Automated Inference of Likely Metamorphic Relations for Model Transformations

Javier Troya*, Sergio Segura, Antonio Ruiz-Cortés

Department of Computer Languages and Systems, Universidad de Sevilla, Spain

Abstract

Model transformations play a cornerstone role in Model-Driven Engineering (MDE) as they provide the essential mechanisms for manipulating and transforming models. Checking whether the output of a model transformation is correct is a manual and error-prone task, referred to as the oracle problem. Metamorphic testing alleviates the oracle problem by exploiting the relations among different inputs and outputs of the program under test, so-called metamorphic relations (MRs). One of the main challenges in metamorphic testing is the automated inference of likely MRs.

This paper proposes an approach to automatically infer likely MRs for ATL model transformations, where the tester does not need to have any knowledge of the transformation. The inferred MRs aim at detecting faults in model transformations in three application scenarios, namely regression testing, incremental transformations and migrations among transformation languages. In the experiments performed, the inferred likely MRs have proved to be quite accurate, with a precision of 96.4% from a total of 4101 true positives out of 4254 MRs inferred. Furthermore, they have been useful for identifying mutants in regression testing scenarios, with a mutation score of 93.3%. Finally, our approach can be used in conjunction with current approaches for the automatic generation of test cases.

Keywords: Model-Driven Engineering, Metamorphic Testing, Metamorphic Relations, Model Transformations, Automatic Inference, Generic Approach

1. Introduction

In Model-Driven Engineering (MDE), models are the central artifacts that describe complex systems from various viewpoints and at multiple levels of abstraction using appropriate modeling formalisms. Model transformations are the cornerstone of MDE [1, 2], as they provide the essential mechanisms for manipulating and transforming models. Model transformations are an excellent compromise between strong theoretical foundations and applicability to real-world problems [2].

The correctness of software built using MDE techniques typically relies on the correctness of the operations executed using model transformations. For this reason, it is critical in MDE to maintain and test them as it is done with source code in classical software engineering. However, checking whether the output of a model transformation is correct is a manual and error-prone task, what is referred to as the *oracle problem* in the software testing literature. Although several approaches address the testing of model transformations with different techniques [3, 4, 5, 6, 7, 8, 9, 10, 11, 12], the oracle problem is still challenging in this domain. One reason may be the lack of formal semantics in transformation languages [8].

As any other software artifact, models are subject to constant change. Therefore, dependent models, which have been derived by the original models by means of transformations, have to be updated appropriately [13]. The most straight-forward way is to re-execute the transformation entirely, i.e., in batch mode. However, when we are dealing with very large models consisting of thousands of elements and some minor changes take place in them, the re-execution of a complete transformation is not the best way to proceed. In fact, only those elements that have been changed should be transformed, what is achieved by the so-called *incremental transformations* [14, 15, 16, 17, 18].

The role of incremental model transformations is the following. A potentially very big model is received as input by a model transformation, which is executed and produces the output model. Then, some small changes are performed in the source model. An incremental transformation detects what the changes have been and propagates only those changes to the already existing output model. However, it cannot be trivially checked that the changes have been properly propagated and that the result of the incremental transformation is the same as having executed the original transformation.

Model transformations are themselves also subject to change. Having a model transformation in a state that is known to be correct, it may be necessary to extend it in order to increase its functionality, but the properties that were satisfied by the original version must be still satisfied in the new version. Therefore, it is important to count on *regression testing* techniques [19] in model transformations. Furthermore, due to the wide variety of model transformation languages available today (cf. Section 2.1.2), it may be necessary to move from some languages to others, i.e., to *migrate* a transformation program to a different

*Correspondence to: University of Seville, ETS de Ingenieria Informatica, room E0.40-A. Avda. Reina Mercedes s/n, 41012 Seville, Spain. E-mail: jtroya@us.es

language [8, 20, 21, 22, 23]. In this scenario, it is also necessary to count on mechanisms for checking the correctness of the migrated transformation.

Metamorphic testing alleviates the oracle problem by providing an alternative when the expected output of a test execution is unknown [24, 25]. Rather than checking the output of an individual program execution, metamorphic testing checks whether multiple executions of the program under test fulfil certain necessary properties called *metamorphic relations* (MRs). Unfortunately, the main limitation of metamorphic testing is the identification of the MRs, which is a manual task that requires a good knowledge of the problem domain. Indeed, the automated inference of likely MRs is recognized as one of the main challenges of metamorphic testing [25]. A few works with proposals for automatic inference of MRs exist, but mostly focused on numerical programs [26, 27, 28].

The feasible application of metamorphic testing in model transformations has been empirically demonstrated by Jiang et al. [29]. In their work, metamorphic relations are manually constructed for a specific model transformation, based on the knowledge of the domain expert.

The context of metamorphic testing in model transformations and the nomenclature we use in this paper are the following. Let us consider we have any model transformation, *mt*, and $SMs = \{sm_1, sm_2, ..., sm_n\}$ is a set of source test case inputs, i.e., source models for *mt*. Based on the knowledge of the models' domain, follow-up test case inputs, sm'_i, can be defined for each source test case input. The way of constructing follow-up test case inputs from source test case inputs is determined by the *metamorphic input relations* (MRs_{IP}). This means that a MR_{IP} indicates which changes must be done in sm_i to obtain sm'_i, but it does not impose any restriction on how sm_i must be. Therefore, the definition of pairs $[sm_i, sm'_i]$ is driven by the MRs_{IP}. For each metamorphic input relation and based on the knowledge of *mt*, several metamorphic output relations (MR_{OP}) that must hold can be defined. A pair $[MR_{IP}, MR_{OP}]$ constitutes a metamorphic relation, and several metamorphic relations can share the same MR_{IP}. Metamorphic testing runs the source and follow-up test case inputs constructed as defined by the MR_{IP} and checks whether the MR_{OP} holds in the source and follow-up test case outputs, regardless of the availability of an oracle for each individual test case.

In this paper we propose the automatic inference of likely metamorphic relations for any model transformation written in ATL [30, 31] – metamorphic relations that are automatically inferred are referred to as *likely* metamorphic relations because we cannot formally demonstrate that they are valid under all circumstances. To this end, a catalogue of 24 so-called *domain-independent metamorphic relations* has been defined. The inference of likely metamorphic relations[1] for a model transformation is possible by taking the trace model produced after the transformation execution as input to our approach.

This approach is targeted at checking the correctness of model transformations in the three application scenarios described before, namely incremental transformations, regression testing and migration to other transformation languages. In fact, we consider that the MRs are inferred from a well-tested version of the model transformation with the intention of using them to detect faults in (i) future versions of the transformation (regression testing), (ii) incremental versions of the original transformation and (iii) the same transformation written in a different language. Out of these three scenarios, the evaluation of the approach performed in this paper focuses on regression testing, although we hypothesize the approach would also be applicable to the other two scenarios.

With our approach, the tester does not need to have any knowledge of the model transformation nor its domain in order to obtain the metamorphic relations for a particular transformation, but only needs to provide a source model for the transformation. This work is the first one that proposes automatic inference of MRs in the context of MDE and model transformations. We have evaluated our approach with seven model transformations from different domains and that differ in their size and complexity. Our initial results are quite promising, we have obtained 4101 true positives out of 4254 inferred likely metamorphic relations, what means a precision of 96.4%. Besides, when using the MRs to kill model transformation mutants, we have obtained a mutation score of 93.3%. Furthermore, our approach is scalable due to the way the catalogue of domain-independent metamorphic relations has been defined, so it can be further extended and improved.

The remainder of this paper is organized as follows. Section 2 presents the essentials for understanding our approach, namely models, metamodels, model transformations and metamorphic testing. Section 3 follows with an in-depth explanation of our approach, which is evaluated in Section 4. The related work is reviewed in Section 5. Finally, we summarize our conclusions and present potential future work in Section 6.

2. Background

In this section we present the basics to understand our approach. We start with a general introduction to model-driven engineering (MDE), where we focus on metamodels, models and, specially, model transformations and the ATL language. Then, we explain metamorphic testing and put it in the context of model transformations.

2.1. Model-Driven Engineering

Model-Driven Engineering (MDE) [32] is a methodology that advocates the use of models as first-class entities throughout the software engineering life cycle. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system. In this section we describe the concepts of MDE necessary to understand our approach.

[1] From hereon, when we write (domain-independent) metamorphic relations, we will always be referring to *likely* (domain-independent) metamorphic relations

2.1.1. Models and Metamodels

A model is an abstraction of a system often used to replace the system under study [33, 34]. Thus, (part of) the complexity of the system that is not necessary in a certain phase of the system development is removed in the model, making it more simple to manage, understand, study and analyze. Models are also used to share a common vision and facilitate the communication among technical and non-technical stakeholders [32].

Every model must conform to a metamodel. Indeed, a metamodel defines the structure, semantics and constraints for a family of models [35]. Like everything in MDE, a metamodel is itself a model, and it is written in the language defined by its meta-metamodel. It specifies the concepts of a language, the relationships between these concepts, the structural rules that restrict the possible elements in the valid models and those combinations between elements with respect to the domain semantic rules.

2.1.2. Model Transformations

Model transformations play a cornerstone role in MDE since they provide the essential mechanisms for manipulating and transforming models [36, 37]. They allow querying, synthesizing and transforming models into other models or into code, so they are essential for building systems in MDE. A model transformation is a program executed by a transformation engine that takes one or more source models and produces one or more target models, as illustrated by the model transformation pattern [1] in Figure 1. Model transformations are developed on the metamodel level, so they are reusable for all valid model instances.

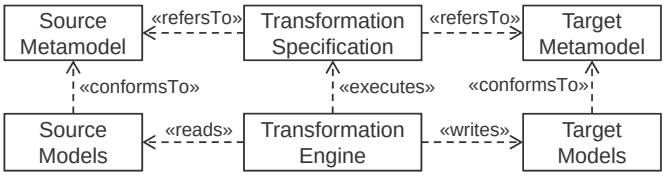


Figure 1: Model transformation pattern (from [1])

Although there are several types of model transformations, such as *model-to-text* (M2T), *text-to-model* (T2M) and *model-to-model* (M2M) transformations, in this paper we focus on the last one. In any case, it can be straightforwardly generalized to consider the other two types of model transformations by using approaches such as the one presented in [38], where a metamodel for representing text is proposed. There are also specific categories within M2M transformations [1, 39]. A transformation is considered *out-place* when it creates new models from scratch, e.g., reverse engineering code as models, or *in-place* if it rewrites the input models until the output models are obtained, e.g., as it is the case in model refactoring. There is a plethora of frameworks and languages to define M2M transformations, such as Henshin [40], AGG [41], Maude [42], AToM^3 [43], e-Motions [44], VIATRA [45], MOMoT [46, 47], QVT [48], Kermeta [49], JTL [50], and ATL [51]. Among these, we focus in this paper on the ATL language due to its importance in both the academic and the industrial arenas.

2.2. Atlas Transformation Language

This model transformation language (named *ATL* for short) has come to prominence in the model-driven engineering community due to is flexibility, support of the main meta-modeling standards, usability that relies on strong tool integration with the Eclipse world, and a supportive development community [30, 31].

ATL is a textual rule-based model transformation language that provides both declarative and imperative language concepts. It is thus considered a hybrid model transformation language. Both in-place and out-place model transformations can be defined in ATL.

An ATL transformation is composed of a set of transformation rules and helpers. Each rule describes how certain target model elements should be generated from certain source model elements. Typically, ATL model transformations are composed of declarative rules, namely *matched rules*. These rules are automatically executed by the ATL execution engine for every match in the source model according to the source patterns of the matched rules. There exist also rules that have to be explicitly called from another rule, namely *(unique) lazy rules* and *called rules*, which gives more control over the transformation execution.

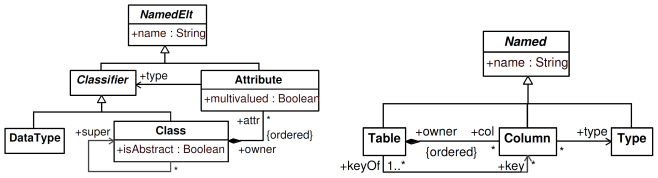
The Object Constraint Language (OCL) is used all throughout ATL transformations as an expression language. A *helper* can be seen as an auxiliary OCL function, which can be used to avoid the duplication of the OCL code at different points in the ATL transformation.

Rules are mainly composed of an *input pattern* and an *output pattern*[2]. The input pattern is used to match *input pattern elements* that are relevant for the rule. The output pattern specifies how the *output pattern elements* are created from the input model elements matched by the input pattern. Each output pattern element can have several *bindings* that are used to initialize its attributes and references.

Listing 1: Excerpt of *Class2Relational* ATL Transformation.

```
1  module Class2Relation;
2  create OUT : RelationMM from IN : ClassMM;
3
4  helper def : objectIdType : Relational!Type =
5    Class!DataType.allInstances() -> select(e | e.name = 'Integer') -> first();
6
7  rule ClassAttribute2Column {
8    from
9      a : Class!Attribute (a.type.oclIsKindOf(Class!Class) and not a.multiValued)
10   to
11     foreignKey : Relational!Column (
12       name <- a.name + 'Id',
13       type <- thisModule.objectIdType)
14 }
15
16 rule Class2Table {
17   from
18     c : Class!Class
19   to
20     out : Relational!Table (
21       name <- c.name,
22       col  <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
23       key  <- Set {key}),
24     key : Relational!Column (
25       name <- 'objectId',
26       type <- thisModule.objectIdType)
27 }
```

[2]Please note that the terms source/target and input/output are used synonymously throughout the paper when talking about model transformations and their elements



(a) Source metamodel. (b) Target metamodel.

Figure 2: Metamodels of the Class2Relational transformation

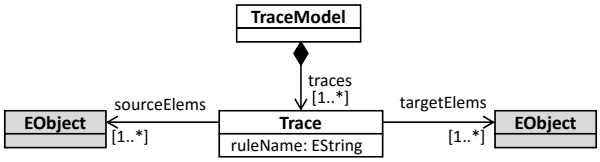


Figure 3: Trace Metamodel.

2.2.1. ATL Transformation Example

An excerpt of the well-known *Class2Relational* model transformation, taken from the ATL Zoo [52], is displayed in Listing 1. The input and output metamodels, necessary to understand the transformation, are displayed in Figure 2.

In the excerpt, we have included two declarative rules (so-called "matched rules" in ATL). The first rule, *ClassAttribute2Column*, takes elements of type *Attribute* whose *type* is a *Class* and whose *multiValued* attribute is *false* (cf. metamodel in Fig. 2(a)), line 9. These elements are transformed into elements of type *Column* (cf. metamodel in Figure 2(b)) in line 11. The value assigned to the *name* attribute is the same as the *name* of the *Attribute* element concatenated with "Id" (line 12). The element referenced by the *type* relationship is retrieved by a helper function in line 13. The second rule, *Class2Table*, takes an element of type *Class* as input and creates two elements, one of type *Table* (line 20) and one of type *Column* (line 24). The *name* given to the *Column* is "objectId", and its *type* is also assigned with the helper. Regarding the *Table*, its *key* points to the new *Column* created. As for its *col* reference, it also points to the *key Column* and to other elements (line 22) of type *Attribute*. Since *Attributes* are elements from the source model, ATL has to resolve which elements are created from such *Attributes*, and these will be the actual elements pointed by *col*.

2.2.2. ATL Internal Traces Mechanism

In order to resolve such elements, ATL uses an internal tracing mechanism. Thereby, every time a rule is executed, it creates a new trace and stores it in the internal trace model. A trace model can be automatically obtained from a transformation execution, e.g., by using Jouault's *TraceAdder* [53], and is composed of a set of traces, one for each rule execution. The metamodel to which a trace model conforms is displayed in Figure 3. A trace captures the name of the applied rule and the elements conforming to those classes from the source metamodel (*sourceElems* reference) that are used to create new elements conforming to classes in the target metamodel (*targetElems* reference). The classes in the source and target metamodels are represented with *EObject*, since they can be any class of said metamodels. When a trace model is created, its traces reference actual elements of the source and target models. This means that we have three models (the source model, the target model and the trace model) linked by several inter-model references. Therefore, by navigating the traces, we can obtain information of which target element(s) have been created from which source element(s) and by which rule. As we explain in Section 3, this is key in our approach.

An example of a trace model is shown in Figure 4. In this example, the transformation has been executed taken as input a source model composed of one *Class* that has one *Attribute*. The model produced as output by the transformation contains two *Columns* and one *Table*. As we see in the figure, two *Trace* instances have been created, one for each of the two rules that compose the transformation. We observe how the attributes and references of the elements in the target model have been initialized as specified in the model transformation of Listing 1.

2.3. Metamorphic Testing

A *test oracle* [54] determines whether a test execution reveals a fault, typically comparing the observed program output to the expected output. Test oracles are not always available or may be too difficult to apply, this is known as the *oracle problem* [55, 56, 54]. For instance, suppose an online search using the keyword "testing" returning 65K results: Is this output correct? Are there any pages containing the keyword not included in the resultset? Do all the pages in the resultset contain the keyword? Answering these questions is challenging and rarely feasible.

Metamorphic testing alleviates the oracle problem by providing an alternative when the expected output of a test execution is unknown [24]. Rather than checking the output of an individual test, metamorphic testing checks whether multiple test executions fulfil certain metamorphic relations. A *metamorphic relation* is a necessary property of the target program that relates two or more input data and their expected outputs. For instance, consider the program $nsearch(k_1)$ that returns the number of matches of the keyword k_1 in a database. Intuitively, the number of results should be smaller when searching for items including both k_1 and another keyword k_2. This can be expressed as the following metamorphic relation: $nsearch(k_1) \geq nsearch(k_1 \wedge k_2)$. A metamorphic relation comprises of a so-called *source test case* ($nsearch(k_1)$) and one or more *follow-up test cases* ($nsearch(k_1 \wedge k_2)$), derived from the source test

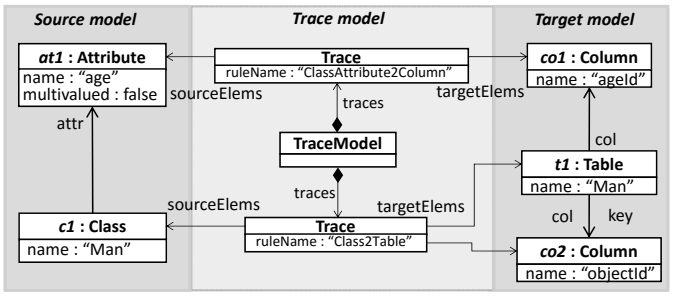


Figure 4: Traces in a specific transformation scenario of *Class2Relational*.

case. A metamorphic relation can be instantiated into one or more *metamorphic tests* by using specific input values, e.g., $nsearch(metamorphic) \geq nsearch(metamorphic \wedge testing)$. If the outputs of a source test case and its follow-up test case(s) violate the metamorphic relation, the program under test must contain a bug.

Metamorphic testing is not only an effective technique to alleviate the oracle problem but it can also be regarded as a test data generation technique. For instance, in the previous example, metamorphic testing could be used together with a random word generator to generate source test cases (e.g., $nsearch(house)$) and their corresponding follow-up test cases (e.g., $nsearch(house \wedge car)$), enabling full test automation, i.e., input generation and output checking.

Metamorphic testing was introduced as a technique to reuse existing test cases back in 1998 by Chen et al. [24]. Since then, researchers have realized of the potential of the technique to address the oracle problem and research contributions have proliferated. In a recent survey, Segura et al. [25] reviewed about 120 papers on metamorphic testing and identified successful applications of the technique in a variety of domains including Web services and applications [57, 58, 59, 60], embedded systems [61, 62], computer graphics [63, 64], compilers [65, 66], simulation [67, 68], machine learning [69, 70] and bioinformatics [71, 72].

This testing technique can also be applied to model transformations [29], as exemplified in Figure 5. Now, the source test case input corresponds to the source model (SM) of the model transformation, and the follow-up test case input is called follow-up source model (fuSM). A fuSM is derived from a SM, by performing some modification in it, such as addition or deletion of elements, modification of attributes, addition or deletion of references, and so on. In this context, a *metamorphic input relation* (MR_{IP}) defines a change that is performed in SM in order to produce fuSM. Therefore, the construction of pairs [SM,fuSM] is driven by the available MRs_{IP}. When the model transformation is executed taking as input SM, we obtain the source test case output, namely the target model (TM) in the context of model transformations, while when we execute it taking as input the fuSM, we obtain the follow-up test case output, referred to as follow-up target model (fuTM) in our context. The comparison of a specific property in TM with respect to fuTM represents the *metamorphic output relation* (MR_{OP}). This means that the MR_{OP} specifies a condition that must hold for the model transformation to be correct.

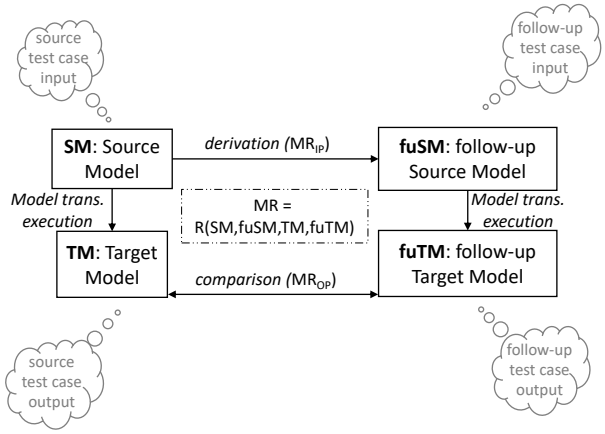


Figure 5: Metamorphic Testing in Model Transformations

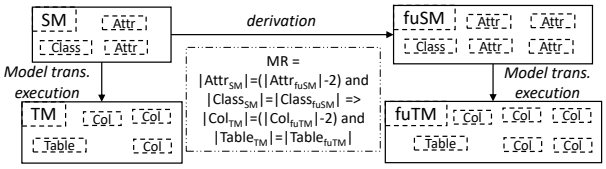


Figure 6: Application Example of Met. Testing in Model Transformations

In this context, a metamorphic relation is defined as a relation among the four parts, those conforming the metamorphic input relation and those of the metamorphic output relation: $MR = R(SM, fuSM, TM, fuTM)$. We have defined a catalogue of 24 likely *domain-independent metamorphic relations* (DIMR), meaning that they are abstractly defined and are to be instantiated in the context of any specific model transformation. A DIMR can be instantiated as one or several MRs in a specific model transformation.

An example of a metamorphic relation for the model transformation described in Section 2.2.1 is graphically shown in Figure 6. The metamorphic input relation applied dictates the addition of two *Attributes*. We observe that SM contains two *Attributes* and a *Class*, and therefore fuSM has four *Attributes* and a *Class*. When we execute the model transformation shown in Listing 1 over SM, we obtain TM, which contains one *Table* and three *Columns*. When the input for the model transformation is fuSM, it produces a fuTM that contains one *Table* and five *Columns*. In the metamorphic relation shown in the center of the figure, $|T_m|$ indicates the number of elements of type T that model m contains. Therefore, the MR in the figure can be read as "If two elements of type *Attribute* are added in fuSM and no element of type *Class* is added (what represents an instantiation of the MR_{IP}), then fuTM must contain two more elements of type *Column* than TM and the same number of elements of type *Table* (this second part must hold and is an instantiation of the MR_{OP})".

3. Automatic Generation of Metamorphic Relationships

The goal of our approach is to automatically infer likely metamorphic relations for any model transformation, such as the one depicted in Figure 6. We consider ATL model transformations [51], although our approach can be trivially extended to any model transformation language able to store the execution in traces. Indeed, as most model transformation languages are composed of model transformation rules, where each rule deals with the construction of part of the target model from part of the source model, it is straightforward to establish traces at the level of transformation rules to store this information. The use of traces storing the execution of a model transformation has not only been applied for ATL [73], but also for other lan-

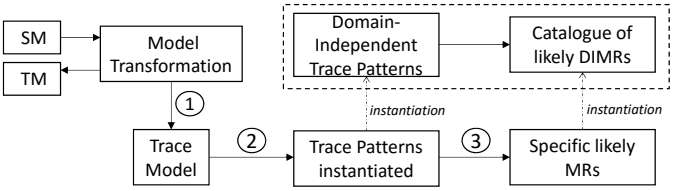


Figure 7: Overview of our approach.

guages able to express model transformations in terms of transformation rules, such as Maude [8].

An overview of our approach is depicted in Figure 7. The fundamental idea is to identify a set of *Domain-Independent Trace Patterns*. A domain-independent trace pattern involves (a part of) one or more traces. For each domain-independent trace pattern identified, we then define one or more likely *domain-independent metamorphic relations* (DIMRs), which we classify as conceptual patterns and put all together in an extensible *Catalogue of likely DIMRs*. We qualified these trace patterns and likely metamorphic relations as *domain-independent* because they are not defined in the context of any specific model transformation.

When a *Model Transformation* is executed, it produces a *Trace Model* (cf. Section 2.2.2), marked as first step in the figure. Then, our approach determines the *Trace Patterns instantiated* among all the traces of the *Trace Model*, in the second step. These patterns are an instantiation of the *Domain-Independent Trace Patterns* in the specific *Model Transformation*. Finally, the third step consists in inferring the *Specific likely MRs* from the *Trace Patterns instantiated* in the *Trace Model*. Said *Specific likely MRs* are instantiations of the *likely DIMRs*.

In this section we first present a running example, namely the *PetriNet2PNML* model transformation, which will be used for the explanation of our approach. Then, we define the generic patterns for the traces and how they are instantiated in our running example, what represents step two in Figure 7. Finally, we describe the catalogue of likely domain-independent metamorphic relations derived from the domain-independent patterns of the traces and instantiate some of them in our running example, represented as the third step in the figure.

3.1. Running Example: PetriNet2PNML

This transformation has been taken from the ATL Zoo [52], a repository containing a large set of ATL model transformations.

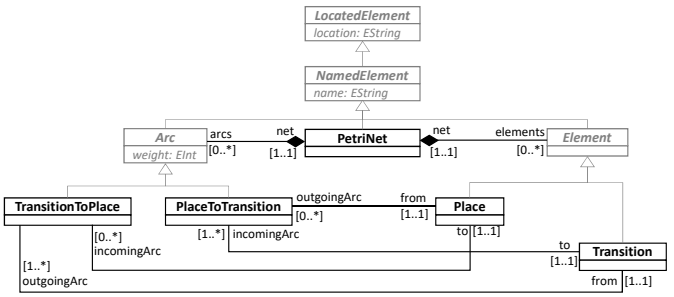


Figure 8: PetriNet Metamodel.

It transforms *PetriNet* models into *PNML* models.

Listing 2: *PetriNet2PNML* ATL Transformation.

```
1  module PetriNet2PNML;
2  create OUT : PNML, TM : TMM from IN : PetriNet;
3
4  rule PNMLDocument {
5    from
6      e : PetriNet!PetriNet
7    to
8      n : PNML!PNMLDocument(
9        location <- e.location,
10       xmlns <- uri,
11       nets <- net),
12     uri : PNML!URI(
13       location <- e.location,
14       value <- 'http://www.informatik.hu-berlin.de/
           top/pnml/ptNetb'),
15     net : PNML!NetElement(
16       name <- name,
17       location <- e.location,
18       id <- e.location,
19       type <- type_uri,
20       contents <- e.elements.union(e.arcs)),
21     name : PNML!Name(
22       location <- e.location,
23       labels <- label),
24     label : PNML!Label(
25       text <- e.name),
26     type_uri : PNML!URI(
27       location <- e.location,
28       value <- 'http://www.informatik.hu-berlin.de/
           top/pntd/ptNetb')
29 }
30
31 rule Place {
32   from
33     e : PetriNet!Place
34   to
35     n : PNML!Place(
36       name <- name,
37       id <- e.name,
38       location <- e.location),
39     name : PNML!Name(
40       labels <- label),
41     label : PNML!Label(
42       text <- e.name)
43 }
44
45 rule Transition {
46   from
47     e : PetriNet!Transition
48   to
49     n : PNML!Transition(
50       name <- name,
51       id <- e.name,
52       location <- e.location),
53     name : PNML!Name(
54       labels <- label),
55     label : PNML!Label(
56       text <- e.name)
57 }
58
59 rule Arc {
60   from
61     e : PetriNet!Arc
62   to
63     n : PNML!Arc(
64       name <- name,
65       location <- e.location,
66       id <- e.name,
67       source <- e."from",
68       target <- e."to"),
69     name : PNML!Name(
70       labels <- label),
71     label : PNML!Label(
72       text <- e.name)
73 }
```

The *PetriNet* metamodel is shown in Figure 8. Note that, for readability purposes, abstract classes and inheritance relationships are depicted in gray, while the rest is shown in black.

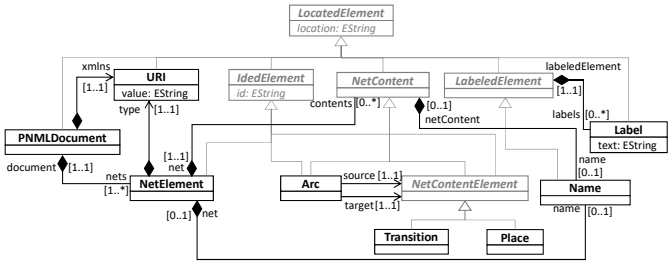


Figure 9: PNML Metamodel.

PetriNet is the root element and represents a Petri net. It is composed of *elements* and *arcs*, which are abstract classes. There are two types of *Elements*, namely *Place* and *Transition*, and two types of *Arcs*, which are *PlaceToTransition* and *TransitionToPlace*. *Arcs* represent the transition between two *Elements*, as indicated by references *from* and *to*. Likewise, *Elements* have the references *incomingArc* and *outgoingArc* in order to indicate the *Arcs* that connect them.

PNML stands for the Petri Net Markup Language [74]. It is a proposal of an XML-based interchange format for Petri nets. In fact, it has become a standard of the ISO/IEC [75]. Originally, it was intended to serve as a file format for the Java version of the Petri Net Kernel. PNML is a concept for defining the overall structure of a Petri net file. Its metamodel is depicted in Figure 9. *PNMLDocument* is the root element that contains Petri nets, and it also contains a *URI*. *NetElement* represents the Petri net and contains a *type*, which is a *URI*. It is also composed of *contents*, of abstract type *NetContent* that can be instantiated as an *Arc*, *Transition* or *Place*. *Arcs* connect, with the *source* and *target* references, *NetContentElements* between them. These can be, in turn, *Transitions* or *Places*. The two kinds of *Arcs* in the *PetriNet* metamodel (Fig. 8: *PlaceToTransition* and *TransitionToPlace*) are not differentiated in this metamodel. *NetElements* and *NetContents* can have a name that is a *LabeledElement* composed of *Label*.

The *PetriNet2PNML* ATL transformation is shown in Listing 2 and is explained in [76].

In order to extract the trace model for this case study, we have created a small source model (a petri net model) as input for executing the transformation. It is composed of three *Elements* and two *Arcs*. This model is graphically depicted in the left-hand side of Figure 10, with dark-gray background. The PNML model resulting from executing the transformation is depicted in the right-hand side of the same figure, also with dark-gray background. Please note that there are several sets of elements surrounded by dotted outlines. They represent elements that have been created by the same rule, as it is indicated by the trace element that points to the set. The only purpose of these clusters in the figure is to avoid having too many references from the traces to the target elements and, in this way, improve the readability of the figure. As mentioned before, when executing an ATL transformation, we can automatically extract a trace model [53] that links the source elements with the target elements. The trace model in this example is depicted in the center of the figure, with a light-gray background.

3.2. Patterns in Traces

In this section we describe the *Domain-Independent Trace Patterns* (cf. Figure 7) that we have identified after studying the trace models resulting from the execution of several model transformations. It is very important to highlight that the patterns are not exclusive, i.e., more than one patterns may appear in the same trace. In order to properly separate and identify each one of them, we use different text formats and types of lines for graphically representing them in the traces (cf. Figures 10 and 11). Furthermore, we will be referring to our running example for clarifying the explanation, so we will show and describe some of the *Trace Patterns instantiated*. The complete set of patterns in the catalogue as well as those inferred in several model transformations are shown on our website [77].

As explained below, the five patterns identified are graphically depicted in Figure 11. There, *SE* stands for *SourceElement*, *TE* for *TargetElement*, *sa* for *source attribute*, *sav* for *source attribute value*, *ta* for *target attribute*, *tav* for *target attribute value*, *tca* for *target constant attribute* and *tcav* for *target constant attribute value*. All these terms refer to abstract concepts, in the sense that they need to be instantiated.

***PatternTR_1*: target elements creation from source element.** This is the most basic pattern. A certain number of elements (NE) of the same type, *TargetElemet* (TE), in the target model are created from an element of the source model of type *SourceElement* (SE). This is depicted graphically by the trace of Figure 11(a). Having a look at Figure 10, we can see that there are 17 instantiations of this pattern in our case study. We show a subset of them in Listing 3. For instance, the first instantiation of the listing indicates that for each *PetriNet* in the source model, two *URIs* are created in the target model.

Listing 3: Some Trace Patterns instantiated for *Pattern_TR1* in *PetriNet2PNML*

```
1 SE=PetriNet NE=2 TE=URI
2 SE=PetriNet NE=1 TE=Name
3 SE=Place NE=1 TE=Label
4 SE=Place NE=1 TE=Place
5 SE=Place NE=1 TE=Name
6 SE=Transition NE=1 TE=Transition
7 SE=Transition NE=1 TE=Label
8 SE=Transition NE=1 TE=Name
```

***PatternTR_2*: target element creation from source element and attribute initialization.** In this pattern, a target element is also created from a source element, so it includes *PatternTR_1*. Besides, now, an attribute of the target element, *ta*, is initialized with a specific value, *tav*, which is derived from the specific value, *sav*, of an attribute in the source element, *sa*. This pattern is shown in Figure 11(b). The features that are specific for this pattern, namely the attributes, are written in italics. There are 18 instantiations of this pattern in the execution of our running example (cf. Figure 10), five of which are shown in Listing 4. For instance, the first instantiation in the listing specifies that a *NetElement* is created from a *PetriNet*. Furthermore, the *id* attribute of the *NetElement* is initialized with the value of the *location* attribute of the *PetriNet*. Specifically, the value *Seville* has been copied.

Listing 4: Some Trace Patterns instantiated for *Pattern_TR2* in *PetriNet2PNML*

```
1 SE=PetriNet TE=NetElement sa=location sav=Seville ta=id tav=Seville
2 SE=PetriNet TE=NetElement sa=location sav=Seville ta=location tav=Seville
```

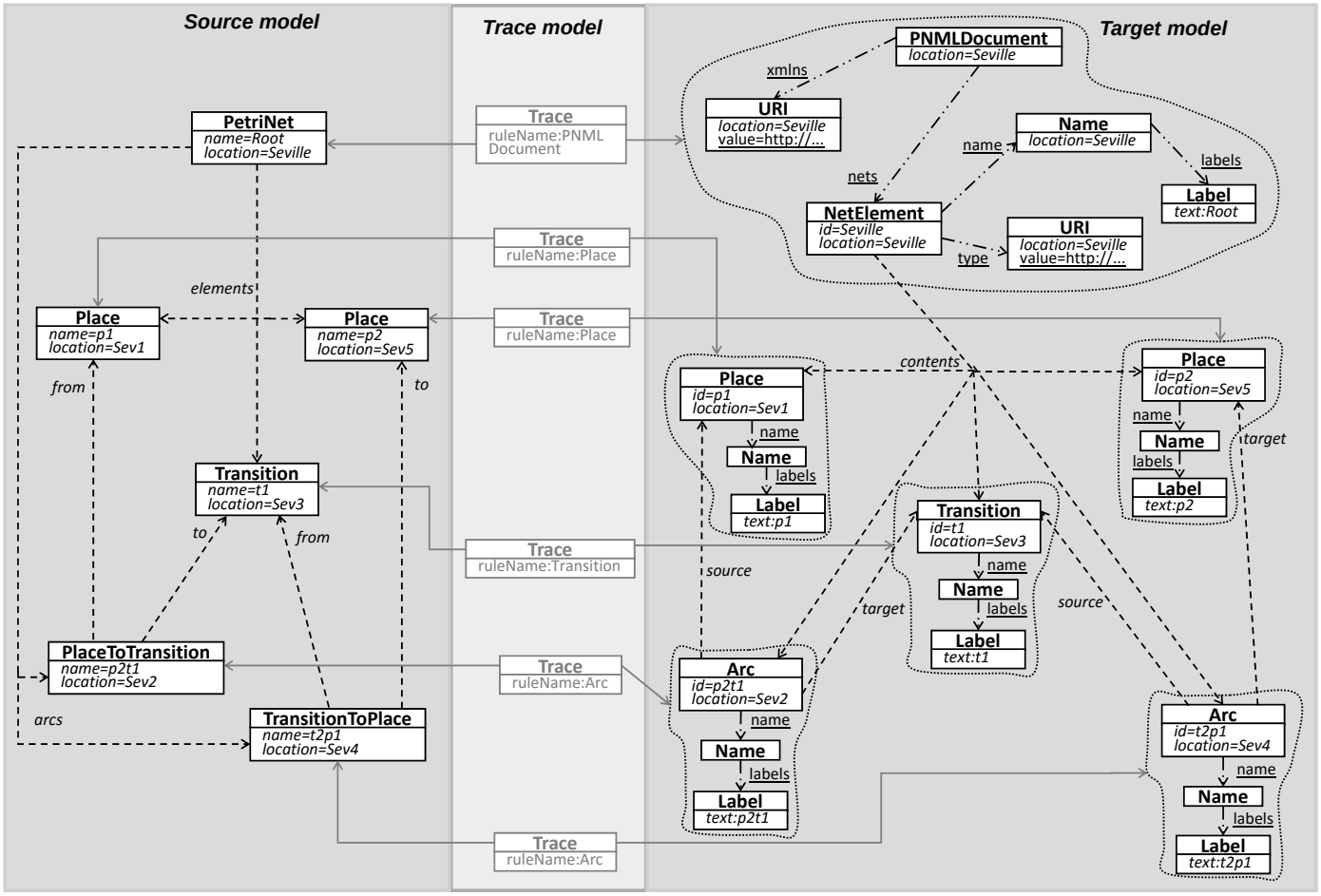


Figure 10: Source, trace and target models for the *PetriNet2PNMML* example and traces patterns instantiated

```
3 SE=PetriNet TE=Label sa=name sav=Root ta=text tav=Root
4 SE=PlaceToTransition TE=Arc sa=name sav=p2t1 ta=id tav=p2t1
5 SE=PlaceToTransition TE=Arc sa=location sav=Sev2 ta=location tav=Sev2
```

***PatternTR_3*: target element with constant value creation from source element.** This pattern is similar to the previous one. A target element is created and the value of an attribute, *tca*, is initialized. However, differently from before, now the value acquired by the attribute does not depend on any attribute from the source element, since the value is a constant value, *tcav*. This pattern is shown in Figure 11(c), and the feature specific for this pattern, namely the attribute in the target element, is underlined. The two instantiations of this pattern in the execution of our running example (cf. Figure 10) are shown in Listing 5. Thus, in the first one we see that a *URI* is created from a *PetriNet* and acquires in its *value* attribute the string "*http...*".

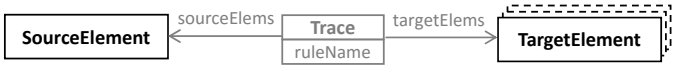
Listing 5: Trace Patterns instantiated for *Pattern_TR3* in *PetriNet2PNML*

```
1 SE=PetriNet TE=URI tca=value tcav=http://www.informatik.hu-berlin.de/top/pnml/
    ptNetb
2 SE=PetriNet TE=URI tca=value tcav=http://www.informatik.hu-berlin.de/top/pntd/
    ptNetb
```

***PatternTR_4*: target element with outgoing references creation from source element.** In this pattern, a target element is also created from a source element, so it includes *PatternTR_1*. Now, the new element created points to other elements in the model, what reflects the creation of outgoing references (*trgRef*). The target elements to which the new references point may have been created by the same rule or by different rules. The cardinalities of the reference can be of four types: *[0..1]*, *[1..1]*, *[0..*]*, *[1..*]*[3]. We consider the four types of cardinalities in the pattern. Depending on the type, the likely DIMRs created from this trace will vary (cf. Section 3.3). This pattern is depicted in Figure 11(d), where the features specific for this pattern, namely the outgoing references (*trgRef*), are depicted with lines composed of small lines and dots. The names and multiplicities of the references are underlined. Our approach finds 38 occurrences of this pattern in our case study. In Listing 6 we show four of them, where *trgRef* indicates the outgoing target references and where we also indicate the multiplicity of the created reference. These multiplicities are defined in the PNML metamodel (cf. Figure 9). For instance, the second instantiation indicates that a *Name* is created from a *PetriNet*, and that it may contain from none to many *labels* as output reference.

Listing 6: Some Trace Patterns instantiated for *Pattern_TR4* in *PetriNet2PNML*

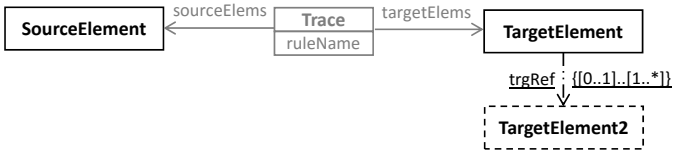
[3] As it is the case in most metamodels, in this version we only consider lower bounds to be 0 or 1 and upper bounds to be 1 or *



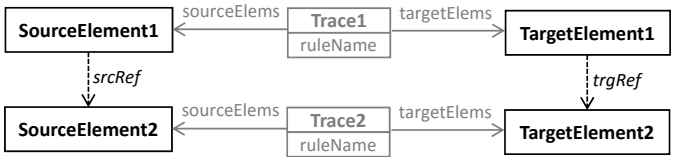
(a) *PatternTR_1*: TE creation from SE.

(b) *PatternTR_2*: TE creation from SE and attribute initialization.

(c) *PatternTR_3*: TE with constant value creation from SE.



(d) *PatternTR_4*: TE with outgoing references creation from SE.



(e) *PatternTR_5*: target reference creation from source reference.

Figure 11: Domain-Independent Trace Patterns (cf. Figure 7)

```
1 SE=PetriNet TE=PNMLDocument trgRef=xmlns multiplicity=[1..1]
2 SE=PetriNet TE=Name trgRef=labels multiplicity=[0..*]
3 SE=Transition TE=Transition trgRef=name multiplicity=[0..1]
4 SE=TransitionToPlace TE=Name trgRef=labels multiplicity=[0..*]
```

***PatternTR_5*: reference creation in the target model from reference in the source model.** This pattern considers the existence of two occurrences of *PatternTR_1* in the trace model. Besides, the source element of the first occurrence of *PatternTR_1* references (*srcRef*) the source element of the second occurrence of *PatternTR_1*, and same thing with the target elements (*trgRef*). The pattern is shown in Figure 11(e), and the references, which are specific for this pattern, are depicted with constant dotted lines. The names and cardinalities of the references are written in italics. There are 2 occurrences of this pattern in our case study, shown in Listing 7. *SE1* and *TE1* indicate the source element and target element of the first trace. *SE2* and *TE2* represent the same but for the second trace. As for the references, *srcRef* is the reference from *SE1* to *TE1* and *trgRef* is the one from *SE2* to *TE2*. Therefore, the first instantiation in the listing specifies that an *Arc* is created from a *PlaceToTransition* and that a *Transition* is created from a *Transition*. Furthermore, in the source model, the *PlaceToTransition* points, with the *to* reference, the *Transition*, while in the target model, the *Arc* points the *Transition* with the *target* reference.

Listing 7: Some Trace Patterns instantiated for *Pattern_TR5* in *PetriNet2PNML*

```
1 SE1=PlaceToTransition SE2=Transition srcRef=to TE1=Arc TE2=Transition
    trgRef=target
2 SE1=TransitionToPlace SE2=Transition srcRef=from TE1=Arc TE2=Transition
    trgRef=source
```

3.3. Automated Inference of Likely Metamorphic Relations

From every *Domain-Independent Trace Pattern* (cf. Figure 7), we derive a set of *likely DIMRs*, and group them all in an extensible *Catalogue of likely DIMRs*. They are abstractly defined, meaning they are to be instantiated in specific model transformations. In fact, when the patterns of the traces are instantiated in a specific model transformation (*Trace Patterns instantiated*), the MRs get also instantiated (*Specific likely MRs*). In order for the generated MRs to be formally expressed and processable by testing tools, we generate them in the well-known Object Constraint Language (OCL) [78].

For the definition of metamorphic relations in the context of model transformations, we keep in mind the schema shown in Figures 5 and 6. In this context, a metamorphic input relation (MR_{IP}) defines a change that is performed in the source test case input (SM) in order to produce the follow-up test case input (fuSM). Typically, when applying metamorphic testing in model transformations, a MR_{IP} defines the addition/deletion of an element of a certain type, together with (optionally) the addition/deletion/modification of some attributes or references, in fuSM with respect to SM. In our approach, the MRs_{IP} are expressed with a sentence in natural language, for instance, "An element of type Place is added in fuSM". Then, the metamorphic output relations (MRs_{OP}) typically compare the content of the target model (TM) with the content of the follow-up target model (fuTM). A pair $[MR_{IP}, MR_{OP}]$ constitutes a metamorphic relation. A MR_{IP} can be part of several MRs, meaning that several MRs_{OP} may share the same MR_{IP}. Each MR_{OP} expresses a condition that must hold for the model transformation to be correct, provided the specified change has been performed in fuSM with respect to SM as specified by MR_{IP}. The MRs_{OP} are expressed in OCL, where we add "TM_" or "fuTM_" before the name of a class to specify whether we are referring to the elements of that class in the target model or in the follow-up target model, respectively.

In the following, we explain each of the 24 domain-independent metamorphic relations (DIMRs) that compose our *Catalogue of likely DIMRs* (cf. Figure 7). They are also shown on our website [77]. For clarification purposes, they are conceptually classified in ten patterns, and they have been named hierarchically. We explain each of the DIMRs defined, and show some instances (*Specific likely MRs*) for our running example, namely the *PetriNet2PNML* model transformation. Several DIMRs and specific MRs are shown in different listings. In those listings showing DIMRs (Listings 8, 10, 12, 14, 16, 18, 19, 21, 23, 25), DIMRs are enumerated according to the pattern they belong to. In the listings that show examples of specific MRs (Listings 9, 11, 13, 15, 17, 20, 22, 24, 26), MRs are enumerated according to their position in the listing, and it is also indicated between brackets the DIMR they are instantiating. All the MRs inferred for the *PetriNet2PNML* case study, as well as for several other model transformations (cf. Section 4), are shown on our website [77].

***PatternMR_1*: addition of a SourceElement.** The MRs in this pattern are to be produced if *PatternTR_1* (cf. Figure 11(a)) is

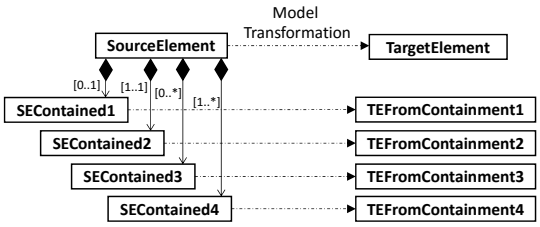


Figure 12: Possible containment associations (metamodel level)

found in the traces. The domain-independent metamorphic relations that conform this pattern are shown in Listing 8[4]. The MR_{IP} defines the addition of an element of type *SourceElement* in fuSM. The consequence is that *NE* elements of type *TargetElement* are created after executing the model transformation in futTM, as specified by the pattern *PatternTR_1* in the traces (cf. Figure 11(a) and Section 3.2). This means that fuTM has *NE* more elements of this type than TM, as it is shown by DIMR_1_1 in the listing.

Furthermore, the element that has been added in fuSM may have containment relationships with other elements that, in turn, may act as input elements in the model transformation in order to generate other output elements (cf. Figure 12). In order to maintain the conformance relationship with the source metamodel in fuSM, we consider that said contained elements are also added in fuSM whenever the lower bound of the association is bigger than 0, specifically:

- If the multiplicity of the association is *[1..1]*, then one element of the generic type *SEContained2* is created in fuSM, what yields DIMR_1_2 in Listing 8.
- If the multiplicity is *[1..*]*, one or more new elements of type *SEContained4* are created in fuSM, expressed by DIMR_1_3.

Finally, the number of instances of those elements that are neither created from *SourceElement* nor from any of its contained classes should remain the same (*AnyOtherType*), as expressed by DIMR_1_4. In order to determine which types are *AnyOtherType*, all classes in the target metamodel are gathered and those that do not appear in a trace as target element for *SourceElement* nor for any of its contained classes are considered as *AnyOtherType*.

Listing 8: Likely DIMRs in *PatternMR_1* of the Catalogue

```
An element of type SourceElement is added in fuSM.
DIMR_1_1: TM_TargElement.allInstances()->size()=fuTM_TargElement.allInstances()
  ->size()-NE
DIMR_1_2: TM_TEFromContainment2.allInstances()->size()=fuTM_TEFromContainment2.
  allInstances()->size()-1
DIMR_1_3: TM_TEFromContainment4.allInstances()->size()<fuTM_TEFromContainment4.
  allInstances()->size()
DIMR_1_4: TM_AnyOtherType.allInstances()->size()=fuTM_AnyOtherType.allInstances
  ()->size()
```

Example. As we see in Figure 10 and in Listing 3, two *URIs* are created from a *PetriNet*. At the same time, no *Arc* is created from a *PetriNet*. We also observe that a *Place*, a *Name* and a *Label* are always created from a *Place*. Therefore, we have the MRs shown in Listing 9. MR1, MR3, MR4 and MR5 are instances of DIMR_1_1, while MR2 is an instance of DIMR_1_4.

Listing 9: Specific MRs for *PatternMR_1* in *PetriNet2PNML*

```
An element of type PetriNet is added in fuSM
MR1 (DIMR_1_1): TM_URI.allInstances()->size()=fuTM_URI.allInstances()->size()-2
MR2 (DIMR_1_4): TM_Arc.allInstances()->size()=fuTM_Arc.allInstances()->size()
An element of type Place is added in fuSM
MR3 (DIMR_1_1): TM_Place.allInstances()->size()=fuTM_Place.allInstances()->size()-1
MR4 (DIMR_1_1): TM_Name.allInstances()->size()=fuTM_Name.allInstances()->size()-1
MR5 (DIMR_1_1): TM_Label.allInstances()->size()=fuTM_Label.allInstances()->size()-1
```

***PatternMR_2*: deletion of a SourceElement.** The MRs in this pattern are to be produced if *PatternTR_1* (cf. Figure 11(a)) is found in the traces. The DIMRs defined in this pattern are shown in Listing 10. The MR_{IP} defines the deletion of an element of type *SourceElement* in fuSM. The effect is that *NE* elements of type *TargetElement* that were created in TM after executing the model transformation are not created now in fuTM. This means that fuTM has *NE* less elements of this type than TM, as it is shown in DIMR_2_1 in the listing. Additionally, the element deleted may have been containing other elements when it was deleted. We take this into consideration for generating several DIMRs. Specifically, we consider the four types of multiplicities considered in this paper and shown in Figure 12:

- If it is *[0..1]*, the deleted element may have either contained another element or not. If such containment association is found, then DIMR_2_2 holds.
- If it is *[1..1]*, the deleted element contained an element, what yields DIMR_2_3.
- If it is *[0..*]*, the deleted element may have contained from none to several elements. This is the reason why the symbol >= is used in DIMR_2_4.
- Finally, if the multiplicity is *[1..*]*, the deleted element contained at least one element, so DIMR_2_5 must hold.

Finally, the number of instances of those elements that are neither created from *SourceElement* nor from any of its contained classes should remain the same (*AnyOtherType*), as expressed by DIMR_2_6.

Listing 10: Likely DIMRs in *PatternMR_2* of the Catalogue

```
An element of type SourceElement is deleted in fuSM.
DIMR_2_1: TM_TargElement.allInstances()->size()=fuTM_TargElement.allInstances()->
  size()+NE
DIMR_2_2: TM_TEFromContainment1.allInstances()->size()=fuTM_TEFromContainment1.
  allInstances()->size() or TM_TEFromContainment1.allInstances()->size()=
  fuTM_TEFromContainment1.allInstances()->size()+1
DIMR_2_3: TM_TEFromContainment2.allInstances()->size()=fuTM_TEFromContainment2.
  allInstances()->size()+1
DIMR_2_4: TM_TEFromContainment3.allInstances()->size()>=fuTM_TEFromContainment3.
  allInstances()->size()
DIMR_2_5: TM_TEFromContainment4.allInstances()->size()>fuTM_TEFromContainment4.
  allInstances()->size()
DIMR_2_6: TM_AnyOtherType.allInstances()->size()=fuTM_AnyOtherType.allInstances()->
  size()
```

Example. As we see in the *PetriNet* metamodel (cf. Figure 8), a *PetriNet* has a containment association, *elements*, with multiplicity *[0..*]* with *Transition*. At the same time, we observe in Figure 10 and in Listing 3 that a *Name* is created from a *PetriNet*. Furthermore, a *Transition*, a *Name* and a *Label* are created from a *Transition*. All this information yields the MRs

[4]In this and remaining listings, each MR is divided in two parts. First, the MR_{IP} is specified, followed by the MRs_{OP} that share the same MR_{IP}.

shown in Listing 11. There, MR1 is an instance of DIMR_2_5; MR2 and MR3 are instances of DIMR_2_4; and MR4, MR5 and MR6 are instances of DIMR_2_3.

Listing 11: Specific MRs for *PatternMR_2* in *PetriNet2PNML*

```
An element of type PetriNet is deleted in fuSM
MR1 (DIMR_2_5): TM_Name.allInstances()->size()>fuTM_Name.allInstances()->size()
MR2 (DIMR_2_4): TM_Label.allInstances()->size()>=fuTM_Label.allInstances()->size()
MR3 (DIMR_2_4): TM_Transition.allInstances()->size()>=fuTM_Transition.allInstances()
  ->size()
An element of type Transition is deleted in fuSM
MR4 (DIMR_2_3): TM_Transition.allInstances()->size()=fuTM_Transition.allInstances()
  ->size()+1
MR5 (DIMR_2_3): TM_Name.allInstances()->size()=fuTM_Name.allInstances()->size()+1
MR6 (DIMR_2_3): TM_Label.allInstances()->size()=fuTM_Label.allInstances()->size()+1
```

***PatternMR_3*: addition of an element with a specific attribute.** The MRs in this pattern are to be produced if *PatternTR_2* (cf. Figure 11(b)) is found in the traces. The metamorphic input relation defines the addition of an element of type *SourceElement* in fuSM, where the attribute *sa* of the element has been set to *sav*. When we execute the transformation, it creates an element of type *TargetElement* in fuTM and initializes its attribute *ta* with value *tav*. In this pattern, attribute *ta* is initialized with the value of attribute *sa*. We consider the basic types Integer and String. For the Integer type, we only consider the case where the value is copied, whereas for String we consider that the value of *tav* can be created in four ways:

1. It is copied from *sav*.
2. It is formed by a string concatenated with the value of *sav*.
3. It is formed by the value of *sav* concatenated with a string.
4. It is formed by a string concatenated with the value *sav* and yet concatenated with another string.

This pattern is a specialization of *PatternMR_1*, meaning that the MR_{IP} defined now implies the one defined in *PatternMR_1*, so the DIMRs generated in *PatternMR_1* are also valid here. Additionally, the DIMR in Listing 12 must hold. It states that in fuTM there is one more element of type *TargetElement* with value *tav* in attribute *ta* than in TM.

Listing 12: Likely DIMR in *PatternMR_3* of the Catalogue

```
An element of type TargetElement is added in fuTM and its attribute ta is
  initialized with value tav
DIMR_3_1: TM_TargetElement.allInstances()->select(te|te.ta=tav)= fuTM_TargetElement.
  allInstances()->select(te|te.ta=tav)-1
```

Example. According to the first, second and third examples of *PatternTR_2* shown in Listing 4, a *PetriNet* creates a *NetElement*, and the value of the *location* attribute of the former is copied into the *id* and *location* attributes of the latter. Furthermore, a *Label* is also created from a *PetriNet*, and its *text* attribute acquires the value of the *name* of the *PetriNet*. This generates the following MRs, all of them instances of DIMR_3_1:

Listing 13: Specific MRs for *PatternMR_3* in *PetriNet2PNML*

```
An element of type PetriNet is added in fuSM with its attribute location set to ''
  value''
MR1 (DIMR_3_1): TM_NetElement.allInstances()->select(te|te.id=''value'')=
  fuTM_NetElement.allInstances()->select(te|te.id=''value'')-1
MR2 (DIMR_3_1): TM_NetElement.allInstances()->select(te|te.location=''value'')=
  fuTM_NetElement.allInstances()->select(te|te.location=''value'')-1
An element of type PetriNet is added in fuSM with its attribute name set to  ''
  value''
MR3 (DIMR_3_1): TM_Label.allInstances()->select(te|te.text=''value'')= fuTM_Label.
  allInstances()->select(te|te.text=''value'')-1
```

***PatternMR_4*: deletion of an element with a specific attribute.** The MRs in this pattern are to be produced if *PatternTR_2* (cf. Figure 11(b)) is found in the traces. The metamorphic input relation in this pattern defines the deletion of an element of type *SourceElement* in fuSM, where its attribute *sa* had the value *sav*. If we execute the transformation the element of type *TargetElement* that was created from such element in TM and whose attribute *ta* had a value of *tav* will not be created in fuTM. In this pattern, attribute *ta* was initialized with the value of attribute *sa*. We consider the same cases as in *PatternMR_3* regarding the values of the attributes.

This pattern is a specialization of *PatternMR_2*, meaning that the MR_{IP} defined now implies the one defined in *PatternMR_2*, so the DIMRs generated in *PatternMR_2* are also valid here. Additionally, the DIMR in Listing 14 is produced. It states that in fuTM there is one less element of type *TargetElement* with value *tav* in attribute *ta* than in TM.

Listing 14: Likely DIMR in *PatternMR_4* of the Catalogue

```
An element of type TargetElement is deleted in fuTM, whose ta had the value tav
DIMR_4_1: TM_TargetElement.allInstances()->select(te|te.ta=tav)= fuTM_TargetElement.
  allInstances()->select(te|te.ta=tav)+1
```

Example. According to the third and fourth examples of *PatternTR_2* shown in Listing 4, an *Arc* is created from a *PlaceToTransition*, and the *id* attribute of the former is initialized with the value of the *name* attribute of the latter. Furthermore, the *location* attribute of the *Arc* is set with the value of the *location* attribute of the *PlaceToTransition*. This generates the following MRs, all of them instances of DIMR_4_1.

Listing 15: Specific MRs for *PatternMR_4* in *PetriNet2PNML*

```
An element of type PlaceToTransition is deleted in fuSM with its attribute name set
  to ''value''
MR1 (DIMR_4_1): TM_Arc.allInstances()->select(te|te.id=''value'')= fuTM_Arc.
  allInstances()->select(te|te.id=''value'')+1
An element of type PlaceToTransition is deleted in fuSM with its attribute location
  set to ''value''
MR2 (DIMR_4_1): TM_Arc.allInstances()->select(te|te.location=''value'')= fuTM_Arc.
  allInstances()->select(te|te.location=''value'')+1
```

***PatternMR_5*: addition of an element that will create a constant attribute.** The MR in this pattern is to be produced if *PatternTR_3* (cf. Figure 11(c)) is found in the traces. In this pattern, the MR_{IP} is the same as in *PatternMR_1*: it consists in adding an element of type *SourceElement* in fuSM. When we execute the transformation, it creates an element of type *TargetElement* in fuTM and initializes its attribute *tca* with the constant value *tcav*. This means that it does not matter how the element of type *SourceElement* added in fuSM is or how its attributes are. The attribute *tca* will always be given the same value. We consider the basic types Integer and String.

This pattern is a specialization of *PatternMR_1*, meaning that the DIMRs generated in *PatternMR_1* are also valid now. Besides, the DIMR in Listing 16 must hold. It states that in fuTM there is one more element of type *TargetElement* with value *tcav* in attribute *tca* than in TM.

Listing 16: Likely DIMR in *PatternMR_5* of the Catalogue

```
An element of type SourceElement is added in fuSM
DIMR_5_1: TM_TargetElement.allInstances()->select(te|te.tca=tcav)=
  fuTM_TargetElement.allInstances()->select(te|te.tca=tcav)-1
```

Example. According to the examples of *PatternTR_3* shown in Listing 5, two *URIs* are created from a *PetriNet*, where their *value* attributes acquire a constant value. This generates the following MRs, which are instances of DIMR_5_1:

Listing 17: Specific MRs for *PatternMR_5* in *PetriNet2PNML*

```
An element of type PetriNet is added in fuSM
MR1 (DIMR_5_1): TM_URI.allInstances()->select(te|te.value="http://www.informatik.hu-
  berlin.de/top/pnml/ptNetb")->size()=fuTM_URI.allInstances()->select(te|te.value="
  http://www.informatik.hu-berlin.de/top/pnml/ptNetb")->size()-1
MR2 (DIMR_5_1): TM_URI.allInstances()->select(te|te.value="http://www.informatik.hu-
  berlin.de/top/pnml/ptNetb/Seville")->size()=fuTM_URI.allInstances()->select(te|te
  .value="http://www.informatik.hu-berlin.de/top/pnml/ptNetb/Seville")->size()-1
```

***PatternMR_6*: deletion of an element that created a constant attribute.** The MR in this pattern is to be produced if *PatternTR_3* (cf. Figure 11(c)) is found in the traces. In this pattern, the MR_{IP} is the same as in *PatternMR_2*: it consists in removing an element of type *SourceElement* in fuSM. Thereby, after executing the transformation, the element of type *TargetElement* that was created in TM and whose attribute *tca* was initialized with the constant value *tcav* is not created in fuTM. This means that it does not matter how the element of type *SourceElement* removed in fuSM was or how its attributes were. The attribute *tca* would have always been given the same value. We consider the basic types Integer and String.

This pattern is a specialization of *PatternMR_2*, meaning that the DIMRs generated in *PatternMR_2* are also valid now. Besides, the DIMR in Listing 18 is produced. It states that in fuTM there is one less element of type *TargetElement* with value *tcav* in attribute *tca* than in TM.

Listing 18: Likely DIMR in *PatternMR_6* of the Catalogue

```
An element of type SourceElement is deleted in fuSM
DIMR_6_1: TM_TargetElement.allInstances()->select(te|te.tca=tcav)=
  fuTM_TargetElement.allInstances()->select(te|te.tca=tcav)+1
```

Example. The same MRs as in the pattern before are generated, but with the symbol '+' instead of '-'. As for its MR_{IP}, it is "An element of type PetriNet is deleted in fuSM".

***PatternMR_7*: addition of an element that will create references.** The MRs in this pattern are to be produced if *PatternTR_4* (cf. Figure 11(d)) is found in the traces. In this pattern, the MR_{IP} is the same as in *PatternMR_1*: it consists in adding an element of type *SourceElement* in fuSM. When we execute the transformation, it creates an element of type *TargetElement* in fuTM together with its outgoing references, *trgRef*, that will point to other elements in the model. We consider four types of cardinality, as explained in Section 3.2: *[0..1]*, *[1..1]*, *[0..*]*, *[1..*]*. This means that more than one reference of the same type can be created in the target model.

This pattern is a specialization of *PatternMR_1*, meaning that the DIMRs inferred in *PatternMR_1* are also valid now. Besides, the DIMRs in Listing 19 must hold, depending on the cardinality of the reference:

- If it is *[0..1]*, the element created may reference none or one element, so DIMR_7_1 must hold.
- If the cardinality is *[0..*]*, the element created may reference none or any number of elements, what yields DIMR_7_2.
- If it is *[1..*]*, it references one or more elements, so DIMR_7_3 must hold.
- Finally, if the cardinality is *[1..1]*, it will reference exactly one element, yielding DIMR_7_4.

Listing 19: Likely DIMRs in *PatternMR_7* of the Catalogue

```
An element of type SourceElement is added in fuSM
DIMR_7_1: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size() or
  TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
  fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()-1
DIMR_7_2: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
  <= fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
DIMR_7_3: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()<
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
DIMR_7_4: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()-1
```

Example. According to the examples of *PatternTR_4* shown in Listing 6, the *PNMLDocument* and *Name* created from a *PetriNet* have the outgoing references *xmlns* and *labels*, respectively, with multiplicities *[1..1]* and *[0..*]*. This produces the MRs in listing 20, where MR1 is an instance of DIMR_7_4 and MR2 is an instance of DIMR_7_2.

Listing 20: Specific MRs for *PatternMR_7* in *PetriNet2PNML*

```
An element of type PetriNet is added in fuSM
MR1 (DIMR_7_4): TM_PNMLDocument.allInstances()->collect(c|c.xmlns)->flatten()->size
  ()=fuTM_PNMLDocument.allInstances()->collect(c | c.xmlns)->flatten()->size()-1
MR2 (DIMR_7_2): TM_Name.allInstances()->collect(c|c.labels)->flatten()->size()<=
  fuTM_Name.allInstances()->collect(c|c.labels)->flatten()->size()
```

***PatternMR_8*: deletion of an element that created references.** The MRs in this pattern are to be produced if *PatternTR_4* (cf. Figure 11(d)) is found in the traces. In this pattern, the MR_{IP} is the same as in *PatternMR_2*: it consists in removing an element of type *SourceElement* in fuSM. When we execute the transformation, the element of type *TargetElement* that was created in TM together with its outgoing references, *trgRef*, that pointed to other elements in the model are not created in fuTM. Again, the cardinalities of the references created in TM can be of the four types mentioned before. This means that, depending on them, a certain number of references that were created in TM are not created in fuTM. Following the same schema as before, we infer the DIMRs shown in Listing 21. Additionally, the DIMRs generated in *PatternMR_2* are also valid now, since this pattern is a specialization of *PatternMR_2*.

Listing 21: Likely DIMRs in *PatternMR_8* of the Catalogue

```
An element of type SourceElement is deleted in fuSM
DIMR_8_1: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size() or
  TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
  fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()+1
DIMR_8_2: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
  >= fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
DIMR_8_3: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()>
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()
DIMR_8_4: TM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()=
    fuTM_TargetElement.allInstances()->collect(te|te.trgRef)->flatten()->size()+1
```

Example. According to the third and fourth examples of *PatternTR_4* shown in Listing 6, a *Transition* produces a *Transition* with outgoing reference *name* and multiplicity *[0..1]*, and a *TransitionToPlace* generates a *Name* with outgoing reference *labels* and multiplicity *[0..*]*. This produces the MRs in Listing 22, where MR1 is an instance of DIMR_8_1 and MR2 is an instance of DIMR_8_2.

Listing 22: Specific MRs for *PatternMR_8* in *PetriNet2PNML*

```
An element of type Transition is deleted in fuSM
MR1 (DIMR_8_1): TM_Transition.allInstances()->collect(c|c.name)->flatten()->size()=
  fuTM_Transition.allInstances()->collect(c|c.name)->flatten()->size()
  orTM_Transition.allInstances()->collect(c|c.name)->flatten()->size()=
  fuTM_Transition.allInstances()->collect(c|c.name)->flatten()->size()+1
An element of type TransitionToPlace is deleted in fuSM
MR2 (DIMR_8_2): TM_Name.allInstances()->collect(c|c.labels)->flatten()->size()>=
  fuTM_Name.allInstances()->collect(c|c.labels)->flatten()->size()
```

***PatternMR_9*: addition of an element with an incoming reference.** The MRs in this pattern are produced if *PatternTR_5* (cf. Fig. 11(e)) is found in the traces. In the scenario seen in the figure, the MR_{IP} in this pattern consists in adding an element of type *SourceElement2* in fuSM. This element has the incoming reference *srcRef* coming from an element of type *SourceElement1*. When the transformation is executed, it creates an element of type *TargetElement2* in fuTM according to *PatternTR_1*. Besides, an incoming reference to such element, *trgRef*, from an element of type *TargetElement1* is also created. The element of type *TargetElement1* from which the reference *trgRef* departs is the one created from *SourceElement1*. Apart from the DIMRs of *PatternMR_1*, the DIMR in Listing 23 is produced. It compares the number of references of type *trgRef* in TM and fuTM, since there must be one more in fuTM.

Listing 23: Likely DIMRs in *PatternMR_9* of the Catalogue

```
An element of type SourceElement2 is added in fuSM with an incoming srcRef
  reference from an element of type SourceElement1
DIMR_9_1: TM_TargetElement1.allInstances()->collect(te|te.trgRef)->flatten()->size()
  =fuTM_TargetElement1.allInstances()->collect(te|te.trgRef)->flatten()->size()-1
```

Example. According to the first example of *PatternTR_5* shown in Listing 7, when a *Transition* is added in fuSM with an incoming *to* reference from a *PlaceToTransition*, the generated *Transition* has an incoming *target* reference from an *Arc*. This produces the following MR, which is an instance of *DIMR*_9_1:

Listing 24: Specific MR for *PatternMR_9* in *PetriNet2PNML*

```
An element of type Transition is added in fuSM with an incoming to reference from
  an element of type PlaceToTransition
MR1 (DIMR_9_1): TM_Arc.allInstances()->collect(ct|ct.target)->flatten()->size()=
  fuTM_Arc.allInstances()->collect(ct|ct.target)->flatten()->size()-1
```

***PatternMR_10*: deletion of an element with an incoming reference.** The MR in this pattern is produced if *PatternTR_5* shown in Figure 11(e) is found in the traces. Therefore, the scenario is the same as the one described in the previous pattern. However, now, the MR_{IP} is defined by deleting an element of type *SourceElement2* in fuSM, which had an incoming reference *srcRef* from an element of type *SourceElement1*. Therefore, the *trgRef* reference that is created in TM will not be created now in fuTM. For this reason, apart from the DIMRs produced in *PatternMR_2*, the DIMR in Listing 25 is generated.

Listing 25: Likely DIMRs in *PatternMR_10* of the Catalogue

```
An element of type SourceElement2 is deleted in fuSM that had an incoming srcRef
  reference from an element of type SourceElement1
DIMR_10_1: TM_TargetElement1.allInstances()->collect(te|te.trgRef)->flatten()->size
  ()=fuTM_TargetElement1.allInstances()->collect(te|te.trgRef)->flatten()->size()+1
```

Example. According to the second example of *PatternTR_5* shown in Listing 7, when a *Transition* is added in fuSM with an incoming *from* reference from a *Transition*, the generated *Transition* has an incoming *source* reference from an *Arc*. This produces the following MR, which is an instance of DIMR_10_1:

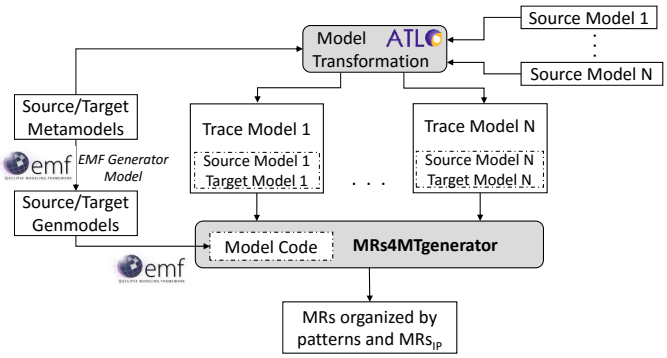


Figure 13: MRs4generator and its inputs/output.

Listing 26: Specific MR for *PatternMR_10* in *PetriNet2PNML*

```
An element of type Transition is deleted in fuSM that had an incoming from
  reference from an element of type TransitionToPlace
MR1 (DIMR_10_1): TM_Arc.allInstances()->collect(ct|ct.source)->flatten()->size()=
  fuTM_Arc.allInstances()->collect(ct|ct.source)->flatten()->size()+1
```

3.4. Considerations and Automation

We have applied a small granularity when defining the domain-independent metamorphic relations (DIMRs) of our catalogue, as it can be observed by having a look at the MRs inferred for the different case studies, available on our website [77]. For this reason, we have tried to define a single DIMR for each error that can be committed in the three scenarios targeted by our approach, namely regression testing, incremental transformations and migration of transformations to other languages. By doing this, we try to maximize the number and types of errors that can be detected. Furthermore, we have separated similar metamorphic input relations (MR_{IP}) into different patterns. For instance, the MR_{IP} "An element of type Transition is added in fuSM with its attribute location set to value" implies the MR_{IP} "An element of type Transition is added in fuSM". However, they are separated in different patterns. Again, this helps find the cause of the errors a transformation may contain. For instance, if a certain MR fails and it contains the second MR_{IP}, one or more MRs that have as MR_{IP} the first one are likely to fail as well. On the contrary, if a MR that has as MR_{IP} the first one fails but no MR containing the second MR_{IP} fails, it may be due to the information added with respect to the second MR_{IP} –namely information related to the attributes–, making it easier to identify the specific problem.

Let us clarify something at this point. Typically, and as it happens in our running example, a metamodel has a root class that contains the rest of classes in the metamodel. For instance, the PetriNet metamodel shown in Figure 8 has *PetriNet* as root class. In the models that instantiate a metamodel and in order to satisfy the conformance relationship, there is typically only one element of the root class, and this element contains the rest of the elements. However, in the Eclipse Modeling Framework (EMF) [79] it is possible to create and visualize models that contain more than one root elements. Furthermore, there are some metamodels in the transformations of the ATL Zoo [52] that do not have a root class. For this reason, in the MRs that are inferred for a specific model transformation, we have included

those that imply the addition/deletion of a root element, in our running example *PetriNet*.

We have automated the inference of metamorphic relations for any model transformation by means of a Java program that we have called *MRs4MTgenerator*, and that is shown together with its inputs and output in Figure 13. As we can see, there are several inputs to our program. Of course, we need the ATL *Model Transformation* from which we will infer MRs, although it is not a direct input to our program. The model transformation takes as input the *Source/Target Metamodels* as well as the *Source Model*. In fact, it can take as input several source models in order to perform several executions. After each execution, it produces an instance of a *Trace Model* (trace model 1, trace model 2, ..., trace model N), represented by the first step in Figure 7. As explained earlier in the paper (cf. Section 2.2.2), the trace model produced by a model transformation points, through inter-model references, to the source and target models. Said trace models, together with the source and target models they point to, are inputs to our program. They are needed in order to identify and instantiate the trace patterns presented in Section 3.2 and represented by the second step in Figure 7.

Furthermore, in order for our program to be able to navigate the models, it needs knowledge of the metamodels they conform to. This is why the *Model Code* is necessary in our program. It is Java code that represents the metamodels. This model code is generated automatically by the EMF tools and integrated in our program as packages and classes. This is done in two steps. First, the so-called *Genmodels* are generated by the *EMF Generator Model* from the metamodels. Second, the model code is automatically extracted from them.

After having instantiated the patterns in the traces, our program infers and prints the MRs specified as OCL expressions, represented by the third step in Figure 7. Each MR inferred is an instance of one of the 24 DIMRs that our catalogue contains. The inferred MRs are printed as they are shown on our webpage [77]: they are organized according to the patterns described in Section 3.3 and, within each pattern, they are classified by different metamorphic input patterns. Therefore, those MRs that share the same MR_{IP} are printed together.

Please note that by not having the ATL model transformation as input to our program, it can be trivially extended to be used with any model transformation language. All we need is for the specific model transformation program to generate an execution trace model. Our *MRs4MTgenerator* is available from our website [77].

4. Evaluation

In this section we present an evaluation of our approach based on seven case studies. In particular, we are interested in answering the following research questions (RQs).

- **RQ1. Feasibility: Is is possible to automatically infer likely metamorphic relations for any model transformation with the user having no knowledge about the transformation?** Since, at the time of writing, there was no proposal to automatically infer metamorphic relations for model transformations, we want to answer if that is feasible.
- **RQ2. Precision: What is the precision of our inferred metamorphic relations?** Due to the fact that the inferred MRs are obtained automatically, without any user intervention, we want to study whether they are accurate and, therefore, can be applied in testing processes.
- **RQ3. Usefulness: Is our approach able to detect faults in regression testing?** In order to test if our approach can be usable in this context, we want to check whether it can identify mutants created with injected faults.

In order to answer RQ1 and RQ2, we use seven case studies that target different problem areas and differ in their level of complexity regarding number and types of features used (number of rules, use of imperative rules, filters, helpers...). The answer to RQ3 is given by producing mutants and several source test case inputs and follow-up test case inputs for one of the seven case studies, namely our running example, the *PetriNet2PNML* model transformation.

The rest of this section is structured as follows [80]. First, we describe the seven case studies used in the evaluation. Second, we define the evaluation metrics for answering RQ2 and RQ3. Third, we describe the execution environment and detail the process followed in this evaluation, which is composed of manual and automated tasks. Fourth, we present the results and the answer to the three RQs. Finally, we discuss some aspects of our approach and present the threats to the validity of this evaluation.

4.1. Case Studies

This section presents the seven case studies used to evaluate our approach and developed solution with respect to the RQs. They are seven model transformations, taken from the ATL Zoo repository [52], master courses and from the literature. They differ regarding the application domains, size of metamodels and in the number and types of features of ATL used. Table 1 summarizes some information regarding the transformations. We can see that the size of the metamodels vary from 6 to 15 classes in the input metamodels and from 5 to 48 classes in the output metamodels. As for the size of the transformations, the number of rules vary between 3 and 13 rules, and the lines of code (LoC) between 70 and 273 lines. Therefore, the smallest transformation is four times smaller than the biggest one. There are transformations using none, 1 or 2 helpers. We have also included further information, namely whether imperative rules, conditions and filters are used within the transformations. An explanation of the domains of each transformation is given in the following.

- **PetriNet2PNML**. This is our running example and is explained in detail in Section 3.1.
- **Class2Relational**. It is a simplified transformation of a class schema model to a relational database model, adapted from [81].

Table 1: Model transformations used as case studies and their characteristics

ID	Transformation Name	# Classes MM Input - Output	# LoC	# Rules	# Helpers	Imperative rules	Conditions	Filters
CS1	PetriNet2PNML	9 - 13	110	4	0	×	×	×
CS2	Class2Relational	6 - 5	100	7	1	×	×	✓
CS3	Grafcet2PetriNet	9 - 9	89	5	0	×	×	×
CS4	IEEE1471_2_MoDAF	14 - 48	229	13	1	×	✓	✓
CS5	ATOM2RSS	14 - 9	70	3	0	×	×	×
CS6	SOOML2SOOPL	15 - 10	273	10	2	×	×	×
CS7	Families2Persons Extended	11 - 12	111	10	0	✓	×	✓

- **Grafcet2PetriNet**. This transformation establishes a bridge between grafcet, a mainly French-based representation support for discrete systems, models and petri net models.
- **IEEE1471_2_MoDAF**. This is a conceptual transformation between IEEE1471 Conceptual Model, a terminology that defines views and viewpoints concepts about architectural descriptions, and MoDAF Architecture View, an architecture framework specified by the British Ministry of Defense that is based on the IEEE1471 terminology.
- **ATOM2RSS**. This transformation permits to get an RSS model from an ATOM model. RSS is a format for syndicating news and the content of news-like sites, including major news sites like Wired, news-oriented community sites like Slashdot, and personal web logs; while ATOM is an XML-based file format intended to allow lists of information, known as "feeds", to be synchronised between publishers and consumers.
- **SOOML2SOOPL**. This model transformation takes as input a model representing an Object Oriented Modeling Language and transforms it into a model representing an Object Oriented Programming Language. This ATL model transformation has been created by the Business Informatics Group of the Institute of Software Technology and Interactive Systems at the Vienna University of Technology (TU Wien). It is used as part of a Master Course.
- **Families2Persons_Extended**. This is an extended version of the original *Families2Persons* model transformation that can be found in the ATL Zoo and that has been discussed in a number of related works on verification and testing [82].

4.2. Evaluation Metrics

To obtain the precision of our approach and therefore to answer RQ2, we compute the precision measure originally defined in the area of information retrieval [83]. In the context of our study, precision denotes the fraction of *correctly inferred* MRs among the set of *all inferred* MRs. When we say that a MR is correctly inferred, we mean that this MR meets the specification of the transformation. In fact, the set of MRs automatically inferred with our approach is nothing but (part of) the specification of the model transformation. Actually, the formal specification of model transformations by means of OCL expressions has been proposed [84, 85, 86, 9], so our approach provides a way to automatically obtain a formal specification for model transformations based on metamorphic relations. Therefore, we assume that a MR is not correctly inferred when it does not meet any part of the specification of the transformation. In practical terms, this means that the MR will not hold for all possible SMs.

For computing precision, we manually classified the inferred likely MRs as true positives (TPs) and false positives (FPs). They have been manually classified by the authors of the paper, who are ATL experts and knowledgeable of the transformations used in the evaluation. It incurred an effort of around 1.5 persons-day. The fact that the MRs are classified and grouped according to their MR_{IP} facilitated this task. A MR is a true positive when it meets (part of) the specification of the model transformation. On the contrary, a MR is a false positive when it specifies something that does not meet the specification of the model transformation. From the number of true positives and false positives –$|TP|$ and $|FP|$–, we compute *precision* as follows:

$$precision = \frac{|TP|}{|TP| + |FP|} \quad (1)$$

In order to answer RQ3 we need to measure the usefulness of our approach. To address this, we have focused on our running example, namely the *PetriNet2PNML* case study. We want to test if the automatically inferred MRs are useful for detecting faults in regression testing. To this end, we have automatically generated 30 different source test case inputs (SM, cf. Figure 5), whose characteristics are described in detail on our webpage [77]. In short, the number of *PetriNets* in the models range from 1 to 3, and the number of *Places*, *Transitions*, *PlaceToTransition* and *TransitionToPlace* range from 0 to 15. In this way, we make sure that we maximize diversity among the different models and we ensure that all elements of the source metamodel are covered.

The next step is to create mutants of the *PetriNet2PNML* model transformation. The idea is that these mutants emulate semantic faults [87] that could be present when evolving the model transformation. In order to create them, we have applied

Table 2: Mutants generated for *PetriNet2PNML*

Mutant	Line number	Mutation operators
M1	56	1 binding deletion
M2	76 80-87	1 binding deletion 2 out-pattern element deletion
M3	100	1 binding value change
M4	33 44-47	1 binding value change 1 out-pattern element addition
M5	42	1 binding deletion
M6	95	1 filter addition
M7	29 44-47	1 binding value change 3 out-pattern element addition
M8	55-56 77-78 100-101	1 pair of binding features swapped 1 pair of binding features swapped 1 pair of binding features swapped
M9	29 93-113	1 binding value change 1 rule deletion
M10	29 95	1 binding value change 1 in-pattern element class change
M11	60 62-65 68-75	1 binding value change 1 out-pattern element deletion 1 rule addition
M12	11, 50, 72, 95	4 filter addition
M13	50, 72	2 in-pattern element class change
M14	29, 102, 103 113-120	3 binding value change 2 out-pattern element addition
M15	50 70-88 102, 103	1 in-pattern element class change 1 rule deletion 2 binding value change

several mutation operators presented in [88]. The 15 mutants produced are summarized in Table 2 and available on our website [77]. These mutants will be used for calculating the mutation score, i.e., the rate of mutants that are killed.

4.3. Execution Environment

All the runs have been executed on a PC running the 64-bits OS Windows 10 Pro with processor Intel Core i7-4770 @ 3.40GHz and 16 GB of RAM. We have used Eclipse Modeling Tools version Mars Release 2 (4.5.2), and we had to install the plugins ATL (we have used version 3.6.0) and ATL/EMFTVM (version 3.8.0). Finally, Java 8 is needed.

4.4. Evaluation Process

This section describes the execution steps needed to obtain the artifacts that have been used in the evaluation of our approach and gives information of the time taken to execute some automated tasks.

In order to answer RQ1 and RQ2, we needed to obtain the MRs for the seven case studies described before. As shown in Figure 13 and described in Section 3.4, our approach automatically infers MRs for any given model transformation. As input, our *MRs4MTgenerator* simply needs the result of at least one execution of the model transformation in terms of a trace

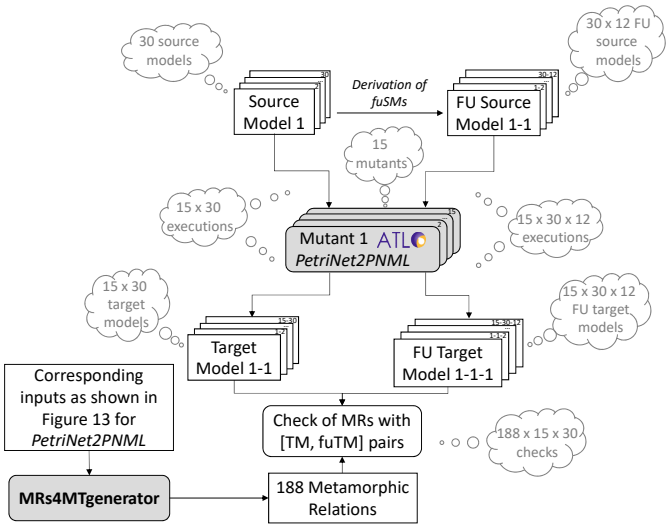


Figure 14: Evaluation Process for Usefulness Study.

model, as well as the Java code generated from the source and target metamodels, which is done by EMF. For the same model transformation with the same trace model(s) as input, our program always infers the same MRs. All the executions of our program for the seven case studies take less than 0.5 seconds in inferring and printing all the MRs organized by patterns and MRs_{IP}.

For answering RQ3, related to the usefulness of our approach, we have carried out a study with several mutants, source test case inputs and follow-up test case inputs. The process followed involves some automatic and some manual steps. The different artifacts in the different parts of the process are displayed in Figure 14.

As we can see in the upper part, we have 30 source models (SMs). They have been generated automatically by a simple Java program, whose execution has taken 0.03 seconds in producing the 30 models. For each of these models, 12 follow-up source models (fuSMs) have been generated. We have implemented a Java class with a method for generating each of the 12 fuSMs. The execution of all of them for all the 30 SMs is done all at once, taking 0.25 seconds.

We have also generated 15 mutants of the *PetriNet2PNML* model transformation described before, what has been done manually. The next step is to automatically execute all the SMs and fuSMs in each of the mutants to obtain all target models (TMs) and follow-up target models (fuTMs) needed. This means to perform $15 \times 30 + 15 \times 30 \times 12 = 5850$ executions, which is done all at once by a Java program that invokes all the transformations with the corresponding input models. The execution of this program takes 65.5 seconds.

For checking the MRs generated, we have used *OCLinEcore*[5]. 198 MRs are inferred for the *PetriNet2PNML* case study. Out of those, we have only taken into consideration those that are TPs, which are a total of 188 MRs (cf. Table 4). Since each MR has been checked for each mutant and each

[5] `https://wiki.eclipse.org/OCL/OCLinEcore`

Table 3: Execution Times in the Evaluation Study

	Task	Time (sec)
RQ1,2	Generation and printout of MRs	< 0.5
RQ3	Generation of 30 SMs	0.03
	Generation of 30×12 fuSMs	0.25
	Execution of $15 \times 30 + 15 \times 30 \times 12$ model transformations	65.5
	Execution of $188 \times 15 \times 30$ checks	71.3

Table 4: Results for precision

ID	# MRs inferred	# TPs	# FPs	# Precision
CS1	198	188	10	94.9%
CS2	60	49	11	81.7%
CS3	122	122	0	100%
CS4	1242	1228	14	98.9%
CS5	118	108	10	91.5%
CS6	256	235	21	91.8%
CS7	124	114	10	91.9%
Overall	4254	4101	153	**96.4%**

SM, this means we have $188 \times 15 \times 30 = 84600$ checks of metamorphic relations, done using *OCLinEcore* in 71.3 seconds.

The major effort has been made for classifying the MRs according to their scenario. Indeed, as we have explained in Sections 2 and 3, several metamorphic relations can share the same metamorphic input relation (MR_{IP}). Recall that a MR_{IP} is used to construct pairs [SM, fuSM], i.e., it represents the change that is performed in fuSM with respect to SM. In our running example, some sets of MRs out of the total of 188 MRs share the same MR_{IP}. Specifically, as described in detail on our website [77], there are 34 different MRs_{IP}. As explained in Section 3.4, some MRs_{IP} may imply others. For this reason and in order to minimize the number of executions for evaluating our approach, we have grouped the 34 MRs_{IP} in the 12 so-called *scenarios* mentioned above, so that, for each source test case input, we only have to define 12 MRs_{IP}, i.e., we only need 12 fuSMs, as detailed on our website [77].

A summary of the execution times of all automated tasks is displayed in Table 3.

4.5. Results

RQ1. Column *# MRs inferred* of Table 4 shows the number of MRs that are inferred for each model transformation. We can assert that our approach has achieved its target of automatically inferring metamorphic relations. In fact, we observe that a high number of them are produced. Our approach has been applied to seven model transformations from different domains and that vary in the size of the source/target metamodels and of the transformation (cf. Table 1). Furthermore, the tester does not need to have any knowledge of the transformations nor the transformation domain, since our approach is able to infer the MRs without any user intervention. It only needs the inputs shown in Figure 13.

This has been possible thanks to the definition of domain-independent metamorphic relations (DIMRs). These DIMRs reflect the behavior that model transformations must have if specific patterns occur in their traces, as explained through Section 3. In the next two RQs we study the precision and usefulness of the inferred MRs.

RQ2. We have manually checked the true positives (TPs) and false positives (FPs) of the metamorphic relations automatically inferred. The specific MRs that result in TPs and FPs are shown on our website [77]. The results are summarized in Table 4. We can see that our approach produces a high number of TPs, being the precision in one case study of 100% and the overal precision of 96.4%. Furthermore, some of the FPs are produced because the trace models that serve as input are not complete enough. If they had contained more traces, there would have likely been less FPs. This is explained in detail in Section 4.6. Summarizing, we can assert that our approach produces accurate metamorphic relations, although there is still room for improvement.

RQ3. We consider that a specific MR is violated for a specific mutant if it is violated, at least, in one out of the 30 source test case inputs. In Table 5 we show how many instances of each of the 24 domain-independent metamorphic relations of our catalogue are violated in each mutant. The second column indicates the number of instances for each domain-independent metamorphic relation that are inferred in the *PetriNet2PNML* model transformation.

The fact that the instances of a specific DIMR are never violated does not mean this DIMR is useless. The reason for the non violation is that we are utilizing a specific model transformation with 15 mutants. Should we have used a different example or different mutants, we would have gotten different results. In fact, the generic nature of our approach, where metamorphic relations are defined in an abstract way in order to be inferable for any model transformation, makes our approach applicable to any model transformation, meaning that, at the same time, it is not tailored at one particular model transformation. This is the reason why some DIMRs do not have any instances or why some instances are never violated for our particular running example.

In the table we can see that there are MRs violated in all mutants but one, mutant 5. We may clarify that mutant 5 is not an equivalent one, i.e., the TMs created by it will differ from those created by the original transformation for some SMs. This mutant is constructed by removing a binding of one of the *URIs* created by the model transformation. These results show that our approach is able to kill 14 out of 15 mutants, obtaining a mutation score of 93.3%. Therefore, we can conclude that our approach is useful for identifying errors in regression testing. Furthermore, due to the granularity used when defining the domain-independent metamorphic relations, as explained in Section 3.4, having a look at the MRs that have failed for a mutant helps identifying the reason for such failure. A detailed spreadsheet document showing the specific MRs that fail for

Table 5: Number of instances of the DIMRs violated in each transformation mutant of the *PetriNet2PNML* transformation

	# Inst	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	**Total**
DIMR_1_1	17	-	2	-	1	-	6	6	-	6	3	1	6	7	-	12	**50**
DIMR_1_2	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_1_3	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_1_4	23	-	-	-	-	-	-	3	-	-	-	-	-	2	2	4	**11**
DIMR_2_1	15	-	2	-	-	-	7	4	-	6	3	1	-	2	-	7	**32**
DIMR_2_2	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_2_3	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_2_4	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_2_5	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_2_6	20	-	-	-	-	-	2	2	-	-	-	-	-	2	4	3	**13**
DIMR_3_1	17	1	1	2	1	-	6	6	8	6	3	2	6	8	-	9	**59**
DIMR_4_1	12	1	-	2	-	-	1	1	9	4	2	1	-	4	-	4	**29**
DIMR_5_1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_6_1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_7_1	15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	13	**13**
DIMR_7_2	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	**3**
DIMR_7_3	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_7_4	16	-	1	-	1	-	8	8	-	12	3	1	8	7	-	10	**59**
DIMR_8_1	13	-	-	-	-	-	7	4	-	-	-	-	-	-	-	7	**18**
DIMR_8_2	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2	**2**
DIMR_8_3	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**0**
DIMR_8_4	15	-	1	-	-	-	5	2	-	4	4	1	-	2	-	6	**25**
DIMR_9_1	2	-	-	-	-	-	-	-	-	2	1	-	-	-	-	2	**5**
DIMR_10_1	2	-	-	-	-	-	-	-	-	2	1	-	-	-	-	2	**5**
Overall	**188**	**2**	**7**	**4**	**3**	**0**	**42**	**36**	**17**	**42**	**20**	**7**	**20**	**34**	**6**	**84**	**324**

each mutant and each test case is available on our website [77].

4.6. Discussion

In the following we discuss some aspects that determine the quantity and quality of the inferred MRs.

Regarding the *quantity*, the number of MRs that are inferred for a specific model transformation is determined by the following factors:

- *Number of classes in source MM*. The more classes the source MM has, the more metamorphic input relations (MRs_{IP}) in *PatternMR_1* and *PatternMR_2* are inferred. In fact, most times one MR_{IP} will be defined for each class.

- *Number of attributes in source and target MMs*. The more attributes both MMs have, the more MRs_{IP} for *PatternMR_3*, *PatternMR_4*, *PatternMR_5* and *PatternMR_6* are likely to be inferred. However, it also depends on the transformation.

- *Number of classes in target MM*. The more classes the target MM has, the more instances of DIMRs in *PatternMR_1* and *PatternMR_2* for each MRs_{IP} are inferred. In fact, the number of MRs inferred in these two patterns for each MRs_{IP} is precisely the number of classes of the target MM. This is the main reason why there are so many MRs inferred in the *IEEE1471_2_MoDAF* case study (cf. Tables 1 and 4).

- *Number of references in source and target MMs*. The more references the target MM has, the more instances of DIMRs in *PatternMR_7* and *PatternMR_8* are inferred. Furthermore, the more references there are in the source MM, the more MRs for *PatternMR_9* and *PatternMR_10* are likely to be inferred. However, the latter also depends on the transformation.

- *Number of rules of model transformation*. The more rules a model transformation has, the more instances of DIMRs in all patterns are likely to be inferred.

- *Number of out-pattern elements*. The more out-pattern elements are produced in the rules of the transformation, the more MRs_{IP} for *PatternMR_1*, *PatternMR_2*, *PatternMR_7* and *PatternMR_8* are inferred.

- *Number of bindings in rules*. The more bindings the rules have, the more MRs_{IP} for *PatternMR_3*, *PatternMR_4*, *PatternMR_5*, *PatternMR_6*, *PatternMR_9* and *PatternMR_10* are likely to be inferred.

Regarding the *quality* of the MRs inferred, the number of TPs and FPs obtained are influenced by some aspects. As explained

in Section 3.4 and captured in Figure 13, our tool takes as input a set of trace models. The more complete, in terms of metamodel coverage, the source models are, the more variability of traces the resulting trace models contain. Having as input for our approach several trace models is very useful, since our tool combines all of them as if it only received a big trace model. Therefore, if several incomplete trace models are received as input, the combination of all of them may yield a complete trace model. By complete we mean that all the possible traces are present.

Not having a complete trace model may produce FPs. For instance, let us suppose that we have a trace that records the generation of an element of type *TargetElement* from an element of type *SourceElement* (cf. Figure 11(a)), and there is no other trace that has a *SourceElement* as input. Our tool would infer an instance of DIMR_1_1 (cf. Listing 8) and another one of DIMR_2_1 (cf. Listing 10) according to this information. However, in the model transformation we could have two rules that take a *SourceElement* as input, and both rules have a filter, so the MRs produced are FPs if the out-pattern elements produced in both rules are of different types. The reason for the generation of these FPs is that the source model that produces this specific trace model is incomplete in the sense that it should have more elements of type *SourceElement* such that all rules that have a *SourceElement* as input are fired.

In the five case studies taken from the ATL Zoo where we have applied our approach (cf. Section 4.1), we have taken as input the source models available in the zoo. In the other two case studies we have produced a source model that contains at least an element of each class in the metamodel. The reason for not creating big source models to use as input of the model transformations, what can be done with approaches such as [6, 89, 5, 4, 3], is that we wanted to evaluate the MRs inferred with the input models that were already available, so that no extra effort was required by the tester.

The following is a summary of the reasons why some FPs are produced when the trace models used as input of our tool are not complete enough:

- *Rules with filters*. In may happen that if there are several rules than contain filters, then some of the instances of the DIMRs in *PatternMR_1* and *PatternMR_2* do not always hold.

- *String binding initialized with condition*. When an attribute of type String is initialized with a condition, it may produce instances of the DIMRs in *PatternMR_3*, *PatternMR_4*, *PatternMR_5* and *PatternMR_6* that are FPs.

- *String binding initialized with concatenation of two attributes*. When an attribute of type String is initialized with the concatenation of two attributes of the rule's input element, it may produce instances of the DIMRs in *PatternMR_3*, *PatternMR_4*, *PatternMR_5* and *PatternMR_6* that are FPs.

- *String attribute set in imperative rule*. When the String value of an attribute is initialized in a rule imperatively called and not in the main rule, our tool may infer instances of DIMRs in *PatternMR_3*, *PatternMR_4*, *PatternMR_5* and *PatternMR_6* that are FPs.

- *More than one nested containment associations*. When a source class contains another class that, in turn, contains one or more classes (and so on), our tool may produce instances of the DIMRs in *PatternMR_4* and *PatternMR_8* that are FPs.

The reasons for each and all of the specific MRs inferred in our seven case studies that result in FPs are explained on our website [77].

4.7. Threats to Validity

4.7.1. Threats to Internal Validity

Are there factors that might affect the results of this evaluation? We consider the following internal threats to the validity of our evaluation based on the executed experiments. First, as source models for the model transformations used as case studies we have mainly used those available in the ATL Zoo [52]. Should we have used smaller models, the precision of the specific MRs inferred would have likely been worse. However, at the same time, we could have generated more complete input models [6, 89, 5, 4, 3], what would have likely resulted in more precise MRs. In any case, a thorough evaluation is needed in order to determine if adding more complete input models would not generate new FPs, or even if it would not remove some of the TPs generated in this evaluation. Second, the precision has been calculated by manually extracting the true positives and false positives for all the generated metamorphic relations in the seven case studies. Despite this process has been carried out by the authors, experts in MDE and model transformations, some TPs or FPs may have been incorrectly identified. In any case, an argument for the mitigation of this threat is that the resulting precision is similar for the seven case studies, being the standard deviation among them of only 6.42%.

Third, the experiment to evaluate the usefulness of our approach requires the generation of several inputs, what may influence the results. In this sense, we have generated 30 source test case inputs that, although it is not a big number, we consider sufficient because the input metamodel is not big. The mutants could have also been designed differently. We have decided to use mutation operators that have been proposed in the literature and whose automation is feasible [88]. Despite more mutants could have been generated, the transformation is not that big, so we do not think the results would have varied significantly. As another remark, the usefulness has been checked only for the regression testing scenario. We leave as future work a deeper evaluation that also considers incremental transformations and transformation migrations.

Finally, the possibility of deriving incorrect MRs if the transformation program is faulty is an intrinsic problem of the automated inference of MRs and program invariants [90, 91, 92, 93]. This is why the scope of these types of techniques is typically regression testing, that is, MRs are inferred from a well-tested version of the program with the intention of using

them to detect faults in future versions of the system. In our approach and evaluation, we assume the model transformation from which the MRs are inferred has been well-tested. Should it had contained errors, many of the MRs obtained would not have met the specification of the transformation.

4.7.2. Threats to External Validity

To what extent is it possible to generalize the findings? The first threat is the limited number of transformations we have evaluated, which externally threatens the generalizability of our results. The results for the precision of our approach are based on the seven case studies summarized in Table 1. To mitigate this threat, we have tried to select model transformations that differ in their domains, size of metamodels and transformation, and variability of features used within the transformation. Furthermore, considering the small deviation among the resulting precision for the seven case studies, it should not vary much if more model transformations are studied. As for the usefulness study, we have chosen our running example to be studied in depth. Further studies with other model transformations may have yielded different results. Indeed, the fact that the instances of a specific DIMR are never violated does not mean this DIMR is useless. The reason for the non violation is that we are utilizing a specific model transformation with 15 mutants. Should we have used a different example or different mutants, we would have gotten different results. In any case, based on the high percentage of mutants killed, 93.3%, we are confident that our approach would have been able to kill many mutants in other model transformations.

Finally, we have applied our approach for the Atlas Transformation Language (ATL) due to its importance both in industria and academia, so it would be interesting to test it with other transformation languages. In fact, we believe our approach would produce similar results for any model transformation language as long as the result of its executions can be stored in a trace model (cf. Figure 3).

5. Related Work

In this section we summarize some works that relate to our approach from different contexts. First, we present approaches that target the testing of model transformations, especially those that propose the generation of test case inputs for model transformations. Second, we describe the only work that, at the time of writing, has applied metamorphic testing to model transformations. Finally, we enumerate some approaches that try to automatically infer metamorphic relations.

5.1. Testing in Model Transformations

Many approaches have proposed the generation of test case inputs for model transformations, where most of them employ a black-box approach. Test case inputs, i.e., source models, are mainly generated either considering the source metamodel [94, 95, 96] or some specified requirements [97, 5]. Most of the approaches are based on constraint satisfaction by means of SAT solvers. There is also an approach to automatically complete test input models. Thereby, the transformation engineer only needs to define a model fragment, which is then automatically completed in order to obtain a valid test case input [6]. There are also approaches that propose white-box methods. For instance, Gonzalez and Cabot [3] propose to generate test input models out of ATL model transformations by extracting OCL constraints and using a model finder to compute the models fulfilling certain path conditions.

These works are complementary to our approach since, as explained in Section 4.6, they could be used in order to produce complete source models to be given as input to the model transformations for which we want to infer metamorphic relations. This would increase the chances to obtain accurate metamorphic relations.

Other than these works, there are several approaches that can be seen as orthogonal to ours for the testing of model transformations. For instance, some works propose to apply static analysis to ATL model transformations [7, 9], so that errors can be spotted without the need to execute the transformation. There are works that propose the translation of model transformation languages to formal domains, where specific analysis can be carried out. For instance, Troya and Vallecillo [8] describe a formal semantics for ATL by translating it to Maude. The work in [21] automatically transforms transformations in a number of transformation languages (such as ATL) to OCL, and the work by Anastasakis et al. [22] transforms QVT model transformations to Alloy in order to verify if given assertions, i.e., properties, hold for the given transformations. Calegari et al. [23] propose an interactive approach to verify contracts for ATL transformations based on the Coq proof assistant. This approach is unbounded, but requires some user guidance. Another approach using the Coq proof assistant to ensure the correctness of model transformations is presented in [98].

Other approaches using theorem provers for model transformation verification go one step further by using modern SMT solvers such as is done in [99, 100]. These approaches do not require user guidance as it was required in the aforementioned Coq-based approaches. They translate the ATL transformations as well as the contracts expressed in OCL into first-order logic expressions and use Z3 for performing the theorem proving. Oakes et al. [20] present an approach to fully prove properties defined as contracts for model transformations expressed in declarative ATL, including advanced features such as lazy rules, by translating the transformations to DSLTrans [101].

Regarding properties specified as contracts, in [82, 102] the authors describe their method where 'Tracts' can be specified for model transformations. These tracts define a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. Our metamorphic relations can be seen similar to these tracts, which are also expressed in OCL. However, while these tracts have to be manually defined and a good knowledge of the transformation domain and purpose are necessary, our metamorphic relations are inferred automatically.

5.2. Metamorphic Testing in Model Transformations

At the time of writing, only Jiang et al. [29] have proposed the application of metamorphic testing to test model transformations, which is the work that has inspired us to keep advancing in the matter. They empirically prove that metamorphic testing is an effective testing method for model transformation programs. However, differently from our approach, they focus on one case study, the popular *Class2Relational* model transformation available on the ATL Zoo [52], for which they manually define metamorphic relations based on their knowledge of the transformation and its domain.

5.3. Automatic Inference of Metamorphic Relations

Enumerating a set of metamorphic relations (MRs) that should be satisfied by a program is a critical initial task in applying metamorphic testing. Typically, a tester or developer has to manually identify MRs using her knowledge of the program under test. This process can easily miss some of the important MRs that could reveal faults as well as produce incorrect MRs. Kanewala and Bieman [103, 104, 26] present the first attempt at developing techniques to automatically infer likely MRs. They propose to apply advance machine learning algorithms to do so. The automated method operates by extracting a set of features from a function's control flow graph, what has similarities to extracting information from the model transformation traces. In further work [105], they explore the effectiveness of several representations of the control flow graphs using the machine learning framework of graph kernels and concluded that a graph kernel that evaluates the contribution of all paths in the graph has the best accuracy.

Zhang et al. [27] propose a search-based approach for the automatic inference of polynomial MRs via dynamically analyzing multiple executions using particle swarm optimization. By *polynomial MRs* they refer to a set of parameters to represent a particular class of MRs. In this way, they convert the problem of inferring MRs into a problem of searching for suitable values of the parameters. On polynominal MRs, the relations between inputs and the relations between outputs are both polynomial equations. With this approach, the authors are able to automatically infer a high number of MRs, as we also achieve. However, the application domain of this approach is drastically different from our application domain, so no fair comparison is possible.

Su et al. [28] present an approach that guides developers to so-called likely *metamorphic properties* (MPs) that may apply to their systems. In this sense, the approach is not completely automatic, since the developers profile executions of the system to detect which MPs might apply to which methods, and then present these properties for testers to either confirm or reject.

Despite approaches for the automatic inference of metamorphic relations are starting to appear, this is still a major challenge in metamorphic testing.

5.4. Synopsis

To the best of our knowledge, in this paper we have presented the first approach to automatically infer metamorphic relations for model transformations. The tester does not need to have any knowledge about the model transformations nor their domains, since the MRs are automatically generated. She only needs to have a source model for the transformation. Finally, compared to related works on the inference of MRs [103, 104, 26, 27, 28], our approach only needs one execution of the program under test in order to produce the MRs.

6. Conclusions

In this paper we have presented the first approach for the automatic inference of likely metamorphic relations in model transformations. We have created a catalogue of 24 likely domain-independent metamorphic relations defined generically and grouped by patterns. This means that they are to be instantiated for specific model transformations. Our approach receives the executions of model transformations as trace models as input and automatically infers a set of specific MRs that are instances of the domain-independent metamorphic relations of our catalogue. The tester does not need to have any knowledge of neither the transformation nor its domain in order to extract the MRs. Furthermore, the way the patterns have been defined makes our approach extensible, so future versions could have more accurate or new MRs. Our approach has been applied to model transformations written in ATL due to its importance in both industry and academia. Nevertheless, it can be applied to any model transformation language as long as it is able to store the result of the execution in trace models.

Together with the approach by Jiang et al. [29], this paper opens the door to a novel way of testing model transformations, so-called metamorphic testing. Since the more complete the input models for the model transformations are, the more accurate the automatically inferred metamorphic relations are likely to be, our approach can be used in conjunction with existing approaches for the automatic generation of test case inputs in model transformations [94, 95, 96, 97, 5, 6, 3]. Besides, there exist several approaches that propose the application of different techniques in the testing of model transformations [7, 9, 8, 21, 22, 23, 98, 99, 100, 20, 101, 82, 102] and that could be complemented with our approach.

Our approach is specifically tailored at detecting faults in model transformations under three application scenarios, namely regression testing, incremental transformations and migrations among different transformation languages. In fact, we assume the MRs are extracted from a well-tested version of the model transformation with the intention of using them to detect faults in (i) future versions of the transformation (regression testing), (ii) incremental versions of the original transformation and (iii) the same transformation written in a different language.

There are several lines of future work that may follow. First, we have proposed an initial set of domain-independent patterns in the traces (cf. Figure 11). It would be interesting to extend this set by identifying new patterns. One possibility is to come up with techniques to detect the presence of filters in the rules. This could be done by studying the value of the attributes and references of the source elements in the traces. Having more patterns in the traces would produce more domain-independent metamorphic relations, so that our catalogue could be extended.

Second, it would be interesting to integrate some of the approaches to automatically generate test case inputs within our tool. In this way, the tester would not need to provide a source model for the model transformation, so the process of generating the MRs could be further automated.

Finally, we would also like to perform a deeper evaluation in order to consider incremental transformations and migration scenarios. It would be interesting not only to check if the MRs inferred are able to detect faults, but also the definition of a methodology to help testers actually locate the faults. The methodology would define a way to locate the errors that produce the violation of the MRs, by analyzing from the more specific to the more general.

Acknowledgements

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project BELI (TIN2015-70560-R), and the Andalusian Government project COPAS (P12-TIC-1867).

[1] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–646. doi:10.1147/sj.453.0621.

[2] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. K. Selim, E. Syriani, M. Wimmer, Model transformation intents and their properties, Software & Systems Modeling 15 (3) (2016) 647–684. doi:10.1007/s10270-014-0429-x.

[3] C. A. González, J. Cabot, ATLTest: A White-Box Test Generation Approach for ATL Transformations, in: Proc. of 15th International Conference on Model Driven Engineering Languages and Systems, MoDELS'12, Springer, 2012, pp. 449–464. doi:10.1007/978-3-642-33666-9_29.

[4] J. M. Küster, M. Abd-El-Razik, Validation of model transformations – first experiences using a white box approach, in: Workshops and Symposia at MoDELS 2006, Springer, 2007, pp. 193–204. doi:10.1007/978-3-540-69489-2_24.

[5] E. Guerra, M. Soeken, Specification-driven model transformation testing, Software & Systems Modeling 14 (2) (2015) 623–644. doi:10.1007/s10270-013-0369-x.

[6] S. Sen, J.-M. Mottu, M. Tisi, J. Cabot, Using models of partial knowledge to test model transformations, in: Proc. of 5th Int. Conference on Theory and Practice of Model Transformations, ICMT'12, Springer, 2012, pp. 24–39. doi:10.1007/978-3-642-30476-7_2.

[7] J. Sánchez Cuadrado, E. Guerra, J. de Lara, Uncovering errors in ATL model transformations using static analysis and constraint solving, in: Proc. of IEEE 25th International Symposium on Software Reliability Engineering, ISSRE'14, IEEE, 2014, pp. 34–44. doi:10.1109/ISSRE.2014.10.

[8] J. Troya, A. Vallecillo, A Rewriting Logic Semantics for ATL, Journal of Object Technology 10 (2011) 5:1–29. doi:10.5381/jot.2011.10.1.a5.

[9] L. Burgueno, J. Troya, M. Wimmer, A. Vallecillo, Static Fault Localization in Model Transformations, IEEE Tansactions on Software Engineering 41 (5) (2015) 490–506. doi:10.1109/TSE.2014.2375201.

[10] F. Büttner, M. Egea, J. Cabot, On Verifying ATL Transformations Using 'Off-the-shelf' SMT Solvers, in: Proc. of the 15th International Conference on Model Driven Engineering Languages and Systems, MODELS'12, Springer-Verlag, 2012, pp. 432–448. doi:10.1007/978-3-642-33666-9_28.

[11] F. Büttner, M. Egea, J. Cabot, M. Gogolla, Verification of ATL Transformations Using Transformation Models and Model Finders, in: Proc. of the 14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM'12, Springer-Verlag, 2012, pp. 198–213. doi:10.1007/978-3-642-34281-3_16.

[12] J. Cabot, R. Clarisó, E. Guerra, J. de Lara, Verification and Validation of Declarative Model-to-model Transformations Through Invariants, Journal of Systems and Software 83 (2) (2010) 283–302. doi:10.1016/j.jss.2009.08.012.

[13] J. Etzlstorfer, A. Kusel, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, M. Wimmer, A Survey on Incremental Model Transformation Approaches, in: Models & Evolution Workshop @ MoDELS, 2013.

[14] A. Razavi, K. Kontogiannis, Partial Evaluation of Model Transformations, in: Proc. of 34th International Conference on Software Engineering, ICSE'12, IEEE Press, 2012, pp. 562–572. doi:10.1109/ICSE.2012.6227160.

[15] F. Jouault, M. Tisi, Towards Incremental Execution of ATL Transformations, in: Proc. of Third International Conference on Model Transformations, ICMT'10, 2010, pp. 123–137. doi:10.1007/978-3-642-13688-7_9.

[16] D. Hearnden, M. Lawley, K. Raymond, Incremental Model Transformation for the Evolution of Model-Driven Systems, in: Proc. of 9th Int. Conf. on Model-Driven Engineering Languages and Systems, MoDELS'06, 2006, pp. 321–335. doi:10.1007/11880240_23.

[17] S. Johann, A. Egyed, Instant and incremental transformation of models, in: Proc. of the 19th IEEE Int. Conf. on Automated Software Engineering, ASE'04, 2004, pp. 362–365. doi:10.1109/ASE.2004.43.

[18] I. Ráth, G. Bergmann, A. Ökrös, D. Varró, Live Model Transformations Driven by Incremental Pattern Matching, in: Proc. of 1st International Conference on Model Transformations, ICMT'08, 2008. doi:10.1007/978-3-540-69927-9_8.

[19] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability 22 (2) (2012) 67–120. doi:10.1002/stvr.430.

[20] B. J. Oakes, J. Troya, L. Lúcio, M. Wimmer, Full Contract Verification for ATL using Symbolic Execution, Journal on Software & System Modeling (2016) 1–35doi:10.1007/s10270-016-0548-7.

[21] F. Büttner, M. Egea, E. Guerra, J. De Lara, Checking Model Transformation Refinement, in: Proc. of 6th International Conference on Model Transformations, ICMT'13, 2013, pp. 158–173.

[22] K. Anastasakis, B. Bordbar, J. M. Küster, Analysis of Model Transformations via Alloy, in: Proc. of 4th Workshop on Model Driven Engineering, Verification and Validation, MoDeVVa'07, 2007.

[23] D. Calegari, C. Luna, N. Szasz, A. Tasistro, A Type-Theoretic Framework for Certified Model Transformations, in: Proc. of 13th Brazilian Symposium on Formal Methods: Foundations and Applications, SBMF'10, 2010, pp. 112–127. doi:10.1007/978-3-642-19829-8_8.

[24] T. Y. Chen, S. C. Cheung, S. M. Yiu, Metamorphic testing: A new approach for generating next test cases, Tech. rep., Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology (1998).

[25] S. Segura, G. Fraser, A. Sanchez, A. Ruiz-Cortes, A survey on metamorphic testing, IEEE Transactions on Software Engineering 42 (9) (2016) 805–824. doi:10.1109/TSE.2016.2532875.

[26] U. Kanewala, Techniques for Automatic Detection of Metamorphic Relations, in: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '14, IEEE Computer Society, 2014, pp. 237–238. doi:10.1109/ICSTW.2014.62.

[27] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, H. Mei, Search-based Inference of Polynomial Metamorphic Relations, in: Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, 2014, pp. 701–712. doi:10.1145/2642937.2642994.

[28] F.-H. Su, J. Bell, C. Murphy, G. Kaiser, Dynamic Inference of Likely Metamorphic Properties to Support Differential Testing, in: Proc. of the 10th International Workshop on Automation of Software Test, AST '15, IEEE Press, 2015, pp. 55–59. doi:10.1109/AST.2015.19.

[29] M. Jiang, T. Y. Chen, F.-C. Kuo, Z. Q. Zhou, Z. Ding, Testing Model Transformation Programs using Metamorphic Testing, in: Proc. of 26th International Conference on Software Engineering and Knowledge Engineering, SEKE'14, 2014, pp. 94–99.

[30] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A Model Transformation Tool, Science of Computer Programming 72 (1-2) (2008) 31–39.

doi:10.1016/j.scico.2007.08.002.

[31] E. M. Project, Atlas Transformation Language – ATL, http://eclipse.org/atl (2015).

[32] A. R. da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139 – 155. doi:http://doi.org/10.1016/j.cl.2015.06.001.

[33] T. Kühne, Matters of (meta-) modeling, Software & Systems Modeling 5 (4) (2006) 369–385. doi:10.1007/s10270-006-0017-9.

[34] J. Ludewig, Models in software engineering – an introduction, Software and Systems Modeling 2 (1) (2003) 5–14. doi:10.1007/s10270-003-0020-3.

[35] S. J. Mellor, K. Scott, A. Uhl, D. Weise, R. M. Soley, MDA distilled: principles of model-driven architecture, Vol. 88, Addison-Wesley, 2004.

[36] S. Sendall, W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software 20 (5) (2003) 42–45. doi:10.1109/MS.2003.1231150.

[37] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Morgan&Claypool, 2012.

[38] M. Wimmer, L. Burgueño, Testing M2T/T2M Transformations, in: Proc. of 16th Int. Conf. on Model-Driven Engineering Languages and Systems, MoDELS'13, Springer, 2013, pp. 203–219. doi:10.1007/978-3-642-41533-3_13.

[39] T. Mens, P. V. Gorp, A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125–142. doi:10.1016/j.entcs.2005.10.021.

[40] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations, in: Proc. of 13th International Conference on Model Driven Engineering Languages and Systems, MoDELS'10, Springer, 2010, pp. 121–135. doi:10.1007/978-3-642-16145-2_9.

[41] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: Proc. of the 2nd Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE'03, Springer, 2003, pp. 446–453. doi:10.1007/978-3-540-25959-6_35.

[42] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-Performance Logical Framework, Vol. 4350 of LNCS, Springer, 2007. doi:10.1007/978-3-540-71999-1.

[43] J. de Lara, H. Vangheluwe, AToM3: A Tool for Multi-formalism and Meta-modelling, in: Proc. of the 5th International Conference on Fundamental Approaches to Software Engineering, FASE'02, Springer, 2002, pp. 174–188. doi:10.1007/3-540-45923-5_12.

[44] J. E. Rivera, F. Duran, A. Vallecillo, A Graphical Approach for Modeling Time-dependent Behavior of DSLs, in: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'09, IEEE, 2009, pp. 51–55. doi:10.1109/VLHCC.2009.5295300.

[45] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, D. Varró, VIATRA - visual automated transformations for formal verification and validation of UML models, in: Proc. of the 17th International Conference on Automated Software Engineering, ASE'02, IEEE/ACM, 2002, pp. 267–270. doi:10.1109/ASE.2002.1115027.

[46] M. Fleck, J. Troya, M. Wimmer, Marrying Search-based Optimization and Model Transformation Technology, in: Proc. of the First North American Search Based Software Engineering Symposium, NasBASE'15, 2015, pp. 1–16.

[47] M. Fleck, J. Troya, M. Wimmer, Search-Based Model Transformations, Journal of Software: Evolution and Process 28 (12) (2016) 1081–1117. doi:10.1002/smr.1804.

[48] J. Greenyer, E. Kindler, Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars, Software and System Modeling 9 (1) (2010) 21–46. doi:10.1007/s10270-009-0121-8.

[49] J.-M. Jézéquel, O. Barais, F. Fleurey, Model Driven Language Engineering with Kermeta, in: Proc. of the 3rd international summer school conference on Generative and transformational techniques in software engineering III, GTTSE'09, Springer, 2011, pp. 201–221. doi:10.1007/978-3-642-18023-1_5.

[50] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, JTL: A Bidirectional and Change Propagating Transformation Language, in: Proc. of Third International Conference on Software Language Engineering, SLE'11, Springer, 2011, pp. 183–202. doi:10.1007/978-3-642-19440-5_11.

[51] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, P. Valduriez, ATL: A QVT-like Transformation Language, in: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA'06, ACM, 2006, pp. 719–720. doi:10.1145/1176617.1176691.

[52] ATL, ATL Zoo, http://www.eclipse.org/atl/atlTransformations (2006).

[53] F. Jouault, Loosely Coupled Traceability for ATL, in: Proc. of the European Conference on Model Driven Architecture Workshop on Traceability, ECMDA'05, 2005.

[54] E. J. Weyuker, On testing non-testable programs, The Computer Journal 25 (4) (1982) 465–470. doi:10.1093/comjnl/25.4.465.

[55] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE Transactions on Software Engineering 41 (5) (2015) 507–525. doi:10.1109/TSE.2014.2372785.

[56] H. Liu, F.-C. Kuo, D. Towey, T. Y. Chen, How effectively does metamorphic testing alleviate the oracle problem?, IEEE Transactions on Software Engineering 40 (1) (2014) 4–22. doi:10.1109/TSE.2013.46.

[57] W. K. Chan, S. C. Cheung, K. R. P. H. Leung, A metamorphic testing approach for online testing of service-oriented software applications., International Journal of Web Services Research 4 (2) (2007) 61–81. URL http://dblp.uni-trier.de/db/journals/jwsr/jwsr4.html#ChanCL07

[58] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, T. Y. Chen, A metamorphic relation-based approach to testing web services without oracles, International Journal of Web Services Research 9 (1) (2012) 51–73. doi:10.4018/jwsr.2012010103.

[59] Z. Q. Zhou, S. Xiang, T. Y. Chen, Metamorphic testing for software quality assessment: A study of search engines, IEEE Transactions on Software Engineering 42 (3) (2016) 264–284. doi:10.1109/TSE.2015.2478001.

[60] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, T. Y. Chen, Testing and validating machine learning classifiers by metamorphic testing, The Journal of Systems and Software 84 (4) (2011) 544–558. doi:10.1016/j.jss.2010.11.920.

[61] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, S. S. Yau, Integration testing of context-sensitive middleware-based applications: a metamorphic approach., International Journal of Software Engineering and Knowledge Engineering 16 (5) (2006) 677–704. doi:10.1142/S0218194006002951.

[62] M.-Y. Jiang, T. Y. Chen, F.-C. Kuo, Z. Ding, Testing central processing unit scheduling algorithms using metamorphic testing, in: Proc. of 4th IEEE International Conference on Software Engineering and Service Science, ICSESS'13, 2013, pp. 530–536. doi:10.1109/ICSESS.2013.6615365.

[63] R. Guderlei, J. Mayer, Towards automatic testing of imaging software by means of random and metamorphic testing, International Journal of Software Engineering and Knowledge Engineering 17 (06) (2007) 757–781. doi:10.1142/S0218194007003471.

[64] W. K. Chan, J. C. F. Ho, T. H. Tse, Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs, Software Testing, Verification and Reliability Journal 20 (2) (2010) 89–120. doi:10.1002/stvr.v20:2.

[65] Q. Tao, W. Wu, C. Zhao, W. Shen, An automatic testing approach for compiler based on metamorphic testing technique, in: 17th Asia Pacific Software Engineering Conference, APSEC'10, 2010, pp. 270–279. doi:10.1109/APSEC.2010.39.

[66] V. Le, M. Afshari, Z. Su, Compiler validation via equivalence modulo inputs, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, 2014, pp. 216–226. doi:10.1145/2594291.2594334.

[67] T. Y. Chen, F.-C. Kuo, H. Liu, S. Wang, Conformance testing of network simulators based on metamorphic testing technique, in: Formal Techniques for Distributed Systems, Vol. 5522 of LNCS, Springer, 2009, pp. 243–248. doi:10.1007/978-3-642-02138-1_19.

[68] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, I. M. Llorente, iCanCloud: A Flexible and Scalable Cloud Infras-

tructure Simulator, Journal of Grid Computing 10 (1) (2012) 185–209. doi:10.1007/s10723-012-9208-5.

[69] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, T. Y. Chen, Application of metamorphic testing to supervised classifiers, in: 9th International Conference on Quality Software, 2009. QSIC '09., 2009, pp. 135–144. doi:10.1109/QSIC.2009.26.

[70] Z. Jing, H. Xuegang, Z. Bin, An evaluation approach for the program of association rules algorithm based on metamorphic relations, Journal of Electronics (China) 28 (4) (2011) 623–631. doi:10.1007/s11767-012-0743-9.

[71] T. Y. Chen, J. W. K. Ho, H. Liu, X. Xie, An innovative approach for testing bioinformatics programs using metamorphic testing, BioMed Central Bioinformatics Journal 10 (1) (2009) 24. doi:10.1186/1471-2105-10-24.

[72] L. L. Pullum, O. Ozmen, Early results from metamorphic testing of epidemiological models, in: ASE/IEEE International Conference on BioMedical Computing (BioMedCom), 2012, 2012, pp. 62–67. doi:10.1109/BioMedCom.2012.17.

[73] A. Bergmayr, J. Troya, M. Wimmer, From Out-place Transformation Evolution to In-place Model Patching, in: Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 647–652. doi:10.1145/2642937.2642946.

[74] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber, The Petri Net Markup Language: Concepts, Technology, and Tools, in: Proc. of the 24th International Conference on Applications and Theory of Petri Nets, ICATPN'03, Springer-Verlag, 2003, pp. 483–505. doi:10.1007/3-540-44919-1_31.

[75] ISO/IEC, Pnml, http://www.pnml.org (2011).

[76] P. Guyard, Bridging grafcet, petri net, pnml and xml, http://www.eclipse.org/atl/atlTransformations/Grafcet2PetriNet/ExampleGrafcet2PetriNet[v00.01].pdf (2005).

[77] J. Troya, S. Segura, A. Ruiz-Cortés, Metamorphic Testing in Model Transformations, https://gestionproyectos.us.es/projects/curso-ice-2016-rest-fp-coordinacioo/wiki (2016).

[78] J. Warmer, A. Kleppe, The Object Constraint Language: Getting your models ready for MDA, Addison Wesley, 2003.

[79] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[80] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting Experiments in Software Engineering, Springer London, London, 2008, pp. 201–228. doi:10.1007/978-1-84800-044-5_8.

[81] M. Lawley, K. Duddy, A. Gerber, K. Raymond, Language features for re-use and maintainability of MDA transformations, in: In OOPSLA Workshop on Best Practices for Model-Driven Software Development, 2004.

[82] M. Gogolla, A. Vallecillo, *Tract*able model transformation testing, in: Proc. of 7th European Conference on Modelling Foundations and Applications, ECMFA'11, Springer, 2011, pp. 221–235. doi:10.1007/978-3-642-21470-7_16.

[83] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[84] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, Y. Le Traon, Model Transformation Testing Challenges, in: Proc. of the ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, 2006.

[85] A. Vallecillo, M. Gogolla, Typing model transformations using tracts, in: Proc. of 5th Int. Conf. on Theory and Practice of Model Transformations, ICMT'12, Springer, 2012, pp. 56–71. doi:10.1007/978-3-642-30476-7_4.

[86] E. Cariou, R. Marvie, L. Seinturier, L. Duchien, OCL for the Specification of Model Transformation Contracts, in: Proc. of the OCL and Model Driven Engineering Workshop, 2004.

[87] J.-M. Mottu, B. Baudry, Y. Le Traon, Mutation analysis testing for model transformations, in: Proc. of 2nd European Conference on Model Driven Architecture – Foundations and Applications, ECMDA-FA'06, Springer, 2006, pp. 376–390. doi:10.1007/11787044_28.

[88] J. Troya, A. Bergmayr, L. Burgueno, M. Wimmer, Towards systematic mutations for and with atl model transformations, in: Proc. of the IEEE 8th Int. Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2015, pp. 1–10. doi:10.1109/ICSTW.2015.7107455.

[89] F. Fleurey, B. Baudry, P.-A. Muller, Y. L. Traon, Qualifying input test data for model transformations, Software & Systems Modeling 8 (2) (2009) 185–203. doi:10.1007/s10270-007-0074-8.

[90] U. Kanewala, J. M. Bieman, A. Ben-Hur, Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels, Softw. Test., Verif. Reliab. 26 (3) (2016) 245–269. doi:10.1002/stvr.1594.
URL http://dx.doi.org/10.1002/stvr.1594

[91] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, H. Mei, Search-based inference of polynomial metamorphic relations, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 701–712. doi:10.1145/2642937.2642994.

[92] F.-H. Su, J. Bell, C. Murphy, G. Kaiser, Dynamic inference of likely metamorphic properties to support differential testing, in: Proceedings of the 10th International Workshop on Automation of Software Test, AST '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 55–59.

[93] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, The daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (1-3) (2007) 35–45. doi:10.1016/j.scico.2007.01.015.

[94] K. Ehrig, J. M. Küster, G. Taentzer, Generating instance models from meta models, Software & Systems Modeling 8 (4) (2009) 479–500. doi:10.1007/s10270-008-0095-y.

[95] E. Brottier, F. Fleurey, J. Steel, B. Baudry, Y. L. Traon, Metamodel-based test generation for model transformations: an algorithm and a tool, in: Proc. 17th International Symposium on Software Reliability Engineering, ISSRE'06, 2006, pp. 85–94. doi:10.1109/ISSRE.2006.27.

[96] S. Sen, B. Baudry, J.-M. Mottu, Automatic Model Generation Strategies for Model Transformation Testing, in: Proc. of 2nd International Conference on Model Transformations, ICMT'09, Springer, 2009, pp. 148–164. doi:10.1007/978-3-642-02408-5_11.

[97] P. Giner, V. Pelechano, Test-driven development of model transformations, in: Proc. of 12th International Conference on Model Driven Engineering Languages and Systems, Vol. 5795 of MoDELS'09, Springer, 2009, pp. 748–752. doi:10.1007/978-3-642-04425-0_61.

[98] I. Poernomo, J. Terrell, Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq, in: Proc. of 12th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM'10, 2010, pp. 56–73. doi:10.1007/978-3-642-16901-4_6.

[99] Z. Cheng, R. Monahan, J. F. Power, A Sound Execution Semantics for ATL via Translation Validation, in: Proc. of 8th International Conference on Theory and Practice of Model Transformations, ICMT'15, 2015, pp. 133–148. doi:10.1007/978-3-319-21155-8_11.

[100] F. Büttner, M. Egea, J. Cabot, On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers, in: Proc. of 15th International Conference on Model Driven Engineering Languages and Systems, MoDELS'12, 2012, pp. 432–448. doi:10.1007/978-3-642-33666-9_28.

[101] B. Barroca, L. Lúcio, V. Amaral, R. Félix, V. Sousa, DSLTrans: A Turing Incomplete Transformation Language, in: Proc. of Third International Conference on Software Language Engineering, SLE'11, 2011, pp. 296–305. doi:10.1007/978-3-642-19440-5_19.

[102] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, L. Hamann, Formal specification and testing of model transformations, in: Proc. of 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-Driven Engineering, SFM'12, Springer, 2012, pp. 399–437.

[103] U. Kanewala, J. M. Bieman, Using machine learning techniques to detect metamorphic relations for programs without test oracles, in: IEEE 24th International Symposium on Software Reliability Engineering, ISSRE'13, 2013, pp. 1–10. doi:10.1109/ISSRE.2013.6698899.

[104] U. Kanewala, J. M. Bieman, Techniques for Testing Scientific Programs Without an Oracle, in: Proc. of the 5th International Workshop on Software Engineering for Computational Science and Engineering, SECSE '13, IEEE Press, 2013, pp. 48–57. doi:10.1109/SECSE.2013.6615099.

[105] U. Kanewala, J. M. Bieman, A. Ben-Hur, Predicting metamorphic re-

lations for testing scientific software: a machine learning approach using graph kernels, Software Testing, Verification and Reliability 26 (3) (2016) 245–269. doi:10.1002/stvr.1594.