Software and Systems Modeling (2022) 21:81–112
https://doi.org/10.1007/s10270-021-00891-0

REGULAR PAPER

Suggesting model transformation repairs for rule-based languages using a contract-based testing approach

Roberto Rodriguez-Echeverria[1] · Fernando Macías[2] · Adrian Rutle[3] · José M. Conejero[4]

Received: 23 June 2020 / Revised: 3 May 2021 / Accepted: 10 May 2021 / Published online: 26 May 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2021

Abstract

Model transformations play an essential role in most model-driven software projects. As the size and complexity of model transformations increase, their reuse, evolution and maintenance become a challenge. This work further details the Model Transformation TEst Specification (MoTES) approach, which leverages contract-based model testing techniques to assist engineers in model transformation evolution and repairing. The main novelty of our approach is to use contract-based model transformation testing as a foundation to derive suggestions of concrete adaptation actions. MoTES uses contracts to specify the expected behaviour of the model transformation under test. These contracts are transformed into model transformations which act as oracles on input–output model pairs, previously generated by executing the transformation under test on provided input models. By further processing, the oracles' output model, precision and recall metrics are calculated for every output pattern (testing results). These metrics are then categorised to increase the user's ability to interpret and act on them. The MoTES approach defines 8 cases for precision and recall values classification (test result cases). As traceability information is retained from transformation rules to each output pattern, it is possible to classify each transformation rule involved according to its impact on the metrics, e.g. the number of true positives generated. The MoTES approach defines 37 cases for these classifications, with each one linked to a particular (abstract) action suggested on a rule, such as relaxation of the rules. A comprehensive evaluation of this approach is also presented, consisting of three case studies. Two previous case studies performed over two model transformations (UML2ER and E2M) are replicated to contrast MoTES with an existing model transformation fault localisation approach. An additional case study presents how MoTES helps with the evolution of an existing model transformation in the context of a reverse engineering project. Main evaluation results show that our approach can not only detect the errors introduced in the transformations but also localise the faulty rule and suggest the proper repair actions, which significantly reduce testers' effort. From a quantitative perspective, in the third case study, MoTES was able to indicate one faulty rule from 19 possibilities for each result case and suggest one or two repair actions from 6 possibilities for each faulty rule.

Keywords Model Transformation · Evolution · Testing · Repairing · Testing Oracle · Adaptations · Verification · Fault Localisation

1 Introduction

In the context of model-driven engineering, models are the primary development artefacts and model transformations

Communicated by Esther Guerra.

✉ Roberto Rodriguez-Echeverria
rre@unex.es

Fernando Macías
fernando.macias@imdea.org

Adrian Rutle
aru@hvl.no

José M. Conejero
chemacm@unex.es

[1] Quercus Software Engineering Group, University of Extremadura, Cáceres, Spain

[2] IMDEA Software Institute, Madrid, Spain

[3] Western Norway University of Applied Sciences, Bergen, Norway

[4] Quercus Software Engineering Group, University of Extremadura, Cáceres, Spain

are crucial elements to define operations over models, such as querying, synthesising and transforming models. Model transformations are essential for Model-Driven Engineering to be practical. However, they are challenging to maintain and adapt when requirements and implementation platforms change [1].

As the size and complexity of model transformations increase, the cost and complexity of their reuse, evolution and maintenance become a challenge, e.g. during their adaptation to new application contexts. Hence, new mechanisms and tools are needed to help engineers with the complex activities involved in model transformation evolution and maintenance.

As a response to this, several model transformation testing approaches have been proposed during recent years. Some of these approaches allow engineers to specify and run regression tests to assess the behaviour of their model transformations, but they only report whether a test has passed or failed, e.g. [2–7]. In other cases, actual fixes are proposed, or even automatically applied, for a given error, e.g. [8–10]. These techniques are based on static analysis of the code, and the fixes they propose are still based on single errors. This kind of unitary information helps find errors in model transformations, yet we think it falls short. Engineers have to review the errors and fix them one by one to detect the source of the problem and figure out how to solve it, which implies interpreting the error correctly within its context. This task is incredibly cumbersome in the case of transformations composed of a large number of rules. Consequently, we believe that there is a need for more meaningful results which provide a deeper understanding of how a model transformation is behaving. In this sense, metrics—i.e. aggregated numeric values—might give additional information about an error, such as its extension for an actual input or some hints about its nature. Although there are previous works that use metrics for pinpointing guilty rules in a model transformation, e.g. [11,12], they neither suggest repairing actions nor offer hints at the cause of the errors.

In this work, we present a **Mo**del **T**ransformation **TE**st **S**pecification (MoTES), an approach to easily leverage testing results to automatically derive proper corrective actions when tests fail. Those adaptations are derived from the computation of precision and recall metrics for every output pattern whose generation is controlled by an invariant condition or contract between input and output patterns. Those metric values are then analysed and categorised to identify a probably guilty model transformation rule (fault localisation) and propose a corrective action (fault repair). As a result, engineers can focus their efforts on applying specific corrective actions to particular model transformation rules. Note that our results are independent of how the contracts are specified and executed and the results collected. That is, the main novelty of this work lies in (1) defining a metric-based test oracle, (2) generating output-centred test results for enhanced interpretation and (3) recommending *abstract* repairing actions [13] based on these results.

This paper is an extension of previous papers presented at the MoDELS 2015 conference [14] and the Fifth International Workshop on the Verification of Model Transformation 2016 [15]. Besides a comprehensive presentation of our approach to make it self-contained, this article introduces a substantial extension over previous papers. The result computation and interpretation section has been extended with the subdivision of False Positive results into input or output-originated. We have also thoroughly rewritten the section on suggesting adaptations to encompass such subdivision, including mutation operators as a base for repairing actions, introducing a new adaptation action, and, finally, providing an extended and revised version of our adaptation cheatsheet. Moreover, a whole new evaluation section has been added, which presents several empirical studies to assess the validity and applicability of our approach. In this sense, we present herein the results obtained by the replication of the mutation test analysis originally performed in [7] for two case studies.

The rest of this paper is organised as follows. Sect. 2 gives an overview of our approach. Contract definition in MoTES is explained in Sect. 3. Sections 4 and 5 present the main contribution of this paper: how the resulting metrics are interpreted and how to map them to adaptation suggestions, respectively. Section 6 presents our evaluation using a replica of two case studies and an application case study. Section 7 discusses the related works. Furthermore, finally, Sect. 8 outlines the main conclusions of this work and indicates future directions for the approach.

2 MoTES overview

The main goal of MoTES is to provide engineers with a fast and lightweight model transformation repairing approach that they can seamlessly use in any stage of their model-driven development process. Therefore, using our approach, engineers can (1) specify what the expected behaviour of a model transformation is, (2) get easy-to-interpret results about its correctness, and (3) automatically obtain a list of suggested repair actions for every transformation rule of interest. In that sense, our approach uses model transformation testing as a foundation to derive suggestions of concrete adaptation actions.

2.1 MoTES structure

In Fig. 1, a graphical overview of our approach is presented. On the left-hand side, a simple model-driven development scenario is depicted, which mainly entails an input model,

the model transformation under test (MTUT), a transformation engine and the products of such transformation: the output model and the execution trace model. Input models and MTUTs are manually specified by modelling engineers, while a particular engine automatically generates output and trace models.

On the right-hand side, our approach's primary artefacts and activities are organised into two fundamental stages: testing and repairing.

At the testing stage, modelling engineers have to manually specify the expected behaviour of the MTUT using contracts (number 1 in the figure), as in [5,6,16]. Contracts are a structured and formal way of defining invariants, pre- and postconditions, which must be fulfilled by the model transformation execution. Although our approach can define pre- and postconditions, we focus on invariants since they are the only relevant properties to compute our metrics upon. It is important to note that a contract can be related to several rules, i.e. it can express a requirement in a concise manner that actually requires multiple rules to be fulfilled, rendering contracts a lightweight method for model transformation testing. As stated by [17], a dedicated language for contract definition allows designers of transformations to make explicit the desired properties of a transformation before implementing or evolving it. The abstract syntax of MoTES is partially shown in Fig. 2. A detailed explanation of contract specification with MoTES is provided in Sect. 3.

Once contracts are defined, test oracles can be automatically derived from them (number 2 in the figure), as [2] suggests. For convenience, we automatically generate test oracle artefacts as model transformations (see Sect. 3.3) whose main mission consists in querying input–output model pairs from the MTUT and yielding a result report model, which conforms to the MoTES Result metamodel (see Fig. 3). That metamodel conveniently allows representing precision and recall values for every output pattern whose generation is constrained by contracts. Note that every `Result` element is labelled as True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN) for metric computation. Basically, given an input model, the execution of the MTUT (number 0 in the figure) generates its corresponding output model, so the input–output model pair is compared against the inclusion and exclusion criteria. Additionally, for easier interpretation, we propose a categorisation of various valid result combinations (see Sect. 4). Testing results are quickly computed utilising a transformation engine taking as input a test oracle (number 3 in the figure), i.e. contracts, an input model and its corresponding output model. Additionally, a testing report provides an overall vision of the MTUT behaviour for a concrete input model because the generation of all the output patterns is correctly reported: error and non-error results. MoTES assumes a complete enough test input model, i.e. covering all the specified constraints, is provided.

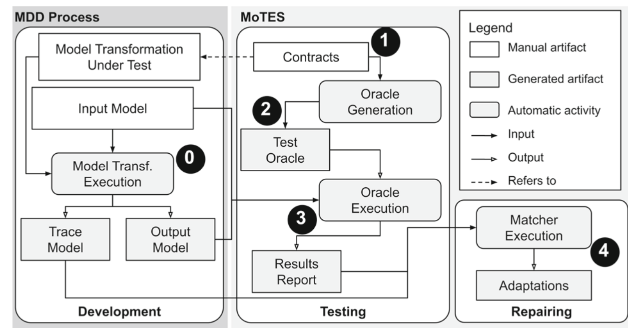

Fig. 1 Approach overview

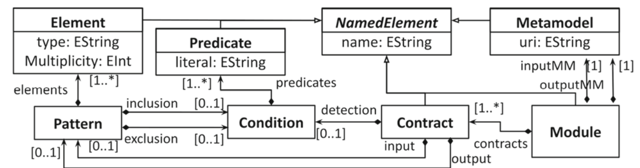

Fig. 2 MoTES contract metamodel

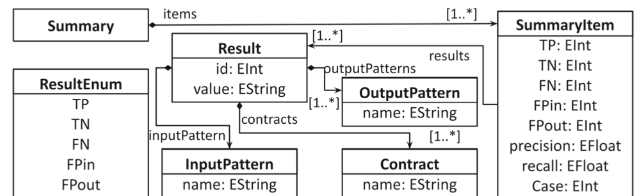

Fig. 3 MoTES result metamodel

Therefore, the generation of test input models is out of the scope of this work.

The final stage of our approach, the repairing stage, is shown on the right side of the figure. In this stage, a specific algorithm (see Algorithm 1) is responsible for matching contract results to MTUT execution traces to derive a particular repair suggestion for every transformation rule involved (number 4 in the figure).

Note that a transformation trace model mainly provides a mapping from output elements to transformation rules. Even though these recommendations refer to specific model transformation rules, our approach can still be considered black-box testing since we do not require the static analysis of the actual model transformation code, but only the traceability information, which is provided as a byproduct after the MTUT is executed. A complete specification is presented in Sect. 5.

Table 1 Star example transformation rules

Rule	Input pattern	Output pattern
S_R1	sq:Square (sq.name!="E")	st:Star st.name=sq.name
S_R2	el:Ellipse (el.light==true)	st:Star st.name=el.name
S_R3	sq:Square (sq.name=="E")	st:Star st.name="F"

2.2 Running example

In the following, we introduce a running example [1] to present our approach better. This example will be used for contract definition, fault localisation, and fault repair.

Let us suppose that a model transformation is defined between a metamodel defining shapes and another one just specifying stars. In the input metamodel there are two types of shapes: `Squares` and `Ellipses`. All `Shape` elements contain two attributes: name (String) and light (Boolean). `Stars` are the only type of elements defined in the output metamodel, and they have just the attribute name (String). Both metamodels are presented in Fig. 4 (at top). Table 1 shows the rules defining such transformation in a language-agnostic manner: rule S_R1 generates stars from squares whose name is not "E"; rule S_R2 generates stars from ellipses with light, and rule S_R3 generates stars with name "F" from squares with name "E". Those rules then define the MTUT.

As aforementioned, our approach, in addition to contract specification, needs the input model, its corresponding output model, and the trace model produced by MTUT execution. Figure 4 shows an example of those three models (at the bottom). In this case, the input model contains four squares (A, B, C and E) and one ellipse (D). According to the MTUT specification, all of them are generating stars in the output model and the trace model stores the name of the particular rule generating each concrete star. Every trace stores the name of the applied rule and relates input (`inElems` reference) to output elements (`outElems` reference). The elements pointed by such references can be of any type of the input and output models because their actual type is `EObject`. For the sake of simplicity, in Fig. 4 those references are pointing to concrete types in the input and output models, but they can reference any type.

3 Contract specification with MoTES

MoTES defines a minimalistic, declarative, domain-specific language (DSL) for the specification of model transformation contracts, initially presented in [15]. The syntax of this language is based on three main elements: *input–output element relationships*, *detection criteria* and *inclusion/exclusion criteria*. This syntax is specifically designed to simplify the calculation of precision and recall metrics.

[1] https://www.eweb.unex.es/eweb/migraria/motes/stars.html.

Note that other approaches have already defined specific languages to define contracts for model testing (see Sect. 7). Although some of them are not clearly tailored to compute our metrics because they lack an explicit definition of pattern, for instance, Tracts [5] (see Sect. 6.2 for more differences), others may be successfully applied, for instance, PAMoMo [6]. Indeed previous contracts already defined by those languages might be reused to apply our approach, playing a complementary role in a testing scenario. Hence, we could provide additional test results for the same model transformations.

Therefore, the MoTES language might not be considered a novel contribution, but it plays a relevant role for the illustrative purposes of our approach. In the following, we present its abstract syntax, concrete syntax, and semantics.

3.1 Abstract syntax

The abstract syntax of MoTES is defined as an Ecore metamodel, partially shown in Fig. 2, whose main concept is `Contract`. A contract contains the following elements:

- Contract name: the identifier for the contract.
- Input pattern: the typed elements to match over in the input model.
- Input exclusion: input elements that fit these criteria will not be matched.
- Input inclusion: only the input elements which fit these criteria will be matched.
- Output pattern: the typed elements to match over in the output model.
- Output exclusion and inclusion: same as for input.
- Detection: it defines a relationship between an input pattern and an output one, i.e. a constraint between the input model and the output model, consequently constraining the model transformation.

Input and output patterns are defined according to a collection of input/output elements and some conditional criteria for their inclusion and exclusion. Note that it is possible to replace exclusion criteria with negative inclusion criteria, but we have found our choice of using both as more intuitive, and it conveys better the set-oriented view of our approach. It is also important to stress that only input metamodel types are allowed in the input pattern, while only output metamodel types are allowed in the output pattern. Conversely, only detection criteria can specify expressions concerning elements from input and output elements because their mission is to express conditions that input–output relationships must hold (invariants).

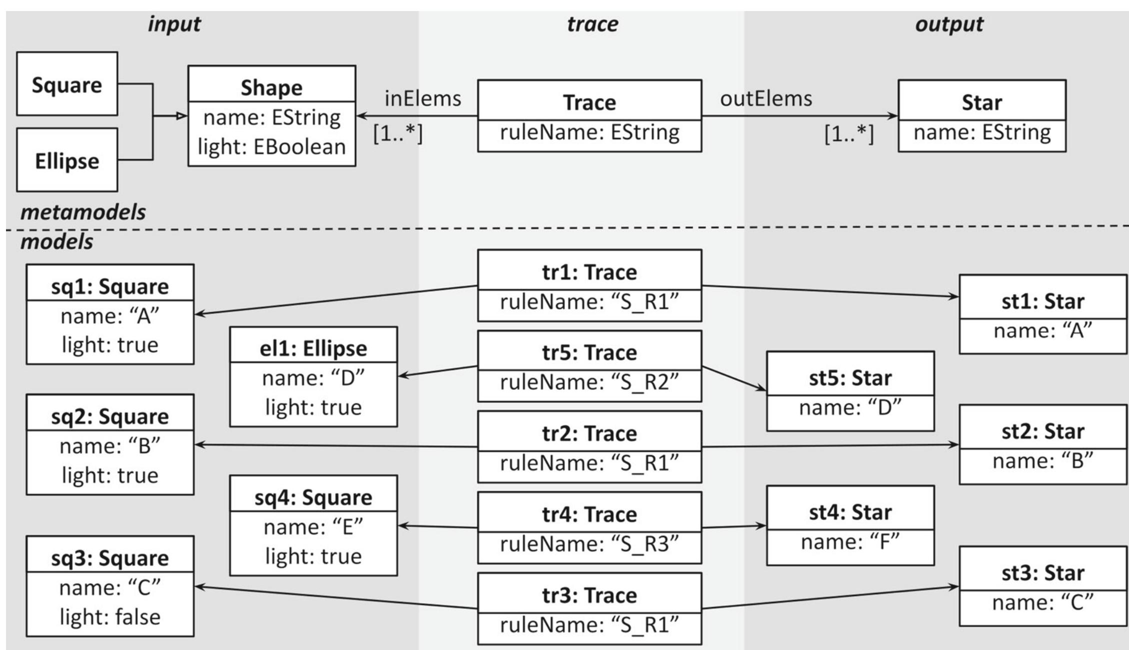

Fig. 4 Star example: input, output, and trace (meta)models

Although patterns allow for the specification of many-to-many relations between input and output elements, each input pattern is associated with exactly one output pattern at the pattern level. The support for pattern cardinality would be very interesting, but it remains as future work. Nevertheless, given that a MoTES contract establishes a one-to-one relationship between an input pattern instance and an output pattern instance, multiplicity (set/size) invariants are implicitly held in MoTES contracts. As a result, this feature would reduce the number of contracts, for example, compared to Tracts (see 6.2 for more details).

Furthermore, the definition of input or output patterns is not mandatory for two special cases: preconditions and postconditions. Only the input pattern needs to be specified to define a precondition contract. Meanwhile, to define a postcondition, only its output pattern needs to be specified. Detection criteria are not necessary in those special cases because there is no need to define a relationship between input and output patterns. Preconditions and postconditions may define additional constraints in input and output models, respectively, but they are not used for precision and recall computation. Therefore, we do not consider preconditions and postconditions in the rest of this work for the sake of brevity and clarity.

3.2 Concrete syntax

The textual concrete syntax of MoTES is defined using Xtext [18]. This simple syntax permits specifying all the relevant concepts of our approach. As shown in Listing 1, input and output patterns are specified as a list of typed in/out elements, while all the different criteria are specified as a list of named predicates.

Listing 1 MoTES Concrete Syntax

```
contract <cname>{
input: { (<iT1>:<iv1>, ... <iTN>:<ivN>)
inclusion: <p_name>(<p_expression>)...
exclusion: <p_name>(<p_expression>)...
}
output: { (<oT1>:<ov1>, ... <oTN>:<ovN>)
inclusion: <p_name>(<p_expression>)...
exclusion: <p_name>(<p_expression>)...
}
detection: <p_name>(<p_expression>)...
}
```

We use predicate expressions for the specification of all the different criteria of a contract (inclusion, exclusion and detection). Specifically, the following subset of first-order logic is used for criteria definition: the universal quantifier (variables of a contract are universally quantified), the existence quantifier (for expressing some conditions of output elements), conjunction, disjunction, negation and equality. Moreover, predicates can be logically connected by conjunction and disjunction operators. Nevertheless, for implementation purposes, we are using OCL because its expressiveness is far more than enough, and we have no intention of defining a new expression language.

Named predicates is another pragmatically convenient feature of the MoTES language. Named predicates provide testers with fine-grained test results by presenting the particular predicate failing in a contract. For instance, let us assume

a contract that defines an invariant between input and output elements to check that output elements are generated and their properties are appropriately bound. If the binding is complex (many properties involved and difficult assignment expressions), providing testers with the specific faulty binding may significantly reduce the repairing effort. By means of named predicates, testers can assign a name to each expression guarding a concrete property binding. Therefore, when an error occurs, MoTES will yield the name of the predicate whose expression is not satisfied, and testers can pinpoint the source of error more accurately.

Another interesting feature of the MoTES language consists of inter-contract invocation. One contract can use the special predicate `invoke` to call a previously defined contract by passing as arguments: the invoked contract name, an input instance matching its input pattern, and another one for its output pattern. The usage of this special predicate is constrained to the detection clause of a contract. Therefore, it works as a modularisation means by fostering code reuse (see 6.2 for more details).

Regarding the star example, let us suppose modelling engineers now decide stars must be only generated from lighted squared shapes. In this sense, the MoTES contract, shown in Listing 2, defines the constraints to consider for the transformation, i.e. squared shapes with its light property True must generate stars with the same name.

Listing 2 An example of a MoTES contract for the star example

```
contract Square2Star{
input { (Square:i)
inclusion: pi1(i.light = True)
}
output { (Star:o) }
detection: p1(i.name = o.name)
}
```

3.3 Semantics

In this section, we will outline a denotational semantics of the contracts definition language. We translate the contracts from the MoTES DSL syntax presented in Fig. 2 and Listing 1 to ATL[2] [19] by means of a code generator built upon Xtend[3].

For each contract, an ATL matched rule will be generated using a template. That rule will generate `Result` elements in the resulting model by querying the input/output models to relate an input pattern instance to an output pattern instance, according to the inclusion/exclusion and detection criteria (see Sect. 4.2 for a detailed description). For example, the test oracle for Listing 2 is shown in Listing 3.

The generated rule takes the name and the input pattern of the MoTES contract (`Square2Star` and `Square`, respectively). As output it generates a `Result` element (line 5) containing input/output patterns, the contract name and the result value of type `ResultEnum`. The `inputPattern` refers to a particular input pattern instance (line 20), while the `outputPattern` may refer to one or many instances of the output pattern (line 21). The function `getOutput` (line 47) is responsible of returning all the output instances related to a concrete input pattern instance.

Contract inclusion/exclusion and detection criteria are used to generate final `isExcluded` (line 26) and `isTransformed` (line 29) helpers for the input type, which act as proxies of the defined predicates. Eventually `resultValue` helper (line 33) computes the proper result (FP, FN, TP, TN) for the matching input–output relationship at instance level (see Sect. 4.1 for a detailed description). For example, if the MTUT has generated a Star (named C) from a lighted Square with the same name, the output of this ATL transformation would be a Result element named "Square2Star", whose input pattern refers to Square C and output pattern to Star C and its result value is TP (True Positive).

The oracle in Listing 3 is executed using an ATL engine, as shown in Fig. 1 (oracle execution), by taking as input: the input and output model of the MTUT and the contract specification (test oracle). As a result, the oracle execution produces an output model conforming to the results metamodel presented in Fig. 3. In Sects. 4 and 5 we detail how we interpret these results and what actions can be performed based on these results, which are also the main contributions of the paper.

3.4 Formalisation

We will use Triple Graph Grammars (TGGs) [20] as the foundation to formalise the MoTES contract specification DSL, the result interpretation and classification of input and output elements into the sets TN, FN, TP, and FP. Details of TGGs are out of the scope of this paper, and the interested reader may consult [20]. Here we only explain how we employ TGGs in our formalisation. TGGs is a declarative formalism for the specification of bidirectional translations between different graph languages. They generate languages of graph triples which consist of a source graph and a target graph, plus a correspondence graph between them (hence the name "triple"). The correspondence graph is used to define the traces between the source and target graphs explicitly. These three graphs ($Source \xleftarrow{src} Corr \xrightarrow{trg} Target$) are connected by two graph morphisms src, trg (i.e. maps which send nodes to nodes and edges to edges while respecting the source and target of the edges) from the correspondence graph to the source and target graphs. In the bidirectional transformations literature, TGGs are used for forward and

[2] We chose ATL for convenience, but MoTES contracts could also be transformed to other transformation languages.

[3] https://www.eclipse.org/xtend/.

Listing 3 MoTES sample Oracle generated from Listing 2

```
1  rule Square2Star{
2    from
3      input : inMM!Square
4    to
5      result : resultMM!Result (
6        inputPattern<-inPattern,
7        outputPattern<-outPattern,
8        contract<-contract,
9        value<-thisModule->resultValue(
10             input.isExcluded,input.is
11       Transformed)
12       ),
13
14   inPattern: resultMM!Pattern
15     (name<-inMM!Square.name),
16
17   outPattern: resultMM!Pattern
18     (name<-outMM!Star.name),
19
20   contract: resultMM!Contract
21     (name<-'Square2Star')
22
23   do {
24     result.inputPattern.object <- input;
25     result.outputPattern.object <-
26       thisModule->getOutput(outMM!Star,
27     input.name);
28   }
29 }
30
31 helper  context inMM!Square def : isExcluded :
32   Boolean = not (self.light=True);
33
34 helper context inMM!Square def : isTransformed
35   Boolean = outMM!Star.allInstances()->
36         one(obj | self.name=obj.name);
37
38 helper def:
39   resultValue(excluded:Boolean,
40     transformed:Boolean):
41     resultMM!ResultEnum =
42     if excluded then
43       if transformed then #FP
44       else            #TN
45       endif
46     else
47       if transformed then #TP
48       else            #FN
49       endif
50     endif;
51
52 helper def :
53   getOutput(e1: OclAny, name: String) :
54     OclAny =
55     e1.allInstances()->select(obj |
56     name=obj.name);
```

backward translations, which take either the source or the target graph as input and produce as output, respectively, an appropriate target or source graph in addition to the correspondence graph. TGGs are also used for incremental synchronisation, which are operational programs that propagate changes made in one artefact (source or target graph) to corresponding changes in another existing artefact (target or source graph), respectively [21].

As in the Graph Transformations (GT) framework [22], the component graphs in TGGs may be typed, attributed graphs with type-inheritance [23]. This kind of graphs is inspired by UML- and EMF-like models in which graph

Fig. 5 TGG rule corresponding to contract in Listing 2

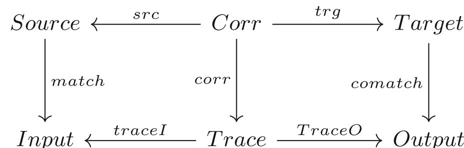

Fig. 6 TGG match

nodes can have (1) attributes with values from a predefined domain, e.g. String or Integer, and (2) inheritance edges which are used in the same sense as in object-oriented models.

Figure 5 shows a TGG rule which represents the contract in the illustrative example in Listing 2, i.e. `Square2Star`. We use some syntactic sugar to indicate that the parts which are marked with ++ will be added in forward translations—i.e. when a match of the $Source$ is found in an $Input$ graph, co-matches of the $Corr$, $Target$, and the corresponding morphisms src, trg will be created, as shown in Fig. 6. As an example, a match of a TGG rule corresponding to the contract in Listing 2 in a trace model like the one shown in Fig. 4 corresponds to a TGG morphism, which in turn consists of a triple of graph morphisms from the source (i:Square), target (o:Star) and correspondence (Sq2:St) graphs into the input, output, and trace models, respectively, in the lower part of Fig. 4.

Considering the concrete syntax in Listing 1, `input` corresponds to $Source$ while `output` corresponds to $Target$. Moreover, `inclusion` and `exclusion` criteria correspond to pre- and postconditions which are formulated as expressions over attribute values. For `input` patterns, negative and positive application conditions can also be used to express these criteria. Finally, `detection` criteria correspond to the graph morphisms src, trg from $Corr$ to $Source$ and $Target$, which may also be augmented with expressions over attribute values. By employing TGG as the formal foundation, we can rely on the rich theory and results of TGG (and GT) to verify the contracts, e.g. one can reason about conflicts, contradictions and relations between contracts [17].

4 Result computation and interpretation

4.1 Metric-based test oracle

A metric-based test oracle and output-centred reports are keys to simplifying result interpretation. We use precision

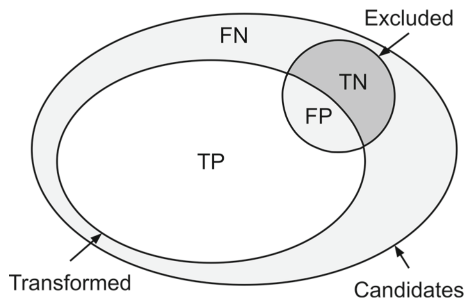

Fig. 7 Candidate set and subsets

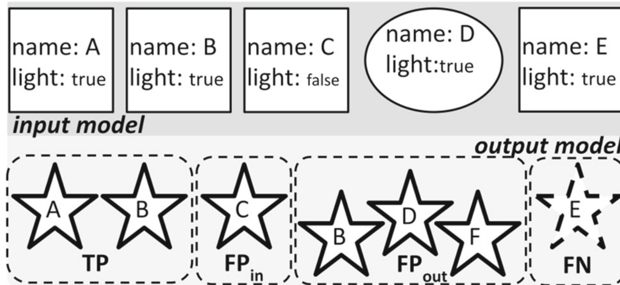

Fig. 8 Illustrative example to define TP, FP and FN

and recall metrics for this purpose, which are calculated for every output pattern. That way, we can get an overall measure of the correctness of all the transformation rules generating that kind of output pattern. Test result reports that are focused on contracts are only helpful to locate failures, while result reporting focused on output can provide additional information, e.g. how many output elements are affected by a specific error. Furthermore, we believe output-focused result reporting aligns better with the way model transformation designers think since they are usually concerned about the output (not contracts) when building transformations.

In the following, we use the star example again to illustrate the different definitions presented herein.

4.1.1 The candidate set

Precision and recall metrics are defined in terms of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). We explain what these four categories represent using the sets of input elements (candidate input pattern for generating a particular output pattern) depicted in Fig. 7. Considering the MoTES contract defined in Listing 2, we can compare the input–output model pair generated by the MTUT, shown in Fig. 8.

The larger set, *Candidates*, represents all the patterns in the input model suitable for generating a concrete output pattern. In the star example, all square shapes in the input model are candidates. Inside the candidate set, two other subsets are depicted: *Transformed* and *Excluded*. All the input patterns generating output patterns belong to the *Transformed* subset. To find them, we propose the definition of detection criteria, as explained in Sect. 3. In the example contract (Listing 2), the criteria used is that both patterns must have the same name. Therefore, squares A, B and C belong to the *Transformed* subset because we can find stars with the same name in the output model. Conversely, the input patterns contained in the *Excluded* subset are those that should not be transformed. Input patterns that are not supposed to be transformed are those which are satisfying the exclusion criteria. To detect them, we propose the definition of inclusion and exclusion criteria, also presented in the previous section. In the example contract (Listing 2), the exclusion criteria used for square shapes is having the light attribute set to false. Therefore, square C is part of the *Excluded* subset.

Using these three sets, it is possible to obtain the values required to calculate the precision and recall metrics for each output pattern. Thus, for every kind of output element, there is a partition of the set of candidates in four subsets, as depicted in Fig. 7.

- *True Positives (TP)*. Number of non-excluded input elements that have been appropriately transformed in output elements. In the example, Stars A and B are correctly generated from Squares A and B, i.e. they are TP.
- *False positives (FP)*. Number of excluded input elements that have been transformed into output elements by mistake. Star C is incorrectly generated, but it can be related to Square C, which is explicitly excluded by the contract, i.e. it is a FP_{in}
- *True Negatives (TN)*. Number of excluded input elements that have not been transformed.
- *False negatives (FN)*. Number of non-excluded input elements that have not been transformed. Star E (dashed line star in the figure) was expected because there is an input element, Square E, eligible to generate a Star and inside the contract scope. So the missing Star E is an FN.

Although we are using the terms TP, FP, TN and FN to name these subsets defined in the candidate set (input model), they classify the output elements generated from elements of each subset. Therefore, output elements are conceptually marked as TP, FP, TN or FN, as illustrated in Fig. 8.

4.1.2 Classifying output elements based on TGG rules

In Sect. 3.4, we outlined how the contracts are formalised as TGG rules and briefly explained how such rules are applied (see Fig. 6). We use these TGG rules in check mode with forward translation semantics, i.e. instead of applying the rules and generating a target model, we check whether for each match of *Source*, there exist co-matches of *Corr* and *Target*. Based on these matches, we classify the output model elements into the four above-mentioned sets as fol-

lows, where $s \in Source$, $t \in Target$, $i \in Input$, and $o \in Output$ are all model elements. To make this classification intuitive, we use the convention $comatch(t) = \perp$ to indicate that t does not have an image in $Output$. Moreover, we simplify the formulae below by assuming that s and t (resp. i and o) are in correspondence.

$$t \in TP\colon match(s) = i \wedge comatch(t) = o$$
$$s \in FN\colon match(s) = i \wedge comatch(t) = \perp$$
$$t \in FP\colon match(s) = \perp \wedge comatch(t) = o$$
$$t \in TN\colon match(s) = \perp \wedge comatch(t) = \perp$$

Note that here we include the TN set as a formula only to explain its meaning in terms of matches of the TGG rule. In practice, this desired situation means that unexpected elements were not generated; hence, no matches are found.

4.1.3 FP_{out}

As a novel extension of this work, we propose herein to divide FP into two different subsets: FP_{in} (FP in Fig. 7) and FP_{out}. Note that in order to explain FP_{out} thoroughly, we have added a Star B to the output model in Fig. 8, whose generation is not covered by the MTUT presented in Table 1 to keep the example simpler. FP_{out} are those output patterns incorrectly generated that cannot be related to any input element from the candidate set, for example, Stars B (second instance), D and F. According to its origin, we distinguish three different types of FP_{out}:

- *Replicas* They are caused because there are more output pattern instances than expected for a concrete input pattern instance. For example, Square B has already generated another star, so the second instance (replica) is not expected.
- *Unrelated* It means that the generation of the output pattern is not constrained by any contract. For instance, Star D has been generated from a different type of shape (input pattern), which is not constrained by any contract.
- *FN-associated* They are a special case of unrelated, which relates the unexpected output pattern instance (FP_{out}) to an FN case. For example, there is no Square named F, so Star F does not match any candidate Square (detection criteria), i.e. FP_{out} unrelated. However, at the same time, there is an FN case involving input element Square E. Star F is supposedly generated from Square E, but an error has been inadvertently introduced in property binding, so their names do not match. Note that such error produces two different consequences in the output model: a FP_{out} and its associated FN.

We have found FP additional subdivision helpful to interpret test results for adaptations suggestion better. However, concerning metrics computation (precision and recall calculation), all of them are aggregated as FP.

4.2 Metrics computation

In this work, we are interested in a fast and practical method to assess if a model transformation is working as expected for a concrete input model. Then, we also check to which degree the generated output model satisfies a set of contracts. From this point of view, a model transformation may be considered an information retrieval process, i.e. patterns of input elements trigger the generation of patterns of output elements. In pattern recognition and information retrieval with binary classification, precision and recall are standard metrics of the relevance of the results obtained. Precision is the fraction of retrieved instances that are relevant, while recall is the fraction of relevant instances that are retrieved. We compute precision and recall using their classic formulae:

$$\text{Precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|} \quad \text{Recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}$$

According to these formulae, their values can range from 0 to 1, but we normalise them to the range 0–100 for the sake of readability.

Metrics computation is implemented as a three-step process (*Results Report* in Fig. 1). First, an intermediate result model is generated by the execution of the test oracles, which are model transformations derived from contracts (see Sect. 3.3). This transformation queries input and output models and marks every input pattern instance as TP, TN, FP_{in} or FN (result types). Basically, it iterates over all the input pattern instances inside the candidate set of every contract to evaluate their `isTransformed` and `isExcluded` predicates. According to such predicates, a particular result type is selected for the Result element generated.

Secondly, we define an additional step to compute FP_{out}, because they need different processing. All output pattern instances not related to an input pattern in the previous step are marked as FP_{out} because they are instances of unrelated or FN-associated FP_{out}. Note that, in this case, neither the contract nor the input pattern can be set in the Result element because it is not inside the candidate set defined by the contracts. Therefore, default values are assigned in this case to be further processed in repairs computation. Conversely, replicas are detected by searching for results containing more than one output pattern for a concrete input pattern. As the final product of those two steps, a result model containing a `Result` element for every relationship between input and output pattern instances is generated. Output patterns are then indirectly marked as TP, TN, FP or FN by marking the `Result` instance defining its input–output relationship.

Table 2 MoTES result cases

TP	FP	FN	PRECISION	RECALL	CASE
>0	0	0	100	100	C_{TP}
>0	>0	0	t_p	100	C_{TPFP}
>0	0	>0	100	t_r	C_{TPFN}
>0	>0	>0	t_p	t_r	C_{TPFPFN}
0	>0	>0	0	0	C_{FPFN}
0	0	>0	NA	0	C_{FN}
0	>0	0	0	NA	C_{FP}
0	0	0	NA	NA	C_0

Finally, this result model is queried to aggregate all the results and provide the developer with an easy-to-understand comprehensive report, grouped by output patterns. All the `Result` instances for a concrete output pattern are summed to count for TP, TN, FP or FN, so precision and recall metrics can be computed for every output pattern. Therefore, precision and recall provide a thorough examination of the MTUT behaviour by output pattern according to all the MoTES contracts defined.

In the star example, after the execution of the first two steps of metrics computation, the generated result model contains 6 `Result` instances (illustrated in Fig. 8) for the output pattern considered (Star): 2 are marked as TP, 1 is marked as FPin, 2 are marked as FPout, and 1 is marked as FN. Note that the replica of Star B, marked as FP_{out}, is not considered here to be consistent with the running example defined. Therefore, by applying the metrics formulae, we get a precision of 40 and a recall of 67 for the Star elements.

4.3 Results categorisation

To simplify result interpretation we reduce the set of possible values for the metrics to four:

– *100*. Perfect result.
– $0 < t < 100$ (t from threshold: t_p for precision and t_r for recall in Table 2).
– *0*. Worst result.
– *NA*. Value cannot be computed (division by zero).

According to these values, Table 2 presents the eight possible result combinations. Those cases provide testers with a uniform method to classify the results obtained for each output pattern. In the star example, the result obtained (precision and recall values) for Stars (output pattern) can be further classified as a C_{TPFPFN} case (see description below). Following, a concise description of every case is introduced:

1. C_{TP} This is the perfect situation. All expected elements were generated, and all generated elements were expected.
2. C_{TPFP} All expected elements were transformed; however, excluded elements were also transformed.
3. C_{TPFN} All transformed elements were expected; however, not all expected elements were transformed.
4. C_{TPFPFN} Neither all expected elements were transformed, nor all transformed elements were expected.
5. C_{FPFN} This is the worst situation. No expected element was transformed, and no transformed element was expected. It is an extreme version of C_{TPFPFN}.
6. C_{FN} There were expected elements, but no element was transformed. It is an extreme version of C_{TPFN}.
7. C_{FP} There were no expected elements, but some elements were transformed. It is an extreme version of C_{TPFP}.
8. C_0 There were no expected elements, and no element was transformed. It represents an exceptional case: all the contracts for that output pattern are not applicable to the input model.

In C_{TPFP}, C_{TPFN} and C_{TPFPFN}, different values for the threshold t may imply diverse adaptation efforts. A proper threshold value allows the user to find a balance between effort and correctness. Therefore, we consider them in a generic way in the following, since t can get different values on each application scenario. For instance, in a reverse engineering process, it might be acceptable to get some FP for particular output patterns because they can be effortlessly filtered out in a following step. Conversely, trying to fix the faulty transformation rules might be comparatively more complex and expensive.

5 Suggesting adaptations

At this point, precision and recall (test results) have been calculated for each output pattern whose generation is constrained by the contracts defined, and the generation of every output pattern then receives a classification according to its test results. Recall that traceability knowledge is kept for each output pattern (see Fig. 4), detailing which rules operate on a particular output pattern. Combined with this classification, MoTES can then suggest a fixing action per output pattern, per contract, and per rule. Those actions have been defined upon the observed consequences of failing transformation rules. A mutations catalogue for model transformations, which defines mutation operators and their consequences in the output model, has been analysed from the perspective of precision and recall metrics complemented by a more detailed examination of TP, FP_{in}, FP_{out} and FN values.

5.1 Mutations in transformations

As a foundation for repair actions, we have used the catalogue of mutations for ATL transformations presented in [24], which are also considered in Sect. 6 to perform an experiment replication. Table 3 shows the set of mutations considered and their consequences in the output model, where a consequence enclosed within square brackets means that it may happen or not.

Since our approach analyses only input and output models, we are primarily interested in getting good coverage of the consequences of the mutations but not full coverage of potential mutations. Those consequences may be: (1) Object Addition (OA), (2) Object Deletion (OD), (3) Object Replacement (OR), (4) Relationship Addition (RA), (5) Relationship Deletion (RD), (6) Object Property Initialisation (OPI), (7) Object Property changing to Null (OPN), and (8) Object Property Modification (OPM).

Additionally, Table 3 presents those consequences in terms of FP_{in}, FN and FP_{out} to better understand them from the point of view of our approach. In the following, we explain them organised by the mutant target.

Target: matched rule and in/out pattern element Mutants with those targets produce a new input–output relationship constrained by no contract. Those kinds of mutation operators generate output pattern instances that cannot be related to any input pattern instances by the defined contracts (candidates in Fig. 7). Therefore, all those output pattern instances are marked as FP_{out}, which may be replicas, FN-associated or unrelated depending on the case.

Target: filter In contrast, these mutants would alter an established input–output relationship constrained by an existing contract. Therefore, they only generate FP_{in} (with or without FN) because either they accept excluded input candidates or filter out too many included ones.

Target: binding Finally, these last mutants are only relevant when they change a property included in the detection criteria of the constraining contract. In that case, they always produce FP_{out} of the type FN-associated, so FN-FP_{out} pairs for the same output pattern appear.

Note that there exist other catalogues of mutant operators, like the one defined in [25] which collects all mutant operators in the literature and proposes new ones derived from the authors' own experiences. They define new mutant operators to cover common syntactic errors in ATL divided into: typing errors or faults causing a runtime error. Typing errors are interesting for static analysis, but they entail no new output consequences. Furthermore, introducing faults makes the transformation not executable, hence not valid for dynamic approaches. In conclusion, there are no other mutant operators that our approach needs to consider as far as we know.

Table 3 Transformation mutation operators and consequences. First four columns defined by [24]

Mutant	Target	Type	Consequences	FP_{in}	FN	FP_{out}	Actions
M.1	Matched rule	Added	OA:[RA]	No	No	Yes (unrelated)	Delete
M.2	Matched rule	Deleted	OD:[RD]	No	Yes	No	Create
I.1	In pattern element	Added	OA:[RA]	No	No	Yes (replica)	Constrain
I.2	In pattern element	Deleted	OD:[RD]	No	Yes	No	Relax
I.3	In pattern element	Class changed	OA:OD:[RD]:[RA]	No	Yes	Yes (FN-associated)	Relax and constrain
F.1	Filter	Added	OD	No	Yes	No	Relax
F.2	Filter	Deleted	OA	Yes	No	No	Constrain
F.3	Filter	Condition changed	OA:OD	Yes	Yes	No	Relax and constrain
O.1	Out pattern element	Added	OA:[RA]	No	No	Yes (replica or unrelated)	Delete
O.2	Out pattern element	Deleted	OD:[RD]	No	Yes	No	Create
O.3	Out pattern element	Class changed	OR:[RA]:[RD]	No	Yes	Yes (FN-associated)	Delete and create
B.1	Binding	Added	OPI:[RA]	No	Yes	Yes (FN-associated)	Check Binding
B.2	Binding	Deleted	OPN:[RD]	No	Yes	Yes (FN-associated)	Check Binding
B.3	Binding	Value changed	OPM:[RA]:[RD]	No	Yes	Yes (FN-associated)	Check Binding

5.2 Repair actions

For the sake of simplicity, we believe that any evolution scenario (or repair process) can be represented as a sequence of six basic actions. The column *Actions* in Table 3 shows how to canonically solve every mutation operator using these basic actions. So, in order to adapt our original transformation to a new application scenario (or repair it), we need to know the proper sequence of actions to perform over the initial transformation rules. Those six basic actions are:

- *Create (Creat.)* A new transformation rule (or output target) should be created to satisfy a contract.
- *Delete (Del.)* A transformation rule (or output target) is useless, and it can be marked for deletion.
- *Constrain (Constr.)* When FP_{in} are generated, additional restrictions may need to be included in the selection criteria for input elements of a concrete transformation rule, i.e. check the rule filter. Conversely, when FP_{out} appear in the test results, incorrect input pattern elements may need to be deleted.
- *Relax (Relax)* Some restrictions may need to be removed from the selection criteria for input elements of a concrete transformation rule, i.e. check the rule input pattern or filter.
- *Check Binding (Bind.)* Property binding should be checked to avoid errors producing FN-FP_{out} tandems.
- *No Action (NoAct.)* When no action needs to be performed on a concrete transformation rule. No action is defined to provide a comprehensive list of actions for all the involved transformation rules in the MTUT. This is the default action for some cases, like the rules that get C_{TP} as result.

Table 4 presents a listing of suggested actions to perform for any possible test result case, according to the number of transformation rules involved in the generation of a concrete output pattern. Possible values for the number of transformation rules involved are 0 (no rules) or n (n>0). In order to provide more concrete repair actions, the latter case may be decomposed as:

- n_{TP} rules generating only TP,
- n_{FP} rules generating only FP,
- n_{TPFP} rules generating both TP and FP,
- n_0 rules not generating anything.

Table 4 can be used as a reference sheet to recommend adaptation actions. It comprises 37 different possibilities: thirty-three are related to concrete transformation rules, and the remaining four are exceptional situations. These exceptions have no value in the fourth column, and they always suggest the action *Create new rules*. Due to their exceptional nature, the recommended action should only be performed when other suggested actions for a concrete output pattern have been already tried, but metric values are still not good enough for our purposes. Additionally, for testing cases C_{TPFPFN} and C_{FPFN} (FN and FP_{out} both appear for the same output pattern), it is worthy of considering whether we obtain a similar number of FN and FP_{out}, which may imply FN-associated FP_{out} for a concrete output pattern. In such a situation, *Check Binding* is with significant probability the right action to take.

Eventually, engineers get a comprehensive list of adaptations grouped by the transformation rules involved in the generation of a particular output pattern or, from another point of view, particular adaptation actions are suggested for every transformation rule. The repairing of model transformations may then be completed with the assistance of the list of recommended actions per transformation rule.

Regarding the star example, given the MTUT defined in Table 1 and the MoTES contract defined in Listing 2, after the execution of MoTES we get the C_{TPFPFN} result for the output pattern `Star`, because MoTES has labelled the generated stars as shown in Fig. 8. By analysing the trace model, we can see that MoTES classifies rule S_R1 into the n_{TPFP} rule subset and rules S_R2 and S_R3 into the n_{FP} subset. As Table 4 suggests for such a result case, we decide to delete S_R2 and S_R3 because they are just generating FP_{out}, while we constrain S_R1 because it is generating both TP and FP_{in}. In this case, the Star labelled as FP_{in} is C, which has been generated from a non-lighted square, and the failing named predicate is `pi1` (checking the light property is true), specified in the input pattern inclusion criteria of the contract. Therefore, the concrete repair action consists in adding the following filter to S_R1: “(sq.light==true)”. Once repair actions have been performed, we rerun MoTES to test new MTUT behaviour and get the C_{TPFN} result, which means there are no more FP errors. Given that we have now only one rule, S_R1, we applied the suggested abstract action *Relax* on that rule. In this case, the failing named predicate is `p1` (contract detection criteria), which checks that input and output pattern instances have the same name for the candidate input element Square E. Therefore, by reviewing S_R1 we derive the concrete action of deleting the filter “(sq.name!=“E”)”. Again, we run MoTES to check whether our repair actions have worked as expected, and we get the C_{TP} result, meaning the MTUT is now working properly.

5.3 Repairs computation

Adaptation reporting is defined as a model query mixing both the MoTES result model and the transformation trace

Table 4 Suggested adaptation/repair actions

Prec.	Rec.	Case	Rules	Actions
100	100	C_{TP}	n	No Action
t_p	100	C_{TPFP}	n_{TP}	No Action
			n_{FP} FP_{in}	Constrain
			n_{FP} FP_{out}	Delete rule/out target
			n_{TPFP} FP_{in}	Constrain
			n_{TPFP} FP_{out}	Delete out target
			n_0	Delete
100	t_r	C_{TPFN}	n_{TP}	Relax or No Action
			n_{FP}	No Action ($n_{FP}=0$)
			n_{TPFP}	No Action ($n_{TPFP}=0$)
			n_0	Relax
				Create out target
			–	Create new rules
t_p	t_r	C_{TPFPFN}	n_{TP}	Relax or No Action
			n_{FP} FP_{in}	Constrain
			n_{FP} FP_{out}	Delete rule/out target
				Check Binding
			n_{TPFP} FP_{in}	Constrain
			n_{TPFP} FP_{out}	Delete out target
				Check Binding
			n_0	Relax
				Delete
			–	Create new rules
0	0	C_{FPFN}	n_{TP}	No Action($n_{TP}=0$)
			n_{FP} FP_{in}	Constrain
			n_{FP} FP_{out}	Delete rule/out target
				Check Binding
			n_{TPFP}	No Action ($n_{TPFP}=0$)
			n_0	Relax
				Create out target
				Delete
			–	Create new rules
NA	0	C_{FN}	0	Create new rules
			n_0	Relax or Delete
			–	Create new rules
0	NA	C_{FP}	n_{FP}	Delete
NA	NA	C_0	0,n	No action

Algorithm 1 Adaptation Actions query

```
1: procedure ADAPTATIONS(out, case, Adaptations)
2:   Results ← getInstances(mmR!Result, results)
3:   Contracts ← getInstances(mmR!Contract, contracts)
4:   ResultAndRules ← []
5:   for all result ∈ Results do
6:     rule ← getRuleId(r, traces)
7:     ResultAndRules.insert(Tuple(rule, result))
8:   end for
9:   ResultsPerContract ← []
10:  for all c ∈ Contracts do
11:    RC ← []
12:    for all tuple ∈ ResultAndRules do
13:      if tuple.result.contract.name = c.name then
14:        RC.insert(tuple)
15:      end if
16:    end for
17:    ResultsPerContract[c] ← RC
18:  end for
19:  ruleIds ← getAll(mmTrace!TraceLinkSet, traces)
20:  ResPerContractPerRule ← []
21:  for all rpcwr ∈ ResPerContract do
22:    RPC ← []
23:    for all rule ∈ ruleIds do
24:      RPR ← []
25:      for all tuple ∈ rpcwr.rules do
26:        if tuple.rule = rule then
27:          RPR.insert(tuple.result)
28:        end if
29:      end for
30:      RPC(rule) ← RPR
31:    end for
32:    ResPerContractPerRule(rpcwr.contract) ← RPC
33:  end for
34:  for all rpcpr ∈ ResPerContractPerRule do
35:    Adaptations[rpcpr] ← getActions(out, case, rpcpr)
36:  end for
37: end procedure
```

model[4], as illustrated by Alg. 1. The procedure defined computes the list of adaptation suggestions for a particular tuple of output pattern and result case, which can be obtained from the SummaryItem elements of the resulting model.

Firstly (lines 4–8), for each Result element, we query the trace model to obtain the identifier of the rule responsible for the transformation of its input pattern instance into its output pattern instance by searching for the Rule element containing the relationship between those two instances (*getRuleId*). In the case of Result elements marked as FP_{out}, we also obtain its input pattern from the trace model, so we can also assign this result to the contract constraining such pattern. When more than one contract is constraining that input pattern, we assign the result to all of them because they may be relevant for any of them. As a result, we get a list of tuples associating each result with the corresponding rule.

Secondly (lines 9–18), all those tuples are grouped by MoTES contract, so for each contract, we create the list of tuples containing the results obtained and their corresponding rules.

Thirdly (lines 19–33), for every contract, we aggregate the results by rule, so for every rule, a list of results is created.

Finally (lines 34–36), we map every combination of result case for an output pattern with the results obtained by every rule involved into a particular list of adaptations suggestions

[4] Any rule-based transformation engine with trace support might be used. Herein we use https://wiki.eclipse.org/ATL/EMFTVM#Advanced_tracing.

for that rule. Tables 9, 11 and 17 show particular examples of adaptation suggestions in an experimental setting.

As illustrated, our approach to suggest repair actions is completely independent of the model transformation language of the MTUT since rule identifiers are obtained by means of the trace model. Therefore, it is not necessary to statically analyse the model transformation rules to locate the faulty ones. However, such independence also means that we cannot execute concrete automatic fixes for model transformation rules. That is, our suggested actions define abstract repairs following the terminology specified by [13], but how to derive concrete repairs for a concrete transformation language from them remains as future work.

5.4 Repair selection

Our approach provides testers with data about transformation success according to the contracts defined, and eventually, it derives some repair suggestions stemming from those results. However, in some situations, human intervention is still needed for the final decision: (1) when several actions are suggested for a particular rule; (2) when conflicting suggestions are provided; and (3) when there is a long list of repair suggestions. Several examples of those situations are illustrated in Sect. 6 and discussed in Sect. 8. Although a detailed analysis of conflicts and prioritisation among repair actions is out of the scope of this paper, in the remainder of this section, we elaborate on them a little bit further.

Sometimes several actions may be suggested for a particular model transformation rule. In those cases, testers are responsible for selecting the final repair action to apply, but our approach helpfully narrows down the possibilities to a shortlist for each faulty rule. Actually, testers always have the final word because they can decide which suggestion makes more sense in light of the particular transformation rule involved.

Additional concerns might need to be taken into account when conflicts appear. For instance, two different result cases might suggest conflicting actions over the same model transformation rule. Furthermore, another source of conflicts might be combining the results obtained from multiple input/output model pairs. A deeper analysis of such conflicting suggestions should then be carried out, and proper solutions should be derived.

In real scenarios, MTUTs may have many complex rules, so many contracts need to be defined. In such cases, the list of test results and repair suggestions might become too long. Moreover, the order of application of repairs might also be relevant because different orders may entail different efforts. Therefore, for testers to select, some priority criteria could be followed to provide them with the best sequence of adaptations to follow.

Finally, human intervention is error-prone, so testers might incorrectly select the wrong repair action from the choices provided. Nevertheless, if a repairing is wrongly selected, the error will not be fixed, and test results will not improve. Therefore, for any action taken, our approach can be executed to see if results are getting better or worse for a concrete output pattern. Our approach is designed to be lightweight and easily interpretable, so it can be quickly re-executed to assess the impact of every repair action applied in the MTUT.

6 Evaluation

In this section, we discuss the validity and limitations of our approach. More specifically, we aim to answer the following research questions:

1. RQ1 - Applicability.
   (a) RQ1.1: Are all the errors appropriately detected?
   (b) RQ1.2: Are there appropriate suggested actions for all the detected errors?
2. RQ2 - Correctness.
   (a) RQ2.1: Are the detected errors (unsatisfied contracts) correct in the sense that all reported errors are representing real model transformation failures?
   (b) RQ2.2: Are the suggested actions correct in the sense that they indicate which rules to fix and how?
3. RQ3 - Usefulness. Could the approach be successfully used in a real application domain?

RQ1 and RQ2 have been answered by replicating a case study aimed at locating errors and suggesting repair actions in faulty model transformations by applying mutation analysis to two different transformation projects. In order to answer RQ3, we have applied our approach to helping us adapt to a new application scenario, an ATL transformation that implements a static analysis in a Model-Driven Reverse Engineering (MDRE) project. In particular, how to adapt it to a different legacy project.

6.1 Experiment replication

Given that our approach is based on model transformation testing, to assess its correctness and completeness, we have performed a comparative experiment with the static analysis approach based on Tracts [7]. Both in the original experiment and here, the tests aim at locating faulty model transformations from the case study in [26]. Additionally, in our case, we also propose repair actions for those faulty model transformations. The authors in [7] used mutation analysis [27]

Table 5 Transformation metrics overview

Metric	UML2ER	E2M	JSP2View
ATL LoC	77	1397	525
#Rules	8	40	19
#Helpers	0	40	14
#Bindings	5	329	99

Table 6 Metamodel Metrics Overview

Metric	UML	ER	Ecore	Maude	StrutsJSP	MVC
#Class	4	8	18	45	16	61
#Atts	3	1	31	17	13	62
#Refs	4	2	34	46	12	87
#Inhs	3	6	16	38	13	57

to inject faults into model transformations [28] systematically and then used their approach to locate those bugs. The purpose of a mutated rule is to emulate a model transformation that contains bugs and can be used to check whether the model transformation testing approach can identify them. To define the possible mutations of ATL transformations, they use the list of transformation change types presented in [29]. We have used the same testing artefacts (model transformations, contracts and mutants) to compute our test oracle and comparatively analyse the results of both approaches.

Setup For this experiment replica, we have used the following model transformation projects: (1) UML2ER, which takes as input a UML Class Diagram and outputs the equivalent Entity-Relationship diagram, and (2) E2M, which generates a Maude metamodel from an Ecore metamodel. Tables 5 and 6 present complexity data for the projects considered. In these case studies, input and output elements are uniquely identified; hence, concrete input–output relationships can be established. The following steps have been taken in order to set both experiments up properly:

1. Manual translation of the considered constraints from Tracts to MoTES.
2. Slight adaptation of the ATL transformations' source code to the specific requirements of the EMFTVM engine [5] (to meet the technical requirement for trace information of our current implementation platform).
3. Analysis and completion of existing test input models in order to check that they cover all the specified constraints.
4. Execution of the transformation under test using EMFTVM engine.
5. Execution of the ATL-based implementation of our approach for the MoTES contracts.
6. Report elaboration with the obtained measures.

The last three steps have been repeated for all the mutations considered in each case. Hence, in each iteration, a different mutation is applied to the original model transformation, one at a time.

Table 7 summarises the mutation operators considered and their consequences. The affected rules are explained in the following. As shown, most possible consequences are considered in this case study.

Measures For each mutation, we collect, grouped by output pattern, the total number of TP, FP, TN and FN yielded by checking all the contracts related to every output pattern. From these data, our test oracle is computed, i.e. precision and recall metrics are calculated. We then use the values those metrics provide to categorise the results obtained for every output pattern (result cases). Moreover, all the ATL rules involved in the generation of every output pattern are identified. By analysing those rules together with the previously collected and derived data, we are able to automatically suggest a list of recommended repair actions for every faulty transformation rule.

Results In this case, we have formatted into tables all the results derived from applying our approach to simplify its understanding. Those tables are presented in the context of each case study.

6.2 UML2ER case

In this case, study[6], the model transformation under test takes as input a UML Class Diagram and outputs the equivalent Entity-Relationship diagram.

6.2.1 Transformation rules

Table 8 shows the input–output pairs of every rule in this transformation, as well as whether the rule is abstract (*Abs*) and the rule it inherits from (*Inh*), if any. In ATL [19], rule inheritance can be used as a code reuse mechanism. Subrules have to match a subset of what their parent rules match, while subrules' target patterns need to extend their parent target patterns. A parent rule can be abstract, which makes it useful for inheritance but also non-executable.

6.2.2 Constraints and contracts

The constraints that should be satisfied by the model transformation are:

1. *U_Co1* All `Package` elements should generate a `Model` element with the same name.

[5] https://wiki.eclipse.org/ATL/EMFTVM.

[6] https://www.eweb.unex.es/eweb/migraria/motes/uml2er.html.

Table 7 ATL mutations and consequences considered (defined by [7])

Case	Mut.	Desc.	Rule	Cons.
UML2ER	U_M1	Binding change	U_R1	OPM
UML2ER	U_M2	Out pattern added	U_R3	OA;RA;OPI
UML2ER	U_M3	Filter addition	U_R8	OD
UML2ER	U_M4	Out pattern class change and binding deletion	U_R5	OR;OPN
E2M	E_M1	In pattern addition	E_R9	OA;RA
E2M	E_M2	Binding value change	E_R9	OPM
E2M	E_M3	Filter deletion	E_R10	OA
E2M	E_M4	Out pattern added	E_R20	OA;OPI
E2M	E_M5	Out pattern deletion	E_R29	OD
E2M	E_M6	Out pattern deletion	E_R1	OD;RD
E2M	E_M7	Filter addition	E_R38	OD

Table 8 Transformation rules (UML2ER)

Rule	Input element	Output element	Abs	Inh
U_R1	NamedElement	Element	✓	–
U_R2	Package	ERModel	–	U_R1
U_R3	Class	EntityType	–	U_R1
U_R4	Property	Feature	✓	U_R1
U_R5	Property	Attribute	–	U_R4
U_R6	Property	Reference	✓	U_R4
U_R7	Property	WeakReference	–	U_R6
U_R8	Property	StrongReference	–	U_R6

2. *U_Co2* For each `Class` in a `Package` an `Entity` in its corresponding `Model` should be generated.
3. *U_Co3* For each `Property` in a `Class` a `Feature` in its corresponding `Entity` should be generated.
4. *U_Co4* There should be as many `Element` instances as `NamedElement` instances.
5. *U_Co5* There should be as many `Model` elements as `Package` elements.
6. *U_Co6* There should be as many `Entity` elements as `Class` elements.
7. *U_Co7* There should be as many `Feature` elements as `Properties` elements.
8. *U_Co8* For each primitive `Property` in a `Class` an `Attribute` in its corresponding `Entity` should be generated.
9. *U_Co9* For each complex non-containment `Property` in a `Class` a `WeakReference` in its corresponding `Entity` should be generated.
10. *U_Co10* For each complex containment `Property` in a `Class` a `StrongReference` in its corresponding `Entity` should be generated.

For these ten constraints, ten contracts are defined in Tracts. Listing 4 shows an excerpt of the Tracts contract definitions. The interested reader might review [26] for more detailed information about these contracts. Those same constraints are defined in only five MoTES contracts (shown partially in Listing 5). Such reduction is possible mainly because multiplicity (set/size) invariants are implicitly held in MoTES contracts. Given that a MoTES contract establishes a one-to-one relationship between an input pattern instance and an output pattern instance, the number of input and output pattern instances has to be the same, whereas Tracts must make such invariants explicit (e.g. Tract U_Tr6). Another interesting difference is that MoTES allows inter-contract invocation. For instance, container correspondence of elements is checked by explicitly invoking a MoTES contract whose input and output patterns fit; meanwhile, Tracts does not provide any means of modularisation. Hence, this container check should be redundantly checked in those Tracts constraining deeper elements. In this sense, U_Tr10 includes the code to check the container correspondence for a concrete `Property-StrongReference` pair but also the code to perform the same check for its container `Class-Entity` pair. Conversely, U_Mo5 performs this check by invoking U_Mo2, which correspondingly invokes U_Mo1.

Listing 4 UML2ER Tracts excerpt

```
--Tract 06 (U_Tr6)
Class.allInstances->size() =
    EntityType.allInstances->size()

--Tract 10 (U_Tr10)
Package.allInstances->forAll(p | ERModel.allInstances
->one(e | p.name = e.name and p.ownedElements
->forAll(class | e.entities->one(entity |
entity.name = class.name and class.ownedProperty
->forAll(prop | prop.complexType <> null
implies entity.features->
        select(f|f.oclIsTypeOf(Reference))
-> one(f | f.name = prop.name and prop.isContainment
implies f.oclIsTypeOf(StrongReference)))))))
```

Listing 5 UML2ER MoTES contracts excerpt

```
contract U_Mo1{
input { (Package:i) }
output { (ERModel:o) }
detection: p1(i.name = o.name)
}

contract U_Mo2{
input { (Class:i) }
output { (EntityType:o) }
detection: p2(i.name = o.name) and
invoke(U_Mo1,i.immediateComposite,
    o.immediateComposite)
}

contract U_Mo5{
input {
(Property:i)
exclusion: pe5_1(i.complexType = null) or
pe5_2(i.isContainment = false)
}
output { (StrongReference:o) }
detection: p5(i.name = o.name) and
invoke(U_Mo2,i.immediateComposite,
    o.immediateComposite)
}
```

6.2.3 Mutations

Regarding the mutation analysis, the following mutations, summarised in Table 7, have been performed in the UML2ER ATL source code:

1. $U_M1(U_R1)$. Modification of the value feature of a binding in U_R1, which results in incorrectly initialised features (*name*) in the target model.
2. $U_M2(U_R3)$. Addition of an out pattern with two bindings in U_R3, which results in the creation of unexpected additional elements in the target model.
3. $U_M3(U_R8)$. Modification of a filter in U_R8, resulting in the generation of fewer elements than expected in the target model.
4. $U_M4(U_R5)$. Modification of the class feature in an out pattern and deletion of a binding in U_R5, which results in incorrectly initialised features in the target model.

6.2.4 MoTES results

For brevity, we do not present the final number of collected TP, TN, FP, and FN. We present rather the values of the metrics calculated from them. Table 9 presents, for each mutation (column *Mut.*), the affected output elements (column *Output*), the result case obtained (column *Case*), the MoTES contracts involved (column *Contract*), the types of result obtained (column *Res.*) and the adaptations suggested (column *Act.*) for every faulty rule (column *Rule*). Note that bold text in the column Act. indicates which action was finally selected in case of multiple suggestions. As shown, all mutations were detected by our approach, and proper repair actions were suggested for each case. In the following, a brief explanation of each analysed mutation is presented.

U_M1 When analysing test results, we find that all the contracts yield the same result: C_{FPFN}, which means that none of the expected elements has been generated, while all generated elements were not expected. All the input elements are marked as FN because no output element has been found to relate to, while all the elements of the output model were marked as FP_{out} elements, which means they could not be related to an input element. In addition, we get the same number of Fn and FP_{out}. In such a situation (C_{FPFN} and $FN{=}FP_{out}$), a thorough review of the bindings of every transformation rule involved is recommended, as stated in Table 4. Moreover, if specific named predicates have been defined for that criteria, MoTES can indicate which predicate is not satisfied and therefore narrow the search space for the incorrect binding.

In the end, for this first mutation, a single change has made all the rules fail. The rationale behind this is that all the rules are bound to the mutated one by an inheritance relationship. Since we use a black-box approach, we can conclude that all the rules are failing, but we cannot mark U_R1 as the source of the problem because this entails statically analyse the transformation rules. However, in such a case, in which all the contracts get the same extreme case, and we previously knew that all the rules are inside the same inheritance hierarchy, we conceivable could conclude the problem might be in the rules at the top of such hierarchy, again named predicates may be helpful (`p1` predicate failing implies the problem is in U_R1).

U_M2 The obtained results indicate that everything is working as expected, except for `EntityType` output elements, whose contract yields a 50% precision (some unexpected `EntityType` elements have been wrongly generated in the output model). Our approach can indicate straightforwardly that U_R3 (`Class2EntityType`) is the only guilty rule involved in the generation of such output pattern and responsible for the TP and FP_{out} results. The default suggested action when FP_{out} appears is to delete the out-

Table 9 Results and suggested adaptations (UML2ER)

Mut.	Output	Case	Contract.	Rule	Res.	Act.
U_M1	ERModel	C_{FPFN}	U_Mo1	U_R2	FN FP_{out}	**Constr.** Del.
U_M1	EntityType	C_{FPFN}	U_Mo2	U_R3	FN FP_{out}	**Constr.** Del.
U_M1	Attribute	C_{FPFN}	U_Mo3	U_R5	FN FP_{out}	**Constr.** Del.
U_M1	WeakRef	C_{FPFN}	U_Mo4	U_R7	FN FP_{out}	**Constr.** Del.
U_M1	StrongRef	C_{FPFN}	U_Mo5	U_R8	FN FP_{out}	**Constr.** Del.
U_M2	ERModel	C_{TP}	U_Mo1	U_R2	TP	NoAct.
U_M2	EntityType	C_{TPFP}	U_Mo2	U_R3	TP FP_{out}	Constr.
U_M2	Attribute	C_{TP}	U_Mo3	U_R5	TP	NoAct.
U_M2	WeakRef	C_{TP}	U_Mo4	U_R7	TP	NoAct.
U_M2	StrongRef	C_{TP}	U_Mo5	U_R8	TP	NoAct.
U_M3	ERModel	C_{TP}	U_Mo1	U_R2	TP	NoAct.
U_M3	EntityType	C_{TP}	U_Mo2	U_R3	TP	NoAct.
U_M3	Attribute	C_{TP}	U_Mo3	U_R5	TP	NoAct.
U_M3	WeakRef	C_{TP}	U_Mo4	U_R7	TP	NoAct.
U_M3	StrongRef	C_{FN}	U_Mo5	U_R8	FN	**Relax** (Creat.)
U_M4	ERModel	C_{TP}	U_Mo1	U_R2	TP	NoAct.
U_M4	EntityType	C_{TP}	U_Mo2	U_R3	TP	NoAct.
U_M4	Attribute	C_{FN}	U_Mo3	U_R5	FN	**Relax** (Creat.)
U_M4	WeakRef	C_{TPFP}	U_Mo4	U_R7	TP	NoAct.
			U_Mo4	U_R5	FP_{out}	**Constr.** Del.
U_M4	StrongRef	C_{TP}	U_Mo5	U_R8	TP	NoAct.

Table 10 Comparison Summary (UML2ER)

Mut.	Description	MoTES results	Detection	Direct
U_M1	Value changed in U_R1 binding	C_{FPFN} for all rules (inheritance)	Both	Both
U_M2	Out pattern element addition in U_R3	C_{TPFP} for U_R3	Both	Both
U_M3	Condition changed in U_R8 filter	C_{FN} for U_R8	Both	MoTES
U_M4	Out pattern class change in U_R5	C_{TPFP} and C_{FN} for U_R5	Both	Both

put target responsible for them. Therefore, our approach can locate the faulty rule and suggest the right repair action.

U_M3 In this case, our approach indicates everything is working as expected for any considered output pattern, except for `StrongReference`. For that output pattern, it yields a C_{FN} result, which means that, although there were expected elements, none of them has been generated. Only U_R8 is responsible for generating that output pattern so that it can be directly located by our approach. Different from U_M1, in this case, there are no extra output elements generated by that rule. Therefore, we can infer that the problem is related to the rule filter and not to the output pattern. As a result, instead of deleting the rule, we would select "Relax" and review the filter to know why it is filtering out too many elements.

U_M4 This is an interesting case because two different but related mutation operators have been applied simultaneously. Therefore, we get two test error cases (C_{FN} and C_{TPFP}) for two different output patterns (`Attribute` and `WeakReference`). However, only one model transformation rule is affected: U_R5. C_{FN} is just indicating that none of the expected elements have been generated (FN), while C_{TPFP} is indicating that some of the generated elements were not expected (FP_{out}), decreasing precision. By analysing both cases at the same time, we find that the source of the problem is just the rule U_R5 (`Property2Attribute`), which is incorrectly generating `WeakReference` elements (FP_{out}) but no `Attribute` elements (FN), and that the number of affected elements are the same for both cases. In this case, different (partially contradicting) actions are suggested:

- C_{FN} suggests to "Relax" rule U_R5.
- C_{TPFP} suggests to "Delete" rule U_R5 or "Delete one of its output targets".

Interestingly, we get that FP_{out}(`WeakReference`) = FN(`Attribute`), which generally means that bindings must be reviewed instead of relaxing or deleting the rule. In the end, we can easily conclude that U_R5 might be incorrectly generating `WeakReference` elements when it should generate `Attribute` elements. Therefore, the suggested actions can be properly applied to fix the rule.

6.2.5 Comparison summary

In this case, we have been able to run all the originally defined mutations under our approach, and the comparative results are presented in Table 10. Basically, both approaches can detect all the mutations (detection column), although slight differences appear when considering how easy it is to find the guilty rule (direct identification column). In the following, we present the comparison of locating the faulty rule for every mutation:

1. U_M1 Since it is a black-box approach, MoTES cannot deal with inheritance relationships between transformation rules. Therefore, it concludes that all the rules are failing according to the provided contracts. However, by using named predicates, it is straightforward to identify the source of error and, therefore, the faulty rule. A similar result is yielded by Tracts in this case: more than half of the Tracts contracts fail, specifically U_Tr1, U_Tr2, U_Tr3, U_Tr8, U_Tr9 and U_Tr10. Therefore, a testing engineer has to thoroughly review the values from three different tables to detect the guilty rule for each of the failing constraints.
2. U_M2 MoTES and Tracts can identify U_R3 as the faulty rule straightforwardly.
3. U_M3 MoTES can directly identify the faulty rule and suggest a repair action; however, Tracts yields four failing constraints: U_Tr3, U_Tr4, U_Tr7 and U_Tr10. So the engineer has to check table values for those 4 constraints to find one guilty rule, in this case. U_Mo4 is reported to be the most cumbersome case taking seven steps to find out the guilty rule.
4. U_M4 MoTES and Tracts can identify U_R5 as the faulty rule straightforwardly.

Summarising, MoTES might not precisely identify the faulty rule when syntactic relationships hold among rules if no named predicates are wisely used, as in U_M1; however, it provides additional information to testing engineers to help them with faulty rule identification. Conversely, MoTES seems to perform better than Tracts when the mutation changes filter conditions of the rules, e.g. U_M3. As a result, we may conclude that, for this case, our approach seems to be as correct and complete as Tracts-based static analysis presented by [7] when localising faulty model transformation rules. However, our approach is not just locating the faulty rules, but it also provides engineers with valuable action suggestions to guide their reparation.

6.3 E2M case

In this case, study[7], the model transformation under test takes as input a model which conforms to the Ecore metamodel and outputs an equivalent model which conforms to the Maude metamodel [26].

6.3.1 Transformation rules

As shown in Table 6, the E2M case is a large case study containing 40 transformation rules and more than one thousand lines of code. Nevertheless, the constraints already defined for this case only affect a subset of those rules. The relevant model transformation rules are shown in Table 11.

6.3.2 Constraints and contracts

The following constraints should be satisfied by the MTUT:

1. E_Co1. All `EPackage`, `EClass`, `EReference`, `EAttribute` and `EEnumLiteral` elements should generate an `Operation` with the same name.
2. E_Co2. For each `Class` a `Sort` in its corresponding `Module` should be generated with the same name.
3. E_Co3. Size constraints should be observed on some of the output element sets.

In this case, three Tracts are defined, as Listing 6 shows. Meanwhile, the same constraints are defined by seven MoTES contracts (shown in Listing 7). In this case, we require more MoTES contracts than Tracts to express the same constraints. Firstly, we should define five different MoTES contracts to express E_Tr1 constraints: one contract for every input pattern that may generate `Operation` elements in the output. Note that those contracts are defined in a reverse manner compared to E_Tr1, checking the input for a concrete output. Additionally, no MoTES contract is defined from E_Tr3, because it specifies a postcondition. Postconditions can be expressed by MoTES, but they are not relevant for metrics computation, just focused on invariants. Therefore, that Tract is not considered in this case study.

6.3.3 Mutations

Regarding the mutation analysis, the following mutations have been performed in the E2M ATL source code (see Table 7):

[7] https://www.eweb.unex.es/eweb/migraria/motes/e2m.html.

Table 11 Transformations (E2M)

Rule	Input element	Output element
E_R1	(entrypoint)	MaudeSpec, SModule, ModImportation, ModuleIdModExp, Sort and operation
E_R4	EPackage	Operation
E_R9	EClass	Sort, SubsortRel and operation
E_R10	EClassifier	Sort
E_R19	EClassifier	Equation, RecTerm and constant
E_R20	EReference	Operation
E_R29	EAttribute	Operation
E_R32	ENum	Sort, SubsortRel and operation
E_R38	EEnumLiteral	Operation

Listing 6 E2M Tracts

```
-- Tract 01 (E_Tr1)
trg_Operation.allInstances->forAll(p|
src_EPackage.allInstances->exists
    (o | p.name = o.name) or
src_EClass.allInstances->exists
    (o | p.name = o.name) or
src_EReference.allInstances->exists
    (o | p.name=o.name) or
src_EAttribute.allInstances->exists
    (o | p.name=o.name) or
src_EENum.allInstances->exists
    (o | p.name = o.name) or
src_EEnumLiteral.allInstances->exists
    (o | p.name=o.name))

-- Tract 02 (E_Tr2)
src_EClass.allInstances->forAll(c |
trg_Sort.allInstances->exists
    (s | c.name = s.name))

-- Tract 03 (E_Tr3)
trg_MaudeSpec.allInstances->size()=1 and
trg_SModule.allInstances->size()=2 and
trg_ModImportation.allInstances->size()=1 and
trg_ModuleIdModExp.allInstances->size()=1 and
trg_Sort.allInstances->size()>0
```

Listing 7 E2M MoTES contracts excerpt

```
contract E_Mo1_1_EPackage2Operation{
input { (EPackage:i) }
output { (Operation:o) }
detection: nameMatch(i.name = o.name)}

contract E_Mo1_2_EClass2Operation{
input { (EClass:i) }
output { (Operation:o) }
detection: nameMatch(i.name = o.name)}

...

contract E_Mo2{
input  { (EClass:i) }
output { (Sort:o) }
detection: nameMatch(i.name = o.name)}
```

1. *E_M1 (E_R9)*. Addition of an input pattern in E_R9, so that more output elements than expected could be generated (Cartesian product of all input patterns).
2. *E_M2 (E_R9)*. Modification of the value in a binding in E_R9, which might result in incorrectly initialised features (`name`) in the target model.
3. *E_M3(E_R10)*. Filter condition in input pattern removed in E_R10, supposedly it might mean to generate more output elements than expected.
4. *E_M4(E_R20)*. An output pattern element added to E_R20 with one binding, which might also mean an overgeneration of output elements.
5. *E_M5(E_R29)*. An output pattern element deletion in E_R29, likely resulting in the generation of fewer elements than expected in the target model.
6. *E_M6(E_R1)*. An output pattern element deleted from E_R1 with the same consequences as the previous mutation.
7. *E_M7(E_R38)*. Filter condition in input pattern added to E_R38, so the model transformation could not generate all the output elements expected.

6.3.4 MoTES results

Table 12 shows that all considered mutations were detected by our approach, and proper repair actions were suggested for each case. Unfortunately, some mutations cannot be considered because of operational issues (see 6.3.5 for a description). Note that, for the sake of consistency, we are keeping the original numbering for the mutations.

E_M2 This case is similar to U_M1, but now the mutation is just affecting one specific rule (`Class2Sort`, E_R9), which can be directly marked as the faulty one. Again, we get C_{FPFN} and FN(`Sort`) = FP_{out}(`Sort`): the involved rule may be generating the expected output elements but some of their detection constraints are not fulfilled, e.g. they are given a different name. In such cases, a thorough revision of the bindings of each transformation rule is recommended,

Table 12 Results and suggested adaptations (E2M)

Mut.	Output	Case	Cs.	Rule	Res.	Act.
E_M2	Operation	C_{TP}	E_Mo1.1	E_R4	TP	NoAct.
			E_Mo1.2	E_R19	TP	NoAct.
			E_Mo1.3	E_R20	TP	NoAct.
			E_Mo1.4	E_R29	TP	NoAct.
E_M2	Sort	C_{FPFN}	E_Mo2	E_R9	FN FP_{out}	**Constr.** Del.
E_M5	Operation	C_{TPFN}	E_Mo1.1	E_R4	TP	Relax **NoAct.**
			E_Mo1.2	E_R19	TP	Relax **NoAct.**
			E_Mo1.3	E_R20	TP	Relax **NoAct.**
			E_Mo1.4	E_R29	FN	Relax **Creat.**
E_M5	Sort	C_{TP}	E_Mo2	E_R9	TP	NoAct.
E_M7	Operation	C_{TPFN}	E_Mo1.1	E_R4	TP	Relax **NoAct.**
			E_Mo1.2	E_R19	TP	Relax **NoAct.**
			E_Mo1.3	E_R20	TP	Relax **NoAct.**
			E_Mo1.4	E_R29	TP	Relax **NoAct.**
			E_Mo1.5	E_R32	TP	Relax **NoAct.**
			E_Mo1.6	E_R38	FN	**Relax** Constr.
E_M7	Sort	C_{TP}	E_Mo2	E_R9	TP	NoAct.

which can be significantly reduced by identifying the failing predicate of a MoTES contract.

E_M5 C_{TPFN} is obtained in this mutation for the output element `Operation`, i.e. some input elements had not been generated, but they were expected. By means of the analysis of the different rules generating `Operation` elements, we find all the rules are generating TP, except for E_R29 `Attribute2Operation`, which is the source of false negatives, because there are `Attribute` elements in the input model not transformed into `Operation` elements. According to the suggested actions, we should select which action to take for both E_R29 and all rules generating TP. For the latter ones, we choose "No Action" because there is another rule causing FN: `Attribute2Operation` (E_R29). In this case, there is no issue related to the filtering condition of input elements, but the problem is in the output pattern of the rule. Both issues look the same from a black-box point of view since we are missing some expected output elements. Therefore, for E_R29 we choose to "Create a new output target" as suggested in Table 4.

E_M7 This mutation defined a new input filter that was not applicable for the input type of the rule. It raised a VM Exception in both EMFTVM and ATL2006 engines. Therefore, we need to redefine this mutant to make it executable without modifying its basic semantics (filtering some valid input candidates out), as shown in Listing 8.

Listing 8 E2M mutation 7 redefined

```
rule EnumLiteral2Operation { -- E_R38
from enumLit : Ecore ! EEnumLiteral
    ( enumLit.value = 0 )
```

Regarding MoTES results, C_{TPFN} is again obtained for the output element `Operation`. Through the analysis of different rules generating `Operation` elements, we find that all rules are generating TP except for rule `EnumLiteral2Operation` (E_R38), which is the origin of the false negatives. In contrast to E_M5 results, in this case, we can safely perform the default suggested action "Relax" for this faulty rule.

6.3.5 Comparison summary

In Table 13, we present the comparison results for the successful mutations, while for the mutations with operational issues, only a brief description of them is included. For this case study, we were not able to run all the originally defined mutations under our approach because different operational issues appeared in some of them (see additional material[8] for more information). In the following, firstly, the comparison of the mutation results are presented, and secondly, the mutations with operational issues are briefly described. Regarding the mutation results to compare to Tracts, they are:

1. *E_M2* MoTES and Tracts can identify E_R9 as the faulty rule straightforwardly.
2. *E_M5* In this case, Tracts is not able to find the guilty rule. According to the authors: *"This leads to a false negative (FN) and the impossibility to detect the guilty rule. This happens in this concrete case because* [E_]*R29 has very*

8 https://www.eweb.unex.es/eweb/migraria/motes/e2m.html.

Table 13 Comparison Summary (E2M)

Mut.	Description	MoTES results	Detection	Direct
E_M1	In pattern element addition in R9	ATL runtime exception		
E_M2	Value changed in E_R9 binding	C_{FPFN} for E_R9	Both	Both
E_M3	Condition deleted in R10 filter	Mutation will never change output		
E_M4	Out pattern element added in R20	Mutation not constrained by contracts		
E_M5	Out pattern element deletion in E_R29	C_{TPFN} for E_R29	MoTES	MoTES
E_M6	Out pattern element deletion in R1	Postcondition (not considered)		
E_M7	Filter addition in E_R38	C_{TPFN} for E_R38	Both	Both

few types, so removing some of them implies a significant loss of information." In contrast, MoTES can easily find the guilty rule E_R29.

3. *E_M7* Both approaches can straightforwardly identify E_R38 as the faulty rule.

Basically, both approaches can detect all the mutations (detection column), although small differences appear when considering how straightforwardly the guilty rule can be found (direct identification column). In conclusion, our approach appears at least as correct and complete as [7] for this second case study. Moreover, MoTES can provide those results while remaining a dynamic testing approach, so no additional static analysis of the model transformation under test is required. Furthermore, our approach is not just locating the faulty rules, but it also provides engineers with practical repair actions to guide them to their fixing.

Finally, a brief description of the commented operational issues is provided for each mutation involved:

1. *E_M1* In its original form, this mutation yields an ATL runtime exception, so no test is needed to detect it. Although we tried to define a valid modified version of this mutation, it was not possible because of the tight coupling between the input and output patterns, i.e. most output pattern bindings depend on concrete elements of the input pattern.
2. *E_M3* This mutation deletes a filter condition of the input pattern of an ATL lazy rule. Because that lazy rule is always invoked by just one rule which is already filtering its input pattern, an implicit filter is present on all its invocations. As a result, it behaves correctly according to contracts, so this mutation does not lead to any negative consequences in the output model.
3. *E_M4* After the mutation, for every `Reference` in the input model, an `Operation` is added to the output model as it should be, but in addition, a `Parameter` is added too. However, no contract is actually constraining the generation of `Parameter` output elements. Hence, no failure has to be detected by any of the tools.
4. *E_M6* This mutant is constrained by E_Co3 that is specifying a postcondition, so it is not relevant for our study, given our focus on invariants.

As aforementioned, mutations are useful to introduce synthetic changes to model transformations. However, sometimes those mutations make transformation rules invalid because they create syntax errors, e.g. E_M1, or they could not have any real or measurable impact on the transformation output, e.g. E_M3 and E_M4. Therefore, although those mutations can be useful for assessing static analysis approaches, they have no real consequences in the output model.

6.4 MDRE case study

This final case study[9] is framed inside the MIGRARIA project [30–32], which defines a model-driven re-engineering process over Legacy Web Applications (LWAs) aimed to obtain Rich Internet Applications (RIAs) and mobile clients (Fig. 9). In the reverse engineering stage inside this process, a static analysis on the software artefacts of the LWA is defined using MTs, in order to abstract its information and organise it according to the structure of our technology-independent metamodel: the MIGRARIA MVC metamodel. Contrary to the two previous case studies, in this one, we do not apply mutations to the MTUT. Nevertheless, the transformation, which is working properly in an application scenario, behaves wrongly for a new one. As a result, we can apply our approach to evolve such transformation to the new application scenario. For the sake of brevity, we just report herein the results and adaptations suggested.

In MDRE, one of the main goals is to increase the reusability of the model transformation chain that implements a static or dynamic analysis of the legacy system. For example, in the MIGRARIA project, it has been necessary to adapt or extend the transformation rules already defined so they contemplate new code patterns. This necessity arises since the code style guidelines used in the development of these LWAs may dif-

[9] https://www.eweb.unex.es/eweb/migraria/motes/mdre.html.

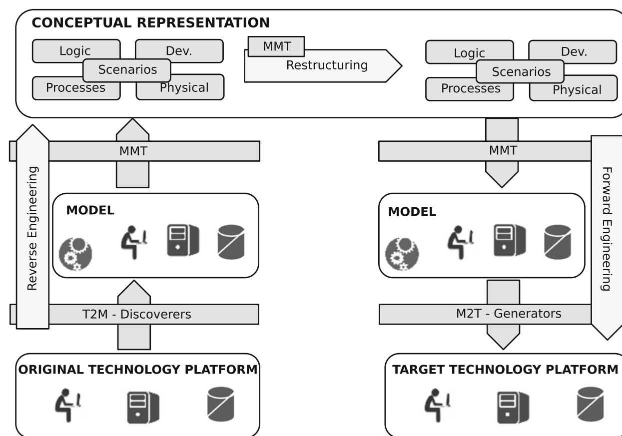

Fig. 9 MIGRARIA Process

fer from the ones considered so far, even if all of them use the same technological platform. The final goal is to reuse an existing reverse engineering (RE) process most optimally and effortlessly.

In this section, we present one of the case studies in MIGRARIA to validate our approach. In this case, the model transformation chain defined for a legacy system named CRS has to be adapted to the characteristics of another legacy system named AL-SIGM. That model chain has two steps. The first one is a text-to-model transformation to get a model representation of the software artefacts of a legacy system. This transformation is implemented by means of the MoDisco project [33]. And the second one is a model transformation that defines a static analysis of the software artefacts (models) and generates a MIGRARIA MVC model. This model transformation is implemented in ATL.

In this paper, we are just considering that model transformation (second step). MIGRARIA MDRE process should execute a model transformation for each of the components of the MIGRARIA MVC metamodel, i.e. Model, View and Controller components. In this work, for brevity, we are just going to focus on the generation of the View Component.

For the original legacy system (CRS), 19 ATL transformation rules had been defined to implement the static analysis. Table 14 lists them.

For the target legacy system (AL-SIGM), new contracts were defined to assess the feasibility of the original model transformation to this new system. For the sake of brevity, Table 15 describes some of them employing the fundamental terms of our approach.

After computing our metrics according to AL-SIGM contracts, we get the results summarised in Table 16. As shown, most of the result cases defined appear in this analysis. Our approach covers all different situations that may appear, being cases C_{TPFN} and C_{FP} the most common ones. This is a clear indicator that many input elements are not being transformed when they should be. One interesting case is the one for `Data_PA` output elements. It has got a perfect precision score and a recall of 91. As shown in Table 14, `Data_PA` elements are generated by six different transformation rules. In these circumstances, a score of 91 seems to be a high enough value to decide not to take any action to increase it. Actually, the threshold defined for the recall of this output element is 80. Values over that threshold mean that the effort to take (analysing and fixing at least six rules) is much greater than the potential benefits.

Table 17 only lists a small group of the adaptations suggested. Here the most interesting case is the `Request` output element, which contains three different contracts and involves four different transformation rules. As shown, different adaptation actions are suggested for different rules according to the kind of results they are generating (TP, FP, FN). Note that this is the final listing of actions performed, so we selected a concrete action (bold in the table) when two alternative ones were suggested. For the rules generating only TP (M_R15 and M_R16) it was suggested to either “Relax” or “No Action”. After the analysis of M_R15 and M_R16 we concluded that the action to take was “No Action”. A similar situation happened for the contract M_Mo4.2. M_R19 was only generating FP, and it was marked for deletion. That way, M_R14 was also marked for deletion because it was not generating anything. And finally, after the suggested adaptation actions were performed, we computed the metrics again, and there were still FN in the output model. So we applied the exceptional case “Create new rules” to solve this situation, as recommended by the approach. In this case, the tester might have taken an educated guess and selected the create action in the first execution. However, given that other actions applied in the same rule or different ones might impact the results, our approach always suggests applying only one repair action per rule by default. Therefore, in this case, the second round of execution would also show the tester that there are no more errors detected except the FN, simplifying the decision making.

In conclusion, after performing this case study, we believe the usefulness of our approach is assessed in real scenarios.

6.5 Evaluation discussion

In the following, we summarise the results obtained from the evaluation experiments, which were guided by the questions at the beginning of this section.

Table 14 Transformations (CRS)

Rule	Input pattern	Output pattern
M_R1	Page	Page
M_R2	BeanWriter	PresentationObject
M_R3	BeanWriter	Data_PA
M_R4	LogicIterate	ObjectPresentationCollection
M_R5	LogicIterate	ObjectPresentationCollection
		DataSet_PA
M_R6	Bean	Derived_PA
M_R7	HTMLForm	PresentationObject
M_R8	HTMLFormTag	Data_PA
M_R9	HTMLFormTag	DataSet_PA
M_R10	HTMLFormSelectOption	LiteralItem
M_R11	HTMLFormTag	LiteralPresentationCollection
M_R12	HTML FormSelectOption	ObjectPresentationCollection
		Data_PA
		Data_PA
M_R13	HTMLFormSelectOption	ObjectPresentationCollection
M_R14	HTML FormTag	Submit_PA
		Request
		ObjectRequestParameter
M_R15	HTMLLinkAction	Request
M_R16	HTMLLinkAction	Request
		ObjectRequestParameter
		Data_PA
M_R17	HTML	ValueRequestParameter
M_R18	HTML	ObjectRequestParameter
		Data_PA
M_R19	HTMLLinkPage	Request

6.5.1 RQ1: Applicability

Results suggest that MoTES is complete enough for the case studies shown in this paper (RQ1.1). MoTES have properly detected all errors introduced by mutations. Indeed, comparing MoTES to Tracts, MoTES was able to detect one case (E_M5) not detected by the static analysis reported in the original case study. Additionally, MoTES is supposed to perform properly in any other case study because, in a more general point of view, it can accurately deal with generic mutation operators. Moreover, in both case studies, MoTES provided relevant and appropriate repair actions for all the errors detected so that the faulty rules may be properly repaired (RQ1.2).

6.5.2 RQ2: Correctness

As aforementioned, MoTES may be seen as a two-step approach: (1) a contract-based model transformation testing step, and (2) a suggestion-based model transformation fixing step. Concerning the first step, MoTES can always detect misbehaving model transformations by contrasting their output with contracts definition. We have replicated two previous case studies performed over two ATL model transformations [7]: UML2ER and E2M. Eleven mutations (see Table 10 and Table 13) have been introduced into those transformations to test if MoTES can properly detect them. Moreover, MoTES' results have been compared to results obtained in the original case studies to assess the correctness and performance of the approach. As a result, in those case studies, errors detected by MoTES were always caused by faulty model transformations (RQ2.1). Additionally, our approach suggested the right set of adaptations or repair actions to guide to their fixing (RQ2.2), as presented.

6.5.3 Q3: Usefulness

In order to assess the usefulness of MoTES, we have applied it in the context of a model-driven migration project. The model transformation chain that the project uses for MDRE

Table 15 Contracts (AL-SIGM)

#	In pattern	Out pattern	Detection
M_Mo1	Page	Page	Same path property
M_Mo2	BeanWriter	PresentationObject (output)	Container pages with same path and name properties in both input and output matches
M_Mo3	HTMLFormSelectOption	LiteralPresentationCollection	Container pages with same path property and container elements with same name property
M_Mo4.1	HTMLLinkAction	Request	Container pages with same path property and value of composed attribute in input matches name of output
M_Mo4.2	HTMLLinkPage	Request	Container pages with same path property and value of composed attribute in input matches name of output
M_Mo4.3	HTMLFormTag	Request	Container pages with same path property and value of value property or composed attribute input matches name of output
M_Mo5	Bean	Derived_PA	Container pages with same path property and value of id property input matches name of output

Table 16 Results (AL-SIGM)

Output element	Pr	Rc	Case
Page	100	100	C_{TP}
PresentationObject (input)	100	100	C_{TP}
PresentationObject (output)	100	25	C_{TPFN}
ObjectPresentationCollection	NA	0	C_{FN}
LiteralPresentationCollection	NA	0	C_{FN}
ObjectRequestParameter	NA	0	C_{FN}
ValueRequestParameter	NA	NA	C_0
LiteralItem	NA	0	C_{FN}
Request	50	14	C_{TPFPFN}
Data_PA	100	91	C_{TPFN}
DataSet_PA	100	100	C_{TP}
Derived_PA	NA	NA	C_0
Submit_PA	NA	0	C_{FN}

required adaptation to a new application scenario: new legacy application models conforming to the same metamodel but featuring slightly different input patterns (see [14]). Once we adapted the contract definitions to the new scenario, MoTES provided us with a clear insight of what was working correctly and which model transformations were misbehaving. Additionally, suggested adaptations were helpful in order to guide the modifications introduced into the original model transformations. Furthermore, the information reported by MoTES was easily actionable to make decisions regarding the trade-offs of repairing model transformations, e.g. when repairing faulty transformations is more expensive than providing a post-transformation solution such as an ad-hoc filter for the output.

From a quantitative perspective, in this case, testers effort has been greatly reduced because: (1) MoTES was able to

Table 17 Selection of adaptations (AL-SIGM) from Table 16

Output	Case	Cs.	Rule	Res.	Act.
Page	C_{TP}	M_Mo1	M_R1	TP	NoAct.
PresentationObject (output)	C_{TPFN}	M_Mo2	M_R2	FN	Relax (Creat.)
LiteralPresentation Collection	C_{FN}	M_Mo3	M_R11	FN	Del.
Request	C_{TPFPFN}	M_Mo4.1	M_R15 M_R16	TP	Relax **NoAct.**
		M_Mo4.2	M_R19	FP	**Del.** Constr.
		M_Mo4.3	M_R14	FN	Del. (Creat.)
Derived_PA	C_0	M_Mo5	M_R6	–	NoAct.

indicate one faulty rule from 19 possibilities for each result case; and (2) MoTES could suggest one or two repair actions from 6 possibilities for each faulty rule. In real application scenarios, when size and complexity are compelling, contract specification overhead pays off.

6.6 Threats to validity

In this section, we elaborate on several factors that may jeopardise the validity of our results. As stated by [34], in applied research, the relationships under study (internal validity) and the generalisation (external validity) are of high priority.

6.6.1 Internal validity

Are there factors that might affect the results of this case study? First, MoTES contracts have to be manually defined. If they do not contain valuable constraints, the results provided by MoTES turn out useless. Nevertheless, given that we have created a DSL tailored for such definitions, according to our approach, the overall complexity is reduced. Moreover, MoTES can be complementary to other testing approaches, providing richer results for the same contracts. Second, concerning the experiments with faulty transformations (UML2ER and E2M cases), we relied on the state-of-the-art of mutation operators for model transformations. However, in the future, we may find that further operators are required to deal with more fine-grained mutations. As a result, these new operators might have an impact on the results obtained in our experiments. Finally, in [14] one of the authors was in charge of both contract definition and result interpretation which might have altered the perceived usefulness of the approach.

6.6.2 External validity

To what extent is it possible to generalise our findings? First, our approach is, to some extent, independent of the transformation language. This is because it was designed for rule-based transformation languages (e.g. ATL) and requires an engine producing transformation trace information in order to be able to suggest repair actions (see Sects. 2, 5.3 for additional details). Therefore it is possible to apply it to other model transformation languages fulfilling those requirements. Second, as a proof of concept, we have implemented it in the ATL language. MoTES contracts are specified in our DSL and later translated to ATL by means of Xtext templates [18]. Nevertheless, other technologies may have been used for this implementation because MoTES executes in its own isolated process without interacting with the existing transformation technology stack used. Regarding its usability, we have assessed it in an MDRE project developed by our own research group. Therefore, additional case studies might be needed to assess it in a broader scope and make our results more generally applicable.

7 Related work

According to [3], model testing presents three main issues: (1) test model generation, (2) test adequacy criteria definition, and (3) test oracle construction. In this work, we assume the existence of real input models for the model transformation under test. Our goal is then to adapt such a model transformation to the specific characteristics of these models, so the tests should just cover all the cases related to those characteristics. We deal with incorrect outputs, i.e. transformation logical errors according to the classification provided by [35]. In that sense, herein, we do not deal with the issue of the generation of artificial input test models nor the definition of test adequacy criteria. Nevertheless, our approach can be used in conjunction with existing approaches [6,16] for the generation of input test models.

7.1 Test oracles

Regarding the aforementioned three model testing issues, this work is focused on the construction of test oracles. According to the different categories of oracle generation approaches, our approach belongs to generic contract specification [36–38].

Among the existing contract-based model testing approaches, the works in [6,16,17,39] present clear similarities with our own. These works test model transformation (light) correctness, focusing on scalability and time-efficiency. They use the concept of contract definition, considering the model transformations as a black box. These approaches propose their own language to specify contracts and can automatically generate test models and test oracles from such specifications, following the approach pointed out by [3]. However, they are mainly focused on the specification of contracts and not on interpreting the test results to suggest concrete adaptation actions. Their test results are just reported as a list of passed/failed tests.

7.2 Faulty rule localisation

As aforementioned, one goal beyond detecting errors in transformation outputs by testing consists of the localisation of the particular transformation rule responsible for such error. For this goal, static and dynamic approaches have been proposed.

7.2.1 Static approaches

Leveraging those previous contract-based approaches, the work in [7] may locate the failing model transformation rule responsible for a contract violation by applying static analysis techniques to match contracts and rules. We have partially evaluated our approach by comparing it to this one (see Sect. 6). Moreover, in [40], ATL model transformations are translated to DSLTrans, and symbolic execution is applied to generate representations of all possible input models to the transformation. Another contract-based work is VeriATL [8,41,42], which proposes a formal (propositional logic) and static approach based on pre- and postconditions (contracts) for ATL transformations. The result is a set of possible failing scenarios, but testers must manually understand them and apply the proper corrections. Compared to MoTES, they just used postconditions to look for violations and cannot explicitly indicate the faulty rule.

There are also some static approaches not based on contracts. For example, the work in [43] also proposes a static analysis of ATL model transformations to discover typing and rule errors, which applies constraint solving to assert whether a source model triggering the execution of a given problem statement can exist.

All of these works define white-box approaches. Conversely, our work defines a black-box approach with a two-fold purpose: (1) to keep its independence from the model transformation platform used for development; and (2) to foster its reusability in different platforms and domains.

7.2.2 Dynamic approaches

More recently, several works [11,12] propose using Spectrum-Based Fault Localisation (SBFL), which is a popular technique used in software debugging for the localisation of bugs [44], to the problem of debugging model transformations. In [11], authors can determine which assertions are not satisfied and can rank the rules according to their probability of being the faulty rule. Another work [12] has recently proposed an improvement of SBFL by adjusting the spectrum information of test models covered rules according to the weights of different test models. Both works use contracts as test oracles and define dynamic techniques, as we do.

Among the dynamic approaches, some can be collected together because they leverage the information in transformation trace models to localise the faulty rule. One of the first approaches in this collection is presented in [45], which proposes a generic trace metamodel for model transformations, the different ways in which models may be generated from it (language-dependent), and the algorithm for error localisation. The paper [35], from a mostly theoretical standpoint, advocates for an offline, a-posteriori verification based on trace models and some sort of oracle capturing the expected output elements according to the input. The authors also adopt two debugging strategies used in Prolog and other declarative programming languages (analysis through queries and partial re-execution) and propose algorithms (in pseudocode) for computing a slice of the program in order to narrow down the scenario causing a bug.

A different approach is to apply metamorphic testing techniques to model transformation testing, whose effectiveness has been empirically proved by [46]. A metamorphic relation is a necessary property of the target program that relates two or more input data and their expected outputs, i.e. a contract. More recently [47] proposes an approach to automatically infer metamorphic relationships from the analysis of transformation trace models, which is specifically tailored at detecting faults in model transformations under three application scenarios, namely regression testing, incremental transformations and migrations among different transformation languages.

7.3 Repairing transformation rules

The repairing of transformation rules has been studied in different contexts: (1) in testing to repair faulty transformation rules; (2) in co-evolution to adapt the rules to changes on metamodels; and (3) in refactoring to improve their quality.

7.3.1 Testing

Although, as previously presented, many works dealing with the localisation of faulty transformation rules can be found,

there is still a significant scarcity of works for their assisted repairing. The work presented in [9,48] is one of the few exceptions to the rule. The authors propose a static analysis approach to detect and fix errors in ATL transformations automatically. For each error, a list of ranked quick fixes is presented, which in their most recent work is sorted based on speculatively applying them in the background and checking their effects and side effects. The paper presents a rich taxonomy of quick fixes (concrete repairs) collected from their own experience and previous works. Furthermore, their repair actions can syntactically fix a transformation, but they do not consider semantic issues (the intent of the transformation developer). To solve that issue, they plan to use contracts in future works. A similar technique can also be found in [49], applied to the Eclipse Java IDE for code refactoring.

7.3.2 Co-evolution

Regarding co-evolution of models and transformations, different approaches have been published. Some techniques also propose (semi)automatically repairing MT rules by analysing changes on the source or target metamodels, based on a predefined set of atomic changes and combinations of these. The authors of [50,51] define a set of atomic actions to adapt ETL rules [52] to changes in the metamodels and a default set of compositions of them which can be extended at will. Other approaches that deal with co-evolution of transformations once the metamodels change are [53] and [54], with a slightly different focus. The former deals with complex changes that border semantic analysis, while the latter prioritises the co-evolution of other artefacts like the models based on the changed metamodels. For the sake of completeness, it is also worth mentioning the works in [55,56], which consider similar scenarios but direct their efforts towards both the estimation of the effort that the adaptation (i.e. co-evolution) of the transformations may entail and the application of co-evolution to model-to-text transformations.

7.3.3 Refactoring

Model transformation refactoring consists of applying a predefined set of actions to improve the code quality of the transformations rules (syntax changes) without modifying its behaviour. The authors of [57] adapt the notion of object-oriented refactoring to model-to-model transformations and propose a catalogue of 24 refactor actions for ATL. Moreover, in [58], an interesting fully-automated search-based approach to refactor model transformations is presented for ATL, which uses a multi-objective algorithm to optimise a set of ATL-based quality metrics.

7.3.4 Comparison to MoTES

All the works aforementioned in this subsection propose some taxonomy or classification of concrete repair actions for model transformations, whose main advantage is their automatic application, and the main disadvantage is their dependency on the transformation language. In testing and co-evolution, most of the actions proposed may be considered a particular implementation of MoTES abstract repair actions, whose goal is aligned with MoTES, i.e. fixing rules with a faulty behaviour. Those works may be considered as the foundation for future research in automating repair actions application. Conversely, in refactoring, though its relevance and interest, the proposed repair actions (refactorings) are not directly applicable to our approach because they are specifically designed to deal with a completely different problem.

7.4 Synopsis

In summary, our approach is, to the best of our knowledge, the first work proposing a categorisation of contract-based test results according to the output pattern generated. Such categorisation provides engineers with a quick diagnosis of the kind of transformation error they are facing. Additionally, rather than just locating the failing rules, our approach can also suggest proper actions to repair them according to the category of errors detected.

8 Conclusions and future work

In this paper, MoTES, a novel approach to assist engineers in repairing model transformations, is presented. The main contributions of this work can be summarised into (1) the definition of a metric-based test oracle, (2) the generation of output-centred test results for enhanced interpretation and (3) the recommendation of specific repairing actions for concrete transformation rules based on testing results.

MoTES uses contracts to specify the expected behaviour of the model transformation under test. Given a test input model, the derived input–output model pair is processed to mark input–output pattern relationships as TP, TN, FP or FN. Those marks are then stored as a result model, queried to calculate precision and recall metrics for every output pattern (testing results). In this work, a classification for the relevant combinations of precision and recall is defined to simplify the interpretation of testing results. Moreover, MoTES allows classifying each transformation rule based on its output pattern classification, according to its testing results, and the traceability information of the MTUT. MoTES defines 37 cases for these classifications, so for each specific rule, a particular (abstract) action is suggested, such as constrain.

Three case studies are presented in great detail for evaluation purposes. In order to assess MoTES applicability and correctness, two previous mutation testing case studies dealing with fault localisation in model transformations [7] (UML2ER and E2M) are replicated. As the main results, our approach has shown to be able to detect the errors introduced in the transformations, localise the faulty rule and suggest the proper abstract repair actions. Regarding MoTES usefulness, a model transformation evolution case study is presented in the context of a reverse engineering project. From a quantitative perspective, MoTES was able to indicate one faulty rule from 19 possibilities for each result case and suggest one or two repair actions from 6 possibilities for each faulty rule. In conclusion, testers' effort can be significantly reduced by applying MoTES.

As future work, we envision the following lines of research.

Priority, dependency and consequences of adaptation actions. In real scenarios, several adaptation actions may be suggested; hence some priority criteria could be followed to indicate to engineers the best sequence of adaptations to follow. Such priority criteria have to be domain-dependent, but it may be partially generic, e.g. actions with a more significant impact in the output model may have a higher priority. Moreover, conflicting actions may be suggested for the same model transformation rule in some cases. Therefore, it could be necessary to properly specify different conflict relationships (dependencies) that might appear among adaptation actions. Furthermore, a concrete repairing action may generate unwanted consequences or side effects in the model transformation, e.g. introducing a new error. Although we are not providing concrete repairing actions, further analysis of those side effects might be insightful. Some works already propose interesting approaches to deal with repair consequences on models [51] and source code [49]. Consequences, dependency and priority should provide engineers with the proper sequence of actions to fix the model transformation.

Automatise adaptations application. We envision automation of the whole process of model transformation evolution once contracts are defined. Every adaptation action taken on the model transformation may imply a change in the sequence of adaptations suggested, e.g. a corrective action on a single model transformation may satisfy several contracts previously violated. Various potential sequences of adaptations define a search space; hence, search-based algorithms may be applied to find the best sequence. As the main trade-off, automatic repairing techniques of model transformations should be defined for the model transformation language used for development, as in [9]. Deriving concrete repairs from our abstract actions for a concrete transformation language remains still as future work.

Acknowledgements The authors wish to acknowledge the collaborative funding support from (i) Spanish Contract RTI2018-098652-B-I00, and (ii) Consejería de Economía e Infraestructuras, Junta de Extremadura (Spain)—European Regional Development Fund (ERDF)-GR18112 Project, and IB18034 project. This work has also been partially supported by Comunidad de Madrid as part of the Project 49/520608.9/18 (MADRIDFLIGHTONCHIP) co-funded by ERDF Funds of the European Union.

References

1. Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B.H.C., Collet, P., Benoit, C., Robert B.F., Rogardt H., James H., Jörg, K., Matthias, S.: The relevance of model-driven engineering thirty years from now. In: Model-Driven Engineering Languages and Systems, pp. 183–200 (2014)
2. Baudry, B., Dinh-trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: Proceedings of IMDT Work Conjunction with ECMDA'06 (2006)
3. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.-M.: Barriers to Systematic Model Transformation Testing. Commun. ACM **53**(6), 139–143 (2010)
4. Rahim, L.A., Whittle, J.: A survey of approaches for verifying model transformations. Softw. Syst. Model. **14**(2), 1003–1028 (2013)
5. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Formal Methods for Model-Driven Engineering, pp. 399–437. Springer, Berlin (2012)
6. Guerra, E., Soeken, M.: Specification-driven model transformation testing. Softw. Syst. Model. **14**(2), 623–644 (2015)
7. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Trans. Softw. Eng. **41**(5), 490–506 (2015)
8. Cheng, Z., Tisi, M.: Slicing ATL model transformations for scalable deductive verification and fault localization. Int. J. Softw. Tools Technol. Transfer **20**(6), 645–663 (2018)
9. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL transformations with speculative analysis. Softw. Syst. Model. **17**(3), 779–813 (2018)
10. Guerra, E., Cuadrado, J.S., de Lara, J.: Towards effective mutation testing for ATL. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 78–88. IEEE (2019)
11. Troya, J., Segura, S., Parejo, J.A., Ruiz-Cortés, A.: Spectrum-based fault localization in model transformations. ACM Trans. Softw. Eng. Methodol. **27**(3), 1–50 (2018)
12. Li, P., Jiang, M., Ding, Z.: Fault localization with weighted test model in model transformations. IEEE Access **8**, 14054–14064 (2020)
13. Kretschmer, R., Khelladi, D.E., Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: From Abstract to concrete repairs of model inconsistencies: an automated approach. In: Proceedings—Asia-Pacific Software Engineering Conference, APSEC, 2017-December, pp. 456–465 (2018)
14. Rodriguez-Echeverria, R., Macías, F.: A statistical analysis approach to assist model transformation evolution. In: 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 226–235, Ottawa, Canada (2015)
15. Rodriguez-Echeverria, R., Macías, F., Rutle, A.: On reducing model transformation testing overhead. In: 2nd Joint International Workshop on Patterns in Model Engineering and the 5th International Workshop on the Verification of Model Transformation,

PAME-VOLT 2016, CEUR Workshop Proceedings. vol. 1693, pp. 58–67 (2016)
16. Gogolla, M., Vallecillo, A.: Tractable model transformation testing. In: France, Robert B., Kuester, Jochen M., Bordbar, Behzad, Paige, Richard F. (eds.) Modelling Foundations and Applications. number 6698 in Lecture Notes in Computer Science, pp. 221–235. Springer, Berlin (2011)
17. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. **20**(1), 5–46 (2013)
18. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, London (2016)
19. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008)
20. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Hartmut, E., Reiko, H., Grzegorz, R., Gabriele T. (eds) Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7–13, 2008. Proceedings, volume 5214 of Lecture Notes in Computer Science, pp. 411–425. Springer (2008)
21. Anjorin, A.: An introduction to triple graph grammars as an implementation of the delta-lens framework. In: Jeremy, G., Perdita, S. (eds.) Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures, pp. 29–72, Springer, Cham (2018)
22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. In: Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2006)
23. Cerioli, M., Margaria, T., Wermelinger, M., de Lara, J., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., Roswitha, B.: Fundamental aspects of software engineering attributed graph transformation with node type inheritance. Theor. Comput. Sci. **376**(3), 139–163 (2007)
24. Troya, J., Bergmayr, A., Burgueño, L., Wimmer, M.: Towards systematic mutations for and with ATL model transformations. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10. IEEE (2015)
25. Guerra, E., Cuadrado, J.S., de Lara, J.: Towards effective mutation testing for ATL. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 78–88 (2019)
26. Troya, J., Burgueño, L., Wimmer, M., Vallecillo, A.: Mutations in ATL Transformations and their Identification with Matching Tables. Technical Report 2, University of Malaga (2014)
27. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
28. Mottu, J.-M.., Baudry, B., Le Traon, Y.: Mutation analysis testing for model transformations. In: Rensink, A., Warmer, J. (eds.) Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10–13, 2006. Proceedings, Lecture Notes in Computer Science, pp. 376–390. Springer, Berlin (2006)
29. Bergmayr, A., Troya, J., Wimmer, M.: From out-place transformation evolution to in-place model patching. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering—ASE '14, ASE '14, pp. 647–652, ACM Press, New York (2014)
30. Rodríguez-Echeverría, R., Conejero, J.M., Clemente, P.J., Preciado, J.C., Sánchez-Figueroa, F.: Modernization of legacy web applications into rich internet applications. In: Andreas, H., Nora, K. (eds.) Current Trends in Web Engineering, pp. 236–250, Springer, Berlin (2012)
31. Conejero, J.M., Rodríguez-Echeverría, R., Sánchez-Figueroa, F., Linaje, M., Preciado, J.C., Clemente, P.J.: Re-engineering legacy Web applications into RIAs by aligning modernization requirements, patterns and RIA features. J. Syst. Softw. **86**(12), 2981–2994 (2013)
32. Rodriguez-Echeverria, R., Pavón, V.M., Macías, F., Conejero, J.M., Clemente, P.J., Sánchez-Figueroa, F.: IFML-based Model-Driven Front-End Modernization. In: Strahonja, V., Vrček, N., Plantak Vukovac, D., Barry, C., Lang, M., Linger, H., Schneider, C. (eds.) 24th Information Systems Development: Transforming Organisations and Society through Information Systems (ISD2014), pp. 226–233, Varaždin, Croatia (2014)
33. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
34. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell (2000)
35. Hibberd, M., Lawley, M., Raymond, K.: Forensic debugging of model transformations. Model Driven Eng. Lang. Syst. **4735**(4735), 589–604 (2007)
36. Mottu, J.-M., Baudry, B., Le Traon, Y.: Model transformation testing: oracle issue. In: IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW '08, pp. 105–112 (2008)
37. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: Robert, B., France, J.K., Ruth, B., Colin, A. (eds.) Model Driven Engineering Languages and Systems, number 7590 in Lecture Notes in Computer Science, pp. 432–448. Springer, Berlin (2012)
38. Gehan, M.K., Selim, F.B., James, R.C., Juergen, D., Shige, W.: Automated verification of model transformations in the automotive industry. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) Model-Driven Engineering Languages and Systems. number 8107 in Lecture Notes in Computer Science, pp. 690–706. Springer, Berlin (2013)
39. García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: Eunit: a unit testing framework for model management tasks. In: Model Driven Engineering Languages and Systems, pp. 395–409. Springer, Berlin (2011)
40. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Full contract verification for ATL using symbolic execution. Software and Systems Modeling, pp. 1–35 (2016)
41. Cheng, Z., Tisi, M.: A deductive approach for fault localization in ATL model transformations. In: International Conference on Fundamental Approaches to Software Engineering, pp. 300–317. Springer, Berlin (2017)
42. Cheng, Z., Tisi, M.: Incremental deductive verification for relational model transformations. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 379–389. IEEE (2017)
43. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Trans. Softw. Eng. **43**(9), 868–897 (2017)
44. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009)
45. Aranega, V., Mottu, J.-M., Etien, A., Dekeyser, J.-L.: Traceability mechanism for error localization in model transformation. ICSOFT **1**, 66–73 (2009)
46. Jiang, M., Chen, T.Y., Kuo, F.C., Zhou, Z.Q., Ding, Z.: Testing model transformation programs using metamorphic testing. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE, 2014-January, pp. 94–99 (2014)
47. Troya, J., Segura, S., Ruiz-Cortés, A.: Automated inference of likely metamorphic relations for model transformations. J. Syst. Softw. **136**, 1339–1351 (2018)
48. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL model transformations. In: 2015 ACM/IEEE 18th International Con-

ference on Model Driven Engineering Languages and Systems (MODELS), pp. 146–155. IEEE, (2015)
49. Muşlu, K., Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: Speculative analysis of integrated development environment recommendations. ACM SIGPLAN Notices **47**(10), 669–682 (2012)
50. Khelladi, D.E., Kretschmer, R., Egyed, A.: Change propagation-based and composition-based co-evolution of transformations with evolving metamodels. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 404–414 (2018)
51. Khelladi, D.E., Kretschmer, R., Egyed, A.: Detecting and exploring side effects when repairing model inconsistencies. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, pp. 113–126 (2019)
52. Kolovos, D.S., Rose, L.M., Paige, R.F.: The Epsilon Book. Eclipse Foundation Inc., Ottawa (2011)
53. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In International Conference on Software Language Engineering, pp. 144–163. Springer, Berlin (2012)
54. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 116–125. IEEE (2015)
55. Ruscio, D.D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and atl transformations. In: International Conference on Theory and Practice of Model Transformations, pp. 60–75. Springer, Berlin (2013)
56. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: ME@ MoDELS, pp. 22–31 (2014)
57. Wimmer, M., Martínez, S., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. J. Object Technol. **11**(2), 2–1 (2012)
58. Alkhazi, B., Ruas, T., Kessentini, M., Wimmer, M., Grosky, W.I.: Automated refactoring of ATL model transformations. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems—MODELS ’16, pp. 295–304 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Roberto Rodriguez-Echeverria PhD, is a senior researcher at the Quercus Software Engineering Group and associate professor of Computer Languages and Systems at the University of Extremadura (Spain). His main research areas include software engineering, web engineering, model-driven engineering, data engineering, and multimedia engineering. He has published more than 50 scientific publications in journals and international conferences. He currently leads the research line on multimedia technologies and engineering.

Fernando Macías is a post-doctoral researcher in the areas of Software Engineering and Computer Science at the IMDEA Software Institute in Madrid. His work includes transferring research results to the aerospace industry to aid in the efficient development of critical embedded software. He holds a PhD in Informatics from the University of Oslo, Norway, earned after extensive research which included the development of open-source tools and the publication of a PhD dissertation in the areas of multilevel modelling, domain-specific languages and model transformation. Fernando also holds an MSc, Major and BSc from the University of Extremadura, Spain.

Adrian Rutle holds a PhD in Computer Science from the University of Bergen, Norway. Rutle is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences (HVL), Bergen. Rutle’s main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has expertise in the development of modelling frameworks and domain-specific modelling languages. He also conducts research in the fields of modelling and simulation for robotics, eHealth, digital fabrication, smart systems and machine learning.

José M. Conejero is an Associate Professor in the Department of Computer and Telematic Systems Engineering at the Universidad de Extremadura (Spain). He received his Ph.D. in Computer Science from Universidad de Extremadura in 2010. He is the author of more than 50 papers of journals and conference proceedings and has also participated in different journals and conferences as a member of the program committee. His research areas include Web Engineering, Big Data and Model Driven Development. He has also been involved in several competitives projects and many contracts with entities and companies.