Autom Softw Eng (2011) 18: 199–224
DOI 10.1007/s10515-010-0079-3

Example-based model-transformation testing

Marouane Kessentini · Houari Sahraoui · Mounir Boukadoum

Received: 19 July 2010 / Accepted: 15 December 2010 / Published online: 4 January 2011

Abstract A major concern in model-driven engineering is how to ensure the quality of the model-transformation mechanisms. One validation method that is commonly used is model transformation testing. When using this method, two important issues need to be addressed: the efficient generation/selection of test cases and the definition of oracle functions that assess the validity of the transformed models. This work is concerned with the latter. We propose a novel oracle function for model transformation testing that relies on the premise that the more a transformation deviates from well-known good transformation examples, the more likely it is erroneous. More precisely, the proposed oracle function compares target test cases with a base of examples that contains good quality transformation traces, and then assigns a risk level to them accordingly. Our approach takes inspiration from the biological metaphor of immune systems, where pathogens are identified by their difference with normal body cells. A significant feature of the approach is that one no longer needs to define an expected model for each test case. Furthermore, the detected faulty candidates are ordered by degree of risk, which helps the tester inspect the results. The validation results on a transformation mechanism used by an industrial partner confirm the effectiveness of our approach.

M. Kessentini (✉) · H. Sahraoui
DIRO, Université de Montréal, Montréal, Canada
e-mail: kessentm@iro.umontreal.ca

H. Sahraoui
e-mail: sahraoui@iro.umontreal.ca

M. Kessentini
College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia

M. Boukadoum
DI, Université du Québec à Montréal, Montréal, Canada
e-mail: Boukadoum.mounir@uqam.ca

Keywords Model transformation testing · Artificial immune system · Traceability

1 Introduction

Model-Driven Engineering (MDE) aims to provide automated support for the creation, refinement, refactoring, and transformation of software models (France and Rumpe 2007). One of the major challenges of MDE is to automate these procedures while preserving the quality of the produced models (Czarnecki and Helsen 2003). In particular, efficient techniques and tools for validating model transformations are needed. One of them is model transformation testing (Lin et al. 2005).

Model transformation testing typically consists of synthesizing a large number of different models as test cases, running the transformation mechanism on them, and verifying the result using an *oracle* function. In this context, two important issues must be addressed: the efficient generation/selection of test cases and the definition of the oracle function to assess the validity of transformed models. This work is concerned with the latter.

Defining the oracle function for model transformation testing is a challenge (Mottu et al. 2008; Baudry et al. 2006). Many problems need to be solved. First, the definition of reference models to compare with the transformation outputs is not obvious (Mottu et al. 2008; Lin et al. 2005; Kolovos et al. 2006). Second, for large models, if the candidate transformation errors are given without any risk quantification, inspecting them could be time and resource-consuming (Baudry et al. 2006). Finally, transformation errors can have different causes such as transformation logic (rules) or source/ target metamodels (Kuster and Abd-El-Razik 2006). Finally, to be effective, the testing process should allow identification of the error causes (Baudry et al. 2006).

The primary contribution of this paper is to generate an oracle function "by example" that addresses the above-mentioned issues. The presented work draws an analogy between the detection of transformation errors and the detection of pathogens in the human body. In the human immune system, the process relies on detecting abnormal conditions; the more abnormal something is, the riskier it is considered. By analogy, we propose an oracle function that compares target test cases with a base of examples containing good quality transformation traces, and then assigns a risk level to the former, based on the dissimilarity between the two as determined by an artificial immune system-based algorithm (Forrest et al. 1994). Consequently, one no longer needs to define an expected model for each test case, and the traceability links help the tester understand the error origins. Furthermore, the detected faults are ordered by degree of risk to help the tester perform further analysis. For this, a custom tool was developed to visualize the risky fragments found in the test cases in different colors, each related to an obtained risk score.

The proposed approach is illustrated and evaluated with the known case of transforming UML class diagrams (CD) to relational schemas (RS). The choice of CD-to-RS transformation is motivated by the fact that it has been investigated by other means and is reasonably complex; this allows focusing on describing the technical aspects of the approach and comparing it with alternatives.

The remainder of this paper is as follows: Sect. 2 presents the relevant background and the motivation for the presented work; Sect. 3 describes the AIS-based algorithm;

an evaluation of the algorithm with industrial validation is explained and its results are discussed in Sect. 4; the benefits and also the limitations of the approach are presented in Sect. 5; Sect. 6 is dedicated to related work. Finally, concluding remarks and future work are provided in Sect. 7.

2 Background and motivation

As showed in Fig. 1, a model transformation mechanism takes as input a model to transform, the *source model*, and produces as output another model, the *target model*. The source and target models must conform, respectively, to specific metamodels and, usually, relatively complex transformation rules are defined to insure this.

We can illustrate this definition of the model transformation mechanism with the case of class diagram to relational schema transformation. Figure 2 shows a simplified metamodel of the UML class diagram (Bezivin et al. 2004), containing concepts like class, attribute, relationship between classes, etc. Figure 3 shows a partial view of the relational schema metamodel (Bezivin et al. 2004), composed of table, column, attribute, etc. The transformation mechanism, based on rules, will then specify how the persistent classes, their attributes and their associations should be transformed into tables, columns and keys.

Once defined, the transformation mechanism needs to be tested to detect potential errors. As described in Fig. 4, the basic testing activities consist of designing test cases, executing the model transformation on them, and examining the obtained re-

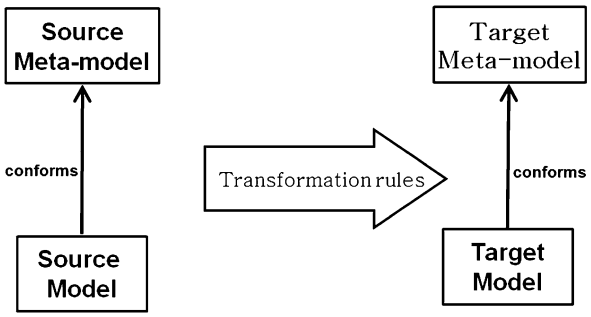


Fig. 1 Model transformation mechanism

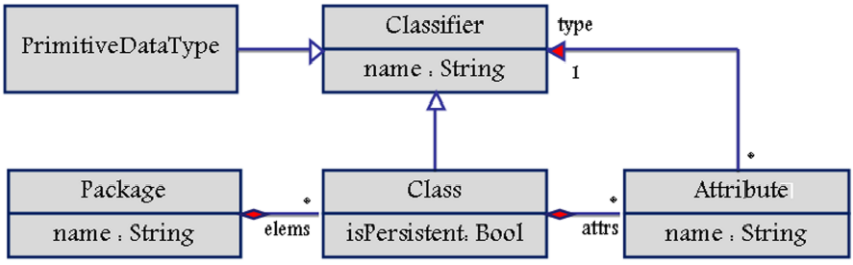


Fig. 2 Class diagram metamodel

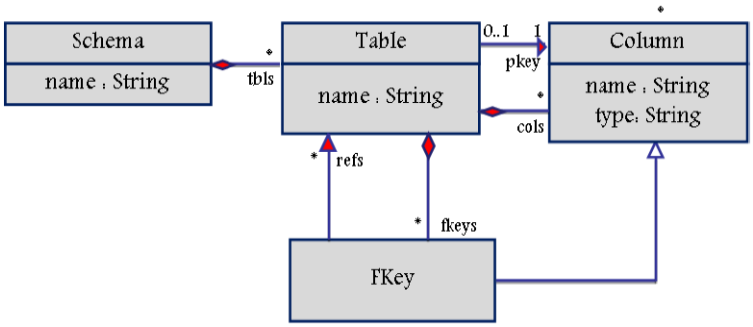


Fig. 3 Relational schema metamodel

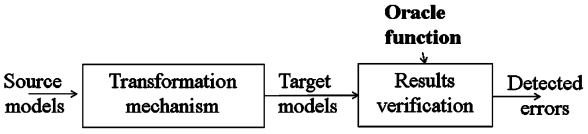


Fig. 4 Model transformation testing process

sults (Lin et al. 2005). This requires an oracle function that analyzes the validity of the transformed models.

Much work has addressed the automatic generation of test cases (Lin et al. 2005; Brottier et al. 2006; Baudry et al. 2002; Fleurey et al. 2004). This paper focuses on the complementary issue of defining the oracle function, assuming that a set of test data can be provided. There are many different ways to define this function, depending on the effort provided and the amount of information that is available (formal specification, expected output, etc.) (Mottu et al. 2008). We distinguish between two main categories of oracle function definitions for model transformation testing: model comparison (Kolovos et al. 2006) and specification-conformance checking (Cariou et al. 2004; Baudry et al. 2006).

For the first category, current MDE technologies and model repositories store and manipulate models as graphs of objects. Thus, when the expected output model is available, the oracle compares two graphs. In this case, the oracle definition problem has the same complexity as the graph isomorphism problem, which is NP-hard (Khuller and Raghavachari 1999). In particular, we can find a test case output and an expected model that look different (contain different model elements) but have the same meaning. So, the complexity of these data structures makes it difficult to provide an efficient and reliable tool for comparison (Baudry et al. 2006). Still, several studies have proposed simplified versions with a lower computation cost (Alanen and Porres 2003). For example, Alanen and Porres (2003) present a theoretical framework for performing model differencing. However, they rely on the use of unique identifiers for the model elements.

To illustrate the specification conformance category, we present two contributions: design by contract (Cariou et al. 2004) and pattern matching (Baudry et al. 2006).

For design by contract, the idea is that the transformation of source models into target models is coupled with a contract consisting of pre- and post-conditions. Hence, the transformation is tested with a range of source models that satisfy the pre-

conditions to ensure that it always yield target models that satisfy the post-conditions. If the transformation produces an output model that violates a post-condition, then the contract is not satisfied and the transformation needs to be corrected. The contract is defined at the metamodel level and conditions are generally expressed in OCL.

The second method of specification-conformance checking uses patterns that are defined as model fragments, instead of pre-conditions, and for each pattern, a set of post-conditions. Then, the process of pattern matching consists in checking the presence of a pattern in a source model. When a pattern is present, the oracle function evaluates the associated post-conditions on the output model. The difference with design by contract approaches is that both patterns and post-conditions are specified in terms of example of models rather than in terms of metamodel concepts.

Specification-based oracles are difficult to define. Indeed, the number of constraints to define can be very large to cover all transformation possibilities (Baudry et al. 2006). This is especially the case of contracts related to one-to-many mappings. Moreover, being formal specifications, these constraints are difficult to write in practice (Cariou et al. 2004). In pattern matching, the constraints are described at the model level and may lead to a fastidious task to define them for each possible instance of the source metamodel (Cariou et al. 2004).

To address the preceding issues, we propose a new oracle definition inspired from the immune system (IS) paradigm that will be described in the next section.

3 Approach

This section describes the principles that underlie the proposed method for model transformation testing. It starts by presenting the metaphor that inspired our work, the artificial immune system (AIS). Then, we provide the details of the approach and our adaptation of the AIS algorithm to the model transformation testing problem.

3.1 Immune system metaphor

The role of an immune system (IS) is to protect its host organism against harmful disease caused by invaders (pathogens) and/or malfunctioning cells. A biological immune system reacts to adverse environmental changes by *identifying* and *eliminating* antigens, which are substances or organisms that are recognized by the body as foreign, and which stimulate the immune *response*. A detailed presentation of the biological immune system is provided in Kuby et al. (1997). This paper adapts the first phase of AIS operation to *identify/detect* transformation traces that present a high-risk of containing errors, when testing a transformation outcome.

The main task of the immune system is to survey the organism using *detectors*, in search of malfunctioning cells and invaders such as bacteria or viruses. Every element that is recognizable by the immune system is called an *antigen*. The original body cells that are harmless to it are termed *self* (or self antigens) while the disease-causing elements are named *non-self* (or antigens). The immune system is able to sort them out.

The classification process into self/non-self is complex and produces a large number of randomly created detectors. A *negative selection mechanism* eliminates detec-

tors that match the cells in a protected environment where only self cells are assumed to be present. Non-eliminated detectors become naive detectors and die after some time. Furthermore, detectors that do match an antigen are quickly multiplied; this accelerates the response to further attacks. Also, the newly-produced detectors are not exact replicates of each other, with the mutation rate being an increasing function of detector-antigen *affinity* Dasgupta et al. (2003).

The elements of the natural immune system that are used in our model transformation testing procedure are mapped as follows.

- *Body*: the transformation mechanism to evaluate.
- *Self-Cells*: model transformation traces without faults.
- *Non-Self Cells* (Antigen): model transformation traces that present a high-risk of having faults.
- *Detector*: example of transformation trace that is very dissimilar to all "clean" traces (self-cells).
- *Affinity*: similarity between a detector and a model transformation trace to evaluate.

The next section presents the principle of our AIS-inspired approach.

3.2 Traceability-based approach for model transformation testing

We start by describing the overall process of the proposed procedure, illustrating it with the case of class diagram to relational schema transformation. Then, we detail our adaptation of the negative selection algorithm to the model transformation testing problem.

3.2.1 Overview

As showed in Fig. 5, our approach can be divided into three important components: the input/output of the testing process, the base of examples, and the main algorithm. We describe these components next.

3.2.1.1 Input/output The *Input* of our testing mechanism is a *test case* (TC). A TC includes a *source model*, its equivalent *target model* generated using the *transformation mechanism* to test, and the *traceability links* between the two models. More formally, a TC is a triple ⟨SMT, TMT, UT⟩, where SMT denotes the source model to test, TMT denotes the generated target model, and UT is a set of *test unit*s. A *test unit* defines the mappings to produce a particular element in the target model (thus, there exists one test unit per element). Since a model element (*e.g.*, Table) may contain sub-elements (*e.g.*, Columns), an element test unit also includes the mapping for the sub-elements.

The creation of a database schema from a UML class diagram, as described in the example of Fig. 6, is a TC where SMT is the class diagram and TMT is the relational schema generated by the transformation mechanism to evaluate. This TC contains five test units UT that correspond to the number of tables.

To ease manipulation of the test cases, the source and target models are described using a set of predicates that encode the included elements. The predicate types correspond to the different concepts of the source and target metamodels (class, attribute,

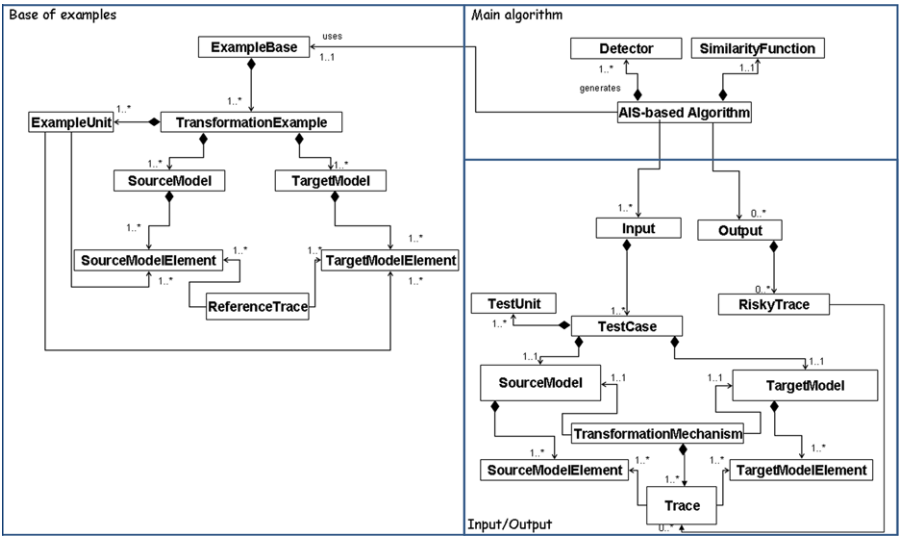


Fig. 5 Overall process of our approach

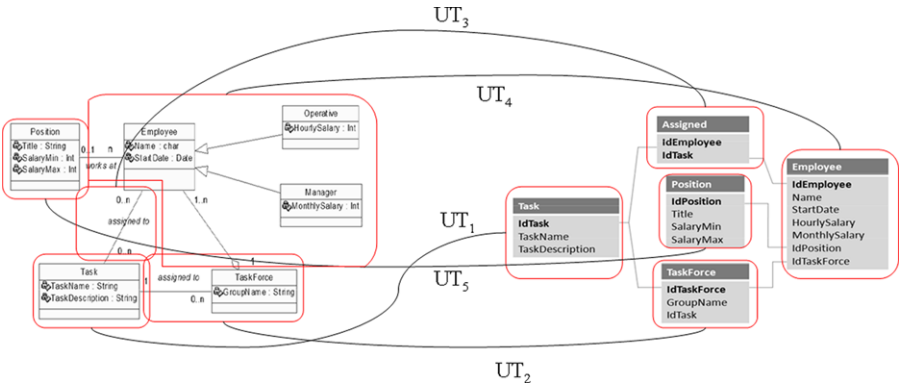


Fig. 6 Test case

etc. for class diagrams). The definition of their parameters has to be decided according to the properties and relationships of these concepts. For example, Class *Position* in Fig. 6 is described as follows:

Class(*Position*).
Attribute(*Title, Position, String*, _).
Attribute(*SalaryMin, Position, Int*, _).
Attribute(*SalaryMax, Position, Int*, _).

The first predicate indicates that *Position* is a class, and the second that *Title* is an attribute of that class with a non-unique value ("_" instead of "unique"). The two other predicates describe the remaining two attributes of class *Position*.

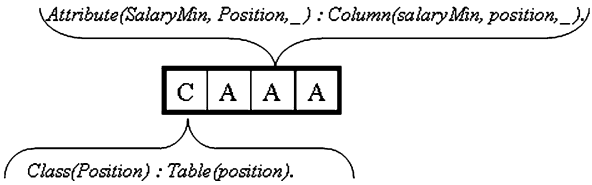


Fig. 7 Transformation unit coding

The traceability links relate the predicates in the source model to their equivalents in the target model. In our work, these links are automatically generated by adapting an existing metamodel, implemented in Kermeta (Falleri et al. 2006). An example traceability link that relates an association link to a column is as follows:

Association(*0, 1, , n, _ , Position, Employee*) : *Column*(*idPosition, employee, fk*).

The mappings are specified by the sign ":". For instance, the mapping between *Association*(*0, 1, , n, _ , Position, Employee*) and *Column*(*idPosition, employee, fk*) means that the association link between *Position* and *Employee* maps to the primary-foreign key (pfk) *idPosition* in table *Employee*.

The different test units are sets of these mappings. For example, UT_5 is described as follows:

Begin UT5
Class(*Position*) : *Table*(*position*).
Attribute(*SalaryMin, Position, Int, _*) : *Column*(*idPosition, position, pk*),
  Column(*salaryMin, position, _*).
Attribute(*SalaryMax, Position, Int, _*) : *Column*(*salaryMax, position, _*).
Attribute(*Title, Position,String, _*) : *Column*(*title, position, _*).
End UT5

Each test unit can be viewed as a sequence (string) composed of the following predicate types: class (C), attribute (A), method (M), generalization (G), aggregation (F), and association (S). For example, in Fig. 7, we present U_5 as the sequence of predicates CAAA, which corresponds to the transformation of a class with three attributes.

The sequence of predicates must follow the specified order of predicate types (C, A, M, G, F, S) to ease the comparison between predicate sequences. When several predicates of the same type exist, we order them according to their parameters. For example, if a class contains several attributes, the corresponding predicates are ordered by considering first the uniqueness, and then the types. In the example of class *Position*, as all the attributes are not unique, the predicates of SalaryMin and SalarityMax (*Int*) appear before the one of Title (*String*).

The *output* of our transformation mechanism is a set of test units containing *risky traces, i.e. traces* with potential transformation errors. Their risk score is determined by an AIS-based algorithm based on dissimilarity with the base of examples. These two components of our approach are described in the next subsections.

3.2.1.2 Base of examples The *base of examples* (BE) is composed of a set of *transformation examples* (TE). A transformation example is a mapping of *model elements* from a source model to a target model. Similar to a test case, a TE is essentially made

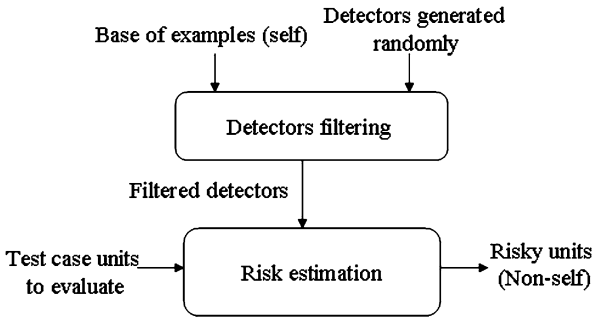


Fig. 8 AIS-based algorithm overview

of transformation *units*. Thus, it is a triple ⟨SME, TME, UE⟩, where SME denotes the *source-model* example, TME denotes the corresponding *target model*, and UE is a set of *example units* that relate model elements in SME to their equivalents in TME. The definition of a transformation example is similar to that of a test case, and the same predicates representation is used, as described above. However, the target model and the test units of TC are generated by the transformation mechanism whereas those of TE exist independently from the mechanism to test.

3.2.1.3 Main algorithm Figure 8 gives the overview of our *AIS-based algorithm*. The detection process has two main steps: *detector generation* and *risk estimation* (similarity function). Detectors are a set of units generated from those in the base of examples. These units define the reference for good transformation traces. The detector generation process is accomplished by using a heuristic search that simultaneously maximizes the difference between the detectors and the units, and between the detectors themselves. The same set of detectors could be used to evaluate different transformation mechanisms based on different formalisms, and it could be updated as the base of examples grows.

The second step of the detection process consists of comparing the test case units to all the detectors. A test case unit that shows similarity with a detector is considered to be risky; the higher the similarity, the riskier the test case unit is. Both detector generation and risk estimation steps use similarity scores. Before detailing the two steps, we describe the similarity function used in this work.

3.2.2 AIS-based algorithm

In this section, we start by explaining how to determine the similarity between two units. The resulting similarity score is used for detector generation and risk estimation as described later.

3.2.2.1 Similarity between transformation units To calculate the similarity between two units, we adapted to our context a dynamic programming algorithm used in bioinformatics to find similar regions between two sequences of DNA, RNA or proteins: the Needleman-Wunsch alignment technique (Carrillo and Lipman 1988). Figure 9 provides an illustration of the algorithm.

The Needleman-Wunsch global alignment algorithm recursively updates a matrix S of similarity scores for already-matched sub-sequences. The dimensions of S are

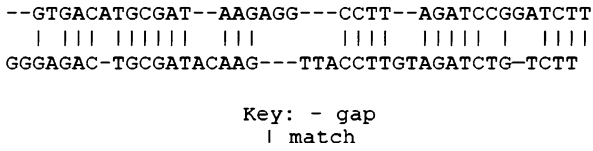


Fig. 9 Global alignment algorithm (Carrillo and Lipman 1988)

set by the lengths of the sequences to align. For two sequences $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$, S is of dimensions $n \times m$, and each of its element $s_{i,j}$ corresponds to the best alignment score for sub-sequences of a and b, a_i to b_j, of lengths i and j, respectively, considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented by "−") to improve sub-sequence matching. The number of introduced gaps corresponds to the number of times that the maximum value for each line in the matrix is not in the diagonal. The alignment algorithm depends on the predicate order in the sequences, hence the precise order that is described in the previous section.

The algorithm operates as follows: If a gap is inserted in a or b, it introduces a penalty of g in the similarity assessment (see below). In our adaptation, we choose the widely-used value of 1 for the penalty g (Carrillo and Lipman 1988). Then the algorithm attempts to match the predicates of each pair of sub-sequences a_i and b_j, by using a similarity function $sim_{i,j}$ to return the reward or cost of matching a_i to b_j, and the similarity score for a_i and b_j is updated. Formally, $s_{i,j}$ is defined as follows:

$$s_{i,j} = \text{Max} \begin{cases} s_{i-1,j} - g & \text{//insertgapfor } b_j \\ s_{i,j-1} - g & \text{//insertgapfor } a_i \\ s_{-1,j-1} + sim_{i,j} & \text{//match} \end{cases}$$

where $s_{i,0} = 0$ and $s_{0,j} = 0$.

Our adaptation of the Needleman-Wunsch algorithm is straightforward. We simply assign a value to g and a way to measure similarity between individual predicates to derive $sim_{i,j}$.

Since our model description uses predicate logic, we define a predicate-specific function to measure similarity. First, if the types differ, the similarity is 0. Since we manipulate sequences of predicates, and not strings, $sim_{i,j}$ behaves as a predicate-matching function PM_{ij} that measures the sought similarity in terms of the parameters of predicates p_k and q_k associated to the different characters of a_i and b_j. This similarity is the ratio of common parameters in both predicates. Formally, $sim_{i,j}$ is defined as follows:

$$sim_{ij} = \frac{PM_{ij}}{\max(|a_i|, |b_j|)}$$

where,

$$PIM_{ij} = \sum_{k=1}^{\max(|a_i|,|b_j|)} \frac{\text{number of equivalent predicates parametres}(p_k, q_k)}{\max(|p_k|, |q_k|)}$$

	C	A	A	A
C	**1**	0	0	0
A	1	**1.66**	1	1
A	1	**1**	**2.66**	1
G	1	1	**1**	**2.66**

$\mathbf{UC_5}$: C A A A -

$\mathbf{UE_{15}}$: C A A - G

Fig. 10 Best sequence alignment between U_5 and T_{15}

The similarity between sequences a and b is obtained by normalizing this absolute measure $s_{n,m}$ with respect to the maximum of their lengths n and m:

$$Sim(a, b) = \frac{s_{n,m}}{\max(n, m)}$$

To illustrate the use of the global alignment algorithm, consider the evaluation of test unit UC_5 described previously, based on its similarity to unit UE_{15} taken as an example unit (reference traces). UE_{15} is defined as follows:

Begin UE15
Class(*Teacher*) : *Table*(*Teacher*).
Attribute(*Level, Teacher, String, _*) : *Column*(*Level, Teacher, _*).
Attribute(*Name, Teacher, String, _*) : *Column*(*Name, Teacher, _*).
Generalization(*Person, Teacher*) : *Column*(*IDTeacher, Person, _*).
End UE15

Using the sequence coding described in Sect. 3.2.1.1, the predicate sequence for UC_5 is CAAA and the one for UE_{15} is CAAG. The alignment algorithm finds the best sequence alignment as shown in Fig. 10. There are three matched predicates between UC_5 and UE_{15}: one class (C), and two attributes (A). If we consider the second matched predicates *Attribute*(*Title, Position, String, _*) : *Column*(*idPosition, position, pk*), *Column*(*title, position, _*) from UC_5 and *Attribute*(*Level, Teacher, String, _*) : *Column*(*Level, Teacher, _*) from UE_{15}, their matching corresponds to element (2, 2) in the matrix. The attribute predicates (and their parameters) are similar, but not the transformation of these attributes since we do not have a primary key created in the second trace. The resulting similarity is consequently $(1 + 1 + 0)/3 = 0.66$, and this value is added to the maximum of elements (1, 2), (1, 1) and (2, 1) which is 1. Thus, the value of the matching is 1.66.

In our example, we have after normalization:

$$Sim(UC_5, UE_{15}) = s_{4,5}/\max(4, 4) = 2.66/4 = 0.65$$

3.2.2.2 Detectors generation This section describes how a set of detectors is produced starting from the base of examples. The generation is inspired by the work of Gonzalez and Dasgupta (2003), and follows a genetic algorithm (Goldberg 1989). The idea is to produce a set of detectors that best covers the possible deviations from

the base of examples. As the set of possible deviations can be very large, its coverage may require a huge number of detectors, which is infeasible in practice. For example, pure random generation was shown to be infeasible in Gonzalez and Dasgupta (2003) for performance reasons.

We therefore consider detector generation as a search problem. A generation algorithm should seek to optimize the following two objectives:

- Maximize the generality of the detector to cover the non-self by minimizing the similarity with the self.
- Minimize the overlap (similarity) between detectors.

These two objectives define the cost function that evaluates the quality of a solution and, then, guides the search. The cost of a solution D (set of detectors) is evaluated as the average cost of the included detectors. We derive the cost of a detector $\mathrm{d_i}$ as an average between the scores of the lack of generality and the overlap, respectively. Formally, we have:

$$\cos t(d_i) = \frac{LG(d_i) + O(d_i)}{2}$$

The lack of generality is measured by the matching score $LG(d_i)$ between the predicate sequence of a detector d_i and those of all units UE_j in the base of examples (BE). It is defined as the average value of the alignment scores $Sim(d_i, \mathrm{UE}_j)$ between d_i and units UE_j in BE:

$$LG_{d_i} = \frac{\sum_{\mathrm{UE}_j \in \mathrm{BE}} Sim(d_i, \mathrm{UE}_j)}{|\mathrm{BE}|}$$

Similarly, the overlap O_i is measured by the average value of the individual $Sim(d_i, d_j)$ between detector d_i and all the other detectors d_j in solution D:

$$O_i = 1 - \frac{\sum_{d_j, j \neq i} Sim(d_i, d_j)}{|D|}$$

The preceding cost function is used in our genetic-based search algorithm. Genetic algorithms (GA) implement the principle of natural selection (Goldberg 1989). Roughly speaking, a GA is an iterative procedure that generates a population of individuals from the previous generation using two operators, crossover and mutation. Individuals having a high fitness have higher chances to reproduce themselves (by crossover), which improves the global quality of the population. To avoid falling in local optima, mutation is used to randomly change individuals. Individuals are represented by chromosomes containing a set of genes.

For the particular case of detector generation, we use the predicate sequences as chromosomes, with each predicate representing a gene. We start by randomly generating an initial population of detectors. The size of this population will be discussed in Sect. 4. It is maintained constant during the evolution. The fitness of each detector is evaluated by the inverse function of cost.

The fitness determines the probability of being selected for crossover. We implement the selection process using a wheel-selection strategy (Goldberg 1989). In fact,

for each crossover, two detectors are selected by applying the wheel selection twice. Even though detectors are selected, crossover only happens with a certain probability. Sometimes, based on a set probability, no crossover occurs and the parents are directly copied to the new population.

The crossover operator allows creating two offspring o_1 and o_2 from the two selected parents p_1 and p_2. We used the 1-point crossover procedure, defined as follows:

- A random position k, is selected in the predicate sequences.
- The first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.
- The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1.

For instance, if $k = 2$, $p_1 = \boldsymbol{CC}$AAGS and $p_2 =$ CA$\boldsymbol{AAS}$, then $o_1 = \boldsymbol{CCAAS}$ and $o_2 =$ CAAAGS.

The mutation operator consists of randomly changing the traceability links associated to some characters. For example, we change a trace that transforms a class to table by another one that transforms an association link to a table.

3.2.2.3 Risk estimation The second step for detecting a potential transformation error is risk assessment. Since the test units are also represented by predicate sequences, each sequence is compared to the detectors obtained in the previous step by using the alignment algorithm. The risk for potential errors associated to test unit UC_i is defined as the average value of the alignment scores $Sim(\mathrm{UC}_i, d_j)$, obtained by comparing UC_i to respectively all the detectors of a set D. Formally,

$$risk_{\mathrm{UC}_i} = \frac{\sum_{d_j \in D} Sim(\mathrm{UC}_i, d_j)}{|D|}$$

By using the previous definition, the test units can be ranked according to their risks of containing potential transformation errors.

4 Evaluation

To evaluate our approach, we conducted an experiment with industrial data. We start this section by presenting the two kinds of transformation errors we considered in this study. Then we describe our experimental setting. Finally, we report and discuss the obtained results.

In addition to our oracle performance, we evaluate the impact of the example base size on transformation error detection quality. Furthermore, we show how a human tester can easily validate the detected faults using our visualization tool. Finally, we discuss the benefits and limitations of the proposed approach to model transformation testing.

4.1 Considered transformation errors

We considered errors belonging to the two following categories:

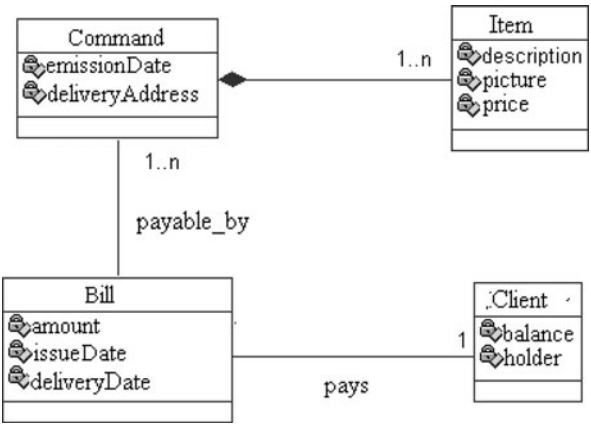


Fig. 11 Transformation input: class diagram

4.1.1 Metamodel coverage

This type of error occurs when the transformation is defined without a complete coverage of the metamodel elements. This leads to the problem that parts of some input models cannot be transformed. To illustrate metamodel coverage errors, consider the class diagram metamodel presented in Fig. 2. Figure 11 shows a class diagram instance that conforms to this metamodel. Suppose that the transformation mechanism does not include rules transforming the metamodel element *Association*. When executing the transformation mechanism, we have these two incomplete traces:

Association(payable_by, Command, Bill, 1..n, _) : _
Association(pays, Client, Bill, 1, _) : _

However, in our base of examples all association links have corresponding transformations. Thus, one of the generated detectors has an example of this faulty trace. The result is that this trace will be considered to be risky.

4.1.2 Transformation logic errors

These errors happen when the transformation, or part of it, is not implemented correctly. This can lead to models that do not conform to the target metamodel. This includes constraints violation. For example, an important constraint in relational models is that each table should have a primary key. Consider a transformation with a rule that maps attributes to columns and another rule that maps unique attributes to primary keys. If we consider class Bill in Fig. 11, this does not contain a unique attribute. We end-up then with a table without a primary key:

Class(*Bill*) : *Table*(*Bill*).
Attribute(*Amount, Bill, _*) : *Column*(*Amount, Bill, _*).
Attribute(*IssueDate, Bill, _*) : *Column*(*IssueDate, Bill, _*).
Attribute(*DeliveryDate, Bill, _*) : *Column*(*DeliveryDate, Bill, _*).

However, in our base of examples, all tables have primary keys. Thus, one of the generated detectors has an example of this faulty trace. Thus this trace will be considered to be risky.

4.2 Experimental setting

We used 12 examples of CD-to-RS transformations, provided by an industrial partner acting in the beverage industry, to build an example base $EB = \{\langle SME_i, TME_i, UE_i\rangle \mid 1 \leq i \leq 12\}$. This company decided to migrate all its existing applications to distributed ones (intra-web) with a common database. As a result, different database schemas had to be generated from the existing applications written in object-oriented code. To this end, the development and maintenance department started by reverse-engineering these projects to class diagrams. Then they transformed the obtained diagrams to relational schema using a commercial tool. In a third step, they completed and corrected the schemas manually.

The projects we obtained from the company are related to three application domains: product management, marketing, and fleet management including geolocalization. For each transformation example, we had the class diagram and the manually corrected relational schema. After receiving the examples, we inspected them manually to ensure that they were free of transformation errors.

As Table 1 shows, the size of class diagrams varies from 28 to 92 elements, with an average of 58. Altogether, the 12 examples defined 193 test units corresponding to the number of tables in the 12 schemas (Sect. 3).

We selected as transformation mechanism to test, MTIP, a tool written in Kermeta (Bézivin et al. 2005). Kermeta implements a state-of-the-art declarative model transformation language suitable for Model-Driven Development (MDD) and data transformation. It is implemented as an Eclipse plugin that leverages the Eclipse Modelling Framework (EMF) to handle models based on MOF, UML2, and XML Schema. The transformation traces are collected automatically by adapting an existing metamodel in Kermeta (Falleri et al. 2006).

We used a 12-fold cross validation procedure. For each fold, we manually introduced different transformation errors into the transformation mechanism (rules) and subsequently transformed one of the 12 examples (test case $\langle SMT_k, TMT_k, UT_k\rangle$). The 11 remaining ones formed the base of examples for the testing ($\{\langle SME_j, TME_j, UE_j\rangle \mid j \neq k\}$). Thus, each fold concerned one different example. The test units were ranked by order of risk, and those that were reported to have a risk higher than 0.75 were checked for correctness. The correctness of our testing method was based on precision and recall capabilities assessments. These were defined as follows:

$$\text{Precison} = \frac{\text{number of true positive transformation errors}}{\text{total number of detected transformation errors}}$$

$$\text{Recall} = \frac{\text{number of true positive transformation errors}}{\text{total number of actual transformation errors}}$$

Are considered as true positive all units that have a risk higher than 0.75 and that were actual errors. For our experiment, we randomly generated 50 detectors (about a quarter of the number of existing units in the base of examples) with a maximum size of 15 predicates (Sect. 3).

4.3 Transformation errors detection results

As showed in Table 1, the riskiest test units detected by our approach contained transformation errors in all folds of the validation procedure. The measured average precision was 91%, with most errors detected with at least 82% precision. The measured average recall of 98% was greater, indicating that nearly all the errors were detected. For over half the total number of folds, 100% recall was obtained, indicating the detection of all expected errors. Furthermore, the precision and recall scores were not correlated with the size of the source model.

We also investigated the types of transformation errors that were identified. As mentioned previously, the possible error sources were during specification of the model transformation mechanism: (i) the metamodels; (ii) the transformation logic (rules). Table 2 shows that, for fold SM5, chosen because it represent the average size and precision/recall scores, our affinity function (risk score) can be a good estimator for detecting transformation errors. In fact, the units located at the top of the list are all true positive, and the unique incorrect (unexpected) detected error is located last. Furthermore, the units containing two kinds of errors are typically detected with higher risk values (UC_{68} and UC_{69}). The same observations can be drawn for all folds, showing that the used risk score offers an effective and efficient manner for the tester to validate the detected errors.

An important consideration is the impact of the example base size on transformation error detection quality. Drawn for SM5, the results of Fig. 12 show that our approach had good precision in situations where only few examples were available. As the results shows, the precision score seems to follow an exponential curve: it rapidly grows to acceptable values and then slows down. First, it improved from 22% to 75% as the example base size increased from 1 to 6 examples. Then, it only grew by an additional 18% as the size went from 6 to 11 examples.

Table 1 12-fold cross validation

Source Model	Number of elements	Number of transformation errors introduced manually	Precision	Recall
SM1	72	13	82%	93%
SM2	83	14	93%	94%
SM3	49	11	92%	100%
SM4	53	16	88%	100%
SM5	38	9	90%	100%
SM6	47	12	100%	100%
SM7	78	16	84%	95%
SM8	34	8	100%	100%
SM9	92	14	82%	93%
SM10	28	9	100%	100%
SM11	59	13	93%	100%
SM12	63	15	94%	100%
Average	58	12	91%	98%

Table 2 Errors detected in SM5

Test units with numbers	Risk	Met-model error	Transformation logic error
UC_{68}	0.93	X	X
UC_{69}	0.91	X	X
UC_{70}	0.96		X
UC_{71}	0.91	X	
UC_{72}	0.89		X
UC_{73}	0.94		X
UC_{74}	0.96		X
UC_{75}	0.89		X
UC_{76}	0.77		

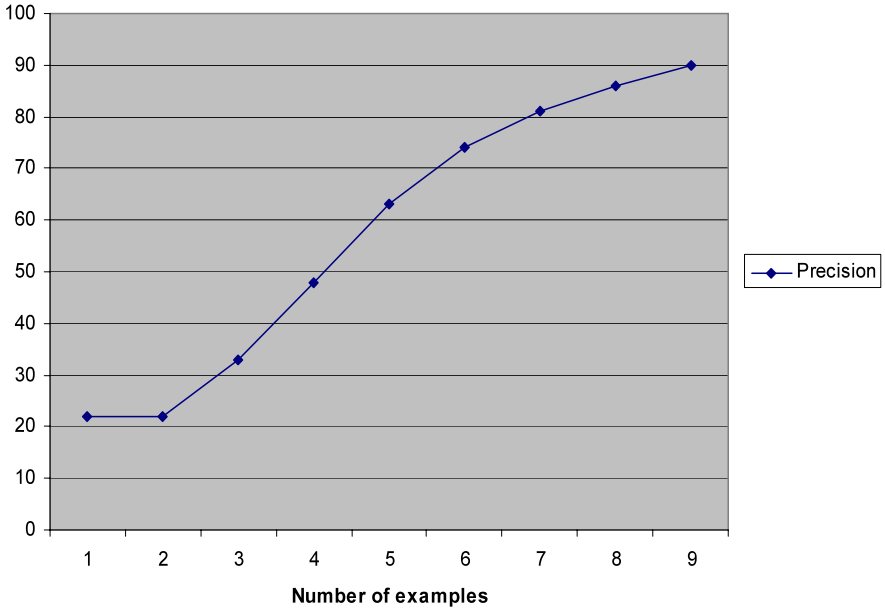


Fig. 12 Example-size variation

We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 1 GB of RAM). The execution time is shown in Fig. 13. As suggested by the curve shape, the time increased linearly with the number of elements. Thus, our approach appears to be scalable from the performance standpoint. Only a few seconds were needed to test the transformation mechanism to evaluate. This execution time does not include that for detector generation since the detectors are only generated once and can serve to evaluate several transformation mechanisms afterwards. This feature is a major advantage of using detectors versus comparing the test units to all units in the base of examples, which can be infeasible in time when the number of units is very large (Forrest et al. 1994).

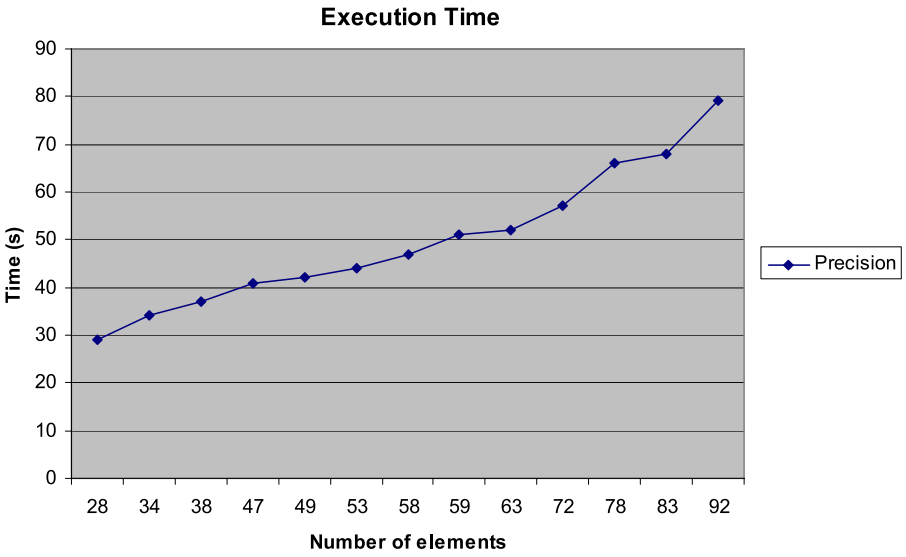


Fig. 13 Execution time

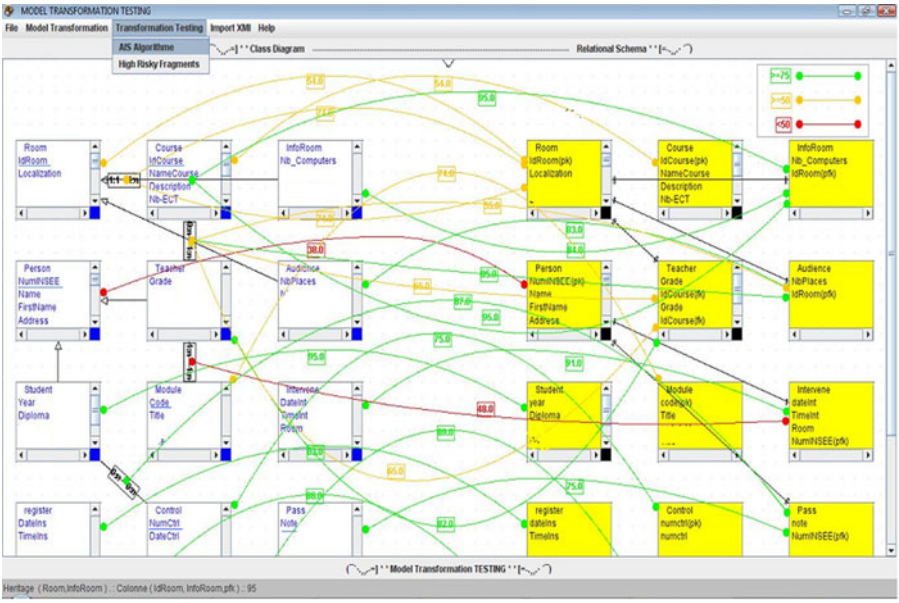
As showed in Fig. 14, a human tester can analyze the detected risky test units with a graphical visualization tool. We developed a custom utility that displays the risky test units with different colors related to the obtained risk score, and with the "clean" traces colored in green. The human tester can validate, for example, only units that present a potential risk that are colored in red. Furthermore, the traces help the tester understand the origin of an error. To allow dealing with the transformation of large models, the traces can be viewed at different levels of granularity. For example, the tester can only show the links between model elements, or between sub-elements. Furthermore, he can only visualize the traces having potential risk (Fig. 14(b)).

5 Discussion

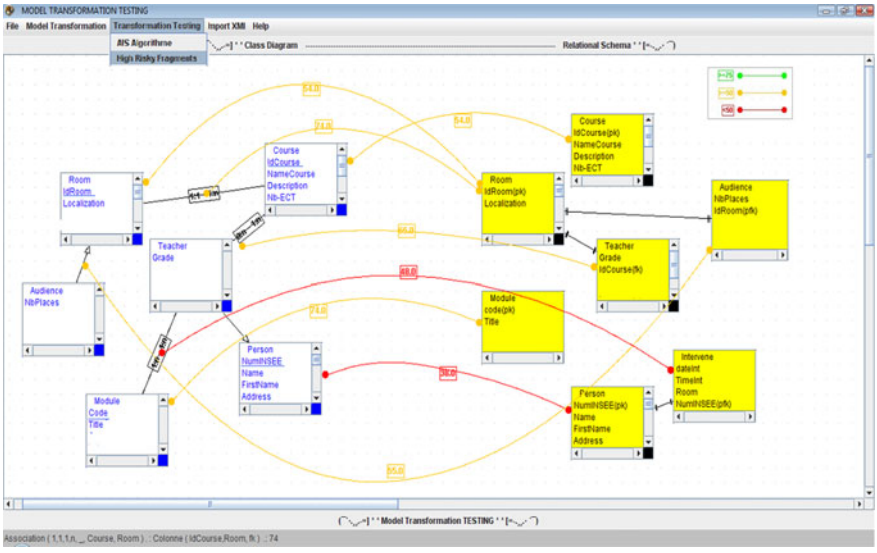
In this section, we discuss several issues concerning the detection of transformation errors. Especially, we describe some advantages and limitations related to our approach.

In our approach, there is no need to define an expected model for each test-case or to define pre- and post-conditions as oracles; we only use similarity to good transformation examples. The approach can be seen to propose a way to detect and order transformation errors by importance, using a risk score. Moreover, our oracle definition is independent from the transformation mechanism to evaluate or the source/target formalisms, and it helps the tester understand the origin of errors by visualizing the traceability links with different colors.

Still, our approach has issues that need to be addressed. First, its performance depends on the availability of good transformation examples, which could be difficult to collect. Second, the assumption that the base of examples does not contain transformation errors may be too strong, and not easily verified in practice. On the positive



(a)



(b)

Fig. 14 Interactive transformation errors detection using our tool: (**a**) all traces and (**b**) only risky traces

side, our results show that a small number of examples may be sufficient to obtain good testing results. This alleviates the two previous limitations, and may even offer a solution because the number of needed examples is small. It consists of generating a few test cases using the transformation mechanism to test and, then, of manually de-

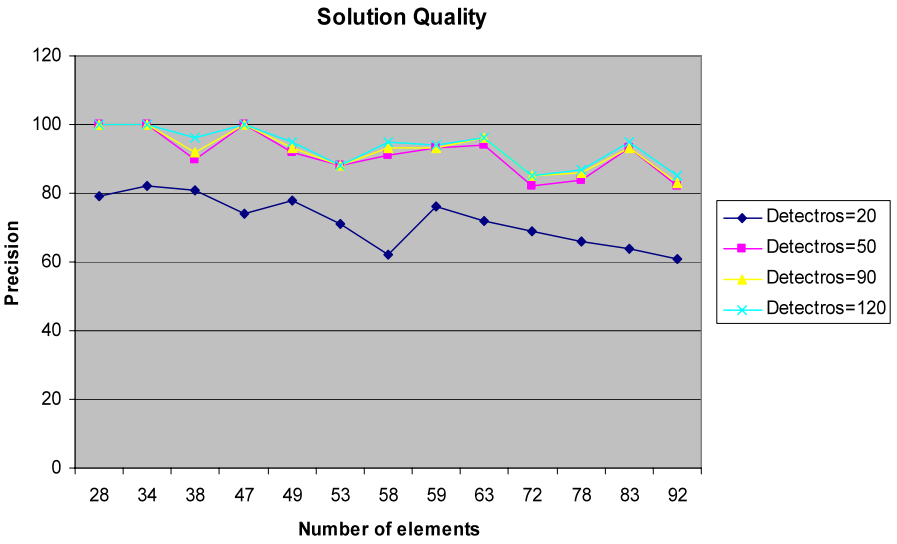


Fig. 15 Detectors variation vs. solution quality (precision)

tecting and correcting potential transformation errors. The resulting cases then form the base of examples.

To reduce the number of necessary examples, these examples are decomposed into units. However, the definition of units sometimes depends on the source/target metamodels of the test-case. Thus, our proposed methodology could sometimes be dependent on the source/target metamodels, but this potential dependency is acceptable in comparison to the state of the art that will be discussed in the next section.

Another potentially important aspect of our detection technique is the generation of a sufficient number of detectors. In our experiments, we generated 50 detectors, which corresponds to a quarter of the units present in the base of examples. We evaluated the precision of our approach when varying the number N_d of detectors, with $N_d = \{20, 50, 90, 120\}$. Our results, shown in Fig. 15, reveal that precision stops improving when the number of detectors is higher than the quarter of the total number of units in the base of examples. In addition, Fig. 16 shows the execution time necessary to generate different numbers of detectors. We observe that this time appears to vary linearly with respect to the diagram sizes for all the number of detectors. In conclusion, our experimentation results indicate that a reasonable number of detectors (quarter of the transformation units in the base of examples), generated in less than one minute, is sufficient to obtain good detection results.

An additional issue is the selection of interesting detectors since the detection results might vary depending on which detectors are used, and ours were randomly generated (though guided by a meta-heuristic). To ensure that our results are relatively stable, we compared the results of multiple executions for detector generation. We found that approximately the same transformation errors are found after every execution and the differences only exist for low-risk test units. We therefore believe

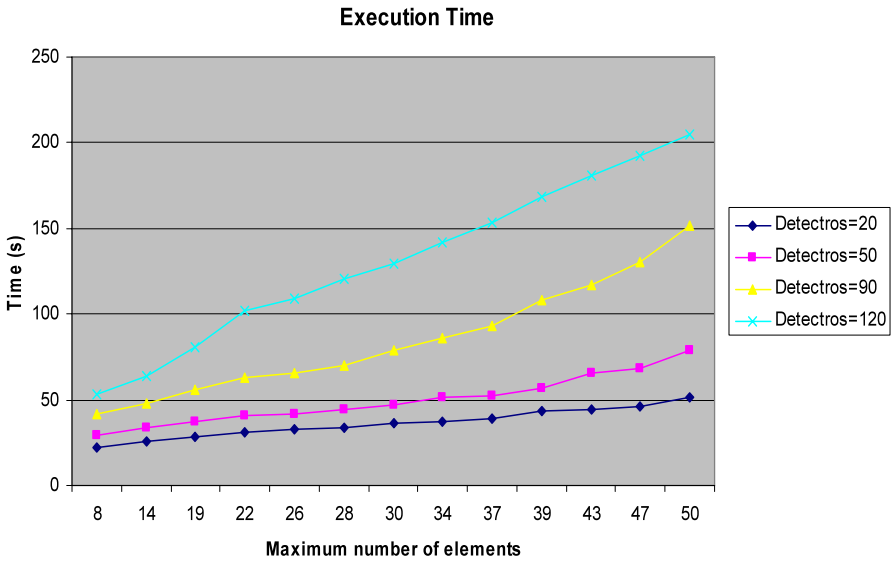


Fig. 16 Detectors variation vs. execution time

that our technique is stable with regard to detector choice since the result variability only relates to the least risky classes.

6 Related work

The work proposed in this paper crosscuts many research topics. In the remainder of this section, we present representative contributions in five of these topics: test-case generation, oracle function definition, search-based testing, by-example model transformation, and traceability and transformation.

6.1 Test case generation

Fleurey et al. (2004, 2008) and Steel and Lawley (2004) discuss the reasons why testing model transformation is distinct from testing traditional implementations: the input data are models that are complex in comparison to simple-type data. Both papers describe how to generate test data in MDA by adapting existing techniques, including functional criteria (Fleurey et al. 2004) and bacteriologic approaches (Baudry et al. 2002). Lin et al. (2005) propose a testing framework for model transformation, built on their modeling tools and transformation engine, that offers a support tool for test case construction, test execution and test comparison; but the test models are manually developed in their work. As our work does not address test case generation, it can be integrated with the previous approaches without the need to define the expected model for each test case.

One of the most widely-used techniques for test-case generation is mutation analysis. Mutation analysis is a testing technique that aims to evaluate the efficiency of a

test set. Mutation analysis consists of creating a set of faulty versions, or mutants, of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. Mottu et al. (2006) have adapted this technique to evaluate the quality of test cases. They introduce some modifications in the transformation rules (program-mutant). Then, using the same test cases as input, an oracle function compares the results (target models). If all the results are the same, we can assume that the input cases were not sufficient to cover all the transformation possibilities. In our work, the goal is not to evaluate the quality of a data set but to propose a generic oracle function to detect transformation errors. Our oracle function compares between some potential errors (detectors) and transformation traces to evaluate. However, in mutation analysis, the oracle function compares between two target models, one generated by the original mechanism (rules) and another after modifying the rules. In addition, our technique does not create program variations (rules modifications) but traces variation that differs from good ones. We modified the transformation mechanism to introduce errors artificially only to validate our approach. Finally, the mutation analysis technique needs to define an expected model for each test case in order to compare it with another target model obtained from the same test case after modifying the rules (mutant).

Some other approaches are specific to test case generation for graph-transformation mechanism. Küster (2006), addresses the problem of model transformation validation in a way that is very specific to graph transformation. He focuses on the verification of transformation rules with respect to termination and confluence. His approach aims at ensuring that a graph transformation will always produce a unique result. Küster's work is concerned with the verification of transformation properties rather than the validation (testing) of their correctness. Darabos et al. (2006) investigate the testing of graph transformations. They consider graph transformation rules as the transformation specification and propose to generate test data from this specification. Their technique focuses on testing the pattern matching activity that is considered the most critical of a graph transformation process. They propose several faulty models that can occur when performing the pattern matching as well as a test-case generation technique that targets those particular faults. Compared to our approach, Darabos' work is specific to graph-based transformation testing. Baudry et al. (2006) propose a technique for generating test cases for code generators. The criterion they propose is based on the coverage of graph transformation rules. The generated test cases consider both individual rules and rule interactions. Sampath et al. (2007) propose a similar method for the verification of model processing tools such as simulators and code-generators. They use a method that generates test-cases for model processors starting from a metamodel. This method, like the previous contributions, is concerned with test-case generation which is not the goal of our contribution.

6.2 Oracle function definition

Mottu et al. (2008) describe six different oracle functions to evaluate the correctness of an output model. These six functions can be classified in the three categories discussed in Sect. 2. Thus, they are completely different from our proposal.

In Brottier et al. (2006), the authors suggest to manually determine the expected outcome of the transformation and compare it with the actual outcome of the transformation by using a simple graph-comparison algorithm, since the compared models conform to the same metamodel. While this makes model transformation testing feasible, our view is that manually constructing the expected outcome is not an efficient and scalable approach.

Varró and Pataricza (2003) have developed a formal framework for describing model transformation. The formal framework relies on models represented as typed attributed graphs. Concerning the transformation correctness, they have developed an approach based on planner algorithms to prove the syntactic correctness of a transformation. Syntactic correctness refers to the property that the result of a transformation corresponds to a certain previously specified syntax, and can be achieved by specifying a graph grammar for both the source and target languages.

More generally, when many test models are necessary, writing an oracle for each test case is time consuming and error prone. Generic oracles are more interesting since they are written only once, and could be used with all the test cases. Another limitation of the existing approaches is that they consider a particular model transformation technique and use its specificities to validate the corresponding transformation mechanisms. This has the advantage of having specific validations but make these approaches difficult to adapt to other transformation techniques. For our approach, the oracle function is generic and independent from the transformation techniques. Moreover, we do not have an explicit specification of the transformation mechanism to evaluate (properties, constraints, or contracts).

6.3 Search-based testing

Our approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) (Harman 2007). SBSE uses search-based approaches to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. Search-based techniques are have been used for problems in software testing (Baresel et al. 2002, 2004; McMinn 2004). Especially, genetic algorithms have been extensively used for test data generation. The general idea behind the proposed approaches is that possible test suites define a search space and that a test adequacy criterion is coded as a fitness function. This later guides the selection of the best test suite in this space. A wide variety of testing problems have been targeted using search techniques, including structural, functional and non functional testing, safety testing, mutation testing, integration testing and exception testing (McMinn 2004). In our work, we use a genetic algorithm with a completely different perspective. Indeed, the idea is to generate artificial situations that are different from known good-transformation traces. Then, these artificial traces are used not as test cases but as oracle functions.

To our knowledge, there exist very few works in software engineering that use an AIS techniques. The closest one to our work proposes a software defect prediction model by means of an artificial immune recognition system (AIRS) along with correlation-based feature selection (CFS) (Catal 2007). In our work, in addition to target a different problem, we do not use AIRS, but the negative selection algorithm.

6.4 By example model transformation

The AIS approach proposed in this paper is based on using examples. Various such by-example approaches have been described in the literature (Varro and Balogh 2007; Kessentini et al. 2008, 2010; Wimmer et al. 2007; Sun et al. 2009; France and Rumpe 2007). The most similar one is Model Transformation By Example (MTBE), which was proposed in Kessentini et al. (2008), France and Rumpe (2007). Varro and Balogh (2007) propose a semi-automated process for MTBE using Inductive Logic Programming (ILP). The principle of their approach is to derive transformation rules semi-automatically from an initial prototypical set of interrelated source and target models. In a previous work (Kessentini et al. 2008, 2010; France and Rumpe 2007) we proposed MOTOE (MOdel Transformation as Optimization by Example), a novel approach to automate model transformation using heuristic-based search. MOTOE uses a set of transformation examples to derive a target model from a source model. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and a quality associated to each one depending on its conformance with the examples at hand. A similar approach to MTBE, called Model Transformation By Demonstration (MTBD), was proposed in Sun et al. (2009). Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, users are asked to demonstrate how the model transformation should be done, through direct editing (e.g. add, delete, connect, update) of the source model so as to simulate the transformation process.

In conclusion, when compared to existing by-example approaches, our proposal appears to present the first contribution that uses examples for model transformation testing.

Despite these efforts in MTBE work, and considering the nature of the algorithms that are used, there is no evidence that a solid base of examples can generate target models without errors.

6.5 Traceability and transformation

In our approach, the definition of transformation examples is based on traceability (Varró and Pataricza 2003). Traceability usually allows tracing artifacts within a set of chained operations, where the operations may be performed manually (e.g. crafting a software design for a set of software requirements) or with automated assistance (e.g., generating code from a set of abstract descriptions). Most work on traceability in MDE uses it for detecting model inconsistency and fault localization in transformations. In our proposal, the goal is not to generate traces but to use clean trace information as input in order to detect transformation errors.

7 Summary

In this article, we presented a new oracle function definition for model transformation testing that does not need to define the expected model for each test case. The technique is based on the metaphor of a biological immune system using negative selection. We propose an oracle function that compares between the targeted test cases

and a base of examples containing good quality transformation traces and assigns a risk level, which will define the oracle function to the former based on the dissimilarity between the two. Furthermore, we use a custom tool to help the human tester visualize the detected risky fragments in test cases, using different colors related to the obtained risk scores.

We illustrated our approach with a transformation mechanism for UML class diagrams to relational schemas. In this context, we conducted a validation with real industrial models. The experiment results clearly indicated that the detected risky fragments (transformation errors) are comparable to those detected by a human tester (precision and recall of more than 90%).

Our method also suffers from some limitations as discussed in Sect. 5. In particular, our oracle function may require considerable effort to find and collect transformation examples.

Future work should validate our approach with more complex transformation mechanisms like sequence diagram to colored Petri nets in order to conclude about the general applicability of our methodology. Also, in this paper, we only looked at the first step of immune systems: the detection of risk. The second step is problem correction. The colonial selection algorithm (Sun et al. 2009) could be adapted for finding the best immune response, i.e. the one corresponding to the optimal sequence of corrections to apply for correcting errors by automatically regenerating some rules from examples.

References

Alanen, M., Porres, I.: Difference and union of models. In: UML'03, USA (2003)

Baresel, A., Sthamer, H., Schmidt, M.: Fitness function design to improve evolutionary structural testing. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1329–1336 (2002)

Baresel, A., Binkley, D.W., Harman, M., Korel, B.: Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In: International Symposium on Software Testing and Analysis (ISSTA 2004), Omni Parker House Hotel, Boston, MA, July 2004, pp. 108–118 (2004). Appears in Softw. Eng. Notes **29**(4)

Baudry, B., Fleurey, F., Jezequel, J.-M., Traon, Y.L.: Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. In: ASE (2002)

Baudry, B., Dinh-Trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Traon, Y.L.: Model transformation testing challenges. In: IMDT Workshop (2006)

Bezivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: OOPSLA/GPCE 2004 Workshop (2004)

Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: MTIP workshop. Available from: http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf (2005)

Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proceedings of SSRE (2006)

Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proceedings of Workshop OCL and MDE (2004)

Carrillo, H., Lipman, D.: The multiple sequence alignment problem in biology. SIAM J. Appl. Math. **48**(5), 1072–1082 (1988)

Catal, C., Diri, B.: Software defect prediction using artificial immune recognition system. In: Proceedings of IASTED international Multi-Conference, pp. 285–290 (2007)

Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOSPLA 2003, Anaheim, USA (2003)

Darabos, A., Pataricza, A., Varro, D.: Towards testing the implementation of graph transformations. In: Proceedings of GT-VMT Workshop Associated to ETAPS'06, Vienna, Austria, pp. 69–80 (2006)
Dasgupta, D., Ji, Z., Gonzalez, F.: Artificial immune system (ais) research in the last five years. In: IEEE Congress on Evolutionary Computation (1), pp. 123–130. IEEE, New York (2003)
Falleri, J.-R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in Kermeta. In: Proceedings of the European Conference on MDA Traceability Workshop, Bilbao, Spain (2006)
Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: 15th IEEE International Symposium on Software Reliability Engineering (2004)
Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.: Qualifying input test data for model transformations. In: Software and Systems Modeling (2008)
Forrest, S., Perelson, A.S., Allen, L., Kuri, R.C.: Self nonself discrimination in a computer. In: Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy (1994)
France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: ICSE 2007: Future of Software Engineering (2007)
Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
Gonzalez, F., Dasgupta, D.: Anomaly detection using real-valued negative selection. Genet. Program. Evolv. Mach. **4**(4), 383–403 (2003)
Harman, M.: The current state and future of search based software engineering. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), 20–26 May, Minneapolis, USA (2007)
Kessentini, M., Sahraoui, H., Boukadoum, M.: Model transformation as an optimization problem. In: Proc. MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Berlin (2008)
Kessentini, M., Sahraoui, H., Boukadoum, M.: Search-based model transformation by example. J. Softw. Syst. Model. (2010). doi:10.1007/s10270-010-0175-7
Khuller, S., Raghavachari, B.: Graph and network algorithms. ACM Comput. Surv. **28**(1), 43–45 (1999)
Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model comparison: a foundation for model composition and model transformation testing. In: Proc. GaMMa (2006)
Kuby, J., Kindt, T.J., Osborne, B.A., Goldsby, R.A.: Immunology, 3rd edn. Freeman, New York (1997)
Küster, J.M.: Definition and validation of model transformations. Softw. Syst. Model. **5**(3), 233–259 (2006)
Kuster, J., Abd-El-Razik, M.: Validation of model transformations—first experiences using a white box approach. In: MoDeVa'06 (2006)
Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Model-Driven Software Development. Springer, Berlin (2005)
McMinn, P.: Search-based software test *data* generation: A survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
Mottu, J.-M., Baudry, B., LeTraon, Y.: Mutation analysis testing for model transformations. In: Proceedings of ECMDA'06 (European Conference on Model Driven Architecture), Bilbao, Spain (2006)
Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: Oracle issue. In: Proc. of ICST08 (2008)
Sampath, P., Rajeev, A.C., Ramesh, S., Shashidhar, K.C.: Testing model-processing tools for embedded systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 203–214 (2007)
Steel, J., Lawley, M.: Model-based test driven development of the Tefkat model- transformation engine. In: ISSRE'04, pp. 151–160. IEEE, New York (2004)
Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: MoDELS09 (2009)
Varro, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: ACM Symposium (SAC 2007) (2007)
Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop, Technical Report, pp. 63–78. Technische Universität, München (2003)
Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: HICSS-40 Hawaii International Conference on System Sciences (2007)