e-Informatica Software Engineering Journal, Volume 15, Issue 1, 2021, pages: 65–84, DOI 10.37190/e-Inf210104

A Framework for the Regression Testing of Model-to-Model Transformations

Issam Al-Azzoni*, Saqib Iqbal*

*Department of Software Engineering and Computer Science, Al Ain University, Al Ain, UAE

issam.alazzoni@aau.ac.ae, saqib.iqbal@aau.ac.ae

Abstract

Background: Model transformations play a key role in Model-Driven Engineering (MDE). Testing model transformation is an important activity to ensure the quality and correctness of the generated models. However, during the evolution and maintenance of these model transformation programs, frequently testing them by running a large number of test cases can be costly. Regression test selection is a form of testing, which selects tests from an existing test suite to test a modified program.

Aim: The aim of the paper is to present a test selection approach for the regression testing of model transformations. The selected test case suite should be smaller in size than the full test suite, thereby reducing the testing overhead, while at the same time the fault detection capability of the full test suite should not be compromised.

Method: approach is based on the use of a traceability mapping of test cases with their corresponding rules to select the affected test items. The approach is complemented with a tool that automates the proposed process.

Results: Our experiments show that the proposed approach succeeds in reducing the size of the selected test case suite, and hence its execution time, while not compromising the fault detection capability of the full test suite.

Conclusion: The experimental results confirm that our regression test selection approach is cost-effective compared to a retest strategy.

Keywords: Model Transformation, Regression Testing, MDE

1. Introduction

Model-Driven Engineering (MDE) refers to representing, designing, and developing a system in the form of models [1]. A model is a core artifact of MDE, which refers to an abstract representation of data and behavior of a system. Model transformations are currently used in a variety of industrial projects [2] and ensuring their correctness is important [3, 4]. Model transformation programs are frequently changed during the evolution and maintenance phases of their life cycle. Several techniques for testing model transformations have been proposed in the literature [5, 6]. However, these techniques generally require executing a large number of test cases to ensure the desired coverage criteria. This can be time-consuming and may require days or even weeks to complete.

During the regression testing of model transformations, a test suite is generally available for reuse [7]. However, a retest-all approach in which all tests are rerun may consume excessive time and resources. In contrast, regression test selection techniques aim to reduce the time required to retest a modified program by selecting a smaller subset of the existing test suite [7, 8].

There have been many regression test selection techniques proposed for conventional software written in regular programming lan-

Submitted: 15 August 2020; Revised: 31 January 2020; Accepted: 09 February 2021; Available online: 29 March 2021

guages [8]. However, only a few exist for testing model transformation programs. For example, the work of Alkhazi et al. [3] proposes an approach for test case selection for model transformations based on a multiobjective search. Another work by Shelburg et al. [9] examined the issue of determining which test cases become invalid when changes occur in a model transformation program. The work of Troya et al. [4] presents an approach for fault localization for model transformations.

In this paper, we present a framework for the regression testing of model transformations, which is based on the use of a metamodel that links test cases to their corresponding test items and test artifacts. To the best of our knowledge, this is the first paper specifically proposing a framework for the regression testing of model transformation programs based on traceability models. We present a tool that can automatically create the traceability model, given the source meta-model and a set of input test models. The tool can also be queried to obtain the set of selected test cases for a changed rule.

Regression testing approaches can be classified into three main categories [8]: test suite minimization, test case selection, and test case prioritization. Test suite minimization approaches aim to reduce the size of a test suite by permanently eliminating redundant test cases from the test suite. Test case selection approaches aim to select a subset of test cases that will be used to test the changed parts of the software. Finally, test case prioritization approaches attempt to order test cases in a way that maximizes desirable properties, such as early fault detection. In the context of model transformation testing, our proposed approach can be classified into the test case selection category.

The main contributions of this paper are summarized as follows:

1. We present a test case metamodel that can be exploited in the regression testing of model transformations.
2. We present a tool that can automatically build the required traceability models and select test cases based on the names of changed rules.
3. We demonstrate the effectiveness of the proposed framework using several experiments which involve introducing several mutations to the model transformation program and being able to kill all the mutants using only a subset of the test case set that is chosen based on the framework. The experiments also demonstrate the time saving benefit of the proposed approach.

The organization of the rest of this paper is as follows: First, we provide the necessary background and a motivating example in Section 2. Our proposed approach for regression testing is presented in Section 3. Section 4 discusses and evaluates our experiments. The related literature is discussed in Section 5. The conclusion and future work are discussed in Section 6.

2. Study background and motivating example

This section provides a description of the core concepts and terms used in this research. In addition, we present a model transformation example which motivates the regression test selection framework presented in the paper.

2.1. Models and model transformation

Models play a central role in MDE. A model represents a simplified or abstract representation of a part of a world (system) [10]. Models of a system help to analyze certain properties of the system without the need to consider its full details. Models help designers and architects to deal with the complexity present in systems. The model needs to conform to a metamodel. This means that the model needs to satisfy the rules defined in the metamodel and it must respect its semantics. A meta-modeling language is used to specify metamodels. For example, Ecore is a meta-modeling language used to specify metamodels in the Eclipse Modeling Framework (EMF) [11].

Models can be transformed into other models allowing for several types of analysis at different levels of abstraction. Model transformation refers

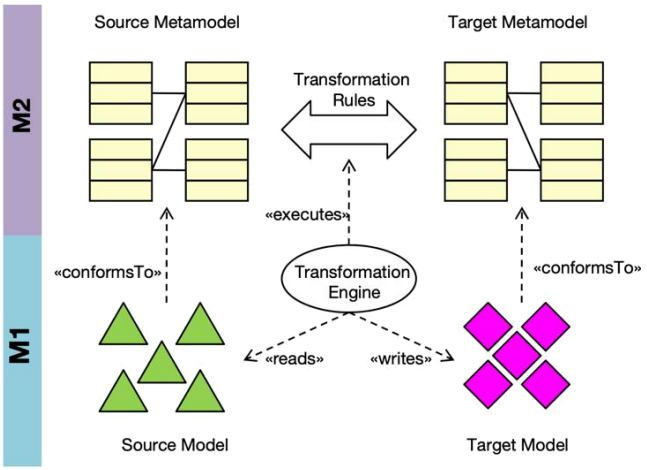
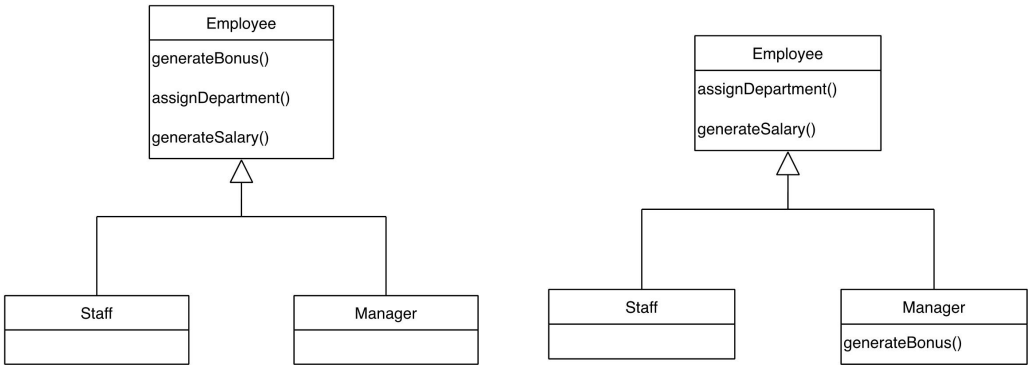


Figure 1: Model transformations pattern [12]



(a) Class diagram before executing the push down method transformation

(b) Class diagram after executing the push down method transformation

Figure 2: Class diagram before and after executing the push down method transformation

to automatically creating a target model from an existing source model by following transformation rules. A source model that conforms to a source metamodel is transformed into a target model that conforms to a target metamodel. A model-to-model transformation language is used to specify the transformation rules. The Epsilon Transformation Language (ETL), one of the Epsilon platform [13, 14] set of languages and tools, is a model-to-model transformation language that can be used to transform models specified in metamodels conforming to Ecore. ETL builds on the Epsilon Object Language (EOL), which is the main language in Epsilon for providing common model management facilities upon which several Epsilon-based languages are based. The ATLAS Transformation Language (ATL) is another model transformation language developed on top of the Eclipse platform [15].

Figure 1, taken from [12], depicts the general pattern of model transformations. *M1* in Figure 1 represents the models, which are instances of the metamodels represented by *M2*. Transformation rules are specified to transform the model elements of the source model to the model elements of the target model. The transformations are performed by a transformation engine, which reads a source model conforming to a source metamodel and produces a target model that conforms to a target metamodel.

Figure 2 shows an example of a model transformation where a push-down method transformation has been applied to a class diagram. The transformation pushed one of the methods from the super class to a child class while implementing better cohesiveness. Figure 2 (a) shows the source model and Figure 2 (b) shows the target model. The automatic model transformation

saves time and energy compared to the manual conversion and brings in the consistency of the information provided the model transformation is correctly defined.

In this paper, we used several languages and tools from Epsilon. Epsilon [13, 14] is a family of languages and tools for automating model management tasks, such as model transformation and model testing. Epsilon is built on top of the Eclipse platform and supports EMF-based model management tasks. One of the languages of Epsilon is the Epsilon Unit Testing Framework (EUnit)'s language. EUnit [16, 17] is a unit testing framework, specifically designed to test model transformations. It is based on EOL and Ant's workflow task description. EUnit provides a language for comparing models. Test cases can be defined, reused, and automatically run on different sets of models. The results of the test cases can be viewed, and the differences between the expected and the actual models are graphically visualized. EUnit uses EMF Compare as the comparison engine [18]. For example, if the value of an attribute of an element in the source model is not equal to the value of the same element in the target model, then the two models are considered not equivalent. In this case, EUnit reports the test case as a failed one.

2.2. Motivation example

In this example, we consider a model transformation example that uses ETL to transform a model that conforms to an Object-Oriented (OO) metamodel to a model that conforms to a Database (DB) metamodel. The code of the example was obtained from [19] and is presented in the Appendix. Model transformation from OO to DB models is a classic example which has been used by several authors in the literature to evaluate new techniques and approaches [20–22].

Figure 3 shows the OO metamodel. We only include the metamodel elements that are involved in the ETL transformation. The names of abstract classes are shown in italics. There are four concrete classes in the OO metamodel: *Class*, *Attribute*, *Reference*, and *Package*. In an OO model, a package can be composed of other packages and classes. A class can have features: attributes and references. A feature can have a type: a *DataType* in the case of an attribute and a *Class* in the case of a reference. A class may extend another class. In this case, the class will inherit all features of the parent class.

Figure 4 shows the DB metamodel. We only include the metamodel elements that are involved in the ETL transformation. The names of ab-

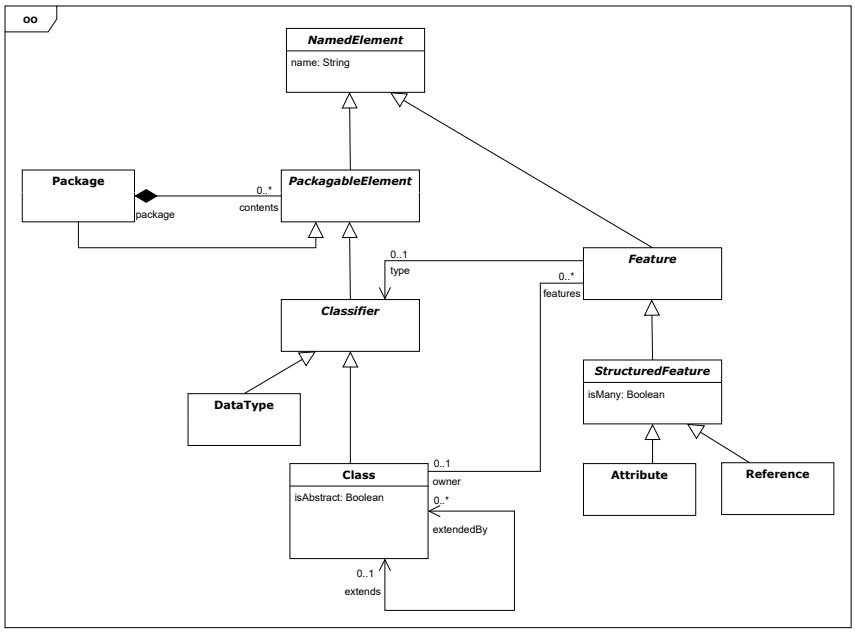


Figure 3: The OO metamodel

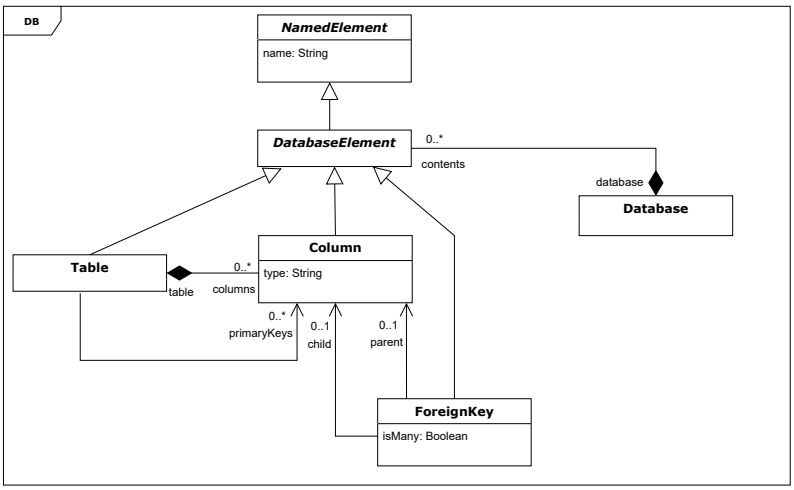


Figure 4: The DB metamodel

stract classes are shown in italics. There are four concrete classes in the DB metamodel: *Table*, *Column*, *ForeignKey*, and *Database*. In a DB model, a database is composed of database elements: tables, columns, and foreign keys. A table is composed of zero or more columns. For a table, a set of columns represent the primary keys for the table. A foreign key is associated with two columns: a child column and a parent column that could be in different tables. The table has a name attribute. Columns and foreign keys have name and type attributes.

The ETL code for the OO to DB model transformation consists of four rules: *Class2Table*, *SingleValuedAttribute2Column*, *MultiValuedAttribute2Table*, and *Reference2ForeignKey*. A brief description of each rule is provided in the comments. In ETL, each rule has a unique name and also specifies one source element and at least one target element. A rule can optionally define a guard which is a Boolean expression specifying a condition that must be met by the source element in order to apply the rule on that element. The body of a rule contains the logic specifying how to populate the values of the features of the target model elements. Some of the rules invoke one or more of the three user-defined EOL operations listed at the end of the code: *primaryKeyName*(), *valuesTableName*(), and *toDbType*(). The operation *primaryKeyName*() returns a string that represents the name of the primary key of the new table. The operation *valuesTableName*() returns a string that represents the name of the new table corresponding to a multivalued attribute in the OO model. The operation *toDbType*() returns a string that represents a type in the DB model.

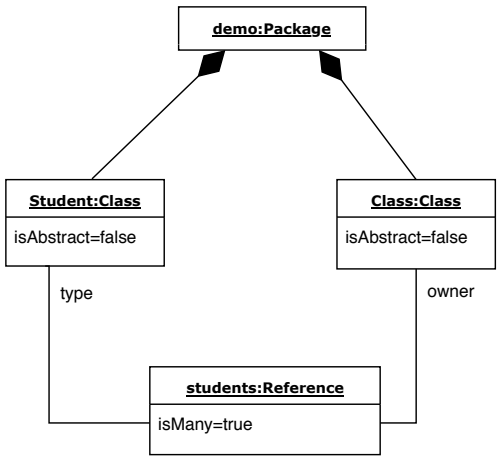


Figure 5: A sample source test model

During the testing of the model transformation program, a tester typically creates a large number of test cases. In each test case, the tester defines two test models: the first one is the source test model and the second one is the target test model. The source test model conforms to the source metamodel, while the target test model conforms to the target metamodel. The assumption is that in order for the test case to pass, the model generated by the model transformation program when the input is the source model must be equivalent to the target model. Figure 5

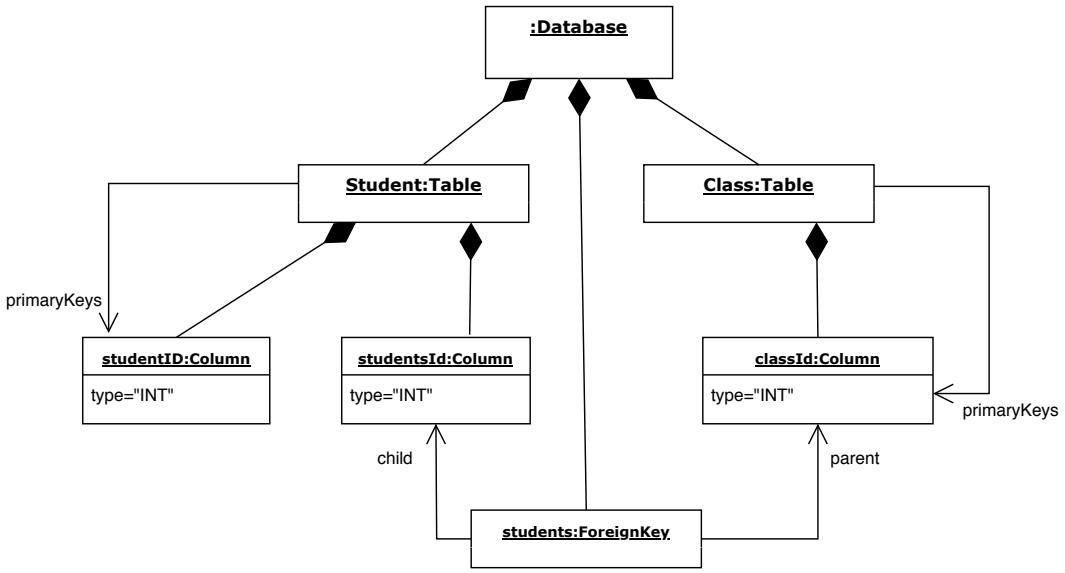


Figure 6: The target model corresponding to the source model in Figure 5

shows a sample source test model. It is an OO model containing two *Classes*: *Student* and *Class* contained in the *Package* named *demo*. Both *Classes* are not abstract (the value of *isAbstract* is *false* in both *Classes*). There is a *Reference* named *students* whose *owner* is *Class* and *type* is *Student*. The target DB model obtained when executing the model transformation is shown in Figure 6. As expected, the *Database* contains two *Tables*: *Student* and *Class*. The primary key columns, *classID* and *studentID*, are created by the rule *Class2Table*. The *ForeignKey* named *students* and the foreign key column *studentsId* are created by the rule *Reference2ForeignKey*. The *parent* and *child* references of the *students ForeignKey* are also set by the same rule to *classId* and *studentsId*, respectively.

The problem that this paper aims to tackle can be summarized as follows. Suppose that a tester has created a large number of test cases, each with its corresponding source and target models. During the maintenance and evolution of the model transformation program, several changes can be made to the program. A change usually involves one or more of the transformation program's rules. When a change is made to the program, a tester needs to rerun the test cases to ensure the correctness of the program. However, instead of rerunning the whole test case suite, this paper proposes a new framework that selects a subset of the test cases in a way that does not reduce the fault detection capability of the original suite. The framework exploits the traceability links between the rules in the transformation program and the source and target models of the related test cases. By applying this regression test selection framework, the tester benefits from the overhead and execution time that are saved when using a smaller subset of test cases for testing.

3. Approach

The proposed framework for regression test selection for model transformation relies on the use of a metamodel that links test cases with their corresponding test items and test artifacts. A test item represents an item under test. In the context of model transformation, a test item corresponds to a rule (or a statement block within the rule). A test case has a name that is used as an identifier for the test case. The test artifacts identify the test models related to the test item, including the source (input) and target (expected) models.

A metamodel that meets such requirements is shown in Figure 7. The metamodel is named as *TestCasesMM*. A test case set consists of zero or more test cases. Each test case is associated with a test item which corresponds to the rule being

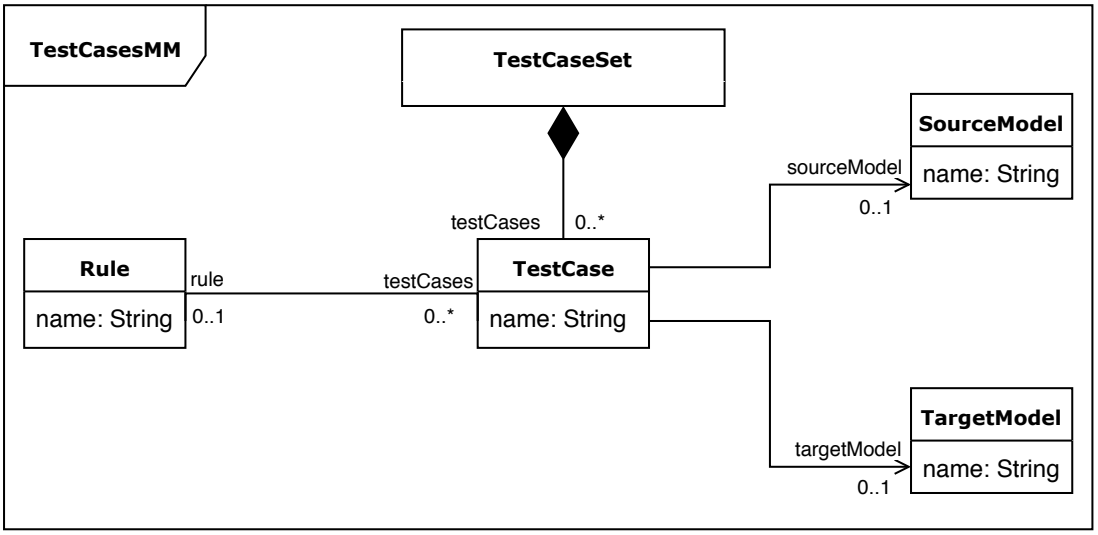


Figure 7: The test cases metamodel

tested by the test case. In cases where a test case is related to more than one rule, the metamodel definition allows to have repeated instances of the *TestCase* meta-class, with the same name, where each one is linked to an instance of the *Rule* meta-class. In addition, a test case has two test artifacts: a source model and a target model. When a rule is changed, the model of the test case set which conforms to *TestCasesMM* can be used to identify the test cases that need to be reexecuted. In addition, the model can be used to identify the source and target models required by each test case.

The traceability links in a test case model can also be used to provide information on the adequacy of the test cases. One example is checking that every rule is linked to at least one test case. If a rule is not linked to any test case, this could indicate that this rule is not covered by the test case set. In this case, a tester would need to reexamine the test case set and add new test cases to improve its coverage. In the Object Constraint Language (OCL) [23] which is the common standard to express constraints on models specified using object-oriented metamodeling technologies, the constraint that every rule is linked to at least one test case can be stated as follows:

```
context Rule
inv RuleLinkedToAtLeastOneTestCase:
self.testCases->notEmpty()
```

Here, for a test case model to be valid, this constraint requires that the collection of test cases linked to any rule is not empty. The keyword **context** is used to specify the model element to which the statement of the constraint applies. This statement is a Boolean expression referred to as invariant. The keyword **inv** is used to specify the invariant: first the name of the invariant is provided, followed by a colon, which is followed by the Boolean expression. In the expression, *self* refers to the model element on which the constraint is evaluated (i.e., whose name follows the keyword **context**). Note that this example is just one possible option for describing constraints on test case models and that our approach does not require it.

We have developed a tool to automate the regression test selection process. The main objective of the tool is to automatically create the test case model which conforms to the metamodel shown in Figure 7. The project files are available on GitHub at https://github.com/ialazzon/RegressionTestSelectionTool. The tool is a Java program that does the following. First, it parses the model transformation program and the EUnit file containing the test case definitions. It also reads all source test model files that are used by the test cases defined in the EUnit file. Then, it automatically creates the test case model, as specified by our approach. Finally, when a tester enters the name of a modified rule in the model

transformation program, the tool shows the contents of a new EUnit file defining the test cases selected by our approach. The tester can run this EUnit file to execute the selected test cases, rather than rerunning the original EUnit file containing the full set of test cases. The tool was used in all experiments conducted when validating the approach (see Section 4).

4. Validation

To evaluate our approach, we conducted several experiments using four different model transformation programs. This section starts with a list of two research questions that the experiments attempt to address. This is followed by a discussion of the experimental setup and results. Finally, a discussion on the threats to validity is provided.

4.1. Research questions

We defined two research questions as follows:

- **RQ1:** How does our approach of regression test selection compare with a re-test-all strategy? Investigating this question serves as a sanity check step. If the results of our experiments show no benefit of using our approach, then we can conclude that our approach is not needed and hence a tester would be better off rerunning all test cases in a test suite.
- **RQ2:** How cost-effective is our approach? We want to ensure that the fault detection capability when rerunning a selected set of test cases is not compromised, while at the same time there is a significant saving in the overhead involved when running these test cases.

4.2. Experimental setup and results discussion

To evaluate these research questions, we applied our approach using the automated tool discussed in Section 3 on four different transformation programs. The first one is the *OO2DB* transformation which has been presented as a motivating example in Section 2.2. The second one is the *QN2QPN* transformation which transforms queueing networks into queueing Petri nets. The *QN2QPN* transformation is written in ATL [15]. It was developed by one of the authors and is presented in [24]. The third one is the *BibTeX2DocBook* transformation which transforms a BibTeX model to a DocBook composed document [25]. The fourth one is the *CPL2SPL* transformation which transforms a CPL model to an SPL model [26]. CPL and SPL are two domain-specific languages used in telephony systems. Both *BibTeX2DocBook* and *CPL2SPL* transformations are written in ATL and available in the ATL Zoo [27].

Consider the *OO2DB* transformation. For regression testing, we adopt the test case metamodel, *TestCasesMM*, shown in Figure 7. Figure 8 shows part of the test case set model created by our tool. It shows three test cases: *TC1*, *TC2*, and *TC3*, with their associated rules and test artifacts. For example, the test case *TC1* is designed to test the rule *Reference2ForeignKey*. For this test case, the source model is *OO1* and the target model is *DBExpected1*. Note that the model *OO1* is the model shown in Figure 5 and the model *DBExpected1* is the model shown in Figure 6.

We manually created a total of 15 test cases using EUnit [17]. Five of the test cases were designed to achieve full line coverage. These test cases apply a white-box approach for testing the model transformation code. For example, test case *TC1* whose input model is shown in Figure 5 covers the rule *Reference2Foreign*. Also, two test cases *TC3* and *TC5* are needed to cover the rule *Class2Table* since it has a Boolean condition: one for the case where a class extends some other class and one for the case where no such class exists. The remaining test cases were designed to achieve full partition coverage for all classes, attributes, and associations in the *OO* metamodel. This test design applies the coverage adequacy criteria as defined by the black-box model transformation testing approach in [28]. For example, in the *OO* metamodel (see Figure 3) a *Class* can have zero or more features. Following the approach in [28], if a property

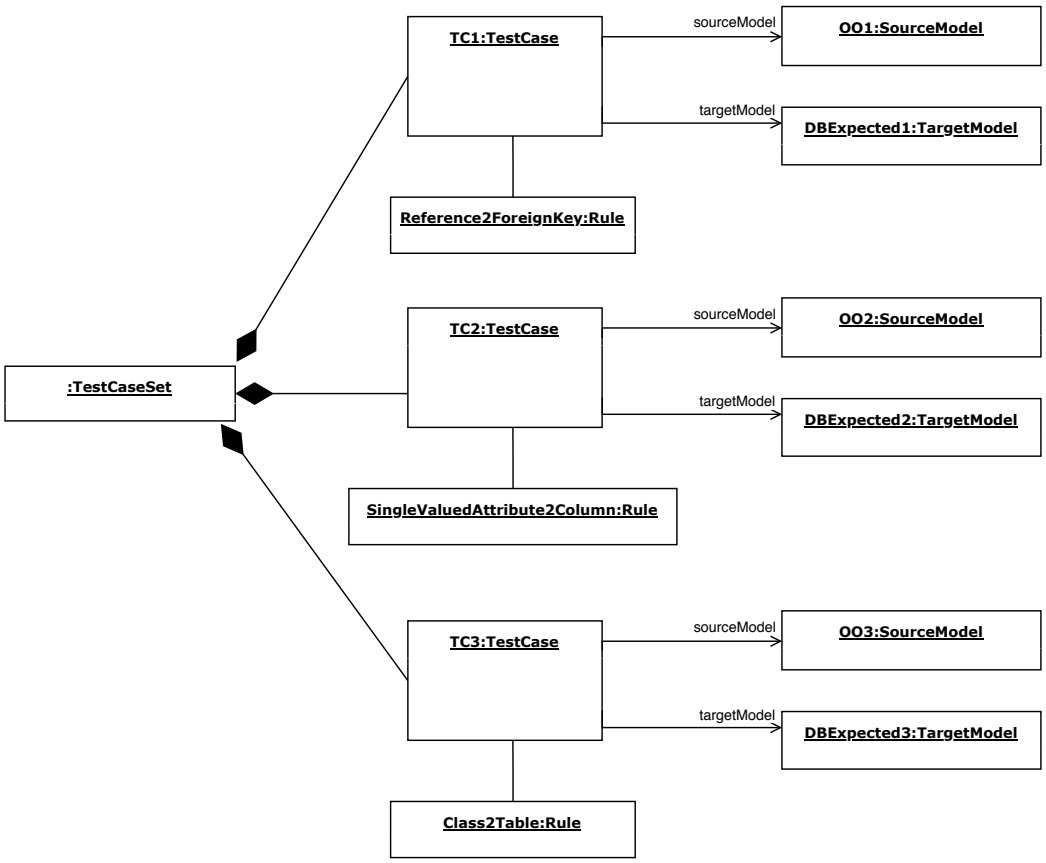


Figure 8: A partial model of the test case set

Table 1: The list of mutants

Mutant Number	Line Number	Original Code	Modified Code
1	102	fkCol.type=”INT”;	fkCol.type=”INTX”;
2	54	c.name=a.name;	c.name=”name”;
3	86	fkCol.type=”INT”;	fkCol.type=”INTX”;
4	19	t.columns.add(pk);	Line removed
5	36	childFkCol.type=”INT”;	childFkCol.type=”INTX”;

Table 2: The test cases corresponding to the mutants in Table 1 and their results

Mutant Number	Affected Rule	Selected Test Cases	No. of Selected Test Cases	Coverage	No. of Failed Test Cases
1	Reference2ForeignKey	*TC1*, *TC9*	2	13.33%	2
2	*SingleValuedAttribute2Column*	*TC2*, *TC6–8*	4	26.67%	3
3	MultiValuedAttribute2Table	*TC4*	1	6.67%	1
4	Class2Table	*TC3*, *TC5-15*	12	80.00%	12
5	Class2Table	*TC3*, *TC5-15*	12	80.00%	3

has a multiplicity of 0..*, a partition such as $\{\{0\},\{1\},\{x\}\}$, with $x \geq 2$, is defined to ensure that the test model contains instances where this property holds with zero, one and more than one object. Test cases *TC6*, *TC7*, and *TC8* include a *Class* which has zero, one, and two attributes, respectively.

We introduced several mutations to model transformation. For each mutant, our tool builds the model of the test case set to trace the affected rules to the test cases that need to be rerun. Table 1 shows a list of the five mutants used in the experiments. For each mutant, Table 2 shows the set of test cases selected by our tool in addition to their total number and coverage and the test case results. Here, coverage refers to the proportion of the number of selected test cases out of the total number of available test cases. The results show that our tool was successful in reducing coverage in every case. It can also be observed that at least one test case failed for each mutant. This indicates that using the traceability links in the test case model was effective in identifying a subset of test cases that need to be rerun and that are able to kill the mutants without requiring to rerun the complete set of test cases. In other words, the reduction in coverage of the selected test cases did not compromise the fault detection capability of our tool.

Consider the *QN2QPN* model transformation. This transformation consists of 10 rules. Table 3 shows the information and results concerning ten experiments using the *QN2QPN* transformation. We created a mutation on a single rule in every experiment. For each experiment, the table shows which rule was mutated and the type of mutation that was applied. The experiments covered the four main types of mutation operators on ATL transformations presented by Troya et al. [29]. These mutation operators were introduced by previous researchers to resemble common semantic faults that programmers make in model transformations [4]. Our experiments were designed to have a good coverage of all mutation operators proposed in literature. For each experiment, the table shows the number of test cases that were run and the execution time in milliseconds (ms) when rerunning all the test cases and when rerunning only the test cases selected by our tool. The percentages shown in red are the reductions (savings) that were achieved when rerunning the selected test cases only vs. rerunning all test cases. Larger values for the reductions represent higher savings.

To report the execution times, we conducted the experiments on a desktop with 4-core CPU running at 2.7 GHz and 8 GBs of RAM. Every measurement was repeated 10 times and the average values are presented here. Before each repetition, a warm-up session was conducted to eliminate the overhead caused by initialization processes in EUnit. The execution time of running a test suite was reported by EUnit as the wallclock time of executing the whole test suite.

Table 3: Results for the QN2QPN model transformation example

ID	Rule Affected	Type of Change	Rerun All		Rerun Selected Only	
			# of test cases	Execution time (ms)	# of test cases	Execution time (ms)
1	Main	Binding – Value Change	34	13064	34 (0%)	13255 (−1%)
2	Server	Filter – Addition	34	13762	23 (32%)	9015 (34%)
3	SourceNode	Binding – Deletion	34	12982	17 (50%)	6804 (48%)
4	SinkNode	In Pattern Element – Class Change	34	11890	7 (79%)	2937 (75%)
5	NonServerNode	Binding – Deletion	34	13375	13 (62%)	5442 (59%)
6	ThinkDevice	Matched Rule – Deletion	34	12256	13 (62%)	5157 (58%)
7	Arc	Out Pattern Element – Addition	34	13446	23 (32%)	8982 (33%)
8	ServiceRequest	Binding – Deletion	34	13277	22 (35%)	8263 (38%)
9	WorkloadRouting	Binding – Value Change	34	12898	22 (35%)	8474 (34%)
10	Workload	Out Pattern Element – Addition and Deletion	34	12013	30 (12%)	10462 (13%)

Table 4: Results for the BibTeX2DocBook model transformation example

ID	Rule affected	Type of change	Rerun all		Rerun selected only	
			# of tests cases	Execution time (ms)	# of tests cases	Execution time (ms)
1	Author	Binding – Value Change	25	7038	24 (4%)	6710 (4.66%)
2	TitledEntry_Title_NoArticle	Binding – Deletion	25	6765	9 (64%)	2813 (58.42%)
3	Article_Title_Journal	Binding – Deletion	25	7007	19 (24%)	5269 (24.80%)
4	Article_Title_Journal	Matched Rule – Deletion	25	6737	19 (24%)	5157 (20.26%)
5	Main	Out Pattern Element – Addition	25	6690	25 (0%)	6690 (0%)
6	UntitledEntry	Binding – Deletion	25	5844	25 (0%)	5844 (0%)
7	TitledEntry_Title_NoArticle	Binding – Value Change	25	6648	9 (64%)	2666 (59.90%)

The results in Table 3 show good reduction in terms of the number of selected cases and the execution time. The exception is in the first rule, *Main*, where there is no reduction in the number of selected test cases. This is because this rule is executed by the transformation on every input model, and hence any input test model will be automatically selected by our tool. The reductions reached more than 75% in Experiment 4. It is also important to mention that in each experiment at least one of the test cases failed the test (as reported by the EUnit tool) in both the Rerun all and Rerun Selected-only cases. This indicates that the fault detection capability was not diminished when rerunning the selected test cases. At the same time, a good savings in terms of the testing execution time were materialized.

Table 5 presents the results concerning the *BibTeX2DocBook* transformation. Note that this transformation consists of nine rules. In our experiments, we randomly selected a total of 25 test cases out of the 100 test cases available in the test suite taken from a previous work on spectrum-based fault detection in model transformations [4]. The test cases in the test suite were semi-automatically created using model generation scripts. We obtained the test suite from [30]. These 25 test cases represent the full test suite that is input to our tool for test case selection. The table shows good reductions in the size of the selected test cases and the execution time as well. However, there is little or no reduction in cases 1, 5, and 6. For case 1, the *Author* rule applies on *Author* elements which appear in almost all of the test cases. Hence, our tool selects all of these test cases resulting in a very small reduction in the size of the selected test cases. For cases 5 and 6, the *Main* and *UntitledEntry* rules apply on source meta-classes whose instances appear in all of the test cases. For example, the *Main* rule which applies on *BibTeXFile* elements is executed on every input model since every *BibTex* model has a root element of type *BibTeXFile*. Also, the *UntitledEntry* rule applies on elements of the *BibTeXEntry* source meta-class which is a superclass for many of the source meta-classes. Note that in each of the cases in Table 5 at least one of the test cases failed when running the selected test cases.

Table 6 presents the results concerning the *CPL2SPL* transformation. This transformation consists of 19 rules. Similar to the BibTex2DocBook transformation case, we randomly selected a total of 35 test cases from the test suite taken from [30]. In all cases, the table shows good reductions in the size of the selected test cases and the execution time as well. In addition, in all cases, rerunning the test cases selected by our tool resulted in at least one test case failure, indicating that the test suites selected by our tool were able to discover the mutants.

Table 7 shows the execution time results for the four case studies. For each case study, the table shows the time it took our tool to generate the test case model. The table shows the confidence intervals on a 95% confidence level. These intervals were obtained using the one-sample t-test [31] which is valid to be used in our case

Table 5: Results for the BibTeX2DocBook model transformation example

ID	Rule affected	Type of change	Rerun all		Rerun selected only	
			# of tests cases	Execution time (ms)	# of tests cases	Execution time (ms)
1	Author	Binding – Value Change	25	7038	24 (4%)	6710 (4.66%)
2	TitledEntry_Title_NoArticle	Binding – Deletion	25	6765	9 (64%)	2813 (58.42%)
3	Article_Title_Journal	Binding – Deletion	25	7007	19 (24%)	5269 (24.80%)
4	Article_Title_Journal	Matched Rule – Deletion	25	6737	19 (24%)	5157 (20.26%)
5	Main	Out Pattern Element – Addition	25	6690	25 (0%)	6690 (0%)
6	UntitledEntry	Binding – Deletion	25	5844	25 (0%)	5844 (0%)
7	TitledEntry_Title_NoArticle	Binding – Value Change	25	6648	9 (64%)	2666 (59.90%)

Table 6: Results for the CPL2SPL model transformation example

ID	Rule affected	Type of change	Rerun all		Rerun selected only	
			# of test cases	Execution time (ms)	# of test cases	Execution time (ms)
1	NoAnswer2SelectCase	Binding – Value Change	35	12656	2 (94.29%)	1118 (91.17%)
2	Busy2SelectCase	Filter – Addition	35	14130	3 (91.43%)	1419 (89.96%)
3	NoAnswer2SelectCase	Binding – Deletion	35	14459	2 (94.29%)	1131 (92.18%)
4	StringSwitch2SelectStat	In Pattern Element – Class Change	35	13224	11 (68.57%)	4458 (66.29%)
5	SwitchedAddress2SelectCase	Binding – Deletion	35	13039	1 (97.14%)	739 (94.33%)
6	Outgoing2Method	Matched Rule – Deletion	35	13191	2 (94.29%)	1087 (91.76%)
7	SubAction2Function	Out Pattern Element – Addition	35	13529	4 (88.57%)	1858 (86.27%)
8	StringSwitch2SelectStat	Binding – Deletion	35	14792	11 (68.57%)	5052 (65.85%)
9	Incoming2Method	Binding – Value Change	35	12716	4 (88.57%)	1889 (85.14%)
10	Proxy2Select	Out Pattern Element – Addition and Deletion	35	14064	5 (85.71%)	1251 (91.10%)

Table 7: The execution times for test case model generation

Case Study	Mean (ms)	The 95% Confidence Interval
OO2DB2	29.73	(25.46, 34.00)
QN2QPN	62.17	(54.62, 69.72)
BibTeX2DocBook	54.63	(48.73, 60.54)
CPL2SPL	75.13	(65.25, 85.01)

since the same size is small (30 execution time results in each case study) and the normality check provides good support for the assumption that the population is normally distributed. It is of interest to note that these observed execution times are very small compared with the test case execution time results noted in the previous tables in this section. In addition, our tool was able to automatically generate the test case models with no considerable cost on part of the tester.

4.3. Threats to validity

There are four basic types of validity threats that can affect the validity of the conclusions of our experiments [32]:

1. *Conclusion Validity:* Threats to the conclusion validity are concerned with factors that affect the ability to draw the correct conclusions based on the observed data. To address this threat, we used confidence intervals based on a 95% confidence level. These intervals were obtained using the one-sample t-test after passing the normality check. Also, we used the wallclock times reported by the EUnit tool for all test case suite execution time results reported in the paper. We have also used a variety of mutation operators in the experiments.
2. *Construct Validity:* This validity is concerned with the relationship between theory and observation. To address this threat, we used standard performance measures and metrics, including rule coverage and execution time. These metrics have been used in other similar work on regression testing for model transformations, such as [3, 4].
3. *Internal Validity:* This validity is concerned with establishing a causal relationship between the treatment (in this case, the application of our approach) and the results of our evaluation. Threats to this validity include any disturbing factor that might influence the results. As our experiments demonstrated, the execution time of a test case suite is directly proportional to its size. Therefore, the observed reductions in execution time can be justified by the selection of a smaller number of test cases by applying our approach. In addition, in every experiment we made a mutation to a single rule only. Every mutation applied one of the mutation operators proposed in literature [29, 33]. If a rule is mutated, then this would affect any input test model that includes an element on which the rule is applied. Hence, our tool would automatically select the corresponding test case for rerun. Hence, the fault detection capability of the original test case suite is not compromised by the suite of the test cases selected by our tool.
4. *External Validity:* This validity is concerned with generalization. The evaluation applied our approach on two model transformations written in two languages: ETL and ATL. We applied different types of mutation operators, and the test models were of different sizes. The test cases were created manually to achieve a variety of coverage criteria. Yet, we cannot make a firm conclusion that our results can be generalized for all model transformations. More experiments are needed in the future to confirm our findings on a wider scope of model transformations, including different languages, coverage criteria, types of mutations and faults, and input test models.

5. Related work

Alkhazi et al. [3] propose an approach for test case selection for model transformation based on multiobjective search. The approach enables a tester to find the best tradeoff between two conflicting objectives: maximizing rule coverage and minimizing the execution time of the selected test cases. A multiobjective algorithm (NSGA-II) is used to find the Pareto-optimal solutions for this problem. The approach was validated using different transformation programs written in ATL. In comparison to their approach, our approach aims to be a safe test case selection technique. A test selection technique is said to be safe if it selects all modification-revealing tests [7]. Our proposed technique will select any test case for rerun when its input model contains an element that could be affected by a change in one of the model transformation rules. Our approach does not consider the trade-off between rule coverage and execution time. However, our experiments show good saving in test execution time when applying our approach while not compromising the fault detection capability of the full test case suite.

In [20], model transformation traceability is used to enhance the automation of qualifying and improving a set of test models in the mutation analysis of model transformation. The

approach relies on a representation of different mutation operators and a traceability mechanism to establish links between the input and output models of each transformation. Patterns are used to identify cases where an input test model lets a given mutant alive. Subsequently, heuristics provide recommendations to generate new test models that are able to kill the mutant. Several aspects of the approach are independent from the transformation language being used, including the traceability and the mutation operator representation. Our approach focuses on regression testing of model transformation rather than mutation analysis. Hence, our approach uses a traceability model linking test cases to the rules in a given model transformation. When a rule is changed, the traceability model can be used to identify the test cases that need to be rerun. Our traceability model differs from that in [20] which maintains links between elements of the input and output models for each mutant. This is a more detailed model suitable for mutation analysis of model transformation.

A multiobjective optimization algorithm is employed in [9] to generate test models for the regression testing of model transformations. The proposed approach assumes that the changes occur in the input metamodel only. In this case, a test model may become invalid when it does not conform to the updated input metamodel. The optimization algorithm has three objectives that define the characteristics of a good solution: maximize coverage of the updated metamodel, minimize the number of input model elements that do not conform to the updated metamodel, and minimize the number of refactorings used to refactor the existing test models. In our approach, we assume that the input and output metamodels are fixed and only the model transformation program may change. While the approach in [9] is useful to determine and update the test models that become invalid due to a change in the input metamodel, our approach utilizes traceability links to determine the test cases that need to be rerun due to an update in the model transformation program.

Honfi et al. [34] presented a method on how model-based regression testing can be achieved in the context of autonomous robots. The method uses optimization for selecting the minimal subset of tests that have a maximum test coverage of the changed components. Although the method is presented in the context of robots, it is applicable to other domains which employ Model-Driven Development (MDD) paradigm. In MDD, models are adopted as the main development artifacts. These models are commonly created using domain specific languages (DSLs). When a model is changed, this can impact the system functions and properties and hence the influenced parts of the system need to be retested. A prototype tool that implements the method using the Eclipse framework [35] and its modeling platform EMF [11] is presented. The tool supports model checkpointing and automatic change detection. Our approach is similar to [34] in terms of employing a metamodel to represent the relationship between the test items and the test cases. However, our approach is focused on applying regression testing to a model transformation. We consider the issues specific to a model transformation which can be more useful to a tester of model transformation programs.

A survey of model transformation testing approaches is provided in [6] and [36]. González and Cabot [37] developed a tool, called ATLTest, to generate test input models for ATL transformations. The tool applies a white-box approach for model transformation testing. In the work of Fleurey et al. [28], model fragments are used as a test adequacy criterion which forms the basis of the black-box approach proposed by the authors. Other approaches rely on formal methods to verify the transformation and its associated properties [38–40]. The problem with these approaches is their computational complexity which becomes cumbersome with the scale of the model transformation program.

Zech et al. proposed a model-based regression testing method based on OCL [41]. The method derives test cases for a given system under test (SUT) based on the availability of a class diagram that captures its system design. The approach is based on a Model Versioning and Evolution (MoVE) framework and uses UML testing profile (UTP) to model test cases. The

method calculates a *delta* from a base model (the initial development model) and the working model (the current development model). The resulting change set (*delta*) then contains the differences between the two versions of the model. There are three main differences between the method by Zech et al. and our method. First, our method presents a regression testing approach for model transformation while the method by Zech et al. presents a regression testing approach of a software system using its design model. Second, Our approach requires a metamodel of the test case set while the method by Zech et al. does not need any metamodel, but rather uses the facilities provided in an MoVE framework. The last difference is that our approach exploits traceability links between models while the method by Zech et al. does not use any traceability link. A similar model-based regression testing for software systems that works with sequence diagrams is presented in [42]. In [43], Al-Refai *at al.* provide a framework for model-based regression test selection supporting modifications to UML class and activity diagrams. Using mutation testing, their experimental results demonstrate that the selected test cases achieve the same fault detection capability as that achieved by the complete set of test sets.

The work of Troya et al. [44] proposes an approach for automatically inferring metamorphic relations for testing ATL model transformations. The inferred metamorphic relations can be used to detect faults in model transformations in several application scenarios including regression testing. The metamorphic relations are inferred by exploiting the trace model produced when a transformation is executed. The trace model is composed of traces. When a rule is executed, a trace can be automatically obtained by using tools such as TraceAdder [45]. For the executed rule, a trace links the name of the rule with the elements instantiating the classes of its source metamodel and the new elements that are created by the rule and hence instantiate classes in the target metamodel. The authors used mutation-based testing, similar to what is done in this paper, to evaluate how effective the approach is in detecting faults in regression testing.

In [4], Troya et al. presented an approach for debugging and fault localization for model transformations by applying spectrum-based fault localization techniques. The approach is based on the use of a trace model that can be obtained when the test cases are executed. When a test case fails, the approach ranks the transformation rules according to how much they are suspected of having the fault causing its failure. Mutation-based testing is applied to validate the effectiveness of their approach in fault localization.

In [46], Naslavsky *et al.* presented an approach for selective regression testing that is model-based. In this approach, test cases are selected for retesting based on modifications to the model, rather than to the source code. The approach uses traceability links between model elements and test cases that traverse such elements. As a modeling perspective, the approach adopts UML class and sequence diagrams. While their presented approach is designed for testing general software programs, our work is designed for testing model-to-model transformation programs. In our approach, we exploit the traceability links between test cases and model transformation program elements such as rules.

Our approach for regression testing of model transformation utilizes traceability links between test cases and test artifacts. Traceability has been studied by researchers in the areas of requirements engineering and model-driven development (MDD) for a long time. Winkler and Pilgrim [47] provide an extensive literature survey on traceability in both areas.

Due to the continuous increase in the size and complexity of software, model-based engineering is gaining a lot of interest from the industry and research community [48]. Model-based testing, which is an important part of model-based engineering tests the consistency of information and behavior of source models by applying transformation mechanisms. The automatic nature of model-based testing makes it a more adoptable approach for detecting software defects fast and effectively [49, 50]. Regression testing is a quite a tedious work which is repeated with every sizeable refactoring of the model. Model-based

approaches can make this process easier by automating, managing, and documenting efficiently. Yoo *et.al* [8] suggest, based on their extensive survey, that model-based regression testing techniques increase the effectiveness and scalability of the overall testing of the system. Model-based regression testing approaches have several advantages over code-based testing [51, 52]. The effort could be estimated at a very early stage and the tools are largely language independent. The models are mostly abstract, which makes the size of the testable data considerably smaller than the code.

The majority of model-based regression testing approaches exploit UML models for developing test suites for regression testing. UML class models have been used for this purpose along with state machines, sequence diagram and activity diagrams [51–53]. Farooq et al. [52] use state machines to represent changes in the tested parts of the system. The method is also automated using an Eclipse-based tool. Briand et al. [51, 54] presented a sequence diagram-based technique to classify and analyze regression test suites. The approach is complemented with a tool to evaluate the presented models. Finite State Machines (FSMs) have also been used to generate regression test suites. Chen et al. [55] have deployed extended FSMs to model the effects of changes and generate test suites for the modified parts of the system. Korel *et al* [56] have used a similar approach and have exploited extended FSMs to reduce the size of an existing regression test suite. In both approaches, modifications (updates, additions and deletions) are represented as transitions of extended FSMs. In one study, Vaysburg et al. [57] used extended FSMs for the dependency analysis of the system that represent various interactions between components for the regression test selection process. In another study, Almasri *et al* [58] conducted an impact analysis using extended FSMs to identify the parts of the system that are affected the most. Feldere et al. [59, 60] also used FSMs to represent all model elements of the system. They proposed a process to identify model elements which trigger change events. The identified models are then changed to make the system consistent.

6. Conclusion and future work

In this paper, we have presented a framework for the regression test selection for model transformation programs. The framework exploits the traceability links in a test case model. In the evaluation, we applied the framework to several model transformation examples and showed the effectiveness and time-saving benefit of the framework. The experiments were performed in the context of the Epsilon platform of integrated tools and languages for model management. We also presented a tool that can automatically build the test case model and thus facilitate the implementation of our proposed framework.

Following this work, there are several avenues for future research. First, the proposed framework needs to be integrated with existing model transformation tools and technologies. For instance, when a designer makes a change to the model transformation code, the relevant test cases identified by our framework can be automatically rerun by the integrated tools. In this case, the designer does not need to manually rerun the test cases. Second, it is recommended to apply the proposed framework to industrial case studies involving larger models and more complex model transformation logic.

7. Acknowledgement

This work has not received any funding.

References

[1] A.R. da Silva, "Model-driven engineering: A survey supported by the unified conceptual mode," *Computer Languages, Systems & Structures*, Vol. 43, 2015, pp. 139–155.

[2] P. Mohagheghi and V. Dehlen, "Where is the proof? – A review of experiences from applying mde in industry," in *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications*, 2008, pp. 432–443.

[3] B. Alkhazi, C. Abid, M. Kessentini, D. Leroy, and M. Wimmer, "Multi-criteria test cases selection for model transformations," *Automated Software Engineering*, Vol. 27, No. 1, 2020, pp. 91–118.

[4] J. Troya, S. Segura, J.A. Parejo, and A. Ruiz-Cortés, "Spectrum-based fault localization in model transformations," *ACM Transactions on Software Engineering and Methodology*, Vol. 27, No. 3, 2018.

[5] D. Calegari and N. Szasz, "Verification of model transformations: A survey of the state-of-the-art," *Electronic Notes in Theoretical Computer Science*, Vol. 292, 2013, pp. 5–25.

[6] G.M.K. Selim, J.R. Cordy, and J. Dingel, "Model transformation testing: The state of the art," in *Proceedings of the First Workshop on the Analysis of Model Transformations*, 2012, pp. 21–26.

[7] G. Rothermel and M.J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, 1996, pp. 529–551.

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, Vol. 22, No. 2, 2012, pp. 67–120.

[9] J. Shelburg, M. Kessentini, and D.R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *Proceedings of the International Symposium on Search Based Software Engineering*, 2013, pp. 209–223.

[10] E. Seidewitz, "What models mean," *IEEE Software*, Vol. 20, No. 5, 2003, pp. 26–32.

[11] *Eclipse Modeling Framework (EMF)*, Eclipse Foundation. [Online]. https://www.eclipse.org/modeling/emf/ [Accessed December 2020].

[12] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, Vol. 45, No. 3, 2006, pp. 621–645.

[13] *Eclipse Epsilon*, Eclipse Foundation. [Online]. https://www.eclipse.org/epsilon/ [Accessed December 2020].

[14] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, *The epsilon book*. Eclipse Foundation., 2018. [Online]. https://www.eclipse.org/epsilon/doc/book/

[15] *ATL – A model transformation technology*, Eclipse Foundation. [Online]. https://www.eclipse.org/atl/ [Accessed December 2020].

[16] *The Epsilon Unit Testing Framework*, Eclipse Foundation. [Online]. https://www.eclipse.org/epsilon/doc/eunit/ [Accessed December 2020].

[17] A. García-Domínguez, D.S. Kolovos, L.M. Rose, R.F. Paige, and I. Medina-Bulo, "EUnit: A unit testing framework for model management tasks," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2011, pp. 395–409.

[18] *EMF Compare*, Eclipse Foundation. [Online]. https://www.eclipse.org/emf/compare/ [Accessed December 2020].

[19] *Epsilon – Examples*. [Online]. https://www.eclipse.org/epsilon/examples/ [Accessed December 2020].

[20] V. Aranega, J. Mottu, A. Etien, T. Degueule, B. Baudry, and J. Dekeyser, "Towards an automation of the mutation analysis dedicated to model transformation," *Software Testing, Verification and Reliability*, Vol. 25, No. 5-7, 2015, pp. 653–683.

[21] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: Testing model transformations," in *Proceedings of the First International Workshop on Model, Design and Validation*, 2004, pp. 29–40.

[22] S. Sen, B. Baudry, and J. Mottu, "On combining multi-formalism knowledge to select models for model transformation testing," in *Proceedings of the First International Conference on Software Testing, Verification, and Validation*, 2008, pp. 328–337.

[23] *Object Constraint Language*, Object Management Group, 2014. [Online]. http://www.omg.org/spec/OCL/2.4 [Accessed December 2020].

[24] I. Al-Azzoni, "ATL transformation of queueing networks to queueing Petri nets," in *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2017, pp. 261–268.

[25] *ATL Transformation Example: BibTeXML to DocBook*, 2005. [Online]. https://www.eclipse.org/atl/atlTransformations/BibTeXML2DocBook/ExampleBibTeXML2DocBook[v00.01].pdf [Accessed December 2020].

[26] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry, "Building DSLs with AMMA/ATL, a case study on SPL and CPL telephony languages," in *Proceedings of the ECOOP Workshop on Domain-Specific Program Development*, 2006.

[27] *ATL Transformations Zoo*, Eclipse Foundation. [Online]. https://www.eclipse.org/atl/atlTransformations/ [Accessed December 2020].

[28] F. Fleurey, B. Baudry, P. Muller, and Y.L. Traon, "Qualifying input test data for model transformations," *Software and System Modeling*, Vol. 8, No. 2, 2009, pp. 185–203.

[29] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer, "Towards systematic mutations for and with ATL model transformations," in *Proceedings of International Conference on Software Testing, Verification and Validation Workshops*, 2015, pp. 1–10.

[30] J. Troya, *Implementation of the Spectrum-Based Fault Localization in Model Transformations*, 2018. [Online]. https://github.com/javitroya/SBFL_MT [Accessed December 2020].

[31] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers, 6th Edition*. John Wiley and Sons, 2013.

[32] C. Wohlin, P. Runeson, M. Hst, M.C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, 2012.

[33] E. Guerra, J. Sánchez Cuadrado, and J. de Lara, "Towards effective mutation testing for ATL," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2019, pp. 78–88.

[34] D. Honfi, G. Molnár, Z. Micskei, and I. Majzik, "Model-based regression testing of autonomous robots," in *Proceedings of the International System Design Language Forum*, 2017, pp. 119–135.

[35] *Eclipse*, Eclipse Foundation. [Online]. https://www.eclipse.org/ [Accessed December 2020].

[36] L.A. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Software and Systems Modeling*, Vol. 14, No. 2, 2015, pp. 1003–1028.

[37] C.A. González and J. Cabot, "ATLTest: A white-box test generation approach for ATL transformations," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2012, pp. 449–464.

[38] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, Vol. 83, No. 2, 2010, pp. 283–302.

[39] K. Ehrig, J.M. Küster, and G. Taentzer, "Generating instance models from meta models," *Software and Systems Modeling*, Vol. 8, No. 4, 2009, pp. 479–500.

[40] J. Troya and A. Vallecillo, "Towards a rewriting logic semantics for ATL," in *Proceedings of the International Conference on Theory and Practice of Model Transformations*, 2010, pp. 230–244.

[41] P. Zech, P. Kalb, M. Felderer, C. Atkinson, and R. Breu, "Model-based regression testing by OCL," *International Journal on Software Tools for Technology Transfer*, Vol. 19, No. 1, 2017, pp. 115–131.

[42] L. Naslavsky, H. Ziv, and D.J. Richardson, "A model-based regression test selection technique," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2009, pp. 515–518.

[43] M. Al-Refai, S. Ghosh, and W. Cazzola, "Supporting inheritance hierarchy changes in model-based regression test selection," *Software and Systems Modeling*, Vol. 18, No. 2, 2019, pp. 937–958.

[44] J. Troya, S. Segura, and A. Ruiz-Cortés, "Automated inference of likely metamorphic relations for model transformations," *The Journal of Systems and Software*, Vol. 136, 2018, pp. 188–208.

[45] F. Jouault, "Loosely coupled traceability for ATL," in *Proceedings of the European Conference on Model Driven Architecture Workshop on Traceability*, 2005, pp. 29–37.

[46] L. Naslavsky, H. Ziv, and D. Richardson, "MbSRT2: Model-based selective regression testing with traceability," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2010, pp. 89–98.

[47] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software and Systems Modeling*, Vol. 9, No. 4, 2010, pp. 529–565.

[48] B. Legeard, "Model-based testing: Next generation functional software testing," in *Practical Software Testing: Tool Automation and Human Factors*, M. Harman, H. Muccini, W. Schulte, and T. Xie, Eds. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010.

[49] A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, 2007, pp. 31–36.

[50] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann, 2007.

[51] L.C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Information and Software Technology*, Vol. 51, No. 1, 2009, pp. 16–30.

[52] Q. Farooq, M.Z.Z. Iqbal, Z.I. Malik, and A. Nadeem, "An approach for selective state machine based regression testing," in *Proceedings of the International Workshop on Advances in Model-based Testing*, 2007, pp. 44–52.

[53] Y. Chen, R.L. Probert, and D.P. Sims, "Specification-based regression test selection with risk analysis," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 2002.

[54] L.C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proceed-*

ings of the International Conference on Software Maintenance, 2002, pp. 252–261.

[55] Y. Chen, R.L. Probert, and H. Ural, "Regression test suite reduction using extended dependence analysis," in *Proceedings of the International Workshop on Software Quality Assurance*, 2007, pp. 62–69.

[56] B. Korel, L.H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proceedings of the International Conference on Software Maintenance*, 2002.

[57] B. Vaysburg, L.H. Tahat, and B. Korel, "Dependence analysis in reduction of requirement based test suites," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2002, pp. 107–111.

[58] N. Almasri, L. Tahat, and B. Korel, "Toward automatically quantifying the impact of a change in systems," *Software Quality Journal*, Vol. 25, No. 3, 2017, pp. 601–640.

[59] M. Felderer, B. Agreiter, and R. Breu, "Evolution of security requirements tests for service-centric systems," in *Engineering Secure Software and Systems*, Ú. Erlingsson, R. Wieringa, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 181–194.

[60] M. Felderer, B. Agreiter, and R. Breu, "Managing evolution of service centric systems by test models," in *Proceedings of the Tenth IASTED International Conference on Software Engineering*, 2011, pp. 72–80.

Appendix: ETL OO2DB code

```
pre {
  "Running ETL".println();
  var db : new DB!Database;
}

// Transforms a class into a table and
// a primary key column
 rule Class2Table
  transform c : OO!Class
  to t : DB!Table, pk : DB!Column {

  t.name = c.name;
  t.database = db;

  // Fill the details of the primary key
  // of the table
  pk.name = t.primaryKeyName();
  pk.type = "INT";
  t.columns.add(pk);
  t.primaryKeys.add(pk);

  // If the class extends some other class
  // create a foreign key pointing towards
  // the primary key of the parent class
  if (c.`extends`.isDefined()){

    var fk : new DB!ForeignKey;
    var childFkCol : new DB!Column;
    var parentFkCol : DB!Column;
    var parentTable : DB!Table;

    parentTable ::= c.`extends`;
    parentFkCol = parentTable.primaryKeys.first();

    childFkCol.name = parentFkCol.name;
    childFkCol.type = "INT";
    childFkCol.table = t;

    fk.database = db;
    fk.parent = parentFkCol;
    fk.child = childFkCol;
    fk.name = c.name + "Extends" + c.`extends`.name;
  }
}

// Transforms a single-valued attribute
// to a column
 rule SingleValuedAttribute2Column
  transform a : OO!Attribute
  to c : DB!Column {

  guard : not a.isMany

  c.name = a.name;
  c.table ::= a.owner;
  c.type = a.type.name.toDbType();
}

// Transforms a multi-valued attribute
// to a table where its values are stored
// and a foreign key
rule MultiValuedAttribute2Table
  transform a : OO!Attribute
  to t : DB!Table, pkCol : DB!Column, valueCol :
  DB!Column, fkCol : DB!Column,
  fk : DB!ForeignKey {

  guard : a.isMany

  // The table that stores the values
  // has an "id" column and a "value" column
  t.name = a.valuesTableName();
  t.database = db;

  pkCol.name = "id";
  pkCol.table = t;
  pkCol.type = "INT";
  valueCol.name = "value";
  valueCol.table = t;
  valueCol.type = a.type.name.toDbType();

  // Another column is added into the table
  // to link with the "id" column of the
  // values table
  fkCol.name = a.name + "Id";
  fkCol.table ::= a.owner;
  fkCol.type = "INT";

  // The foreign key that connects
  // the two columns is defined
  fk.parent = pkCol;
  fk.child = fkCol;
  fk.database = db;
}

// Transforms a referecne into a foreign key
rule Reference2ForeignKey
  transform r : OO!Reference
  to fk : DB!ForeignKey, fkCol : DB!Column {

  fkCol.table ::= r.type;
  fkCol.name = r.name + "Id";
  fkCol.type = "INT";
  fk.database = db;
  fk.parent = r.owner.equivalent().primaryKeys.first();
  fk.child = fkCol;
  fk.name = r.name;

}

operation DB!Table primaryKeyName() : String {
  return self.name.firstToLowerCase() + "Id";
}

operation OO!Attribute valuesTableName() : String {
  return self.owner.name + "_" +
   self.name.firstToUpperCase() + "Values";
}

operation Any toDbType() : String {
  var mapping : OO2DB!TypeMapping;
  mapping = OO2DB!TypeMapping.allInstances().
    select(tm|tm.source = self). first ;
  if (not mapping.isDefined()){
    ("Cannot find DB type for OO type " + self +
      ". Setting the default .").println();
    return OO2DB!TypeMap.allInstances().first().
    `default`.target;
  }
  else {
    return mapping.target;
  }
}
```