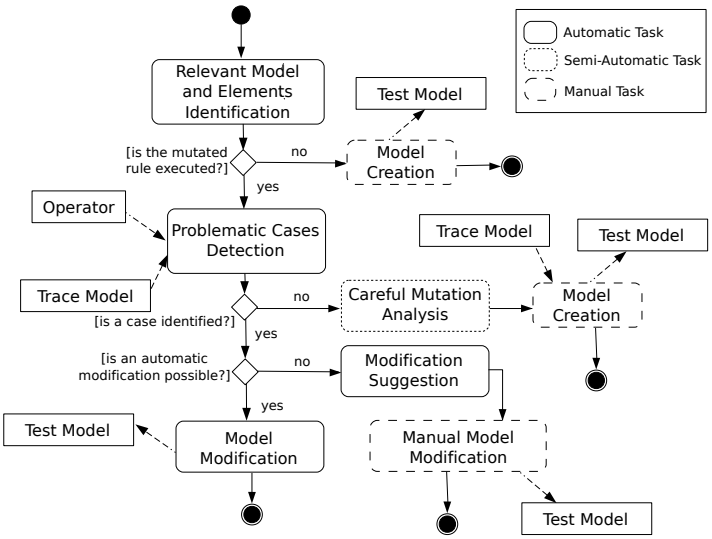


Figure 19. Choices During the New Test Model Creation

analysis is required. This analysis then relies on the results of the "relevant model and element identification" step (cf. 5.3).

To summarize the different steps of the proposed approach and their order, Figure 19 shows an activity diagram corresponding to the choices made by the final algorithm to support the creation of new test models. The process starts when *relevant models* and subparts of them are automatically identified. If the algorithm does not identify any relevant model, this means that the mutated rule has never been executed: this is the first information provided by the approach that helps testers. Therefore, the mutated rule must be analysed to determine the elements which are required to activate the rule execution, and a new test model is generated. If the mutated rule has been executed and if the mutant remains alive, the context must be analysed. The process then continues by trying to automatically identify whether the test model contains one of the proposed patterns. To perform this task, the modeled mutation operator and the model trace are used. When this activity is ended, if no pattern has been identified, the tester must semi-automatically analyse the mutant. To assist her in this task, the trace as well as the results of the "relevant model and element identification" step are provided. Once the mutant has been analysed, the tester can manually create the new test model. However, if a pattern has been identified, the proposed recommendation can be automatically implemented, a new test model is automatically created. For patterns where the change cannot be automatically performed, a modification suggestion is reported to the tester to help her to create a new test model.

Although the process is not fully-automated, assistance is provided thanks to the identification of relevant models and elements, the trace that helps to understand the transformation or the mutant execution, and the recommendations associated to the patterns.

8. CASE STUDY

In this section, the approach is deeply illustrated with the class2rdbms transformation. An initial test set is provided. In order to increase the mutation score, the approach is used to propose new test models owning the ability to *kill* a mutant with a minimal or even without any manual analysis. The input and the output metamodels of the transformation have been presented in section 2. Finally, in Section 8.9, the approach is experimented on a second case study to kill all the mutants of the fsm2ffsm transformation.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

8.1. Case Study Implementation

A language dedicated to model transformation such as QVTo would perhaps have been the most adapted as case study. Nevertheless, Kermeta has been chosen to implement the class2rdbms transformation. It is an imperative, object oriented language, with a model oriented type system. Indeed, some works extended and enhanced in this paper are based on Mottu and Sen works [9, 20]. At that time, the case study was also class2rdbms and it was already implemented using Kermeta. Using the same case study enables the comparison and the evaluation of the approach proposed in this paper.

The overview of the Kermeta implementation of the transformation is extracted from Muller *et al.* paper [53]. The transformation is implemented in three steps:

Table Creation Tables are created from each *Class* marked persistent in the input model.

Column Creation For each persistent *Class*, all *Attribute*s and outgoing *Association*s are transformed to create corresponding *Column*s. The *FKey*s (foreign keys) are created but their *cols* property cannot be filled, and the corresponding *Column* cannot be created because primary keys of references table cannot be known before they have been created, at the end of this step.

Foreign-Key Update Foreign-key columns are created in the *Table* that contains the *FKey*, and the property *cols* of *FKey* is updated.

The transformation has been implemented in a class named *Class2RDBMS*. As a general-purpose language, Kermeta does not manage transformation rules but class methods called *operations*. Moreover, the *Class2RDBMS* class provides a method `transform` that takes the input model as a parameter and returns the corresponding output model. Listing 2 presents an excerpt of the Kermeta code of the transformation.

The three steps of the transformation appear in the body of `transform` *operation* (lines 12 to 31). First, tables are created for each persistent `Class` (lines 19 to 24). Second `Columns` are created in the tables (lines 26 to 27) and, finally, `FKeys` are updated (line 30). The mapping between classes and tables is represented by the `reference class2table` in class `Class2RDBMS` (line 8). The `FKeys` reference is used to store all the created foreign keys during step 2 in order to be able to update them at step 3. The other operations (*e.g.* `createFKeyColumns`) have been implemented but only `createColumns` is presented here. Kermeta implementation of class2rdbms is composed of 113 lines of code in 11 operations. In the shown code, the trace link creation is also present (line 8). As explained in Section 5.1, the trace links creation has been manually inserted in the transformation, in the same way Falleri *et al.* did in their work [48]. With other languages like QVTo, the creation of the trace would have been automatic.

Faults designed with the mutation operators have been injected into the original transformation leading to 200 mutants. The approach focuses on the test set improvement for which mutation operator models and mutants already exist. For the sake of space, the specification of the mutation operator models and the mutants relative to the case study is not detailed in the paper, but available online [52]. The number of mutants per operator is synthesized in Table IV. Only the *CCCR - Classes compatible creation replacement* operator is not applied because there is no inheritance in the output metamodel.

Operator Type	Operator	Number of Mutants
Navigation	$ROCC$ - Relation to another class change	12
	$RSCC$ - Relation to the same class change	9
	$RSMA$ - Relation sequence modification with addition	72
	$RSMD$ - Relation sequence modification with deletion	12
Filtering	$CFCP$ - Collection filtering change with perturbation	38
	$CFCD$ - Collection filtering change with deletion	18
	$CFCA$ - Collection filtering change with addition	19
Creation	$CACD$ - Classes association creation deletion	11
	$CACA$ - Classes association creation addition	9

Table IV. Number of mutant per operators created for the class2rdbms transformation

DOI: 10.1002/stvr

For the experimentation, the mutation analysis process is launched on 200 mutants with a test set initially composed of 8 models. The way they have been created is independent from the approach that as any mutation approach requires an initial test set. In this case study, they have been manually created. The old version (0.4.1) of Kermeta is used, but it corresponds to the one used by initial works [9, 20]. The following of the Section details the process to create new test models using the proposed approach in order to kill a mutant produced by the *RSCC* mutation operator.

Listing 2: Extract of the class2rdbms transformation

```
1  package Class2RDBMS;
2  require kermeta // The kermeta standard library
3  require "LocalTrace.kmt" // The trace framework
4  require "../ClassMM.ecore" // Input metamodel in Ecore
5  require "../RDBMSMM.ecore" // Output metamodel in Ecore
6        [...]
7  class Class2RDBMS {
8    reference class2table : LocalTrace // The trace of the transformation
9    reference fkeys : Collection<FKey> // Set of keys of the output model
10   operation transform(inputModel : ClassModel) : RDBMSModel is do
11     class2table := LocalTrace.new // Initialise the trace
12     class2table.create
13     fkeys := Set<FKey>.new
14     result := RDBMSModel.new
15     // Create tables
16     getAllClasses(inputModel).select{ c | c.is_persistent }
17       .each{ c | var table : Table init Table.new
18                  table.name := c.name
19                  class2table.storeTrace(c, table)
20                  result.table.add(table)}
21     // Create columns
22     getAllClasses(inputModel).select{ c | c.is_persistent }
23       .each{ c | createColumns(class2table.getTargetElem(c), c, "")}
24     // Create foreign keys
25     fkeys.each{ k | k.createFKeyColumns }
26   end
27       [...]
28  operation createColumns(table : Table, cls : Class, prefix : String) is do
29     // add all attributes
30     getAllAttributes(cls).each{ att |
31                               createColumnsForAttribute(table, att, prefix) }
32     // add all associations
33     getAllAssociation(cls).each{ asso |
34                            createColumnsForAssociation(table, asso, prefix) }
35   end
36       [...]
37 }
```

Listing 3: *RSCC_class2RDBMS_33* mutant extract

```
1  operation createColumnsForAssociation(table : Table, asso : Association,
       prefix : String) is do
2    // if isPersistentClass(asso.dest) then // original
3    if isPersistentClass(asso.src) then // mutant

15   else

17   end
18 end
```

8.2. Mutation Operator Modelisation: RSCC_class2RDBMS_33 example

Among all the produced mutants, the focus is put on the mutant *RSCC_class2RDBMS_33* (this number has no sense in itself, it is just a way to identify one mutant among the 200). Figure 20 shows the mutation operator model (RSCC_4) which leads to its production. This model expresses,

DOI: 10.1002/stvr

thanks to its *initNavigation* reference, that initially, a *dest* reference was navigated and that its mutation leads to the navigation of an *src* reference (thanks to its *newNavigation* reference). These two references belong to the *Association* class and point to the *Class* class.

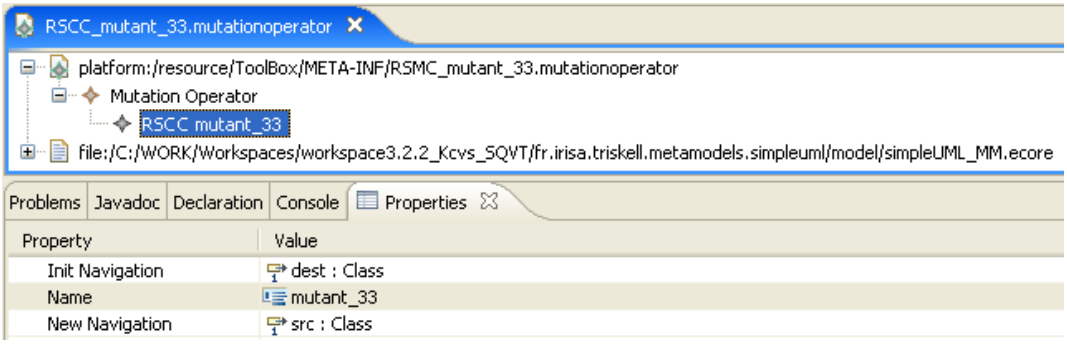


Figure 20. *RSCC_4* Mutation Operator Modelisation

The mutation operator model RSCC_4, used for this mutant, is applied each time the navigation $dest$ occurs producing each time a new mutant. For instance, Listing 3 is an excerpt of *RSCC_class2RDBMS_33* mutant. Line 2 represents the initial instruction whereas line 3 represents the instruction mutated by the *RSCC* operator application.

8.3. First Iteration of the Mutation Analysis Process

To start the mutation analysis process, the test models built by the tester are successively executed on the transformation under test and on all its mutants. During the executions, the traces are automatically generated. For each mutant and each test model, the resulting output model is compared to the one generated by the original transformation in order to determine if the test model has killed the mutant or not. Results of the model comparisons are stored in a mutation matrix. Figure 21 shows an extract of the produced mutation matrix corresponding to the mutant *RSCC_class2RDBMS_33*. It contains information about the mutant:

- the name of the modified rule: *createColumnsForAssociation*,
- the mutant name: *RSCC_class2RDBMS_33.kmt*,
- the file path,
- a link to the modeled mutation operator (via *MutationOp*).

Furthermore, the results of the comparison are specified through the `Cell` reference. For example, for this mutant, each of the associated `Cell` has the value `true`. This means that each test model has left the mutant alive. Since no test model kills the mutant, it is considered alive.

The analysis of the mutation matrix indicates that 144 mutants were killed and 56 remained alive, giving a mutation score of 72% for the 200 mutants and the 8 test models.

To increase the mutation score, a live mutant is chosen. Here the mutant *RSCC_class2RDBMS_33* has been chosen and the approach is applied in order to add a new test model that will kill it.

8.4. Identification of Relevant Model and Elements

The application of the approach identifies relevant models, *i.e.* those for which the mutated rule of the *RSCC_class2RDBMS_33* mutant is executed. The results are given in Table V, that summarizes the input models identified as relevant, their model elements and the resulting (destination) elements created by the mutated rule.

Among the 8 test models in the initial test set, four were identified as relevant: *ClassModel02*, *ClassModel04*, *ClassModel05* and *ClassModel06*. It means that these models own elements (identified in the *Source Elements* column) that are handled by the mutated rule. However, they all let the mutant alive, the results of their execution are the same as the one of the original transformation. They do not kill the mutants, but they are candidate to be improved.

DOI: 10.1002/stvr

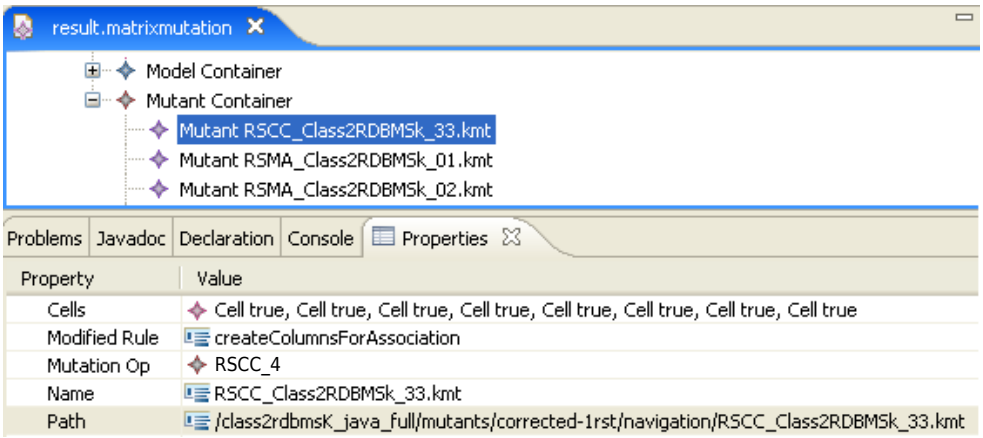


Figure 21. Mutation Matrix Extract

TEST MODEL	SOURCE ELEMENTS	DESTINATION ELEMENTS
ClassModel02	*next: Association*	*next: FKey*
ClassModel04	*a: Association*	*a: FKey*
ClassModel05	*b: Association*	*b: FKey*
ClassModel06	*a: Association* *b: Association*	*a: FKey* *b: FKey*

Table V. Identified Relevant Model and Elements for *RSCC_class2RDBMS_33*

8.5. *Test Models Analysis*

To understand why the mutant is not killed, further analysis of these identified test models is needed. Based on the identified element, the automatic patterns detection is launched for the mutation operator *RSCC_class2RDBMS_33*. The results of the pattern detection, including the pattern number, is summarized in Table VI.

TEST MODEL	PATTERN NUMBER
ClassModel02	*RSCC_P*1

Table VI. Identified Patterns for *RSCC_class2RDBMS_33*

From the 8 initial test models, the "relevant model and element identification" step results in the identification of 4 relevant models. From these 4 models, the pattern detection identifies only 1 of them. The detected pattern is the number 1 for *RSCC* mutation operators. This pattern indicates that the element recovered by the original navigation and the mutated one is the same. It means that for test model *ClassModel02*, the *src* and *dest* reference of the *next Association* point to the same instance.

8.6. *Test Model Creation*

The proposed modification for this pattern is: add a new instance of *Class* in the model, radically different from the *Class* recovered by the mutant and the original transformation. Then either the *src* or the *dest* reference need to be updated to refer the newly created instance.

The *Class* recovered by the original *Association* and the mutated one is the class named *customer* whose *is_persistent* attribute is set to `true`. Therefore, a new *Class* instance, called *newInstance1* is created. Given the structural constraints expressed in the metamodel, an *Attribute* must be added to this new *Class*. In order to maximize the chance to kill the mutant, the characteristics of this

DOI: 10.1002/stvr

attribute are defined differently from the one owned by the *customer Class*. The new instance has its *NewInstance1 is_persistent* attribute set to `false` and contains an *Attribute newAttribute* of type *customer*. Figure 22 represents the created model.

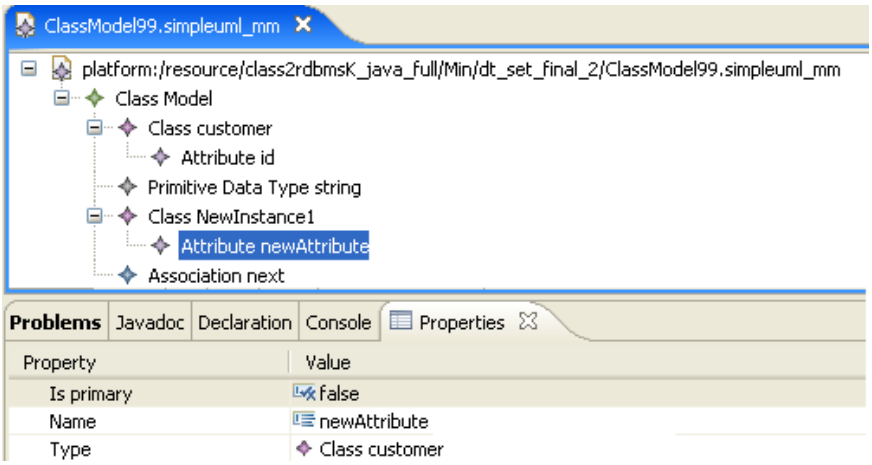


Figure 22. New Test Model Produced

Once the new test data is added to the set of test models, the mutation analysis process is launched again. This time, the mutation score is 78.5%. In the mutation matrix (partially shown in Figure 23), the *RSCC_class2RDBMS_33* mutant contains a `false` *Cell*, indicating that it is henceforth killed. The added test model has effectively killed the mutant.

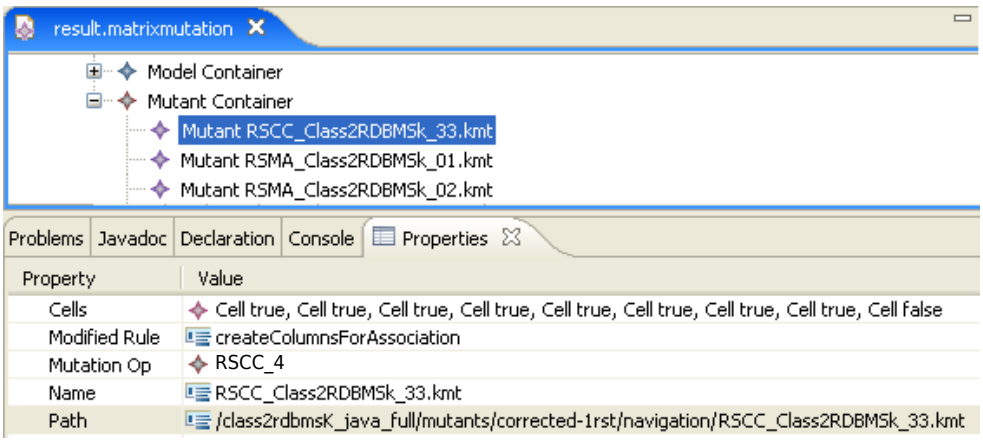


Figure 23. Produced Mutation Matrix Extract

8.7. New Test Models Productions

This time the analysis focuses on the mutant *CFCA_class2RDBMS_10* obtained by applying the filtering operator *CFCA* on the *attrs* reference of the input meta-model. Among the 9 test models, 5 are identified as relevant (*i.e.* they allow the execution of the mutated rule). These 5 models are concerned by the pattern 2 of the operator *CFCA* (*cf.* Section 7.1). For the record, this pattern indicates that the filtered collection (*attrs*) contains only one element. According to the solution sketched in Section 7.1, an element has to be added to the *attrs* collection. So one of these models is copied and then modified to produce a new test model. The mutation analysis process is started again with 10 test models; the mutation score is 89%. Once again, the considered mutant *CFCA_mutant_10* is actually killed by the new test model.

DOI: 10.1002/stvr

The new iteration deals with the *RSMA_mutant_09* mutant obtained by applying the mutation operator *RSMA* that adds an extra navigation (named *parent*) to the initial navigation (named *dest*). Among the 10 test models, 4 are identified as relevant and among these 4 models, only 3 satisfy one of the patterns associated to the applied mutation operator. The three cases respect the pattern 3. This pattern states that instances recovered by the *parent* and *dest* reference own attributes with the same value. The trace identifies the *is_persistent* attribute. The solution proposed for this pattern is a modification of the value of one of the attributes achieved by the navigation. Following these indications, one of the three models is copied and modified to provide an 11^{th} test model. The mutation analysis process is launched again, and the mutation score is 91.5%. To exceed 90% the test set contains 11 models. The process can be executed again to increase the mutation score.

8.8. Improvement Comparing To Manual Approach

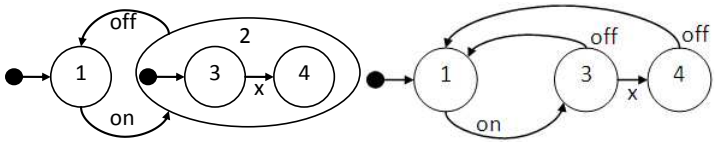
Without the approach proposed in this paper, when a mutant remains alive, the user must (i) look for the mutated rule, (ii) study each execution of this mutant and (iii) understand why it has not been killed. She has to think about what the output model should look like in case the mutant would be killed in order to create an input model leading to such an output. For this purpose, she manually executes the mutant on potentially each model to understand how each element of the output models has been created. She may go through all the elements of each model. Possibly all models and all their elements are studied by the user who has no indication. Then she has to perform some modifications, verify that the new models kill the mutant and continue until succeeding. This has to be performed for each live mutant; the process is long and fastidious.

Using the proposed assistant may lead to the construction of a test model set finally containing as many test models as mutants and potentially more than the set manually obtained. Indeed, sometimes killing a mutant can lead to kill other ones as a side effect. However, it is possible to automatically prune the test model set by "reducing" the mutation matrix. This reduction relies on the premise that a test model killing mutants already killed by other test models is useless.

This case study aims to illustrate the different steps of the proposed approach and their iteration. In the course of this description, the gain in terms of studied models and model elements and the benefit of the patterns are shown. For example, in the case of the mutant 33, only 4 models among 8 are considered relevant. For each of them, the relevant elements are highlighted (here only 1 or 2 per model). Moreover, using the patterns associated to the operator leads to the identification of a single model, and indications are given on the way to improve it. Next section introduces a new experiment, on a quantitative point of view.

8.9. An Experiment Evaluating the Number of Elements To Be Analysed

In this subsection, the proposed approach is experimented on another example: a model transformation which flattens finite state machines. This transformation takes as input a hierarchical state machine and produces as output an equivalent flattened state machine. Figure 24(a) shows a hierarchical state machine, and Figure 24(b) shows its equivalent flattened state machine. After a brief description of the preliminary step, the approach is used to create new test models with less effort in order to kill live mutants. Finally, the gain is analyzed on the tester's effort in terms of studied models and model elements.



(a) Hierarchical FSM Model Example (b) Flattened FSM Model Example

Figure 24. One hierarchical Finite State Machine Model (FSM) sample and one flattened FSM

DOI: 10.1002/stvr

Creation of mutants and initial test models. Using the mutation operator metamodels, the possible mutations are designed based on the finite state machine metamodel [52] and generate 76 mutation operator models. The operators are applied to the transformation implemented in Kermeta and obtain 126 mutants (details are given in the Annex [52]).

Based on the partitioning approach proposed in [19], a set of 9 test models, respectively labelled $M1$ to $M9$, is created. Table VII shows their respective sizes, calculated upon the number of elements (*i.e.* objects, attributes, associations) they are made of.

MODEL NAME	$M1$	$M2$	$M3$	$M4$	$M5$	$M6$	$M7$	$M8$	$M9$	Total
# OF ELEMENTS	14	31	31	14	16	12	10	10	25	163

Table VII. Respective sizes of the initial test data set models

Results and Analysis. Applying the proposed approach on this case study, the mutation score of the test model set is increased from 45% to 100% in 8 iterations, producing 8 new test models. Each iteration is dedicated to kill one live mutant. Table VIII shows final results of our test model improvement process. Several observations can be drawn from this case study:

- Our traceability approach allows the tester to drastically reduce the search space to find which models and which elements of these models are the most relevant for creating new test models. Whereas a tester should entirely analyse the existing test models (*e.g.* 163 model elements at the first iteration) to create new models, with the proposed approach, only two models (corresponding to 39 elements) are relevant. The analysis can even be focused on less elements (only 11 of a single model). The gain column compares the total number of model elements with the number of studied elements (*e.g.* 93.25%). The global average gain in terms of elements to be covered by the full approach is around 87%.
- The common patterns identified on mutation operators provide to the tester some simple modifications that can be performed on the test models. In this case study, five mutants have been killed by simply applying the recommended modification on the test models. Among the three remaining mutants, the trace indicated for two of them that their mutated part was not executed. Finally, only one of the eight studied mutants required an in-depth analysis.

ITERATION	MODELS SET SIZE	TOTAL MODELS SIZE	STUDIED MUTANT	STUDIED MODELS SIZE	STUDIED ELEMENTS SIZE	GAIN (%)	PATTERN DETECTED
1	9	163	$CFCD\_4$	39	11	$93,25$	$CFCD\_P2$
2	10	180	$CACA\_4$	90	36	80	–
3	11	205	$CFCA\_10$	25	5	$97,56$	$CFCA\_P2$
4	12	235	$CFCA\_8$	55	10	$95,74$	–
5	13	260	$CFCD\_8$	80	15	$94,23$	$CFCD\_P2$
6	14	288	$CACA\_6$	198	56	$80,56$	–
7	15	313	$CACD\_5$	223	60	$80,83$	$CACD\_P2$
8	16	338	$CFCA\_7$	248	68	$79,88$	$CFCA\_P1$
Result	17	368	$100\%killed$	–	–	–	–

Table VIII. Final results of the process on the *fsm2ffsm* transformation

9. CONCLUSION

The mutation analysis process suffers from the manual and tedious character of the test data set improvement activity. In this paper, a solution to this issue is provided in order to go towards a full automation of the mutation analysis process adapted to model transformation. More particularly, it has been proposed a process based on transformation traceability, an abstract representation of the mutation operators and problematic specific configurations to enhance the automation of the test data set improvement activity.

In a previous work, an algorithm based on the model transformation traceability has been proposed to identify, for each live mutant, relevant test models and their involved elements [46]. The

DOI: 10.1002/stvr

major drawback of this algorithm was that it could not help in the study of what the output model should be if the mutant were killed and thus not providing possible modifications. To solve this issue and enhance the previous method, an approach independent from the tested transformation and the used transformation language has been proposed. This independency results from the modelling of the mutation operators. Some specific problematic configurations (patterns) are identified for each mutation operator and recommendations are proposed.

The proposed approach has been illustrated with the class2rdbms transformation written in Kermeta with an initial set of 8 test models. Three live mutants have been studied in order to produce new test model killing them. The initial test model set reached a 72% mutation score, and the 3 new test models increase the score to 90%. Each time, the new test model has been easily and quickly created thanks to the proposed algorithms. The approach has been experimented on a second case study. It highlights how the approach drastically reduces the number of model elements to study compared to the manual approach where no indication to reduce the investigation in the models was given.

The approach has been applied to the Kermeta language but is not dedicated to Kermeta. Indeed, the mutation matrix representation, the traceability and the mutation operator representation are free from any transformation language. Moreover, the identified patterns come from observations about Kermeta and QVTo [16] transformations. They thus do not depend on a specific transformation language and can be used with another one.

Currently, an automatic modification of the existing test models to create a new test model cannot be provided each time. Indeed, in order to automatically create a new test model by simple modification, structural constraints of the input metamodels must be taken into account. The arisen challenges to obtain an automatic test models generation can be also found in general automatic test model set generation challenges. Thus, it is planned to look towards test model generation techniques in order to provide a full automatic test model creation.

REFERENCES

1. Gamatié A, Le Beux S, Piel E, Ben Atitallah R, Etien A, Marquet P, Dekeyser JL. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst.* Nov 2011; .
2. DeMillo R, Lipton R, Sayward F. Hints on test data selection: Help for the practicing programmer. *Computer* 1978; **11**(4).
3. Voas JM, Miller KW. The revealing power of a test case. *Softw. Test., Verif. Reliab.* 1992; **2**(1):25–42.
4. Murmane T, Reed K, Assoc T, Carlton V. On the effectiveness of mutation analysis as a black box testing technique. *Software Engineering Conference*, 2001; 12–20.
5. Baudry B, Le Traon Y, Sunyé G, Jézéquel JM. Measuring and improving design patterns testability. *Proceedings of Metrics Symposium 2003*, 2003.
6. Frankl PG, Weiss SN, Hu C. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software* 1997; .
7. DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol.* 1993; .
8. Baudry B, Ghosh S, Fleurey F, France R, Le Traon Y, Mottu J. Barriers to systematic model transformation testing. *Communications of the ACM* 2009; .
9. Mottu JM, Baudry B, Le Traon Y. Mutation analysis testing for model transformations. *ECMDA 06*, Spain, 2006.
10. Bézivin J, Rumpe B, Schürr A, Tratt L. Model transformations in practice workshop. *Satellite Events at the MoDELS 2005 Conference*, 2005.
11. van Gigch JPv. *System design, modeling and metamodeling*. Plenum press, New York, 1991.
12. Gonzalez-Perez C, Henderson-Sellers B. *Metamodelling for Software Engineering*. Wiley Publishing, 2008.
13. Jeusfeld MA, Jarke M, Mylopoulos J ((eds.)). *Metamodeling for Method Engineering*. MIT Press: Cambridge, MA, USA, 2009.
14. Sendall S, Kozaczynski W. Model transformation: the heart and soul of model-driven software development. *Software, IEEE* 2003; .
15. Czarnecki K, Helsen S. Classification of model transformation approaches. *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
16. Object Management Group, Inc. MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/07-07-07.pdf Jul 2007. OMG paper.
17. Mottu JM, Baudry B, Le Traon Y. Model transformation testing : oracle issue. *MoDeVVa workshop colocated with ICST'08.*, Lillehammer, Norway, 2008.
18. Finot O, Mottu JM, Sunye G, Attiogbe C. Partial test oracle in model transformation testing. *International Conference on Model Transformation*, Budapest, Hungary, 2013.

DOI: 10.1002/stvr

19. Fleurey F, Baudry B, Muller PA, Traon YL. Qualifying input test data for model transformations. *Software and System Modeling* 2009; **8**(2).
20. Sen S, Baudry B, Mottu JM. Automatic model generation strategies for model transformation testing. *International Conference on Model Transformation, ICMT09.*, Zurich, Switzerland, 2009.
21. Sen S, Moha N, Baudry B, Jézéquel JM. Meta-model pruning. *Model Driven Engineering Languages and Systems, MODELS, Lecture Notes in Computer Science*, vol. 5795, Schürr A, Selic B (eds.), Springer, 2009.
22. Mottu JM, Sen S, Tisi M, Cabot J. Static analysis of model transformations for effective test generation. *IEEE International Symposium on Software Reliability Engineering, ISSRE 2012*, IEEE: Dalls, USA, 2012.
23. Guerra E. Specification-driven test generation for model transformations. *ICMT*, *Lecture Notes in Computer Science*, vol. 7307, Hu Z, de Lara J (eds.), Springer, 2012; 40–55.
24. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 2010; .
25. Li N, Praphamontripong U, Offutt J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.
26. Smith B, Williams L. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software* 2009; **82**(11):1819 – 1832.
27. Smith B, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Softw. Engg.* 2009; **14**(3).
28. Pimont S, Rault JC. A software reliability assessment based on a structural and behavioral analysis of programs. *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, 1976.
29. Clarke LA, Podgurski A, Richardson DJ, Zeil SJ. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng.* Nov 1989; .
30. Ammann P, Offutt J. *Introduction to software testing*. Cambridge University Press, 2008.
31. Barbosa EF, Maldonado JC, Vincenzi AMR. Toward the determination of sufficient mutant operators for c. *Softw. Test., Verif. Reliab.* 2001; **11**(2):113–136.
32. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 1996; **5**(2).
33. Ma YS, Kwon YR, Offutt J. Inter-class mutation operators for java. *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002.
34. Argrawal, DeMillo, Hathaway, Hsu, Krauser, Martin, Mathur, Spafford. Design of mutant operators for the c programming language. *Technical Report*.
35. Tisi M, Jouault F, Fraternali P, Ceri S, Bézivin J. On the use of higher-order model transformations. *Model Driven Architecture - Foundations and Applications, 5th European Conference*, 2009.
36. Fraternali P, Tisi M. Mutation analysis for model transformations in atl. *International Workshop on Model Transformation with ATL*, Nantes, France, 2009.
37. Ferrari FC, Maldonado JC, Rashid A. Mutation testing for aspect-oriented programs. *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008.
38. Simão A, Maldonado JC, da Silva Bigonha R. A transformational language for mutant description. *Comput. Lang. Syst. Struct.* 2009; **35**(3).
39. Fleurey F, Steel J, Baudry B. Validation in model-driven engineering: testing model transformations. *Proceedings of MoDeVa*, 2004; 29–40.
40. Baudry B, Fleurey F, Jézéquel JM, Le Traon Y. From genetic to bacteriological algorithms for mutation-based testing. *STVR Journal* Jun 2005; **15**(2):73–96.
41. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, ACM: New York, NY, USA, 2010; 147–158.
42. Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software. *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SIGSOFT FSE, 2011.
43. Ayari K, Bouktif S, Antoniol G. Automatic mutation test input data generation via ant colony. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM, 2007; 1074–1081.
44. IEEE. *IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society Press: New York, NY, USA, 1990.
45. EMFcompare. www.eclipse.org/emft/projects/compare.
46. Aranega V, Mottu JM, Etien A, Dekeyser JL. Traceability for mutation analysis in model transformation. *Proceedings of MODELS'10 Workshops and Symposia*, Springer-Verlag: Berlin, Heidelberg, 2010; 259–273. URL `http://dl.acm.org/citation.cfm?id=2008503.2008537`.
47. Jouault F. Loosely coupled traceability for atl. *ECMDA Workshop on Traceability*, Germany, 2005.
48. Falleri JR, Huchard M, Nebut C. Towards a traceability framework for model transformations in kermeta. *ECMDA-TW Workshop*, 2006.
49. Yie A, Wagelaar D. Advanced traceability for ATL. *1st International Workshop on Model Transformation with ATL (MtATL 2009)*, Nantes, France, 2009.
50. Vanhooff B, Ayed D, Baelen SV, Joosen W, Berbers Y. Uniti: A unified transformation infrastructure. *MoDELS*, USA, 2007.
51. Aranega V, Etien A, Dekeyser JL. Using an Alternative Trace for QVT. *Workshop on Multi-Paradigm Modeling*, Olso, Norway, 2010.
52. Aranega V, Mottu JM, Etien A, Degueule T, Baudry B, Dekeyser JL. Annexe and experimentation material. `https://sites.google.com/site/mutationtesttransfo/`.
53. Muller PA, Fleurey F, Vojtisek D, Drey Z, Pollet D, Fondement F, Studer P, Jézéquel JM. On executable meta-languages applied to model transformations. *Model Transformations In Practice Workshop*, Jamaica, 2005.

DOI: 10.1002/stvr