Debugging Model Transformations: A Spectrum-Based Fault Localization Approach

(Double-Blind Review)

ABSTRACT

Model transformations play a cornerstone role in Model-Driven Engineering (MDE) as they provide the essential mechanisms for manipulating and transforming models. The correctness of software built using MDE techniques typically relies on the correctness of the operations executed using model transformations. However, it is challenging and error prone to debug them, and the situation gets more critical as the size and complexity of model transformations grow, where manual debugging is no longer possible.

Spectrum-Based Fault Localization (SBFL) uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component (e.g., statements) of being faulty. In this paper we present an approach to apply SBFL for locating the faulty rules in ATL model transformations. The approach resides in the Eclipse Modeling Framework and is supported by the corresponding toolkit. We also evaluate the accuracy of the approach and compare the effectiveness of different SBFL techniques at locating faults in model transformations. Our experiments conclude that the technique that behaves best is able to indicate the faulty rule in 81.5% of cases.

CCS CONCEPTS

•Software and its engineering → Domain specific languages; Software testing and debugging;

KEYWORDS

Model Transformation, Spectrum-based, Fault Localization, Debugging, Model-Driven Engineering

ACM Reference format:
(Double-Blind Review). 2017. Debugging Model Transformations: A Spectrum-Based Fault Localization Approach. In *Proceedings of ACM ISSTA conference, Santa Barbara, California USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

In Model-Driven Engineering (MDE), models are the central artifacts that describe complex systems from various viewpoints and at multiple levels of abstraction using appropriate modeling formalisms. Model transformations (MTs) are the cornerstone of MDE [15, 36], as they provide the essential mechanisms for manipulating and transforming models. They are an excellent compromise between strong theoretical foundations and applicability to real-world problems [36]. Most MT languages are composed of model transformation rules[1]. Each MT rule deals with the construction of part of the target model. They match input elements from the source model and generate output elements that compose the target model. The ATLas transformation language (ATL) [34, 40] has come to prominence in the MDE community. This success is due to ATL's flexibility, support of the main metamodeling standards, usability that relies on strong tool integration within the Eclipse world, and a supportive development community [38]. Due to the importance of ATL in both the academic and the industrial arenas, the testing of ATL transformations is of prime importance.

The correctness of software built using MDE techniques typically relies on the correctness of the operations executed using MTs. For this reason, it is critical in MDE to maintain and test them as it is done with source code in classical software engineering. However, checking whether the output of a MT is correct is a manual and error-prone task, known as the oracle problem. In order to alleviate the oracle problem, the formal specification of MTs has been proposed by the definition and use of *contracts* [5, 8, 11, 51], i.e., assertions that the execution of the MTs must satisfy. These assertions can be specified on the models resulting from the MTs, the models serving as input for the MTs, or both, and they can be tested in a black-box manner. These assertions are usually defined using the Object Constraint Language (OCL) [54].

However, even when using the assertions as oracle to test if MTs are correct, it is still challenging to debug them and locate what parts of the MTs are wrong. The situation gets more critical as the size and complexity of MTs grow, where manual debugging is no longer possible, so there is an increasing need to count on methods, mechanisms and tools for debugging them.

Some works propose the debugging of model transformations by bringing them to a different domain such as Maude [50], DSLTrans [38] or Colored Petri Nets [55], where some specific analysis can be applied. The problem with these approaches is that the user needs to be familiar with such domains. The only work that addresses the debugging of ATL model transformations based on contracts proposes a static approach to identify the *guilty* rule [8]. It statically extracts the types appearing in the contracts as well as those of the MT rules and decide which rules are more likely to contain a bug. Their *static* approach achieves good results on several case studies (except one). On the other hand, the effectiveness of *dynamic* approaches, where the information obtained from the execution of MTs can help identify the faulty rules, is an open question.

Spectrum-Based Fault Localization (SBFL) is a popular technique used in software debugging for the localization of bugs [1, 56]. It uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component (e.g., statements) of being faulty. A program spectrum details the execution information of a program from a certain perspective,

[1]Throughout the paper, we may also refer to *model transformation (MT) rules* as *transformation rules* or merely *rules*

such as branch or statement coverage [27]. SBFL entails identifying the part of the program whose activity correlates most with the detection of errors.

This paper proposes the first approach that applies spectrum-based fault localization in ATL model transformations. Being SBFL a dynamic approach, our approach takes advantage of the information recovered after a MT run, what may help improve the results over static approaches [8] and at the same time complement them. We follow the approaches in [5, 8, 11, 51] and use the previously described contracts (assertions) as oracle function to determine the correctness of MTs. Given a MT, a set of assertions and a set of source models, our approach indicates the violated assertions and uses the information of the MT coverage to rank the transformation rules according to their suspiciousness of containing a bug. Also, out of the many existing techniques proposed in the literature for computing the suspiciousness values [41, 56, 58], we select five of them, namely *Tarantula* [32], *Ochiai* [1], *Kulczynski2* [37], *Cohen* [37] and *Russel-Rao* [41], and compare their effectiveness in the context of MTs. Evaluation results revealed that the most effective technique positions the faulty rule first in the rank in 81.5% of cases. This contrasts with the not-so-good results obtained in [8] for the same case study. The conclusions from our experiments serve as a proof-of-concept of the effectiveness of our approach to debug model transformations, and is to be complemented with further studies with more case studies.

Like ATL, our approach is compliant with the Eclipse Modeling Framework and is completely automated and executable, dealing with Ecore metamodels and XMI model instances and tailored at iteratively debugging ATL model transformations, although it could be trivially extended to support other transformation languages based on rules.

The remainder of this paper is organized as follows. Section 2 presents the basics of our approach, namely it describes spectrum-based fault localization, model transformations and the ATL language, and it presents the model transformation serving as case study. Then, Section 3 details our approach for applying SBFL in MTs, followed by its evaluation in Section 4. Then, Section 5 presents and describes some works related to ours, and the paper finishes with the conclusions and the outlook to future work in Section 6.

2 BACKGROUND

In this section we present the basics to understand our approach. First, a description of model transformations and the ATL transformation language is given, followed by the introduction of the ATL MT that serves as case study. Then, we explain what spectrum-based fault localization is with a general introduction.

2.1 Model Transformations

Model transformations play a cornerstone role in MDE since they provide the essential mechanisms for manipulating and transforming models [6, 46]. They allow querying, synthesizing and transforming models into other models or into code, so they are essential for building systems in MDE. A model transformation is a program executed by a transformation engine that takes one or more input models and produces one or more output models, as illustrated by the model transformation pattern [15] in Figure 1[2]. Model transformations are developed on the metamodel level, so they are reusable for all valid model instances. Most MT languages are composed of model transformation rules, where each rule deals with the construction of part of the target model. They match input elements from the source model and generate output elements that compose the target model.

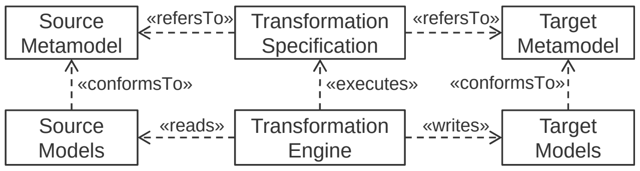


Figure 1: Model transformation pattern (from [15])

There is a plethora of frameworks and languages to define MTs, such as Henshin [3], AGG [47], Maude [13], AToM^3 [16], e-Motions [42], VIATRA [14], MOMoT [18, 19], QVT [25], Kermeta [31], JTL [12], and ATL [35]. In most of these frameworks and languages, model transformations are composed of transformation rules. Among them, we focus in this paper on the ATL language due to its importance in both the academic and the industrial arenas.

2.1.1 *ATLas Transformation Language.*

ATL has come to prominence in the model-driven engineering community due to its flexibility, support of the main meta-modeling standards, usability that relies on strong tool integration with the Eclipse world, and a supportive development community [34, 40].

ATL is a textual rule-based model transformation language that provides both declarative and imperative language concepts. It is thus considered a hybrid model transformation language. An ATL transformation is composed of a set of transformation rules and helpers. Each rule describes how certain output model elements should be generated from certain input model elements.

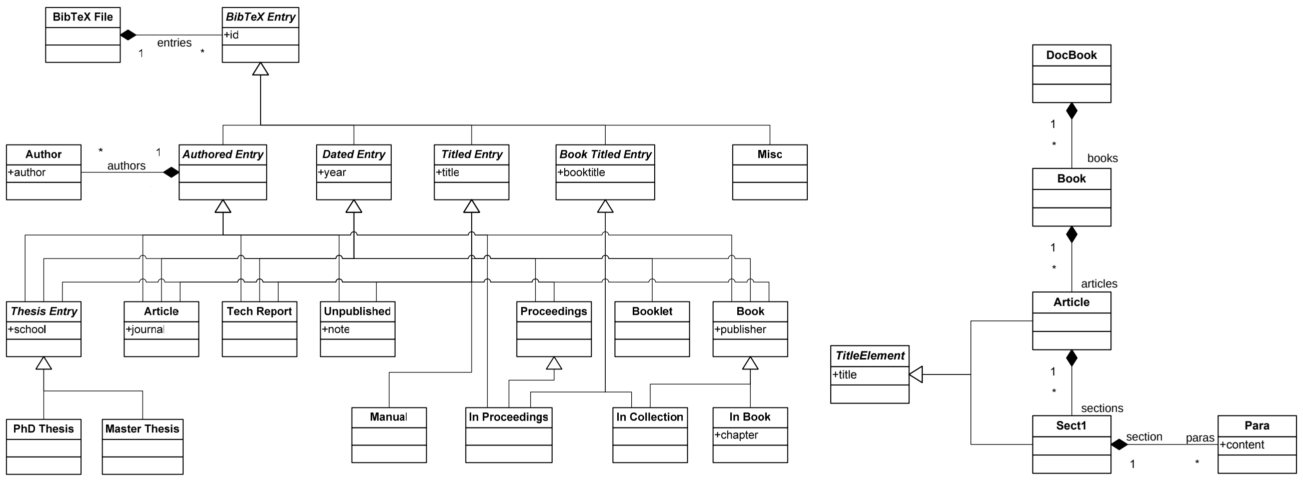
Rules are mainly composed of an *input pattern* and an *output pattern*. The input pattern is used to match *input pattern elements* that are relevant for the rule. The output pattern specifies how the *output pattern elements* are created from the input model elements matched by the input pattern. Each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

2.1.2 *Transformation Example.*

The *BibTeX2DocBook* model transformation [30], taken from the ATL Transformation Zoo [4], is used throughout this paper as case study. It transforms a BibTeXML model to a DocBook composed document. BibTeXML[3] is an XML-based format for the BibTeX bibliographic tool. DocBook [53] is an XML-based format for document composition.

The aim of this transformation is to generate, from a BibTeXML file, a DocBook document that presents the different entries of the bibliographic file within four different sections. The first and second sections provide the full list of bibliographic entries and the sorted list of the different authors referenced in the bibliography,

[2]In the paper, we use the terms *input/output* models/metamodels and *source/target* models/metamodels indistinctly.
[3]http://bibtexml.sourceforge.net/



(a) Input metamodel. (b) Output metamodel.

Figure 2: Metamodels of the BibTeX2DocBook transformation (from [30])

respectively, while the third and last section present the titles of the bibliography titled entries (in a sorted way) and the list of referenced journals (in article entries), respectively.

Listing 1: Excerpt of the *BibTeX2DocBook* MT.

```
module BibTeX2DocBook;
create OUT : DocBook from IN : BibTeX;

rule Main {                               -- tr1
  from
    bib : BibTeX!BibTeXFile
  to
    doc : DocBook!DocBook (books <- boo),
    boo : DocBook!Book (articles <- art),
    art : DocBook!Article (title <- 'BibTeXML_to_DocBook',
        sections_1 <- Sequence{se1, se2, se3, se4}),
    se1 : DocBook!Sect1 (title <- 'References_List',
        paras <- BibTeX!BibTeXEntry.allInstances()->sortedBy(e
             | e.id)),
    se2 : DocBook!Sect1 (title <- 'Authors_List',
        paras <- thisModule.authorSet),
    se3 : DocBook!Sect1 (title <- 'Titles_List',
        paras <- thisModule.titledEntrySet->collect(e |
             thisModule.resolveTemp(e, 'title_para'))),
    se4 : DocBook!Sect1 (title <- 'Journals_List',
        paras <- thisModule.articleSet->collect(e | thisModule.
             resolveTemp(e, 'journal_para')))
  }

rule Author {                          -- tr2
  from
    a : BibTeX!Author (...)
  to
    p1 : DocBook!Para (content <- a.author)
}
...
rule TitledEntry_Title_NoArticle {    -- tr4
  from
    e : BibTeX!TitledEntry (...)
  to
    entry_para : DocBook!Para (content <- e.buildEntryPara()),
    title_para : DocBook!Para (content <- e.title)
}
...
rule Article_Title_Journal {            -- tr6
  from
    e : BibTeX!Article (...)
  to
    entry_para : DocBook!Para (content <- e.buildEntryPara()),
    title_para : DocBook!Para (content <- e.title),
    journal_para : DocBook!Para (content <- e.journal)
} ...
```

The metamodels of this transformation are displayed in Figure 2. The BibTeXML metamodel (Fig. 2(a)) deals with the mandatory fields of each BibTeX entry (for instance, author, year, title and journal for an article entry). A bibliography is modeled by a *BibTeXFile* element. This element is composed of *entries* that are each associated with an *id*. All entries inherit, directly or indirectly, from the abstract *BibTeXEntry* element. The abstract classes *AuthoredEntry*, *DatedEntry*, *TitledEntry* and *BookTitledEntry*, as well as the *Misc* entry, directly inherit from *BibTeXEntry*. Concrete BibTeX entries inherit from some of these abstract classes according to their set of mandatory fields. There are 13 possible entry types: *PhDThesis*, *MasterThesis*, *Article*, *TechReport*, *Unpublished*, *Manual*, *InProceedings*, *Proceedings*, *Booklet*, *InCollection*, *Book*, *InBook* and *Misc*. An authored entry may have several authors.

The DocBook metamodel (Fig. 2(b)) represents a limited subset of the DocBook definition. Within this metamodel, a DocBook document is associated with a *DocBook* element. Such an element is composed of several *Books* that, in turn, are composed of several *Articles*. An *Article* is composed of sections (class named *Sect1*) that are ordered. A *Sect1* is composed of paragraphs (class *Para*) that are also ordered within each section. Both *Article* and *Sect1* inherit from the *TitledElement* abstract class.

An excerpt of the *BibTeX2DocBook* model transformation [30] is shown in Listing 1, which contains (part of) 4 out of 9 rules. The first rule, *Main*, creates the structure of a *DocBook* from a *BibTeXFile* and creates four sections with their corresponding titles. The paragraphs of each section are to be resolved when the remaining rules are executed. This rule uses the helpers *authorSet*, *titledEntrySet* and *articleSet*, not shown here due to space limitations.

The remaining rules in the excerpt are simplified due as well to space limitations. The simplification of the second rule, *Author*, creates a paragraph for each author having as content the author name. In this excerpt, the fourth rule, *TitledEntry_Title_NoArticle*, creates two paragraphs for each entry inheriting from *TitledEntry*, using for one of the paragraphs the helper *buildEntryPara*. Finally,

Table 1: An example showing the suspiciousness value computed using Tarantula (taken from [56])

Statement	Code	tc_1	tc_2	tc_3	N_{CF}	N_{CS}	Suspiciousness	Ranking
s_1	input(a)	●	●	●	1	2	0.5	3
s_2	i = 1;	●	●	●	1	2	0.5	3
s_3	sum = 0;	●	●	●	1	2	0.5	3
s_4	product = 1;	●	●	●	1	2	0.5	3
s_5	if (i < a) {	●	●	●	1	2	0.5	3
s_6	sum = sum + i;			●	1	0	1	1
s_7	product = product x i; // **BUG: 2i → i**			●	1	0	1	1
s_8	} else {	●	●		0	2	0	10
s_9	sum = sum + i;	●	●		0	2	0	10
s_{10}	product = product / i;	●	●		0	2	0	10
s_{11}	}	●	●		0	2	0	10
s_{12}	print(sum);	●	●	●	1	2	0.5	3
s_{13}	print(product);	●	●	●	1	2	0.5	3
Execution Results		Successful	Successful	Failed				

the sixth rule, *Article_Title_Journal*, creates three paragraphs for each article. As mentioned, please note that some rules and parts of the displayed rules have not been shown, so the behavior of the transformation is more complex than the one shown in the excerpt. We refer the interested reader to the document explaining the complete model transformation [30]. In the evaluation of our approach we use the complete transformation (cf. Section 4).

2.2 Spectrum-Based Fault Localization

Spectrum-Based Fault Localization (SBFL) uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component (e.g., statements) of being faulty. A program spectrum details the execution information of a program from a certain perspective, such as branch or statement coverage [27]. Table 1 depicts an example showing how the technique is applied to a sample program [56]. Horizontally, the table shows the code statements of the program, i.e., components. Note that a bug is seeded in statement s_7. Vertically, the table shows three test cases of the program. For each test case (i.e., tc_1, tc_2, and tc_3), a cell is marked with "•" if the program statement of the row has been exercised by the test case of the column, creating what is known as *coverage matrix* [2]. Additionally, the final row depicts the outcome of each test case, either "Successful" or "Failed", conforming the so-called *error vector* [2]. Based on this information, it is possible to identify which components were involved in a failure (and which ones were not), narrowing the search for the faulty component that made the execution fail.

Once a coverage matrix as the one shown in Table 1 is constructed, a number of techniques can be used to rank the program components according to their *suspiciousness*, that is, their probability of containing a fault. A popular fault localization technique is *Tarantula* [32], which for a program statement is computed as $(N_{CF}/N_F)/(N_{CF}/N_F + N_{CS}/N_S)$, where N_{CF} is the number of failing test cases that cover the statement, N_F is the total number of failing test cases, N_{CS} is the number of successful test cases that cover the statement, and N_S is the total number of successful test cases. The suspiciousness score of each statement is in the range [0,1], i.e., the higher the suspiciousness score of each component, the higher the probability of having a fault. The values of N_{CF}, N_{CS} and the *Tarantula* suspiciousness value for each statement are given in the sixth, seventh and eighth columns of Table 1. The last column indicates the position of the statement in the suspiciousness-based ranking where top-ranked statements are more likely to be faulty. In the example, the faulty statement s_7 is ranked first.

It is noteworthy that suspiciousness techniques may often provide the same value for different statements, being these tied for the same position in the ranking, e.g., statements s_6 and s_7 in Table 1. Under this scenario, different approaches are applicable such as measuring the effectiveness in the best and worst scenarios, using an additional technique to break the tie, or using some simple heuristics such as alphabetical ordering [56].

The effectiveness of suspiciousness metrics is usually measured using the EXAM score [58, 59], which is the percentage of statements in a program that has to be examined until the first faulty statement is reached, i.e.,

$$EXAM_{Score} = \frac{Number\ of\ statements\ examined}{Total\ number\ of\ statements}$$

For example, assuming that the statement s_7 is examined in second place (worst case), the EXAM score of *Tarantula* in the previous example would be $(2/13) = 0.153$, i.e., 15.3% of the statements must be examined to locate the bug.

3 SPECTRUM-BASED FAULT LOCALIZATION IN MODEL TRANSFORMATIONS

In this section we describe our SBFL approach for debugging model transformations. We first describe how the coverage matrix and the error vector are constructed. This is followed by an explanation of the suspiciousness calculation of the different transformation rules and the metric used for measuring the effectiveness of SBFL techniques. The section ends with the description of the methodology to apply our approach.

Table 2: Tarantula [32] suspiciousness values for the simplified *BibTeX2DocBook* MT when OCL_2 fails

T. Rule	tc_{02}	tc_{12}	tc_{22}	tc_{32}	tc_{42}	tc_{52}	tc_{62}	tc_{72}	tc_{82}	tc_{92}	N_{CF}	N_{UF}	N_{CS}	N_{US}	N_C	N_U	Susp	Rank
tr_1	●	●	●	●	●	●	●	●	●	●	9	0	1	0	10	0	0.5	3
tr_2 **(BUG)**	●		●	●	●	●	●	●	●	●	9	0	0	1	9	1	1	1
tr_3	●	●	●	●	●		●		●	●	7	2	1	0	8	2	0.44	7
tr_4	●	●	●	●	●	●	●	●	●	●	9	0	1	0	10	0	0.5	3
tr_5	●	●		●	●	●	●	●	●	●	8	1	1	0	9	1	0.47	6
tr_6	●	●	●	●	●	●	●	●	●	●	9	0	1	0	10	0	0.5	3
tr_7	●						●				2	7	0	1	2	8	1	1
tr_8	●	●			●		●	●	●		5	4	1	0	6	4	0.36	8
tr_9		●					●		●		2	7	1	0	3	7	$0.\overline{18}$	9
Test Result	F	S	F	F	F	F	F	F	F	F								

3.1 Constructing the Coverage Matrix and Error Vector

The construction of the coverage matrix requires information about the execution of the MT with a set of source models: $S = \{s_1, s_2, ..., s_n\}$. These models must conform to the source metamodel. The oracle that determines whether the result of a MT is correct or not is a set of OCL assertions: $O = \{ocl_1, ocl_2, ..., ocl_m\}$. These assertions are defined by specifying the expected properties of the output models of the transformation or properties that the <input, output> models pairs must satisfy. As an example, Listing 2 shows three OCL assertions for the model transformation described in Section 2.1.2. A *test case* is a pair composed of a source model and an OCL assertion: $tc_{ij} =< s_i, ocl_j >$. Therefore, the test suite is composed by the cartesian product of source models and OCL assertions: $T = S \times O = \{tc_{11}, tc_{12}, ..., tc_{nm}\}$. The test case tc_{ij} produces an error if the result of executing the MT with the source model s_i does not satisfy the OCL assertion ocl_j. It is worth noting that OCL assertions must hold for any source model. An OCL assertion is not satisfied for a MT when there is, at least, one test case where it is violated. This means, for instance, that for ocl_1 to be satisfied, it must be satisfied in $\{tc_{11}, tc_{21}, ..., tc_{n1}\}$.

We may recall that this paper focuses on debugging and not testing. Thus, we do not impose any constraint in how the source models are generated, either manually or automatically, neither in the type of OCL constraints used.

Table 2 depicts a sample coverage matrix constructed using our approach. Horizontally, the table shows the transformation rules in which we aim to locate bugs. In particular, we list the 9 transformation rules $< tr_1, tr_2, \dots, tr_9 >$ of the *BibTeX2DocBook* example [30], where a bug was seeded in tr_2. Vertically, the table shows 10 test cases using the constraint OCL_2 in Listing 2, $< tc_{02}, tc_{12}, \dots, tc_{92} >$. A cell is marked with "•" if the transformation rule of the row has been exercised by the test case of the column. The information about the rules triggered by a given test case can be collected by inspecting the trace model, e.g., using Jouault's *TraceAdder* [33]. The final row depicts the error vector with the outcome of each test case, either successful ("S") or failed ("F"). In the example, all test cases failed except tc_{12}, i.e., OCL_2 is violated.

Note that by grouping those test cases using the same OCL assertion we can simplify debugging by providing not only the most suspicious transformation rules, but also the specific assertion revealing the failure. In practice, this means that our approach generates a coverage matrix and an error vector for each violated OCL assertion.

Listing 2: OCL assertions for the *Class2Relational* MT.

```
--OCL1. The main Article must be properly created and named
TrgBook.allInstances()->forAll(b|b.articles->exists(a|a.title='
      BibTeXML_to_DocBook'))
--OCL2. For each author, there must be a paragraph with the
      name of the author within a section named 'Authors List'
SrcAuthor.allInstances()->forAll(a|TrgPara.allInstances()->
      exists(p|p.content=a.author and p.section.title='Authors_
      List'))
--OCL3. The titles of all publications must appear in the
      content of a paragraph of a section
SrcTitledEntry.allInstances()->forAll(te|TrgSect1.allInstances
      ()->exists(s|s.paras->exists(p|p.content=te.title)))
```

3.2 Calculating Suspiciousness

The following notation will be used throughout the paper to compute the suspiciousness of transformation rules from the information collected in the coverage matrix and the error vector. This notation is directly translated from the context of SBFL in software programs [56] by using transformation rules as the components (e.g., instead of statements), namely:

N_{CF}	number of failed test cases that cover a rule
N_{UF}	number of failed test cases that do not cover a rule
N_{CS}	number of successful test cases that cover a rule
N_{US}	number of successful test cases that do not cover a rule
N_C	total number of test cases that cover a rule
N_U	total number of test cases that do not cover a rule
N_S	total number of successful test cases
N_F	total number of failed test cases

Table 2 shows the values of N_{CF}, N_{UF}, N_{CS}, N_{US}, N_C and N_U for each transformation rule. The values of N_S and N_F are the same for all the rows, 9 and 1 respectively, and are omitted. Based on this information, the suspiciousness of each transformation rule using *Tarantula* is depicted in the column "Susp", followed by the position of each rule in the ranking. In the example, the faulty rule tr_2 is ranked first, tied with tr_7 (c.f. Section 4 for details about how ties are managed in our approach and evaluation). Assuming that the faulty rule was inspected in the second place (worst scenario), the EXAM score would be calculated as $2/9 = 0.222\%$, i.e., 22.2% of the transformation rules would have to be examined to locate the bug.

3.3 Methodology

In this section, we described the proposed methodology to debug ATL model transformations with our SBFL approach. It is graphically depicted in Figure 3.

(1) The inputs have to be provided, namely the *Model Transformation* under test as well as the sets of *Source Models* and *OCL Assertions*.
(2) The approach executes and indicates whether there is *any failure*, ending the process if there is none.
(3) If there is a failure, it indicates the *set of non-satisfied OCL assertions*, i.e., those that are violated for at least one test case. As explained in Section 3.1, it constructs a coverage matrix and an error vector for each non-satisfied assertion and returns *the rules of the MT ranked by suspiciousness* in each case.
(4) The user picks the ranking of one of the OCL assertions in order to *locate and fix the faulty rule* that made the assertion fail.
(5) Now, the user has a *Fixed Model Transformation* that has potentially less bugs than the original *Model Transformation.* The user can decide whether use it as input for the approach, together with the *Source Models* and *OCL Assertions*, or to keep repairing it according to the suspiciousness rankings obtained for the different non-satisfied OCL assertions.
(6) In the upcoming execution of the approach with the *Fixed Model Transformation*, less OCL assertions should be violated, and the user would repeat the process to keep fixing the bugs. This process is repeated iteratively until all bugs have been fixed.

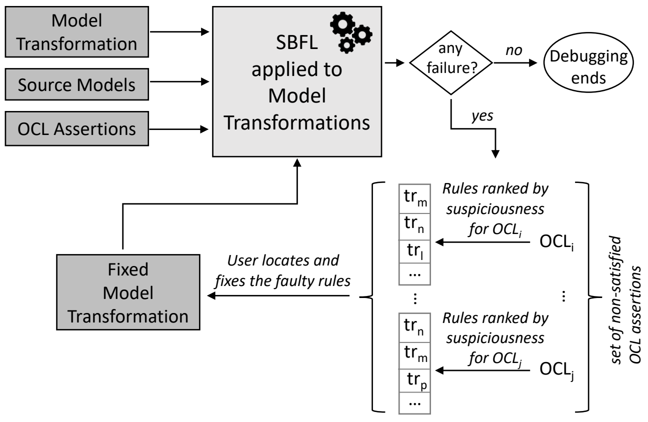

Model Transformation
Source Models
OCL Assertions
SBFL applied to Model Transformations
any failure?
no
Debugging ends
yes
Rules ranked by suspiciousness for OCL$_i$
OCL$_i$
Rules ranked by suspiciousness for OCL$_j$
OCL$_j$
tr$_m$
tr$_n$
tr$_l$
tr$_p$
set of non-satisfied OCL assertions
User locates and fixes the faulty rules
Fixed Model Transformation

Figure 3: Debugging of a MT applying our SBFL approach

4 EVALUATION

4.1 Research Questions

The research questions (RQs) that we want to answer in this work are the following:

- *RQ1.- Is our approach able to accurately locate faulty rules in model transformations?*
- *RQ2.- Which technique for suspiciousness computation, out of the five studied, is more effective?*

4.2 Experimental Setup

4.2.1 Case Study. As case study for the experiments we use the *BibTeX2DocBook* ATL MT described in Section 2.1. This case study presents bad precision results in a related work that proposes a static approach for the location of faulty rules [8], so we want to study if our approach produces more accurate results. An excerpt of the transformation is shown in Listing 1, and a complete description is available on [30]. The transformation is composed of nine rules and four helpers. In ATL, helpers are functions used in order to avoid the duplicity of code. This means that the same transformation with the same behavior can be written with and without helpers [38]. Since the treatment of helpers is left for future work, we have slightly modified the MT by including the content of the helpers directly in the rules.

4.2.2 Test Suite. We have developed a model generator that, given any metamodel, produces a user-defined number of model instances. The idea of our model generator is to produce a set of models with a certain variability degree. It creates an instance of the root class of the metamodel and, from such instance, it traverses the metamodel and randomly decides, for each containment relationship, how many instances to create for each contained class. This process is repeated iteratively until the whole metamodel is traversed. After all instances and containment relationships are set, non-containment relationships are created, respecting the multiplicities indicated in the metamodel. Also, attributes are given random values. Alternatively, it is possible to generate models with some predefined structure, by indicating the minimum and maximum number of entities to create. The values to be given to specific attributes can also be preset by the user.

For our evaluation, we have created 100 models conforming to the *BibTeX* metamodel (cf. Fig. 2(a)). In the generation of these models, when the generator finds a contained class in its metamodel traversal, it creates from 0 to 3 entities of such class. As for the attributes (journals, titles, authors...), most of them are randomly given significant names taken from the top-100 highly-cited papers in software engineering [22]. We have also constructed 26 OCL assertions that conform a formal specification of the model transformation and that are satisfied by the *BibTeX2DocBook* MT, three of which are shown in Listing 2.

This means, according to Section 3.1, that $|S| = 100$ and $|O| = 26$, so we have 2600 test cases: $|T| = |S| \times |O| = 2600$.

4.2.3 Mutants. In order to test the usability and effectiveness of our approach, we have produced mutants of the *BibTeX2DocBook* MT where bugs have been introduced. We have produced 26 mutants by using mutant operators as presented in [49], where each mutant is a variation of the original *BibTeX2DocBook.* Our set of

Table 3: Mutation operators used in 26 mutants

Mutant Operator (from [49])	**#Mutants**
In-pattern elm. Addition	2
In-pattern elm. class change	1
Filter deletion	1
Filter condition change	3
Out-pattern elem. addition	3
Out-pattern elem deletion	2
Out-pattern elm. class change	1
Out-pattern elm. name change	2
Binding addition	2
Binding deletion	7
Binding value change	7
Binding feature change	1

OCL assertions is complete enough as to kill all 26 mutants, i.e. all of them make at least one OCL assertion fail, what indicates there is an error in the MT. Table 3 presents the mutation operators that have been used for creating the mutants.

4.2.4 Set of Non-Satisfied OCL Assertions. As described, we have produced 26 mutants that correspond to buggy versions of the *BibTeX2DocBook* MT. Each one of them may violate one or more of the 26 OCL assertions. In total, the 26 mutants make 146 OCL assertions fail, so the results of our evaluation are extracted from the 146 rankings obtained, one for each violated assertion. These rankings are the results of suspiciousness values calculated with 146 coverage matrices of size 9 -transformation rules- $\times$ 100 -source models- and with the corresponding 146 error vectors.

4.2.5 Techniques for Suspiciousness Computation. We are interested in studying how different techniques for computing the suspiciousness of program components behave in the context of model transformations. Those dealt with in this paper are displayed in Table 4 (ignore for now the last row). *Tarantula* [32] is one of the best-known fault localization techniques. It follows the intuition that statements that are executed primarily by more failed test cases are highly likely to be faulty. Additionally, statements that are executed primarily by more successful test cases are less likely to be faulty. The *Ochiai* similarity coefficient is known from the biology domain and it has been proved to outperform several other coefficients used in fault localization and data clustering [1]. *Kulczynski2*, taken from the field of artificial intelligence, and *Cohen* have showed promising results in preliminary experiments [37, 58]. Finally, *Russel-Rao* has shown different results in previous experiments [41, 57, 58].

4.2.6 Evaluation Metric. In order to compare the effectiveness of the different SBFL techniques, we apply the EXAM score described in Section 2.2. In the context of this work, this score indicates the percentage of transformation rules that need to be examined until the faulty rule is reached. When there are ties in the suspiciousness values of several rules, we break them with the confidence-based strategy proposed in [59], where the formula for its computation is shown in the last row of Table 4. In case both the suspiciousness of several rules and their confidence are tied, we apply the approach

Table 4: Techniques applied for suspiciousness computation

Technique	Formula
Taantular [32]	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F}+\frac{N_{CS}}{N_S}}$
Ochiai [1]	$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF}+N_{CS})}}$
Cohen [37]	$\frac{2\times(N_{CF}\times N_{US}-N_{UF}\times N_{CS})}{(N_{CF}+N_{CS})\times(N_{US}+N_{CS})+(N_{CF}+N_{UF})\times(N_{UF}+N_{US})}$
Kulczynski2 [37]	$\frac{1}{2}\times(\frac{N_{CF}}{N_{CF}+N_{UF}}+\frac{N_{CF}}{N_{CF}+N_{CS}})$
Russel-Rao [41]	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}+N_{US}}$
Confidence [59]	$Max(\frac{N_{CF}}{N_F},\frac{N_{CS}}{N_S})$

proposed in [37], i.e., we average the best and worst cases and reflect this in the EXAM score.

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

4.3 Experimental Results

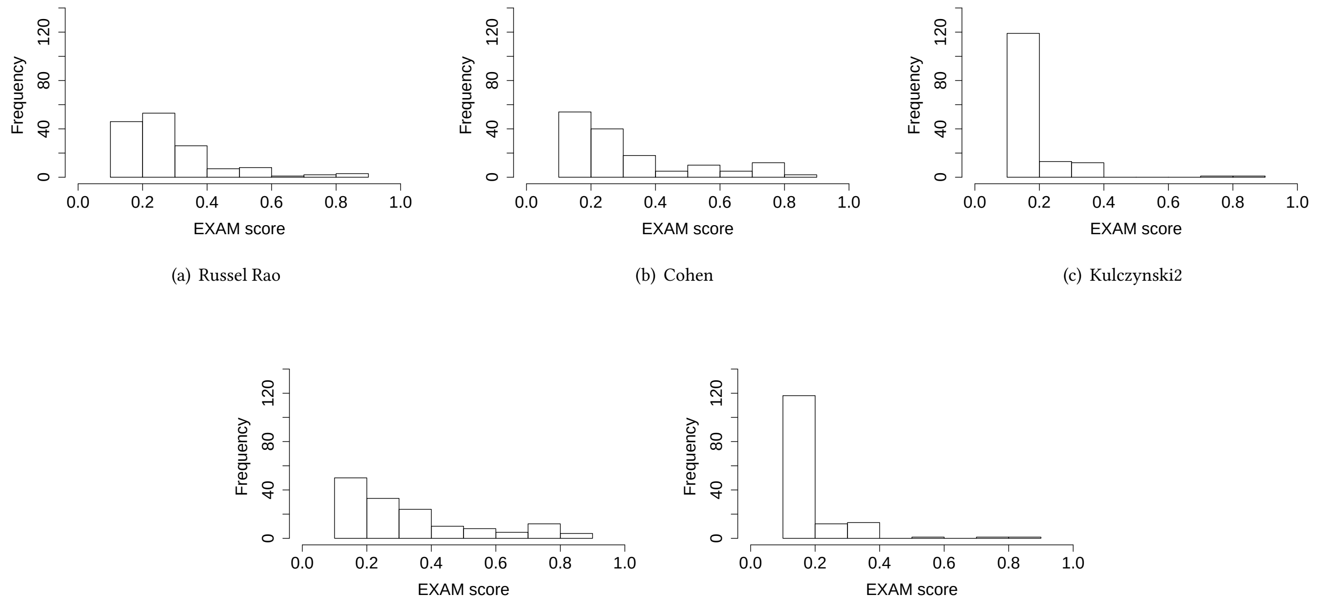
Table 5 shows the descriptive statistics of the EXAM score for each suspiciousness computation technique when applied to the 146 violated OCL assertions (cf.Section 4.2.4). Let us highlight that, since our MT under test consists of nine rules, the best possible value for the score is $\frac{1}{9} = 0.\overline{1}$ (the faulty rule is ranked first in the suspiciousness rank) while the worst value is $\frac{9}{9} = 1$ (the faulty rule is ranked the last). Having a look at the median values in the table, we can see that all the techniques found the faulty rule in their first or second guess for a majority of the test cases. Furthermore, *Kulczynski2* and *Ochiai* ranked the faulty rule first in 81.5% and 80.8% of cases, respectively, being additionally the techniques with the smallest standard deviation.

The differences among the results of each technique are graphically depicted in the histograms of Figure 4. The X axe indicates the EXAM score and the Y axe indicates the number of occurrences (out of the set of 146) where the EXAM score is obtained. It can be easily appreciated that *Kulczynski2* and *Ochiai* are the techniques that provide better results, i.e., a lower value of EXAM score. Having a look for instance at the chart for *Kulczynski2*, we can see that in 98.3% of the cases the exam metric is below 0.4, i.e., the faulty rule is ranked, as much, third.

Each run of our approach has taken between 9 and 11 seconds. This is the time taken to execute the MT with all 100 source models, print in the console the violated OCL assertions and compute and print in *cvs* files all the coverage matrices, error vectors and suspiciousness rankings for all five techniques together with the automatically computed EXAM score for each violated assertion.

4.4 Statistical Results

4.4.1 Null Hypothesis tests. Despite the data shown in Table 5 and Figure 4 indicate that *Kulczynski2* and *Ochiai* behave the best



(a) Russel Rao (b) Cohen (c) Kulczynski2

(d) Tarantula (e) Ochiai

Figure 4: Histograms of the values for the EXAM score obtained by each technique

Table 5: Descriptive statistics of the EXAM score

	Rus Rao	Cohen	Kul2	Tarantula	Ochiai
Min.	0.1111	0.1111	0.1111	0.1111	0.1111
1st Qu.	0.1111	0.1111	0.1111	0.1111	0.1111
Median	0.2222	0.2222	0.1111	0.2222	0.1111
Mean	0.2603	0.2953	0.1492	0.3151	0.1530
3rd Qu.	0.3333	0.3333	0.1111	0.4444	0.1111
Max.	0.8889	0.8889	0.8889	0.8889	0.8889
Std. Dev.	0.1673	0.2205	0.1047	0.2275	0.1108

in terms of the EXAM score, the MT mutants with which they have been tested have been randomly generated. For this reason, we need to study whether the differences observed are due to chance or not using statistical analysis.

The null hypothesis (H_0) states that there is not a statistically significant difference between the results obtained by the different suspiciousness computation techniques, while the alternative hypothesis (H_1) states that at least for one pair of techniques such difference is statistically significant. Statistical tests provide a probability (named *p-value*) ranging in [0, 1]. Researchers have established by convention that p-values under 0.05 represent so-called statistically significant values and are sufficient to reject the null hypothesis. In our study, since the results do not follow a normal distribution as can be appreciated in the histograms of Figure 4 (we also confirmed this assumption using Shapiro-Wilk normality tests), we used the Friedman test for the analysis [21]. The resulting

Table 6: Pairwise comparison of results

	Russel Rao	Cohen	Kulczynski2	Tarantula
Cohen	0.45	-	-	-
Kulczynski2	$< 2e^{-16}$	$< 2e^{-16}$	-	-
Tarantula	$< 2e^{-16}$	$< 2e^{-16}$	$< 2e^{-16}$	-
Ochiai	$< 2e^{-16}$	$< 2e^{-16}$	0.35	$< 2e^{-16}$

p-value was $< 2.2e^{-16}$, leading us to reject the H_0. In order to find the specific techniques with statistically significant differences, pairwise comparisons were performed using Conover's Test [28]. Additionally, we applied correction of the p-values using the Holm post-hoc procedure [29] as recommended in [17]. The corrected p-values of the pairwise comparisons of all techniques are shown in Table 6.

Since the p-value is smaller than $2.2e^{-16}$ in the majority of the comparisons, we can conclude that some techniques actually perform better than others. Specifically, we conclude that the differences are statistically significant for all the pairs except for <*Russel Rao*, *Cohen*> and <*Ochiai*, *Kulczynski2*>.

4.4.2 Effect size estimation. To further investigate the differences between the different suspiciousness computation techniques, we used Vargha and Delaney's $\widehat{A_{12}}$ statistic [52] to evaluate the effect size, i.e., determine which technique performs better and to what extent. Table 7 shows the effect size statistic for every pair of techniques. Each cell shows the $\widehat{A_{12}}$ value obtained when comparing the suspiciousness computation technique in the columns

Table 7: Effect size estimations

	Russel Rao	Cohen	Kulczynski2	Tarantula
Cohen	0.4907	-	-	-
Kulczynski2	**0.7509**	**0.7329**	-	-
Tarantula	0.4564	0.4713	**0.2467**	-
Ochiai	**0.7437**	**0.7270**	0.4954	**0.7474**

against the technique in the rows. Vargha and Delaney [52] suggested thresholds for interpreting the effect size: 0.5 means no difference at all; values over 0.5 indicates a small (0.5, 0.56), medium (0.57, 0.64), large (0.65, 0.71) or very large (0.72, 1) difference in favor of the technique in the row; values below 0.5 indicates a small (0.5, 0.44), medium (0.44, 0.36), large (0.36, 0.29) or very large (0.29, 0.0) difference in favor of the single technique in the column. Cells revealing very large differences in the table are highlighted in boldface.

We can see that *Ochiai* and *Kulczynski2* perform definitely better than any other technique. However, the difference between these two is significant, and the table simply suggests that *Kulczynski2* might be slightly better than *Ochiai*, but this is no guaranteed and should be further investigated in the future.

4.5 Discussion

The results of the experiments described in the previous sections allow us to answer the two research questions. Regarding the question "*RQ1: Is our approach able to accurately locate faulty rules in model transformations?*", we can answer it affirmatively. In fact, one of the two best techniques, *Kulczynski2*, ranks the guilty rule first in 81.5% of cases, what is a good result. As for the question "*RQ2: Which technique for suspiciousness computation, out of the five studied, is more effective?*", we can conclude that *Ochiai* and *Kulczynski2* provide better results than the other techniques, being the differences among them negligible.

We want to recall that the case study chosen to serve as proof-of-concept of our approach has given quite good results in comparison with the same case study in a related work with the same purpose as ours but proposing a static approach [8], where a precision of 25% is achieved.

4.6 Threats to Validity

4.6.1 Threats to Internal Validity. Are there factors that might affect the results of this evaluation? We consider the following internal threats to the validity of our evaluation based on the executed experiments. First, we have used as test suite 100 source models generated with our model generator and 26 OCL assertions. Should we have used different sets of source models and assertions, the results could have varied. We have tried to minimize this threat by constructing most OCL assertions as those proposed in [8] for the same case study. As for the model generator, we have developed it so that it can generate models with a certain degree of variability for any given metamodel. As second threat, we have used a set composed by 100 source models in the test suite. Having used larger and smaller sets of models could have resulted in likely better and worse results, respectively. Third, although our approach is able to locate several faults in MTs by treating each OCL assertion independently, in our experiments we have only *contaminated* one rule in each mutant. This means that even if a mutant makes more than one OCL assertions fail, all of them fail due to the same faulty rule (although we do have inserted several faults within the same rules). Furthermore, we have not considered all the mutation operators presented in [49] for constructing our mutants, such as *matched rule deletion*. Finally, we have gotten rid of the helpers in the MT by inserting them directly in the rules from which they are called. A more in-depth evaluation of our approach with a varied number of source models, the consideration of helpers, more mutants with more mutation operators and the injection of faults in several rules within the same mutant is left for future work.

4.6.2 Threats to External Validity. To what extent is it possible to generalize the findings? The first threat is that our experiments have only been realized with one case study, which externally threatens the generalizability of our results. To mitigate this threat, we have selected the case study that gives worse results in [8], having obtained much better results with our approach. Also, we have applied our approach for ATL due to its importance both in industria and academia, so it would be interesting to test it with other transformation languages. However, we do believe our approach would produce similar results for any model transformation language based in rules as long as the result of its executions can be stored in traces, what allow to construct the coverage matrix and error vector and, therefore, apply SBFL.

5 RELATED WORK

Due to the lack of oracles and formal semantics in transformation languages, some approaches propose to translate the transformation specifications to other domains where formal treatment is possible. For instance, Troya and Vallecillo propose to translate ATL to the rewriting logic framework Maude [50], where some formal analysis can be performed, although the translation is not fully automated. Oakes et al. propose to translate the declarative part of ATL to the visual graph-based model transformation engine DSLTrans [38]. Visual contracts similar to our OCL assertions but less expressive can be then tested for satisfaction in DSLTrans. Anastasakis et al. propose to translate QVT model transformations to Alloy in order to verify if some properties hold for the transformation, and there are also approaches for verifying contracts for ATL transformations based on the Coq proof assistant [10, 39]. The major difference of our approach with these is that we do not need to leave the ATL environment in order to check for the correctness of the MTs, so our approach stays within the Eclipse Modeling Framework dealing with Ecore metamodels and XMI models and the user does not need to be familiar with any other domain-specific language such as Maude, DSLTrans, Alloy or Coq. Furthermore, our approach helps locate the faulty rules, what is not proposed in these approaches.

Burgueño et al. [8] propose an approach with a similar purpose as ours, but their approach is static. They also count on ATL model transformations and OCL assertions that must be checked for correctness. Their approach proposes to locate the faulty rules based on matching functions that automatically establish alignments among the metamodels footprints appearing in the transformation rules and those present in the OCL assertions. They present three case studies where the approach behaves quite well. However, for the *BibTeX2DocBook* model transformation, subject of study in this paper, the results for the precision are not good when using OCL assertions similar to those used in our study, meaning the approach cannot manage to properly locate the faulty rules. This may be due to the way the calculation of the faulty rules is done, by matching the metamodel footprints in MT rules and OCL assertions, since the source metamodel of this MT presents a very elaborated inheritance structure, while in the MT rules only a few classes are used as input pattern elements in the rules. Furthermore, this approach does not check if an OCL assertion is satisfied, but it resorts to the Tracts tool [9]. Contrarily, our approach presents very good results for this case study with the *Ochiai* and *Kylczynski2* techniques and do not need any input from external tools. In any case, this work and ours can complement each other, and we leave as future work a thorough comparison of both approaches.

There are other approaches that propose static analysis for debugging model transformations. Sánchez-Cuadrado et al. [44] combine static analysis and constraint solving in order to discovery errors in ATL model transformations such as navigation errors (like invalid collection operations and operators), disconformities between the types used in the transformation and those declared in its source/-target metamodels, integrity constraints regarding the semantics of ATL, problems related to dependencies between transformation rules and, in summary, any error that the current syntactic checker of ATL is not able to identify. They even provide possible suitable quick fixes based on speculative analysis [43]. These approaches have meant an important milestone in the evolution of ATL. Our approach is orthogonal to these and, consequently, can serve to complement them.

As for model generators for producing test suites, some proposals for generating models have been proposed in the literature, where some of them require input by the tester. For instance, the model generator in [7] requires the tester to provide metamodel fragments as input, or the one in [45] requires input from the MMCC external tool [20] to provide model fragments. Other approaches propose the generation of models in different formats such as the Human Usable Textual Notation [23], so they need to be transformed prior to their use as input for model transformation languages integrated in the Eclipse Modeling Framework such as ATL. Some other more sophisticated model generators try to derive a set of input models from model transformations [24, 26], what is not desired in our case because we may be debugging erroneous transformations. Contrarily to these approaches, we needed for our approach an easy-to-use and light-weight model generator that, given a metamodel, it returns a set of models conforming to such metamodel, where the models present certain variability among them. This way, we assure that different models fire different rules in a MT, obtaining a varied spectra, key for SBFL.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented the first approach for the debugging of model transformations following a spectrum-based fault localization (SBFL) approach, and have implemented and evaluated it. Taking as input the model transformation under test and a set of source models and OCL assertions that serve as oracle function, our approach determines which assertions are not satisfied and, for each of them, in ranks the transformation rules according to their suspiciousness of being the faulty rule causing the failure. We have studied the effectiveness of five techniques proposed in the literature for the suspiciousness computation of program components (e.g., statements) in the context of model transformations, and have concluded that two of them, namely *Ochiai* and *Kulczynski2*, perform better than the rest. For instance, *Kulczynski2* has ranked the faulty rule first in a 81.5% of the cases. Our approach has been applied to model transformations written in ATL due to its importance in both industry and academia. Nevertheless, it can be trivially applied to any model transformation language based on transformation rules as long as it is able to store the result of the execution in traces. The program that implements our approach as well as all its inputs for the *BibTeX2DocBook* case study and all the generated results are available on [48].

There are several ways how we plan to improve and extend this work. First, in this paper we have treated the transformation rules as program components. For future work, we would like to also consider helpers as program components. This way, our approach would rank not only the rules according to their suspiciousness, but also the helpers. This is useful in those transformations that use helpers called from several different rules. Second, we would like to carry out experiments where bugs are introduced in more than one rule and in the helpers, and study how our approach behaves. Also, we would like to consider more techniques for the suspiciousness computation from those proposed in the literature [56] in order to discover if any of them behaves better than *Ochiai* and *Kulczynski2* in the context of model transformations. Integrating new techniques in our implementation is straightforward. It is also interesting to study how the variation in the size of the set of source models affect the quality of the results in the different techniques. Finally, we plan to consider more case studies from those in the ATL Zoo [40] and perform a more in-depth comparison with the approach presented in [8].

VERIFIABILITY

For the sake of verifiability, the program that implements our approach, together with the 26 mutants of the *BibTeX2DocBook* MT as well as the 100 source models, the file with the 26 OCL assertions and the *csv* files with all the computed results are available on [48].

ACKNOWLEDGMENT

Not included in this version as it might help reveal authors identity.

REFERENCES

[1] Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software 82*, 11 (2009), 1780 – 1792.

[2] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)* (2007), pp. 89–98.

[3] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. of MODELS* (2010), vol. 6394, pp. 121–135.

[4] ATL. ATL Zoo. http://www.eclipse.org/atl/atlTransformations, 2006.

[5] Baudry, B., Dinh-Trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., and Le Traon, Y. Model Transformation Testing Challenges. In

Proc. of the ECMDA workshop on Integration of Model Driven Development and Model Driven Testing (2006).

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

[7] Brottier, E., Fleurey, F., Steel, J., Baudry, B., and Traon, Y. L. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proc. of ISSRE 2006* (2006), pp. 85–94.

[8] Burgueño, L., Troya, J., Wimmer, M., and Vallecillo, A. Static Fault Localization in Model Transformations. *IEEE Tansactions on Software Engineering 41*, 5 (May 2015), 490–506.

[9] Burgueño, L., Wimmer, M., Troya, J., and Vallecillo, A. Tractstool: Testing MTs based on contracts. In *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition (MODELS 2013* (Oct. 2013), S. Ghosh, B. Baudry, E. Jackson, D. D. Ruscio, Y. Liu, and S. Zschaler, Eds., CEUR. http://ceur-ws.org/Vol-1115/poster5.pdf.

[10] Calegari, D., Luna, C., Szasz, N., and Tasistro, A. A Type-Theoretic Framework for Certified Model Transformations. In *Proc. of SBMF* (2010), pp. 112–127.

[11] Cariou, E., Marvie, R., Seinturier, L., and Duchien, L. OCL for the Specification of Model Transformation Contracts. In *Proc. of the OCL and Model Driven Engineering Workshop* (2004).

[12] Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering*, vol. 6563 of *LNCS*. Springer, 2011, pp. 183–202.

[13] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. *All About Maude – A High-Performance Logical Framework*, vol. 4350 of *LNCS*. Springer, 2007.

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

[15] Czarnecki, K., and Helsen, S. Feature-based survey of model transformation approaches. *IBM Systems Journal 45*, 3 (2006), 621–646.

[16] de Lara, J., and Vangheluwe, H. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proc. of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, vol. 2306 of *LNCS*. Springer, 2002, pp. 174–188.

[17] Derrac, J., Garca, S., Molina, D., and Herrera, F. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation 1*, 1 (2011), 3 – 18.

[18] Fleck, M., Troya, J., and Wimmer, M. Marrying Search-based Optimization and Model Transformation Technology. In *Proc. of NasBASE* (2015), pp. 1–16.

[19] Fleck, M., Troya, J., and Wimmer, M. Search-Based Model Transformations. *Journal of Software: Evolution and Process 28*, 12 (2016), 1081–1117.

[20] Fleurey, F., Baudry, B., Muller, P., and Traon, Y. L. Qualifying input test data for model transformations. *Software and System Modeling 8*, 2 (2009), 185–203.

[21] Friedman, M. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics 11*, 1 (1940), 86–92.

[22] Garousi, V., and Fernandes, J. M. Highly-cited papers in software engineering: The top-100. *Information and Software Technology 71* (2016), 108 – 128.

[23] Giner, P., and Pelechano, V. Test-driven development of model transformations. In *Proc. of MODELS'09*, vol. 5795 of *LNCS*. Springer, 2009, pp. 748–752.

[24] González, C. A., and Cabot, J. *ATLTest: A White-Box Test Generation Approach for ATL Transformations.* Springer, 2012, pp. 449–464.

[25] Greenyer, J., and Kindler, E. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and System Modeling 9*, 1 (2010), 21–46.

[26] Guerra, E., and Soeken, M. Specification-driven model transformation testing. *Software & Systems Modeling 14*, 2 (2015), 623–644.

[27] Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability 10*, 3 (2000), 171–194.

[28] Hollander, M., Wolfe, D. A., and Chicken, E. *Nonparametric statistical methods.* John Wiley & Sons, 2013.

[29] Holm, S. A simple sequentially rejective multiple test procedure. *Scand. J. Statist. 6*, 2 (1979), 65–70.

[30] INRIA. ATL Transformation Example: BibTeXML to DocBook, 2005. https://www.eclipse.org/atl/atlTransformations/BibTeXML2DocBook/ExampleBibTeXML2DocBook[v00.01].pdf.

[31] Jézéquel, J.-M., Barais, O., and Fleurey, F. Model Driven Language Engineering with Kermeta. In *Generative and Transformational Techniques in Software Engineering III*, vol. 6491 of *LNCS*. Springer, 2011, pp. 201–221.

[32] Jones, J. A., and Harrold, M. J. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2005), ASE '05, ACM, pp. 273–282.

[33] Jouault, F. Loosely Coupled Traceability for ATL. In *Workshop Proc. of ECMDA* (2005).

[34] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. ATL: A Model Transformation Tool. *Sci. Comput. Program. 72*, 1-2 (2008), 31–39.

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

[36] Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., and Wimmer, M. Model Transformation Intents and Their Properties. *Software and System Modeling* (2014), 1–35.

[37] Naish, L., Lee, H. J., and Ramamohanarao, K. A Model for Spectra-based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol. 20*, 3 (Aug. 2011), 11:1–11:32.

[38] Oakes, B. J., Troya, J., Lúcio, L., and Wimmer, M. Full Contract Verification for ATL using Symbolic Execution. *Journal on Software & System Modeling* (2016), 1–35.

[39] Poernomo, I., and Terrell, J. Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In *Proc. of ICFEM* (2010), pp. 56–73.

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

[41] Qi, Y., Mao, X., Lei, Y., and Wang, C. Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. In *Proc. of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, ACM, pp. 191–201.

[42] Rivera, J. E., Duran, F., and Vallecillo, A. A Graphical Approach for Modeling Time-dependent Behavior of DSLs. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09)* (2009), IEEE, pp. 51–55.

[43] Sánchez-Cuadrado, J., Guerra, E., and de Lara, J. Quick fixing ATL transformations with speculative analysis. *Software & Systems Modeling* (2016), 1–35.

[44] Sánchez-Cuadrado, J., Guerra, E., and de Lara, J. Static analysis of model transformations. *IEEE Transactions on Software Engineering PP*, 99 (2016), 1–32.

[45] Sen, S., Baudry, B., and Mottu, J.-M. Automatic Model Generation Strategies for Model Transformation Testing. In *Proc. of ICMT'09* (2009), vol. 5563 of *LNCS*, Springer, pp. 148–164.

[46] Sendall, S., and Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software 20*, 5 (2003), 42–45.

[47] Taentzer, G. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proc. of the 2nd Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, vol. 3062 of *LNCS*. Springer, 2003, pp. 446–453.

[48] The_Authors. Downloadable compressed file with our executalbe program and artifacts generated, 2017. https://WeNeedToAddIt.

[49] Troya, J., Bergmayr, A., Burgueno, L., and Wimmer, M. Towards systematic mutations for and with atl model transformations. In *Proc. of the IEEE 8th Int. Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (April 2015), pp. 1–10.

[50] Troya, J., and Vallecillo, A. A Rewriting Logic Semantics for ATL. *Journal of Object Technology 10* (2011), 5:1–29.

[51] Vallecillo, A., and Gogolla, M. Typing model transformations using tracts. In *Proc. of 5th Int. Conf. on Theory and Practice of Model Transformations (ICMT 2012)* (2012), Springer, pp. 56–71.

[52] Vargha, A., and Delaney, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics 25*, 2 (2000), 101–132.

[53] Walsh, L. M. N. *DocBook: The Definitive Guide.* O'Reilly & Associates, 1999.

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

[55] Wimmer, M., Kappel, G., Schönböck, J., Kusel, A., Retschitzegger, W., and Schwinger, W. A Petri Net based debugging environment for QVT Relations. In *Proc. of ASE'09* (2009), IEEE, pp. 3–14.

[56] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering 42*, 8 (2016), 707–740.

[57] Xie, X. *On the Analysis of Spectrum-based Fault Localization.* PhD thesis, Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia, 2012.

[58] Xie, X., Chen, T. Y., Kuo, F.-C., and Xu, B. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Trans. Softw. Eng. Methodol. 22*, 4 (Oct. 2013), 31:1–31:40.

[59] Yu, Y., Jones, J. A., and Harrold, M. J. An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proc. of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 201–210.