Model Transformation Testing and Debugging: A Survey

JAVIER TROYA, ITIS Software, Universidad de Málaga, Malaga, Spain
SERGIO SEGURA, SCORE Lab, I3US Institute, Universidad de Sevilla, Seville, Spain
LOLA BURGUEÑO, IN3, Open University of Catalonia, Barcelona, Spain
MANUEL WIMMER, CDL-MINT, Johannes Kepler University, Linz, Austria

Model transformations are the key technique in Model-Driven Engineering (MDE) to manipulate and construct models. As a consequence, the correctness of software systems built with MDE approaches relies mainly on the correctness of model transformations, and thus, detecting and locating bugs in model transformations have been popular research topics in recent years. This surge of work has led to a vast literature on model transformation testing and debugging, which makes it challenging to gain a comprehensive view of the current state of the art. This is an obstacle for newcomers to this topic and MDE practitioners to apply these approaches. This paper presents a survey on testing and debugging model transformations based on the analysis of 140 papers on the topics. We explore the trends, advances, and evolution over the years, bringing together previously disparate streams of work and providing a comprehensive view of these thriving areas. In addition, we present a conceptual framework to understand and categorise the different proposals. Finally, we identify several open research challenges and propose specific action points for the model transformation community.

CCS Concepts: • **Software and its engineering → Model-driven software engineering**; **Domain specific languages**; **Software testing and debugging**.

Additional Key Words and Phrases: Model Transformation, Testing, Debugging, Survey

ACM Reference Format:
Javier Troya, Sergio Segura, Lola Burgueño, and Manuel Wimmer. 2022. Model Transformation Testing and Debugging: A Survey. *ACM Comput. Surv.* 37, 4, Article 111 (August 2022), 39 pages. https://doi.org/10.1145/3523056

1 INTRODUCTION

In Model-Driven Engineering (MDE) [174], models are the central artifacts to represent complex systems from various viewpoints and at multiple levels of abstraction using appropriate modeling formalisms. Model transformations (MTs) are the cornerstone of MDE [189, 207, 225], as they provide the essential mechanisms for manipulating and constructing models. They are considered

We would like to thank Prof. Antonio Vallecillo for his help in an earlier version of this manuscript. This work is partially supported by the European Commission (FEDER) and Junta de Andalucia under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación) under projects HORATIO (RTI2018-101204-B-C21), COSCA (PGC2018-094905-B-I00) and LOCOSS (PID2020-114615RB-I00), by the Austrian Science Fund (P 28519-N31, P 30525-N31), and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).
Authors' addresses: Javier Troya, ITIS Software, Universidad de Málaga, Malaga, Spain, jtroya@uma.es; Sergio Segura, SCORE Lab, I3US Institute, Universidad de Sevilla, Seville, Spain, sergiosegura@us.es; Lola Burgueño, IN3, Open University of Catalonia, Barcelona, Spain, lburguenoc@uoc.edu; Manuel Wimmer, CDL-MINT, Johannes Kepler University, Linz, Austria, manuel.wimmer@jku.at.

0360-0300/2022/8-ART111 $15.00
https://doi.org/10.1145/3523056

an excellent compromise between strong theoretical foundations and applicability to real-world problems [207, 225]. Existing MT languages often provide dedicated language concepts to realize MT rules. Each rule matches input elements from the source model and deals with either the construction of part of the new target model—in the case of out-place MTs—or the evolution, i.e., modification, of the source model—in the case of in-place MTs [189].

The correctness of software systems built with MDE approaches largely depends on the correctness of the operations executed using MTs. Therefore, it is critical in MDE to test and debug MTs as it is done with source code in classical software engineering, but it has to be emphasized that MTs also come with their own challenges in this respect [167]. MT testing aims to reveal failures by executing the MT under test with a set of input models and checking whether it produces the expected output; if it does not, then a bug has been detected. Once one or more unexpected outputs are observed (i.e., bugs), MT debugging focuses on isolating the bug causing it and fixing it.

Selim et al. [244] reviewed the state of the art in MT testing in 2012. They organized the papers according to the phase of the testing process they belong to. They considered 29 works in their study and concluded that more research is needed.

This article presents a comprehensive survey on testing and debugging MTs, providing a unified view and a classification of the vast literature on the topics. Testing and debugging are closely related activities as explained above, and thus we decided to cover both to make our survey more thorough and helpful for readers—a similar approach is followed in other surveys, such as on compiler testing [183]. Overall, the survey is based on the analysis of 140 papers published between 2004 and 2020. This represents a largely updated survey with respect to the survey by Selim et al. [244] from 2012, since our study includes more than 100 additional papers. As a part of our survey, we first propose a conceptual framework for classifying current and future contributions on MT testing and debugging. Then, we report the trends, advances and evolution of MT testing and debugging over the years, and some of the open research challenges and specific action points to be addressed in the future. This article also aims to serve as a reference point for future contributions, and thus, special emphasis is put on how the approaches are evaluated, pointing readers to the most popular case studies and tools.

The remainder of this paper is structured as follows. Section 2 briefly describes some concepts related to MDE, presents an MT excerpt serving as a running example throughout the paper and discusses previous surveys related to MT testing and debugging. Then, Section 3 presents our conceptual framework for MT testing and debugging, while Section 4 formulates our research questions and describes the review methodology followed in our survey. The state of the art in MT testing and debugging is described in Sections 5, 6 and 7, and the research challenges identified are described in Section 8. Finally, Section 9 concludes the paper and Appendix A presents some interesting statistics about the selected papers.

2 BACKGROUND

Model-Driven Engineering (MDE) [174, 190] advocates models as first-class entities throughout the system life-cycle. It is meant to increase productivity by maximizing automation and interoperability, simplifying the design process and promoting communication between stakeholders. The use of MDE principles and techniques is growing, being well established, for instance, in the development of embedded and production systems. This section introduces MDE's main building blocks, namely (meta)models and MTs. It also presents a running example used throughout the paper and describes the existing surveys related to MT testing and debugging.

2.1 Models and Metamodels

A *model* is an abstraction of a system used to replace the system under study for a particular purpose [222, 226]. This abstraction process allows to better manage, understand, study, and analyze

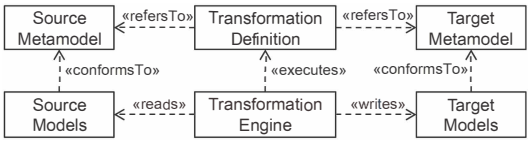

Fig. 1. Model transformation pattern (taken from [189])

models in contrast to the full system under study. Of course, this also helps for communication and discussion means. Thus, models are frequently used to share and communicate a common vision among technical and non-technical stakeholders [190].

In MDE, it is common that a model must conform to its metamodel. A *metamodel* defines the structure for a family of models [229]. Technically, metamodels are just a special type of models. Thus, they have to conform again to another model—the so-called meta-metamodel. Thus, metamodels are written in the language defined by their meta-metamodel. A metamodel specifies the concepts of a language, the relationships between these concepts, the structural rules that restrict the possible elements in the valid models and those combinations between elements [174]. Metamodels are typically expressed with class diagrams, and they can be extended with textual expressions that add further constraints, typically expressed using languages like OCL [181, 221].

2.2 Model Transformations

Model transformations play a key role in MDE [174, 207, 246]. They allow querying, synthesizing and transforming models into other models or code. Thus, they are essential for building systems with MDE approaches. A *model transformation* (MT) can be considered as a program executed by a dedicated transformation engine that takes one or more source models and produces one or more target models [174, 246], as illustrated by the model transformation pattern [189] in Fig. 1. As MTs are specified on the metamodel level, they are reusable for all models of the source metamodel or at least for the subset of models that qualify for a transformation in case the transformation has additional pre-conditions. OCL often plays an important role in MTs as expression language [180].

Depending on the nature of the source and target artifacts, there are *model-to-text* (M2T), *text-to-model* (T2M) and *model-to-model* (M2M) transformations [174]. M2T transformations are typically used to implement code and documentation generators, model serialization, and model visualization [237]. Among the frameworks and languages to define M2T transformations, we can find Acceleo [170], EGL [238], MOFScript [233] and Xtend [172]. T2M transformations are typically used for reverse engineering [134], e.g., transforming legacy applications to models for model-driven software modernization. MoDisco [175] is currently the most popular tool for defining this kind of transformation. Most research on MTs is devoted to M2M transformations. There are different classifications for M2M transformations [189, 230], such as *out-place* and *in-place*. A transformation is considered out-place when it creates new models from scratch, e.g., transforming a class diagram into a relational model [224]. We say a transformation is in-place if it rewrites the source models to produce the target models, as it is, for instance, the case in model refactoring. There is currently a plethora of frameworks and languages available to define M2M transformations, such as AGG [249], ATL [216], AToM^3 [191], e-Motions [236], Henshin [165], JTL [185], Kermeta [214], Maude [187], MOMoT [196, 197], QVT [203] and VIATRA [188].

Running example. Listing 1 displays an excerpt of the *Class2Relational* MT in the ATL language available on the ATL Zoo [142], and its source and target metamodels are displayed in Fig. 2. The *Class2Relational* MT is a simple yet complete scenario traditionally used as a case study to present new approaches or languages for the development of MTs. It was proposed as the challenge of the *Model Transformations in Practice* workshop of 2005 [178] and has been used as a benchmark for MT approaches ever since. According to rule 1, every object of type *DataType* in the source model

(line 5) is transformed to an object of type *Type* in the target model (line 6) with the same *name* (line 7). As for the second rule, it receives as input objects of type *Attribute* whose *type* reference points to an object of type *DataType* and whose *multiValued* attribute is set to *false* (line 10), and it creates an object of type *Column* with the same *name* (line 12) and whose *type* reference points to the *Type* object created from the attribute's type (line 13).

Listing 1. Excerpt of *Class2Relational* MT [142].

```
1 module Class2Relation;
2 create OUT : RelationalMM from IN :
      ClassMM ;
3
4 rule DataType2Type{   -- Rule 1
5   from dt : Class!DataType
6   to t : Relational!Type(
7            name<-dt.name)
8 }
9 rule SingleValuedDataTypeAttribute2Column{
          -- Rule 2
10   from at : Class!Attribute (at.type.
       oclIsKindOf(Class!DataType) and not
        at.multiValued)
11   to co : Relational!Column(
12           name<-at.name,
13           type<-at.type)
14 }
```

An example of a source model and the target model created by this MT excerpt is shown in Fig. 3. Note that we have included a so-called *trace model* in the figure. MT engines typically create a trace model and populate it during MT execution. A trace model basically registers which elements in the target model are created from which elements in the source model and by which rule. Trace models are specifically useful in some testing and debugging approaches, as we shall see throughout the paper. Looking at the figure, please note that object *c* of type *Class* (in the left-most part of the figure) has not been transformed, as we see no trace pointing to this object. The reason is that there is no rule in our MT excerpt that takes objects of type *Class* as input in our MT excerpt.

2.3 Previous Surveys on MT Testing and Debugging

To the best of our knowledge, there is no paper presenting a study of the literature concerning MT debugging. Regarding MT testing, Selim et al. [244] published a paper in 2012 reviewing the state of the art. They organized 29 primary studies according to the phases of the testing process and concluded that more research into all testing phases would be useful. These phases are also identified by Baudry et al. [168], and they are (*i*) model generation, (*ii*) oracle function, and (*iii*) test adequacy criteria. These are the MT testing phases we consider in our survey too.

Although the scope is different from our survey (cf. Section 4.2), it is worth mentioning surveys on MT verification. In 2013, Calegari and Szasz [182] performed a survey on the state of the art of MT verification. Their survey analyzes three components in MT verification: the MT itself, the properties of interest, and the verification techniques used to establish the properties. Later, in 2015, Amrani et al. [164] explored the question of the formal verification of MT properties through a three-dimensional approach that dealt with the same three components proposed in [182]; and Rahim and Whittle [235] published a survey of approaches for verifying MTs. They presented a coarse-grained classification based on the technical details of the approaches and a finer-grained classification according to criteria such as MT languages supported or properties verified.

This article differs from the surveys about MT verification [164, 182, 235] in its scope: testing and debugging. Also, our survey largely updates and complements the work of Selim et al. [244] by

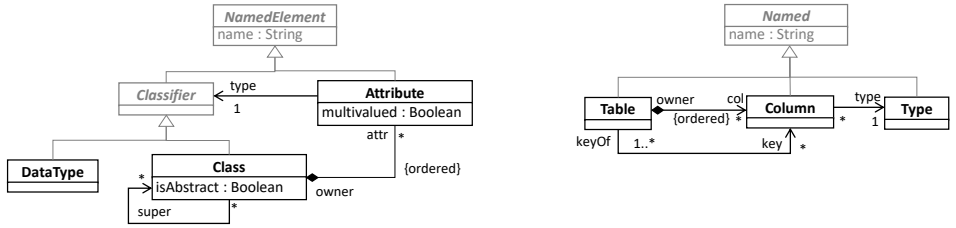

Fig. 2. *Class* metamodel (left) and *Relational* metamodel (right) of the *Class2Relational* MT (from [142])

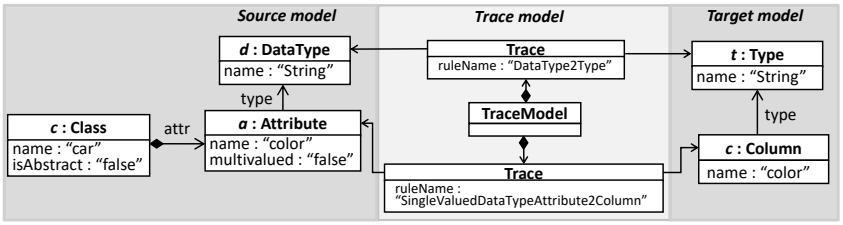

Fig. 3. Execution example of the *Class2Relational* MT excerpt

also considering debugging and reviewing about 100 additional papers, most of them published from 2012 onward.

3 TESTING AND DEBUGGING MODEL TRANSFORMATIONS

To survey the state of the art in testing and debugging of MTs, we propose the conceptual framework displayed in Fig. 4, whose goal is to provide a way to understand the different proposals and how they are connected, as well as to categorise current and future contributions. As illustrated, we have used the original model transformation pattern presented in Fig. 1, and we have augmented it to include testing and debugging concepts, highlighted with grey color in the figure. The parts under study are described below and they are exemplified with our running example:

- *Transformation Definition (SUT).* This is the actual MT, typically implemented with MT languages. In the context of our survey, we will refer to this as the SUT (System Under Test). An example is the MT implementation of our running example (cf. Listing 1).
- *Model Transformation Testing.* This refers to the execution of the MT with the aim of revealing failures (i.e., unexpected outputs). Testing approaches are typically classified according to the testing phase to which they contribute [168, 244], namely:
  - *Model Generation.* This is referred to as the generation of so-called *test models*, which are input models that conform to the input metamodels of the transformation. A test model in our running example is shown on the left-hand side of Fig. 3. Results of our survey on model generation are provided in Section 5.1.
  - *Oracle Function.* In software testing, a test oracle determines if the result of a test case is correct [258] by obtaining *Oracle Outputs*. A test case includes source and target models. There are different ways to construct a test oracle for a model transformation. It typically depends on the available artifacts. For instance, if the expected model is available, a straightforward oracle compares the obtained target model with the expected model. In our running example, the target model obtained from the model on the left-hand side of

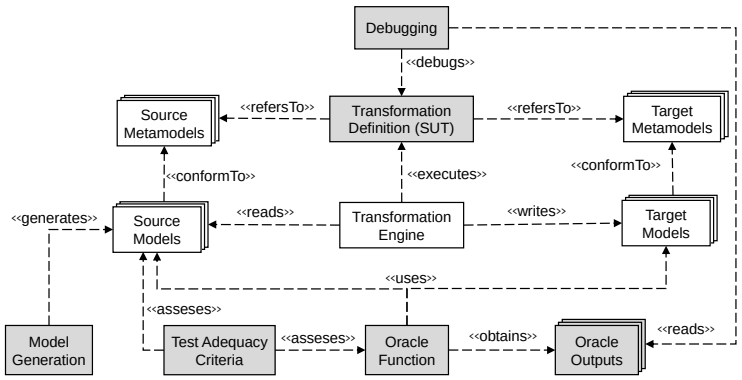

Fig. 4. Model transformation pattern (from [189]) augmented for testing and debugging

Fig. 3 is shown on the right-hand side of the same figure. If we do not have the expected model, an alternative oracle defines properties that the obtained model must satisfy. For instance, in our running example, we can check that if the source model contains objects of type *DataType*, the target model must contain objects of type *Type* with the same *name*. Results of our survey on oracle function are provided in Section 5.2.

  - *Test Adequacy Criteria.* Test adequacy criteria measure the quality of a test suite with respect to one or more objectives. They help define testing goals to be achieved as a result of software testing, e.g., covering a certain percentage of code [168]. In the context of MT testing, test adequacy criteria can be based, for example, on how well the input metamodel is covered or on how effective the oracle functions are on identifying synthetic bugs (so-called mutants) introduced in the MT under test. For example, to cover all the transformation rules of our running example, the union of all test models should contain at least one instance of each non-abstract class in the source metamodel (left-hand side of Fig. 2). Results of our survey on test adequacy criteria are provided in Section 5.3.

- *Debugging.* This consists in locating and fixing bugs in the model transformation specification, typically starting from the failures observed during testing (*Oracle Outputs* in our figure). For instance, if we observe that the target model does not contain an object of type *Type* in the target model with the same *name* as one of the objects of type *DataType* in the source model, then lines 5 and/or 6 in Listing 1 likely contain a bug that needs to be identified and fixed. Debugging techniques can be classified as *dynamic* or *static*, depending on whether they require running the MT or not, respectively. Results of our survey on debugging are provided in Section 6.

4 REVIEW METHOD

To collect the papers related to MT testing and debugging, we followed a structured method partially inspired by the guidelines of Kitchenham [220] and Webster et al. [257]. In addition, we took inspiration from existing surveys on related topics such as formal verification of static software models [202], MT design patterns [223], UML model execution [186], and MT tools [217].

4.1 Research Questions

We aim to answer the following research questions (RQs):

- ***RQ1 - Testing.*** *In which part of the testing process do the studies focus and what do they propose?* We aim to classify the papers according to the three phases devoted to testing (cf. Section 3), namely model generation, test oracle definition and test adequacy criteria. These will be further categorized into subcategories within each phase. Answered in Section 5.
- ***RQ2 - Debugging.*** *What are the approaches for debugging MTs?* We aim to identify the papers focusing on MT debugging, classifying them in subcategories of dynamic and static approaches. Answered in Section 6.
- ***RQ3 - Experimental Evaluations.*** *How are testing and debugging approaches on MTs evaluated?* We aim to provide insights on the current practices for evaluating research proposals in the context of MT testing and debugging. To this end, we explore different dimensions of the evaluations, such as tools proposed and MTs employed as case studies. Answered in Section 7.
- ***RQ4 - Challenges.*** *What are the research challenges for the future?* Based on our survey results, we aim to identify and categorize open research challenges in the field of model transformation testing and debugging and give concrete action points. Answered in Section 8.

Apart from these RQs, in Appendix A we show the trends in testing and debugging of MTs and some interesting statistics from the surveyed papers, such as the number of publications per year and country, top co-authors, frequently used transformation languages, etc.

Table 1. Search engines and number of studies retrieved

Digital Library	ACM	DBLP	Elsevier	IEEE Xplore	Scopus	SpringerLink	Web of Science
# Studies	309	227	242	748	716	369	478

4.2 Inclusion and Exclusion Criteria

We scrutinized the existing literature looking for papers focusing on MT testing or debugging including methods, tools or guidelines. Specifically, we focus on the steps identified in Fig. 4, namely (*i*) model generation, (*ii*) test adequacy criteria, (*iii*) oracle function, and (*iv*) debugging (bug location and fix). Surveys and exploratory papers (e.g., [162, 168, 244]) have not been included as primary studies, but have been considered for setting the scope of this paper, as explained before.

As described in the proposed conceptual model, and inline with the widely adopted notion of testing [198], we focus on testing approaches running the MT under test to identify failures, often referred to as "dynamic testing" [211]. Formal and static (i.e., those not running the MT under test) approaches for identifying bugs in MTs such as formal verification [194, 199, 231, 245, 255, 256] and model checking [262] have been largely studied in related papers and surveys [164, 182, 235] and are out of the scope of our work. We also excluded papers on model-based testing [208, 213, 218], where MTs are often used as a means to test other programs. Regarding debugging, a topic not studied in previous surveys to the best of our knowledge, we included both static and dynamic approaches to provide a complete view of the topic. As we can see from Figure 4, we focus on debugging approaches that read the output of the testing phase.

As models can be treated as graphs [246], we include approaches for testing and debugging graph transformations since they can be considered MTs in this setting. Besides, graph transformations are applied to the problem of instance generation [207]. Therefore, we include the term "*graph transformation*" in the search (cf. Section 4.4). Note, however, that graph transformations are often used as a suitable formalism for verification [207], and that is the reason why many papers on graph transformations are not considered in this survey. Finally, we excluded PhD theses, papers not related to computer science, not written in English, or not accessible from the Web.

4.3 Data Sources

The search was performed in the online repositories[1] of ACM, DBLP, Elsevier, IEEE Xplore, Scopus, SpringerLink, and Web of Science. They all provide an advanced search engine, which fits our purpose very well, as explained later on. We also selected repositories supporting batch retrievals of the bibliographical references. This allowed us to use reference managers for managing the extracted references such as JabRef and Zotero. This drastically reduces the time for processing the references and removing duplicated entries compared to using repositories in isolation.

4.4 Search Strategy and Paper Selection

Table 1 summarizes the number of publications retrieved from each digital library. We used different engines for executing the conceptual query *("model transformation" OR "graph transformation") AND ("test*" OR "debug*" OR "validat*" OR "verificat*")* in title, abstract and keywords. Testing is considered one of the Verification & Validation (V&V) activities [210] and that is why we included the terms *"validat*"* and *"verificat*"* in our searches. This way, we try to gather papers focused on testing that might be referring to the terms "verification" or "validation" and not explicitly to "testing". Also, our search query uses wildcards. Please note that not all search engines support this type of queries. However, we aimed to match the concrete search terms for a particular search engine as closely as possible when queries containing wildcards were not supported. In addition, not all engines allow searching in the title, abstract, and keywords of the papers. For instance, SpringerLink only supports searching in the papers' title and main text.

[1]Throughout the paper, we use the terms "repository", "search engine" and "digital library" indistinctly.

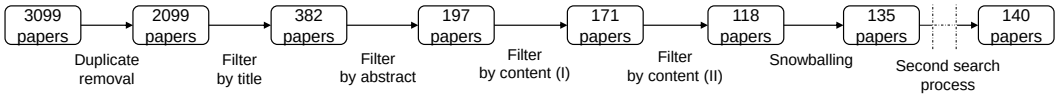

Fig. 5. Overview of selection process

We did not add any time constraints on the search since we did not know the exact point in time when research on the survey topic began. Finally, selected papers were published between 2004 and 2020. Please note that this time frame is reasonable, since the first workshop on model transformations was held in 2005 and this topic started to gain more attention ever since.

We generated one *BibTeX* file for each digital library by following the process reported in [253]. As SpringerLink only allows to produce *CSV* files, we performed a pre-processing step using Zotero to obtain a *BibTeX* file. Subsequently, we collected all references of the individual files together in one single common *BibTeX* file. As illustrated in Fig. 5, we started with an initial set of 3099 papers. Next, we employed JabRef and Zotero to remove duplicates, resulting in 2099 papers.

Then, we read the title of all 2099 publications to decide which ones to discard according to the title. Two authors of the paper were in charge of this process. We identified and removed 1717 papers that were clearly not related to the survey topic according to their title, so we kept a set of 382 publications. The next step was to read the abstract of the papers to keep discarding unrelated papers, this meant 185 more works were dropped out. There were still some papers whose adequacy to this survey was not apparent according to their title and abstract, so we needed to glance at the text for reasoning about their inclusion. After this step, 26 more papers were discarded, having a set of 171 papers. Every relevant step of the review process was followed by meetings where all the authors discussed the doubts and minor disagreements until reaching a consensus.

Then, the 171 papers were distributed among the four authors of the article, who read them and extracted information out of them, as explained in Section 4.5. In this process, 53 more papers were discarded, having a total of 118 papers. This step also included performing a process of backward snowballing [259] by considering the related works described in the papers selected. The rationale of this process is to "rescue" papers that had either not been obtained by any search engine or that were discarded by mistake in the filtering process. 17 papers were obtained after this process.

Since most papers were in the context of M2M transformations, and considering that T2M and M2T approaches can be referred to with the terms *reverse engineering* and *code generator*, respectively, we repeated the complete search process, this time with the query *("reverse engineer*" OR "code generat*") AND ("test*" OR "debug*" OR "validat*" OR "verificat*")*. Five more papers were obtained, so the final set of publications is composed of **140 papers**, henceforth referred to as *primary studies*. The list of primary studies is publicly available on a companion website [254].

4.5 Data Extraction

All 140 primary studies were carefully analyzed to answer our RQs. For each work, we extracted: the full reference, brief summary, type of contribution, context (model generation, oracle function, test adequacy criteria, or debugging), testing dimension (functional vs non-functional), type of MT considered (M2M, M2T, T2M), MT language supported, tool support, characteristics of the experimental evaluation (including techniques, case studies, and availability of experimental assets), and challenges reported. Primary studies were read at least twice by two authors to reduce misunderstandings or missing information. We recorded all the information collected in a spreadsheet. The few disagreements that arose were handled in group discussions involving all the authors.

As a sanity check, we shared a preliminary and a final version of this article with the authors of the primary studies to confirm that the information collected was correct. Some minor changes were proposed and integrated.

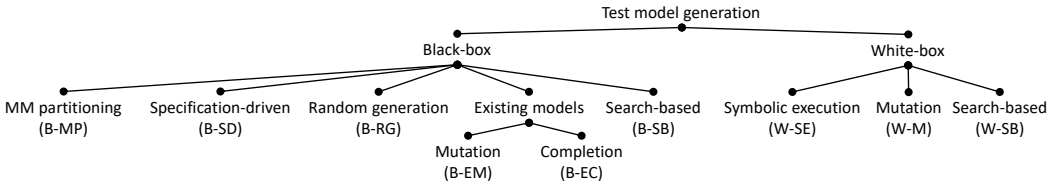

Fig. 6. Categorization of test model generation

4.6 Limitations

The main threat to validity of our work lies in the review method, which did not follow existing systematic guidelines strictly. As a result, there is a chance that some of the reviewed papers do not meet a minimal quality criteria or that we missed some papers. To mitigate this threat, we performed a comprehensive review process, including searching in online repositories, snowballing, and contacting the authors of the reviewed papers for inaccuracies or missing publications. A similar procedure was followed in related popular surveys (e.g., [166, 169, 215, 228, 260]). This makes us confident in the correctness of the reported results. In any case, there may be additional approaches, such as commercial ones, which are not accompanied by publications available in the used databases. Thus, finding these may require multivocal literature reviews [200].

5 MODEL TRANSFORMATION TESTING (RQ1)

This section answers RQ1 (cf. Section 4.1). As explained in Section 3, we classify the MT testing process in three phases: input model generation, test oracle definition, and test adequacy criteria. There are papers that target only one phase, and papers that target more than one, in which case we include them in all phases they target. Next, we summarize the papers and their categorization.

5.1 Test data generation

Test data—test models in our domain—are generated to exercise the MT under test as thoroughly as possible. The categorization we propose for the generation of test models, largely based on the classification of testing techniques proposed by Fraser and Rojas [198], is shown in Figure 6; while the papers that fall under each category, together with some relevant features, are displayed in Table 2. Regarding the table columns, *category* indicates the categorization of each study according to Fig. 6. *Inputs* comprises those inputs needed by the approaches ('*' means it is needed), where *seeds* refer to initial models needed as input. *Features* collect some interesting properties of the approaches. Nothing appears in *MT type* and *MT language* when nothing is mentioned in the paper. Regarding *generation*, it indicates whether the studies construct the models with a proposed algorithm or they rely on third-party model finders (such as SAT solvers)—in the latter case, a proposed algorithm likely orchestrates the whole process. Finally, under *output* we display the *model format* in which the models are generated.

We have identified 60 primary studies that address the generation of test models—note that the same study can fall under more than one subcategory, but we put each work in the one that best represents it. For brevity, in the table as well as in the explanations below, we group papers of the same authors that focus on the same line of work. The categories and the works falling under each category are described below.

5.1.1 Black-box approaches. In black-box approaches, only the specification of the system under test is required, which in the context of MTs refers to the source/target metamodels and, in some cases, a specification of the MT. However, some primary studies proposing a black-box approach focus on an MT language (cf. Table 2) for exemplary purposes. We further categorize black-box studies in the following categories.

Table 2. Approaches for test model generation (**Testing Type** => F: Functional; NF: Non-Functional; **MT Language** => ACG: Auto-Code Generator; GTL: Graph Transformation Languages; **Generation** => AI: Algorithm; MF: Model Finder)

Primary Study		Inputs						Features					Output
Reference	Category (Fig. 6)	Constraints	MM partitions	MM	Seeds	MT imp.	MT spec.	Testing Type	MT Type	MT Language	OCL supported	Generation	Model Format
Burgueño et al. [20–22]	B-MP	*	*	*				F	M2M		*	MF	USE
Gogolla et al. [45, 61]	B-MP	*	*	*				F	M2M		*	MF	USE
Jahanbin et al. [63]	B-MP		*	*				F	M2M		*	AI	.model
Motu et al. [82]	B-MP	*	*	*				F	M2M	Kermeta	*	MF	
Nguyen et al. [87]	B-MP	*		*			*	F	M2M	RTL	*	MF	USE
Sen et al. [109, 110]	B-MP	*	*	*				F	M2M		*	MF	EMF
Wu et al. [138, 139]	B-MP	*	*	*				F			*	MF	
Gogolla et al. [43]	B-SD	*		*				F	M2M		*	MF	USE
Guerra and Soeken [48, 52]	B-SD	*		*			*	F	M2M	ATL	*	MF	USE
Lamari et al. [73]	B-SD	*		*			*	F	M2M		*	AI	
Popoola et al. [90]	B-SD			*				F	M2M			AI	
Runge et al. [94]	B-SD			*			*	F	M2M			AI	
Sampath et al. [91, 96]	B-SD	*		*				F	M2T	ACG		AI	
Scheidgen et al. [101]	B-SD			*				F	M2M			AI	EMF
Almendros-Jiménez et al. [3]	B-RG	*		*				F	M2M	ATL	*	AI	EMF
Ehrig et al. [36, 37]	B-RG			*				F				AI	GGX
Fiorentini et al. [40]	B-RG	*		*				F	M2M			AI	
He et al. [56–58, 140]	B-RG	*		*				NF	M2M			AI	EMF
Nassar et al. [86]	B-RG			*				NF	M2M	Henshin		AI	EMF
Gómez-Abajo et al. [46, 54]	B-EM	*		*	*			F			*	AI	EMF
Sen et al. [108]	B-EM			*	*			F			*	AI	
Brottier et al. [17]	B-EC			*	*			F	M2M			AI	
Min-Hue et al. [62]	B-EC			*	*			F	M2M		*	MF	TCSL
Semerath et al. [106, 107]	B-EC	*		*	*			F	M2M		*	MF	EMF
Sen et al. [111]	B-EC			*	*			F	M2M	Any	*	MF	EMF
Batot et al. [11]	B-SB			*	*			F			*	AI	EMF
Rose and Poulding [93]	B-SB			*	*			F	M2M	ETL		AI	HUTN
Shelburg et al. [112]	B-SB			*	*			F	M2M			AI	
Wang et al. [130]	B-SB			*			*	F	M2M			AI	
Alsibahi et al. [1]	W-SE	*		*		*		F	M2M	Any		MF	
Calegari and Delgado [23]	W-SE			*		*		F	M2M	QVT-R		AI	
Gonzalez and Cabot [47]	W-SE	*		*		*		F	M2M	ATL	*	MF	
Lengyel and Charaf [74]	W-SE	*		*		*		F	M2M	GTL	*	AI	
Mottu et al. [83]	W-SE	*		*		*		F	M2M	Kermeta	*	MF	
Nguyen et al. [88]	W-SE	*		*		*		F	M2M	RTL	*	MF	USE
Sánchez-Cuadrado [97]	W-SE	*		*	*	*		F	M2M	ATL	*	MF	EMF
Schoenboeck et al. [103]	W-SE	*		*		*		F	M2M	Any	*	MF	
Stürmer et al. [113]	W-SE			*		*		F	M2T	TargetLink		AI	
Wang et al. [129]	W-SE			*		*		F	M2M	Tefkat		AI	EMF
Wieber et al. [131, 133]	W-SE	*		*		*		F	M2M	GTL		AI	EMF
Aranega et al. [5]	W-M	*		*	*	*		F	M2M	Kermeta		AI	
Darabos et al. [32]	W-SE			*		*		F	M2M	GTL		AI	
Guerra et al. [53]	W-M	*		*		*		F	M2M	ATL	*	MF	EMF
Alkhazi et al. [2]	W-SB	*		*		*		F	M2M	ATL	*	AI	
Jilani et at. [65]	W-SB	*		*		*		F	M2M	ATL	*	AI	EMF
Sahin et al. [95]	W-SB	*		*		*		F	M2M	ATL	*	AI	EMF

Metamodel (MM) partitioning. Approaches in this category—based on the well-known testing technique *equivalence partitioning* [198]—split the input metamodel into different *partitions*, so that generated models must cover all these partitions and models that cover the same partitions are considered equivalent. For instance, for the *Class* metamodel in Figure 2, a partition could consider the *DataType* class, and another partition could include *Class* and *Attribute* classes. In the works by Sen et al. [109, 110], metamodel partitions and constraints are both transformed into Alloy [212] to generate a Boolean CNF formula and solve it using a SAT solver to obtain the models. Wu et al. [138, 139] propose an approach in the same line, considering metamodel

partitions and OCL constraints. Gogolla and Burgueño et al. apply metamodel partitioning by employing the so-called classifying terms so that models generated are classified into equivalent classes [20–22, 45, 61]. Classifying terms are represented as arbitrary OCL terms on a class model that calculate a characteristic value for each object model [45, 61]. This approach is implemented in the context of the UML-based Specification Environment (USE) tool [201]. Nguyen et al. [87] also propose to use the classifying terms of Hilken et al. [61] for test models generation. Mottu et al. [82] aim to discover new MT preconditions by generating a set of input models based on input domain partitioning, for which the PRAMANA tool based on Alloy is used. Since some of these models may be incorrect or incomplete, the execution of the transformation is analyzed to correct or complete them. Finally, Jahanbin and Zamani [63] propose to enrich the Epsilon Model Generation language (EMG) [90], which uses random operations for producing test models with equivalence partitioning.

Specification-driven. Approaches in this category propose to generate models through specific domain-specific languages (DSLs) or specifications. We find works that define the specification of test models and others that define the specification of MTs. Gogolla et al. [43] propose to generate input models using ASSL (A Snapshot Sequence Language), which is built as an extension of the USE tool [201]. Lamari [73] proposes a formal language for the specification of MTs (MTSpecL) and an accompanying tool. Guerra and Soeken [48, 52] extend the *PAMOMO* DSL [206] for test model generation. Since a specification of the MT is used, and not the MT itself, the approach is classified by its authors as black-box. This specification is translated to OCL expressions, which together with the metamodel are fed to a SAT solver. Runge et al. [94] also use as input a specification of the MT, this time in the form of visual contracts. They obtain the dependency graph for the contracts and propose an algorithm to generate test cases from the graph based on maximizing coverage. Scheidgen [101] defines a DSL called *rcore* that drives the model generation allowing to specify how to deal with concrete choices, such as concrete multiplicities or chosen alternatives. Popoola et al. [90] present the DSL and framework Epsilon Model Generation (EMG), aiming for a semi-automated model generation approach. The validation of the generated models is left to the tester. Finally, Sampath et al. [91, 96] focus on test model generation for testing auto-code generators (ACG). They focus on covering not only the syntactic aspects of a translation, but its semantics too. The approach needs as input a syntactic and semantic metamodel of a modeling language expressed using inference rules and a test specification in the form of a coverage criterion over the metamodel, and generates a test suite that can be used to test any code generator for this language. In this context, a test case consists of a model, inputs to drive the model and the corresponding outputs from the model.

Random. This category includes works that generate models (pseudo-)randomly, a common testing strategy [198]. Ehrig et al. [36, 37] propose instance-generating graph grammars for creating metamodel instances. They implement an MT algorithm that obtains an operational description of the language defined by the metamodel. This allows deriving instances of an arbitrary metamodel in a systematic and random way. The approach by Fiorentini et al. [40] supports exhaustive and random generation for generating small models. The works by He et al. [56–58, 140] propose the opposite: the generation of large random models for performance testing. Inputs to their approaches are the metamodel and a configuration model that serves to specify structure-related constraints, such as number of elements or constraints in the relationships. A similar performance testing approach was proposed by Nassar et al. [86], where large EMF-conformant models are generated applying transformation rules either randomly or following user preferences. Finally, the model generator by Almendros-Jiménez and Becerra-Terón [3] generates models randomly satisfying input OCL constraints, after the user indicates the number of elements to create in the models.

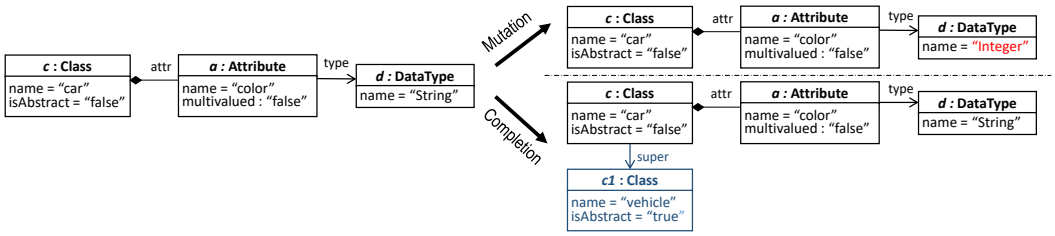

Fig. 7. *Mutation* and *Completion* of existing models

Existing models - Mutation. This category groups those approaches where test models are created by applying changes to existing models. This idea of creating test data by applying changes to existing inputs is often called data perturbation [232] or data mutation [247]. Changes are driven by so-called *mutation operators*, where each operator addresses a specific type of change to the model, e.g., modifying attributes. An example is displayed in Fig. 7, which shows in the left-hand side a model conformig to the *Class* metamodel of Fig. 2. In the top part of the right-hand-side we can see a mutation of the model, where the *name* attribute of the *DataType* class has changed from *"String"* to *"Integer"*. Sen and Baudry [108] synthesize a set of primitive mutation operators from the metamodel as graph-grammar rules [195]. These rules are automatically obtained from any metamodel using a proposed MT, where only operations of addition of elements, relationships and attributes are considered. Further constraints on the metamodel are not considered in the rules. Gómez-Abajo et al. [46, 54] present the domain-specific language WODEL for the specification and generation of model mutants. WODEL provides dedicated language concepts for specifying model mutation (such as deletion and addition of elements), item selection strategies (such as random), and specific concepts for the composition of mutations.

Existing models - Completion. This category considers works where the model generation does not start from scratch, but where existing models, typically called *partial* models, are completed or extended to obtain new models. Please note that the concept of *completing* a model is very similar to *mutating* it, where the mutations performed in the former are related to expanding the model size and complexity. For instance, in the example shown in Fig. 7, we can see in the bottom part of the right-hand-side how the model is completed by adding a new *Class* class, so that the class whose name is *car* now inherits from this one. Brottier et al. [17] propose an algorithm that takes an effective metamodel and fragments of models as input and produces a set of test models. The effective metamodel is the part of the input metamodel that is relevant to the MT. Sen et al. [111] provide a methodology to generate effective test models from partial models with a semi-automated tool. In this context, a partial model is a model conforming to a *relaxed* version of the original source metamodel of the MT. Partial models are automatically completed. Minh-Hue et al. [62] propose to generate models in a modeling language named *Test Case Specification Language* (TCSL). They start with a UML class model and a use case specified in the *Use Case Specification Language* (USL). Test models expressed in TCSL are obtained by means of MTs and the solver in USE, all orchestrated in their USLTG tool. Finally, Semerath et al. [106, 107] integrate a structural graph solver using partial models with the Z3 SMT-solver [192] to generate models that fulfill structural and attribute constraints. The approach is implemented in the VIATRA framework.

Search-based. Some approaches rely on search-based algorithms for the generation of models that optimize one or more objectives, a well-known general testing approach referred to as *search-based testing* [198, 228]. Rose and Poulding [93] adapt Poulding's search-based algorithm [234] for obtaining an optimised probability distribution over the models on which the transformation acts. The optimised distribution is then employed to generate test models by using sampling techniques.

Shelburg et al. [112] propose a multi-objective search-based approach to generate test models from existing ones when the metamodel is modified. Objectives are maximizing the coverage of the updated metamodel, minimizing the number of test model changes, and minimizing the number of test model elements that do not conform to the new metamodel. Wang et al. [130] propose a genetic algorithm to generate test models with the objectives: (*i*) maximizing similarities with given expected metrics' values, (*ii*) maximizing metamodel coverage, and (*iii*) minimizing number of test cases. It uses a mono-objective optimization algorithm. Finally, Batot et al. [11] apply search-based multi-objective model generation. Input models are generated such that they target specific parts of the metamodel tagged as mandatory, i.e., metamodel coverage is one objective. Several minimality criteria are supported as additional optimization objectives in their approach.

5.1.2 White-box approaches. In these approaches, access to the source code of the MT is required for testing. We further categorize white-box studies in the following categories.

Symbolic execution. This category, based on the standard testing technique *symbolic execution* [198], includes proposals that analyze the MT implementation for generating test models that maximize a certain coverage criteria. For instance, if a proposal aims to generate models that trigger all rules, then, for the MT excerpt of Listing 1, there should be at least a model containing an object of type *DataType*, so that rule 1 is triggered, and at least a model containing an object of type *Attribute* that is not multivalued and whose *type* is an object of type *DataType*, so that rule 2 is triggered. There are two papers that explicitly mention *symbolic execution*. First, Schoenboeck et al. [103] propose TETRABox as a generic framework for execution-based white-box testing of MT languages. The path constraints collected from the MT together with the source metamodel are used by the UMLtoCSP constraint solver [179] to generate source models that fulfil all path constraints. The other work is by Alsibahi et al. [1], who present a model finder that uses the relational constraint solver KodKod [252] to check the existence of suitable models.

Some other related approaches are included in this category despite they do not explicitly mention *symbolic execution*. Wang et al. [129] derive an effective metamodel by analyzing the MT rules. They also identify representative values (classes, associations) from the MT rules based on the effective metamodel. Furthermore, they generate so-called coverage items (combinations of representative values). Based on this collected information, they finally generate test input models.

The approach by Gonzalez and Cabot [47] focuses on ATL. The MT is firstly analyzed and a dependency graph is obtained. Then, this graph is traversed a number of times and, finally, test cases are created using the EMFtoCSP tool [179]. Sánchez-Cuadrado [97] also focuses on ATL. He uses static analysis, coverage analysis and model finding to generate test models (pairs of input-output models). His approach needs a seed test model, which can be then extended to cover the MT, for which the USE Model Validator [159] is used. This approach is implemented as a new feature in the AnATLyzer tool [141] and is also used for debugging the MT (cf. Section 6). Mottu et al. [83] propose an approach to statically analyze the MT in order to obtain a metamodel footprint. This, plus other inputs such as OCL invariants, pre-conditions, and the input metamodel are transformed to Alloy [212]. Calegari and Delgado [23] propose to use the dependencies graph for generating test models not covering the whole transformation, but the minimal sets of rules that satisfy every top rule. Similarly, Wieber et al. [131, 133] systematically generate so-called requirement graph patterns from the MT to support test case construction and present a framework for test generation based on Triple Graph Grammars (TGG) [241]. Their test generator produces test cases consisting of pairs of test input models and expected output models. In the work by Nguyen et al. [88], TGG rule dependencies are extracted and test cases are created for covering all rule dependencies of declarative TGG rules. Then, as oracle, patterns for input/output test conditions are transformed into OCL classifying terms with the USE tool [201]. In the approach by

Lengyel and Charaf [74], generated test models cover all execution paths of the MT. To achieve it, the proposed algorithms access the pre and postconditions of the MT rules, so they predict whether the models to be generated exercise all MT rules. Finally, the work by Stürmer et al. [113] deals with M2T transformations, and specifically with model-based code generators. Starting with the transformation, which is expressed as graph rules, they generate test models (they call them first-order test cases) and a set of corresponding test vectors for the models (which they call second-order test cases), the latter being time-dependent. To generate the models, they systematically partition the input space of the graph transformation rules into equivalence classes.

Mutation. This category includes those approaches that propose creating buggy variants (i.e., mutants) of the MT by applying syntactic changes to the implementation of the transformation. This is an application of the well-know testing technique for general-purpose programs *mutation testing* [215]. The MT mutants play an important part in the model generation process. An example of a mutation in rule 1 of the *Class2Relational* MT excerpt of Listing 1 is displayed in Listing 2, where the value given to the *name* attribute of the *Type* object created by the rule has been modified. Three studies fit in this category. Darabos et al. [32] generate models (they call them test graphs) based on a set of mutation rules applied on the rule's preconditions. Aranega et al. [5] propose to apply mutation analysis to semi-automatically improve an initial set of test models. The purpose of this approach is to mutate the MT under test to generate a set of mutants. The initial set of test models serves as input for these mutants. Guerra et al. [53] present a framework for effective mutation testing for ATL. This framework allows to automatically generate mutants for any ATL MT (cf. Section 5.3). Besides, it also allows to synthesize test models able to detect injected bugs in ATL MTs.

Listing 2. Mutation in rule 1 of *Class2Relational* MT.

```
1 rule DataType2Type {   -- Rule 1
2   from dt : Class!DataType
3   to t : Relational!Type (name<-'NewType') -- Mutated; previously: (name<-dt.name)
4 }
```

Search-based. This category includes those papers proposing the use of search-based techniques for test model generation. The approach by Jilani et at. [65] supports the generation of test input models reflecting different coverage criteria such as statement coverage, branch coverage, and multiple condition/decision coverage. Sahin et al. [95] formulate the MT testing problem as a bi-level optimization problem [239] in order to integrate test case generation with mutation testing, and focus on ATL MTs. The objective of the upper level is test case generation to provide a high coverage of the source and target metamodels, and at the same time, to detect the bugs (i.e., mutants) in the MT introduced by the lower level. Hence, the objective is maximizing the number of generated mutants that cannot be detected by the test cases. Finally, Alkhazi et al. [2] present the first approach for test case selection in the context of MTs using multi-objective search. They employ the non-dominated sorting genetic algorithm NSGA-II [193] to find the best trade-offs between two conflicting objectives, namely maximizing MT rule coverage and minimizing execution time.

5.2 Test oracle

Test oracle approaches in the context of MTs depend on the available artefacts. For instance, if the expected model is available, a straightforward oracle is comparing the obtained target model with the expected model. When it is not available, the most common solution is to come up with a set of properties that the generated models must fulfill. These properties are normally called *contracts* or *assertions*. Approaches that propose contracts or assertions vary depending on the way these are obtained. The proposed categorization of test oracle approaches—partially based on the classification of test oracles proposed by Barr et al. [166]— is shown in Fig. 8. Table 3 displays the 43 primary studies that propose test oracle approaches. They are classified according to the categories

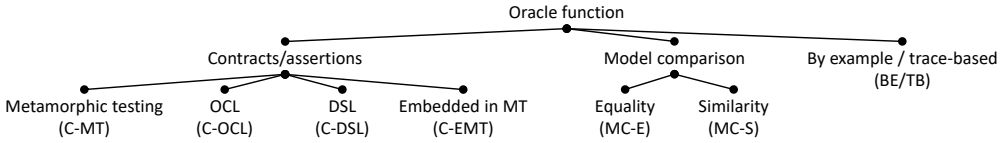

Fig. 8. Categorization of techniques for oracle function

of Fig. 8 and some relevant features are displayed. For instance, we show whether approaches support OCL and propose any DSL. *Oracle* column displays the type of oracle, mainly classified as contracts, expected model or traces; and whether these can be obtained (semi)automatically by the approach or they have to be manually specified—see table caption. In the following we describe all categories and primary studies.

5.2.1 Contracts/assertions. As mentioned before, the so-called contracts or assertions are properties that the models generated by an MT should satisfy. Otherwise, the MT must contain errors. Typically, these assertions are expressed as OCL conditions, so that it is straightforward to evaluate them as true or false. Listing 3 displays a couple of OCL assertions for the MT excerpt of Listing 1. We can see that classes of the source metamodel are prefixed by *"Src"*, while classes in the target metamodel al prefixed by *"Trg"*. This is a common practice in some works [44, 122, 127]. There are approaches that propose a language different than OCL for specifying such conditions, and the way of obtaining the assertions or contracts can also differ depending on the proposal. Thus, we categorize the approaches that propose the use of contracts or assertions as test oracles according to the way in which these are defined or generated. We distinguish the following categories.

Listing 3. Sample OCL assertions for the *Class2Relational* MT.

```
1 --Assertion 1. For each DataType, a Type is created with the same name
2 SrcDataType.allInstances()->forAll(d|TrgType.allInstances()->exists(t|t.name=d.name))
3 --Assertion 2. For each single valued attribute whose type is a datatype, there must exist
      a column with the same name
4 SrcAttribute.allInstances()->collect(a|not(a.multiValued) and a.type.oclIsKindOf(
      SrcDataType))->forAll(at|TrgColumn.allInstances()->exists(c|c.name=at.name))
```

Metamorphic testing. This is a technique to alleviate the oracle problem [184, 242, 243]. It is based on the idea that often it is simpler to reason about relations between two or more executions of the program under test, than to fully formalise its input-output behavior [242]. In the context of MTs, metamorphic testing has been applied to automatically infer assertions (so-called *metamorphic relations*) that should hold between two or more executions of the MT under test. Let us explain it with the example of Fig. 9 for our running example of Listing 1. We have a source model (*SM*) and we do a controlled modification in it, such as adding or deleting elements, in order to obtain the so-called follow-up source model (*fuSM*). When the MT is executed taking as input SM, we obtain the target model (*TM*), while when we execute it taking as input the fuSM, we obtain the follow-up target model (*fuTM*). In this context, a metamorphic relation (*MR*) is defined as a relation among the four parts: $MR = R(SM, fuSM, TM, fuTM)$, such as the MR displayed in the figure. In our example, we observe that the SM contains two *DataTypes* and an *Attribute*, and in the fuSM one more *DataType* and one more *Attribute* have been added. When we execute the MT shown in Listing 1 over the SM, we obtain the TM, which contains two *Types* and one *Column*. When the input for the MT is the fuSM, it produces a fuTM that contains three *Types* and two *Columns*. In the metamorphic relation shown in the figure, $|T_m|$ indicates the number of elements of type T that model m contains. Therefore, the MR in the figure can be read as "If two elements of type *DataType* and one element of type *Attribute* are added in fuSM with respect to SM, then fuTM must contain two more elements of type *Type* and one more element type *Column* than TM".

Table 3. Approaches for oracle function (**Oracle**=> *C-M*: Contract Manual; *C-A*: Contract Automatic; *C-SA*: Contract Semiautomatic; *E-M*: Expected output Manual; *E-A*: Expected output Automatic; *Tr-A*: Trace Automatic)

Primary Study	Category (Fig. 8)	MT Type	MT Lang.	OCL supported	DSL proposed	Oracle
Jiang et al. [64]	C-MT	M2M	ATL	*		C-M
Troya et al. [123, 124]	C-MT	M2M	ATL	*		C-A
He et al. [55]	C-MT	M2M	ATL	*		C-M
Du et al. [34]	C-MT	M2M	ATL	*		C-M
Cariou et al. [25]	C-OCL	M2M	Any	*		C-M
Braga et al. [15, 16]	C-OCL	M2M	Any	*		C-M
Gogolla and Vallecillo (et al.) [44, 127]	C-OCL	M2M	Any	*		C-M
Cariou et atl. [24, 72]	C-OCL	M2M	ATL	*		C-SA
Guerra et al. [48, 49, 52]	C-OCL	M2M	Any	*		C-A
Nguyen et al. [87, 88]	C-OCL	M2M	RTL	*		C-A
Selim et al. [105]	C-OCL	M2M	ATL	*		C-A
Cheng and Tisi [26, 27]	C-OCL	M2M	ATL	*		C-A
Sánchez-Cuadrado et al. [116]	C-OCL	M2M	ATL	*		C-A
Wimmer and Burgueño [134]	C-OCL	M2T/T2M	Any	*		C-M
Guerra et al. [51]	C-DSL	M2M	QVT-R	*	PAMOMO	C-M
Ciancone et al. [28, 29]	C-DSL	M2M	QVTO		MANTra	C-M
Rodriguez-Echeverria et al. [92]	C-DSL	M2M	ATL		MoTe	C-M
Anastasakis et al. [4]	C-DSL	M2M	Alloy	*	Alloy	C-M
Narayan and Karsai [85]	C-DSL	M2M	GReAT		GReAT	C-M
Tiso et al. [119]	C-DSL	M2T	Acceleo	*	*Unnamed*	C-M
Bonfanti et al. [14]	C-DSL	M2T	Xtext		AsmetaL	C-M
Guerra et al. [50]	C-EMT	M2M	ETL	*	transML	C-M
Mazanek et al. [78]	MC-E	M2M	Any			E-M
Wieber et al. [131]	MC-E	M2M	Any			E-A
Lin et al. [76]	MC-E	M2M	ECL			E-M
Tiso et al. [117]	MC-E	M2T	Acceleo			E-M
Stürmer et al. [113]	MC-S	M2T	TargetLink			E-A
Finot et al. [38, 81]	MC-S	M2M	Any			E-M
Kolovos [71]	MC-S	M2M			ECL	E-M
Kessentini et al. [68, 69]	BE/TB	M2M	Kermeta			E-A
Matragkas et al. [77]	BE/TB	M2M	ETL			Tr-A
Jörges and Steffen [66]	BE/TB	M2T	CG			Tr-A

The first work proposing to apply metamorphic testing for MTs is by Jiang et al. [64]. They empirically demonstrated the feasible application of metamorphic testing for MTs, and metamorphic relations were defined manually. Later, Troya et al. [123, 124] automated the generation of metamorphic relations, also expressed in OCL. This was possible by identifying a set of patterns in the execution traces of MTs. He et al. [55] applied metamorphic testing to bidirectional MTs, but they followed a different approach where testers must manually identify generic metamorphic relations of the MT. Finally, Du et al. [34] proposed to combine metamorphic testing and bug localization to debug MTs (cf. Section 6). For the metamporphic testing part, they rely on existing works such as [55, 124].

Object Constraint Language (OCL). Here we describe papers that use or propose OCL contracts (also called "assertions") as test oracles. We first describe approaches that do not automate their generation. The work by Cariou et al. [25] is the first that investigates and discusses the relevance of OCL for defining MT contracts. Then, Braga et al. [15, 16] formalize the concept of contract as transformation contract, which is essentially a transformation model. Similarly, Gogolla and

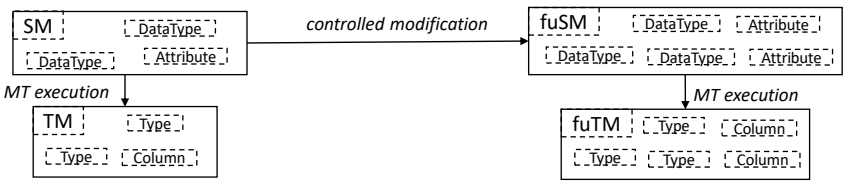

Fig. 9. Application of metamorphic testing in the context of model transformations

Vallecillo (et al.) [44, 127] present the concept of *Tract* as a generalization of the concept of contract. Tracts are OCL conditions that can be used to specify preconditions and postconditions on the transformation as well as constraints that need to be satisfied by any pair of source/target models. They propose the USE tool [201] to check the conformance of the tracts. This approach and tool are used as oracle in some other works [22, 61, 134], where contracts need to be defined manually. Finally, the work by Wimmer and Burgueño [134] proposes to test M2T/T2M transformations by representing text within a generic metamodel.

Regarding works that propose a (semi-)automation in the generation of OCL contracts/assertions, Cariou et al. [24, 72] propose a tool to help in the definition of the contracts, so that these can be semi-automatically built. Guerra et al. [48, 49, 52] generate assertions from MTs specified in PAMOMO. To do this, they provide a script that traverses all invariants and postconditions in the MT specification and generate the corresponding OCL assertions. Nguyen et al. [87, 88] follow a similar approach for the generation of OCL assertions. They apply the same scripts, but this time for MTs specified in RTL (Restricted Transformation Language). There are a number of works that implement approaches for ATL. Selim et al. [105] propose an approach to automatically generate OCL contracts. For this, ATL MTs are first translated to transformation models [177]. Cheng and Tisi [26, 27] design sound natural deduction rules for ATL and apply these rules on the postconditions of the MT to generate further OCL contracts. The approach helps the user pinpoint the bug. Finally, Sánchez-Cuadrado et al. [116] present and automate a method to translate target OCL constraints (constraints defined on the target metamodel of an MT) to the source metamodel using information from ATL transformations. The method allows us to ensure that if a source model satisfies the source constraints, the transformed target model will satisfy the target ones.

Domain-specific languages (DSLs). There are works that propose domain-specific languages to specify assertions/contracts. In [51], PAMOMO is proposed to manually define visual contracts specifying preconditions, postconditions, and invariants for the transformation. The visual contracts are tested by translating them into the QVT-Relations language, and subsequently, using a QVT engine in check-only mode. Ciancone et al. [28, 29] focus on testing QVTO transformations, for which they present the MANTra proposal and tool. Assertions have to be manually defined, for which MANTra provides an assertions API to define them. Rodriguez-Echeverria et al. [92] present a DSL called MoTe for manually defining contracts with a semantics based on graph transformations. The test oracle execution consists in the computation of precision and recall metrics for every relation between input and output patterns defined by a contract. Anastasakis et al. [4] propose to analyze MTs via Alloy. Assertions are manually defined using the so-called *Alloy statements*. Narayan and Karsai [85] use the term *correspondence rules* for the contracts, which are manually expressed as path expressions. In the context of M2T transformations, Tiso et al. [119] formulate the oracles in terms of the properties of the generated text files, such as the structure of text fragments or which files and folders must be present and how they are named. They propose a DSL to manually define such oracles. Finally, Bonfanti et al. [14] present an M2T transformation that implements a code generator from ASMs to C++, for which they use Xtext. Along with the C++ generated code, they also obtain unit tests from the tests defined at model level in the ASMs. The authors argue that these tests can validate the M2T transformation.

Embedded in MT. In this category we consider approaches that inject the definition of assertions into the model transformation code. We include here the work by Guerra et al. [50], which analyzes a formal specification of the MT. Assertions generated from patterns specifying pre-conditions on input models are included in a dedicated *pre* section of the MT definition, while assertions generated from patterns specifying correctness properties of the MT or of the expected output models (so to speak post-conditions) are injected in a dedicated *post* section of the MT.

Model comparison. The most intuitive way to check the output of a system is to compare it with the expected output. Some approaches aim for equality, while others for similarity.

Equality. In the MT context, there are several approaches that propose to compare the output model(s) with expected ones. There are some works that rely on EMFCompare [163] or existing comparison procedures, like hash comparison and link creation, for comparing models [78, 131], while others propose specific algorithms or techniques. For instance, Lin et al. [76] propose an algorithm that provides an output based on new elements, deleted elements and changed elements between the two models. Finally, Tiso et al. [117] present a testing framework for M2T transformations, where the oracle part is delegated to the developer, who needs to manually compare the text generated with the expected text.

Similarity. Stürmer et al. [113] focus on M2T transformations by proposing the systematic testing of model-based code generators. As oracle function, they propose to compare the test outputs of the model with the test outputs of the resulting code. The comparison yields *correct* if a sufficiently similar behavior of the outputs is observed, for which a signal comparison algorithm is proposed. Finot et al. [38, 81] propose an approach to compute the difference between the output model and a partial expected model, returning a difference model. Kolovos [71] presents the Epsilon Comparison Language (ECL), a task-specific model management language that allows to develop language-specific algorithms for establishing matches between different models. The result of comparing two models with ECL is a trace mainly consisting of a number of rule matches that basically indicates if the elements found in a model are present in the other.

By example / trace-based. Approaches in this category need to either inspect the trace model generated by the MT execution—the concept of trace model is explained in Section 2 using our running example—or the traces that the input and output may generate (such as in the case of code generators). We identify three types of approaches in this category. First, Kessentini et al. [68, 69] present an oracle function based on the notion that the more an MT deviates from well-known MT examples, the more likely it is to be faulty. They compare the output model not with a corresponding expected output for the given input model, but with an already available set of examples which contain good quality MT traces from past MTs. The main benefit of this work is that it is not necessary to have an expected output model. Second, Matragkas et al. [77] propose to enrich the execution traces after the MT execution with domain-specific semantics and check for conformance with respect to the MT specification. In this context, an MT generating non-conforming traces is considered as erroneous and requires fixes. Finally, Jörges and Steffen [66] focus on testing code generators. Since their input test models are executable, their approach proposes to obtain execution traces from the input test models as well as execution traces from the source code generated from the input test models by the code generator under test. A matcher compares the two execution traces to check whether the same atomic actions occurred in the exact same order.

5.3 Test adequacy criteria

As explained in Section 3, test adequacy criteria measure the quality of a test suite with respect to one or more objectives. Test adequacy criteria help in defining testing goals to be achieved as a result of software testing, e.g. covering a certain percentage of code [168]. In the context of MT testing, test adequacy criteria can be based, for example, on how well the input metamodel is covered by the test models, or on how effective the oracle functions are on identifying synthetic bugs (so-called mutants) introduced in the MT under test (cf. Fig. 4). Fig. 10 depicts the proposed categorization of test adequacy criteria for MT, mostly inspired on the seminal survey on test coverage and adequacy by Zhu et al. [263]. The 19 primary studies that propose approaches for test adequacy criteria are displayed in Table 4, where they are grouped by categories (cf. Fig. 10). The table also displays some relevant information of the studies, such as whether they support OCL,

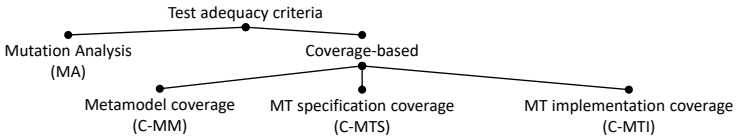

Fig. 10. Categorization of test adequacy criteria approaches

Table 4. Approaches for test adequacy criteria (**Mutation** => the approach proposes Manual (M), Partially Automated (PA) or Automated (A) mutation of the MT; **Coverage**=> the approach proposes the coverage of In/Out MM, Spc (MT Specification) or Imp (MT Implementation))

Primary Study	Category (Fig. 10)	MT Type	MT Language	OCL supported	Mutation
Aranega et al. [5, 8, 9]	MA	M2M	Kermeta		M
Guerra et al. [53]	MA	M2M	ATL	*	A
Kahn and Hassine [70]	MA	M2M	ATL		M
Mottu et al. [80]	MA	M2M	Any		M
Sánchez-Cuadrado et al. [99]	MA	M2M	ATL	*	A
Troya et al. [120]	MA	M2M	ATL		PA
					Coverage
Finot et al. [39]	C-MM	M2M	Any	*	Out MM (Class, Asc, Attr)
Fleurey et al. [41]	C-MM	M2M	Kermeta	*	In MM (Class, Asc, Attr)
Wang et al. [128]	C-MM	M2M	Tefkat		In-Out MM (Class, Asc, Attr)
Bauer et al. [12, 13]	C-MTS	M2M	Any	*	MM (Class, Asc, Attr), Spc (contracts)
Tiso et al. [118]	C-MTS	M2T	Acceleo	*	MM (Class, Asc, Attr), Spc (template)
Arifulina et al. [10]	C-MTI	M2M	GTL		Imp (rule, path)
Calegari and Delgado [23]	C-MTI	M2M	QVT-R		Imp (rule)
García et al. [42]	C-MTI	M2T	MOFScript	*	Imp (rule)
Heckel et al. [59]	C-MTI	M2M	GTL		Imp (dataflow)
McQuillan and Power [79]	C-MTI	M2M	ATL	*	Imp (rule, instruction, decision)
Wieber and Schürr [132]	C-MTI	M2M	GTL		Imp (pattern)

the type of coverage proposed or the mutation activity—see table caption. The different categories and the primary studies are explained in the following.

5.3.1 Mutation analysis. Approaches in this category measure the effectiveness of test cases according to their ability to detect bugs, for which they propose mutation analysis [215] as test adequacy criteria. The idea is to generate buggy variants (i.e., mutants) of the model transformation under test. These mutants contain one or more bugs. An example of mutation for rule 1 of the MT shown in Listing 1 is displayed in Listing 2. The adequacy of the testing approach is measured according to its ability to detect the mutants. It is noteworthy that some works apply mutation analysis in order to evaluate their research contributions (such as [18, 39, 110, 122, 131, 133] and many more)—in fact, mutation analysis is the most frequently used technique in the evaluations. These works are not considered as primary studies in this category, since they do not advance the state of the art in mutation analysis of MT, but they use existing proposals. Here we only include those papers that present a contribution in the context of mutation analysis for MTs.

Mottu et al. [80] were the first authors to explore mutation analysis for MTs. They study potential bugs that developers may introduce in MTs. They do not focus on a specific MT language, but define a set of generic mutation operators for MTs based on model navigation, model's elements filtering, output model creation and input model modification. They give detailed explanations of the mutations proposed. Aranega et al. [5, 8, 9] focus on the mutation operators presented by Mottu et al. [80] and describe a way to systematically and automatically generate them for the Kermeta language.

The works by Kahn and Hassine [70], Troya et al. [120], Sánchez-Cuadrado et al. [99] and Guerra et al. [53] focus on mutation operators for ATL MTs. Kahn and Hassine [70] propose a set of 10 mutation operators that are mainly based on the operators presented by Mottu et al. [80]. They exemplify the operators in an ATL MT example but do not provide means to automate them.

Troya et al. [120] derive a systematic set of 18 ATL mutation operators by proposing a general language-centric synthesis approach. They explain each of the mutation operators proposed and the consequences they have in the generated output model(s). They also automate the generation of MT mutants by realizing a framework that exploits the concept of Higher-Order Transformations [250], but only describe the solution and implementation for a couple of mutants. Sánchez-Cuadrado et al. [99] present a set of 27 mutation operators that they use to evaluate their AnATLyzer tool, and they automate the mutants generation. Finally, Guerra et al. [53] revise mutation operators proposed in the literature and, in addition, propose a new set of operators emulating the most frequent typing errors in ATL transformations. Regarding operators proposed in the literature, they integrate (*i*) the operators presented by Troya et al. [120], which they name *syntactic operators*, (*ii*) the operators proposed by Mottu et al. [80], which they name *semantic operators*, and (*iii*) the operators presented by Sánchez-Cuadrado et al. [99], which they call *typing operators*. Furthermore, they analyze the entire ATL zoo [142] in order to discover new mutation operators derived from most common errors. For this, they use their AnATLyzer tool to discover errors in the MTs available on the Zoo and extract a new set of mutation operators. This approach provides tool support for the automated generation of a wide variety of ATL MT mutants.

5.3.2 Coverage-based. This category includes those papers that propose to measure the effectiveness of a testing approach for MTs according to its ability to cover the input/output metamodels and/or the MT under test. We collect papers in the following three categories.

Metamodel coverage. This includes the papers measuring test adequacy according to the portion of the input/output metamodels covered. As an example of this kind of coverage, the model obtained by completion in the bottom of the right-hand-side of Fig. 7 covers a larger part of the metamodel than the model in the left-hand side because it additionally includes (i) a *Class* instance whose *isAbstract* attribute is set to *false* and (ii) a relationship of type *super*. Wang et al. [128] study how much the MT under test is covering the input and output metamodels. They measure coverage from different perspectives, such as feature coverage, inheritance coverage, association coverage, model element coverage and metamodel coverage; and they propose an algorithm to compute them. Fleurey et al. [41] propose to measure the quality of a set of test models by measuring how much they cover the input metamodel, which they propose to measure in terms of class coverage, attribute coverage, and association coverage. Finot et al. [39] propose to compute the coverage of the output metamodel. They measure the elements of the output metamodel that are exercised by the oracle. The metamodel coverage is measured as proposed by Fleurey et al. [41].

Model transformation specification coverage. Approaches in this category measure the adequacy of the testing approach based on the portion of the MT specification covered. These approaches are typically used to test the adequacy of test oracles. Bauer et al. [12, 13] present a coverage analysis approach for measuring test suite quality for MT chains. They focus on the coverage of the metamodel and a specification of the transformation chain expressed by contracts that specify conditions for the models used and created by the MT. To compute coverage, footprints are extracted for the test cases, which contain the main characteristics of the test case execution. Tiso et al. [118] discuss coverage in the context of M2T transformations. Despite they do not present any specific approach, they describe a coverage criterion that checks that the various templates on the preconditions of the rules are instantiated on the input models.

Model transformation implementation coverage. We include here those works that focus on coverage of the MT under test. Wieber and Schürr [132] focus on the coverage of the pattern matching in graph transformations. Their basic idea is to stimulate the pattern matching engine so that combinations of variable binding and unbinding steps do occur. Heckel et al. [59] and Arifulina et al. [10] also deal with graph transformations. The former propose a data-flow coverage approach

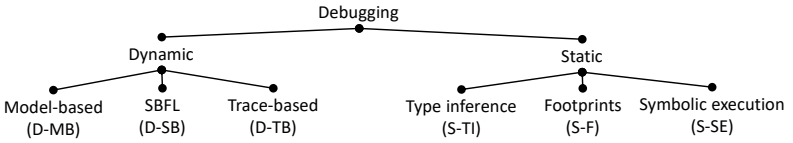

Fig. 11. Classification of debugging techniques

Table 5. Approaches for model transformation debugging (**Contracts**: the approach proposes contracts to check the presence of errors; **SbS**: the approach allows step-by-step debugging; **Activity** => BL: Bug Localization; BF: Bug Fix)

Primary Study	Category (Fig. 11)	MT Type	MT Lang.	OCL supported	DSL proposed	Contracts	SbS	Activity
Ege and Tichy [35]	D-MB	M2M	Any graphical					BL & BF
Jukss et al. [67]	D-MB	M2M	Any	*	AToMPM	*		BL
Mészáros et al. [84]	D-MB	M2M	GTL		VMTS		*	BL
Schönböck et al. [103]	D-MB	M2M	Any	*	PaMoMo	*		BL
Wimmer et al. [102, 104, 135–137]	D-MB	M2M	QVT-R	*	TROPIC	*	*	BL
Du et al. [34]	D-SB	M2M	ATL	*		*		BL
Li et at. [75]	D-SB	M2M	ATL	*		*		BL
Troya et al. [121, 122]	D-SB	M2M	ATL	*		*		BL
Aranega et al. [6, 7]	D-TB	M2M	QVTO			*		BL
Corley et al. [30, 31]	D-TB	M2M	MoTif				*	BL
Dhoolia et al. [33]	D-TB	M2T	Any			*		BL
García et al. [42]	D-TB	M2T	MOFScript	*				BL
Hibberd et al. [60]	D-TB	M2M	Tefkat			*		BL
Ujhelyi et al. [125, 126]	D-TB	M2M	VIATRA2					BL
Sánchez-Cuadrado et al. [97–100, 114, 115]	S-TI	M2M	ATL	*				BL
Burgueño et al. [18, 19]	S-F	M2M	ATL	*		*		BL
Oakes et al. [89]	S-SE	M2M	DSLTrans			*		BL

implemented by generating a dependency graph between the rules that registers whether a rule creates/deletes/updates an element, while the latter produce an invocation graph representing all possible sequences of rules that can result from executing every possible input model. McQuillan and Power [79] define coverage measures for ATL MTs. For this, they first propose processing the compiled ATL transformations (i.e., instructions for the ATL VM) to collect information such as operations, branch locations, etc. Then, they run the transformation and process the resulting log file to estimate the actual coverage for the executed transformation. They present three types of coverage metrics: rule coverage, instruction coverage and decision coverage. Calegari and Delgado [23] focus on QVT-Relations transformations and define a test adequacy criteria based on the coverage of every possible rule chain and, thus, the whole MT. Finally, García et al. [42] deal with M2T transformations in the MOFScript language. They argue that the transformation coverage by the test suite may be informed based on the executed transformation lines, so the approach builds on trace models (cf. Section 2), and coverage is conducted at the rule level.

6 MODEL TRANSFORMATION DEBUGGING (RQ2)

This section aims to answer RQ2. As described in our conceptual model, debugging focuses on locating and fixing bugs in the MT, often starting from the failures observed during testing. Fig. 11 depicts the proposed classification for approaches on MT debugging, partially inspired on the survey on fault localization by Wong et al. [260]. Table 5 displays the 31 primary studies that propose debugging approaches, classified by the categories of Fig. 11. Some interesting properties of the studies are also displayed—see table caption. The different categories as well as the primary studies are summarized in the following.

6.1 Dynamic approaches

This category includes approaches where the model transformation needs to be executed in order to debug it. This means that a model transformation engine as well as a (set of) input model(s) need to be available. We further classify these papers in the following categories:

Model-based. Approaches in this category typically propose a modeling notation in order to debug the model transformation, claiming to perform debugging at model level. The works by Wimmer et al. [135–137] and by Schönböck et al. [102, 104] propose a model-based debugger representing QVT Relations on basis of TROPIC, a transformation language representing a variant of Colored Petri Nets (CPNs). In order to provide a convenient debugging environment, TROPIC aims for a dedicated view on all artifacts of a transformation, i.e., the metamodels, models, and transformation logic. This is possible by having a dedicated runtime model which also enables the investigation of each operational step of a MT. Later, Schönböck et al. [103] employ their Pattern-based Modeling Language for Model Transformations (PAMOMO) to compute failure traces that can be mapped back to the MT implementation as an input for further debugging steps. Jukss et al. [67] describe a layered approach to debugging by mapping familiar debugging operations to different formalisms in order to raise the abstraction for debugging to a similar level as the models being transformed. Declarative queries can be performed during the debugging session. This approach is evaluated by an implementation in AToMPM [227], a browser-based tool for Multi-Paradigm Modeling.

A couple of approaches base their model-based debugging mechanisms on a graphical syntax. Mészáros et al. [84] present a visual model transformation debugger realized in the Visual Modeling and Transformation System. The solution facilitates the step-by-step execution of model transformations, the visualization of the overall state of the transformation and also supports individual matches. In addition, the transformations can be dynamically updated when being executed which allows for interesting debugging possibilities. Ege and Tichy [35] present an approach for debugging visual declarative model transformation languages. It proposes to highlight the parts of the MT that likely need a change. It also proposes changes to the models and MT for repair.

Spectrum-based Fault Localizatoin (SBFL). SBFL is a testing technique that uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component of being faulty [260]. We have included it as a category since several works have applied it in the context of model transformation. The recent works by Troya et al. [121, 122] were the first ones applying this technique in the context of MTs, obtaining promising results. They propose to apply SBFL to MTs by using the model transformation rules as program components under examination. By executing the MT under test with a number of input models, some executions result in success and some other in failure according to an oracle composed of OCL assertions that pairs of <input,output> models must satisfy. With the results given by the oracle as well as the rules triggered when executing the transformation with different input models, the so-called *coverage matrix* and *error vector* can be built. With different mathematical formulae proposed in the literature [260], transformation rules are ranked according to their likelyhood in containing a bug [121, 122]. Later, Li et at. [75] present a similar approach, where they propose to use weighted test models as well as weighted rule coverage to improve the performance of SBFL. Finally, Du et al. [34] propose to apply SBFL without the need to count on an oracle. To achieve this, the oracle is obtained by applying metamorphic testing techniques [242], similar as it is done in [124] (cf. Section 5.2).

Trace-based. We group in this category all dynamic approaches that make use of the trace model in the debugging process (cf. Section 2 for the definition of trace models). Most dynamic approaches for MT debugging make use of traces. For instance, approaches in the category of SBFL need to check which rules were executed, for which they need to look into the traces. However, we include in this *trace-based* category those approaches whose main contribution in MT debugging is the use of traces or the way of constructing them.

Hibberd et al. [60] coin the term *forensic* debugging of MTs to refer to ad-hoc debugging, i.e., debugging once the model transformation execution finishes. They do not propose an approach to forensic debugging, but classify MT bugs and explore debugging approaches. In their study,

they highlight that the trace of a MT is a key-enabling factor. They argue that the information provided by the traces of MTs can be leveraged for more effective post-hoc debugging techniques than it is possible with traditional languages. In the works by Aranega et al. [6, 7], local and global traces are built during the MT execution. When there is an error in the output model, the traces are inspected by an algorithm until a set of likely buggy rules is found. However the exact localization of the actual buggy rule is manually done. Ujhelyi et al. [125, 126] present a dynamic backward slicing approach for MT implementations and their associated models by exploiting the generated execution trace models of MT engines. Corley et al. [30, 31] apply omniscient debugging to MTs. According to the authors, omniscient debugging is a natural extension of stepwise execution that enables reverse execution. The authors define a trace-based omniscient debugger that supports basic omniscient debugging features such as backwards execution for MTs.

There are a couple of works that aim at debugging model-to-text (M2T) transformations. García et al.'s work [42] heavily rests on trace models, which capture a ternary relationship between the source model elements, the elements of the MT, and the produced code (i.e., text). The authors choose MOFScript as the M2T transformation language, and complement MOFScript's native trace model with an additional trace model that enables full fine-grained traceability between MT elements and locations in generated text files. The other approach is by Dhoolia et al. [33], who associate taint marks with input model element, and propagate them using an instrumented model transformer, to generate a taint log, in which taint marks are associated with substrings of the output. Any erroneous substring in the output, or the location of a missing output, may thus be associated with a taint mark and projected back to the related input model element(s).

6.2 Static approaches

This category includes approaches where the MT is not executed in order to debug it. We further classify these papers in the following categories:

Type inference. This category groups the works by Sánchez-Cuadrado et al. [97–100, 114, 115]. They present a method for statically analysing ATL MTs, especially to find typing and other errors such as unresolved bindings, uninitialized features, or conflicting rules. Their approach is based on static analysis and type inference. Furthermore, by using a constraint solver, it can be checked if there is actually a possible source model triggering the execution of a buggy statement. To evaluate the usefulness of the proposed method, the authors have implemented the approach in a tool named AnATLyzer, which aims for a test-driven development of ATL MTs [97], and analysed the entire ATL zoo [142]. AnATLyzer identified a huge number of errors in the existing MTs which shows the need for such tool support for MTs and has been employed for the test-driven development of MTs.

Footprints. The works by Burgueño et al. [18, 19] present a static approach for locating buggy rules in MTs. Tracts [44, 127] are used to express conditions that the MT must satisfy. By extracting the *footprints*, i.e., metamodel elements, used in the tracts and in the MT rules, matching functions are constructed to automatically generate alignments between MT specifications and implementations. Such alignments are key for interpreting the testing results, i.e., the result of the tracts evaluation.

Symbolic execution. There is only one work that applies symbolic execution for debugging MTs [89]. The approach presented is integrated within the SyVOLT tool, which verifies DSLTrans transformations. This is achieved by generating the full state space for a MT, i.e., reflecting all possible executions, which allows to prove structural contracts for the MT. SyVOLT allows to detect and even localize errors in both artefact types: in the implementation and in the contracts of the MT.

7 EXPERIMENTAL EVALUATIONS (RQ3)

In order to reply to RQ3, we have analyzed the evaluations performed in the primary studies, where we specially focus on the case studies used. Besides, we have analyzed the tools proposed for MT testing and debugging and those used in the experimental evaluations.

Table 6. Reported tools

Name	Description	Studies	Language	Last update
AnATLyzer [141]	Static analysis of ATL transformations	[98–100, 114–116]	Java	2020
AToMPM [227]	Generation of modeling web-based tools	[30, 67]	JS/Python	2021
DSLTrans [143]	Contract-based language verification	[89]	Java	2020
Eclipse Epsilon [144]	Model generation and comparison	[71, 90]	Java	
EMF Model Generator [145]	Large EMF models generator	[86]	Java	2019
EMFtoCSP [146]	Automatic verification of UML/EMF models	[47]	Java	2018
MDE Testing [148]	Mutation testing tool for ATL	[53]	Java	2020
MRs4MTgenerator [149]	Generation of metamorphic relations	[124]	Java	2017
PAMOMO [150]	Pattern-based inter-modelling	[49, 50, 52]	Java	2010
PRAMANA [151]	Generation of models using Alloy	[82, 83, 111]	Java	
Not named [156]	Generation of model transformation mutants	[120]	Java	
Not named [157]	Model constraint mutation (USE extension)	[20]	Java	2019
Not named [158]	Automated generation of models	[57, 140]	Java	2018
RandomEMF [152]	Random model generation	[101]	Java	2015
SBFL_MT [153]	Spectrum-based fault localization	[122]	Java	2017
SymexTRON [154]	White-box test case generation	[1]	Scala	2016
TracsTool [155]	Black-box checking of M2M transformations	[19, 134]	Java	
USE [159]	UML/OCL System specification and validation	[20–22, 43–45, 61]	Java	2020
VIATRA Generator [160]	Graph solver for consistent model generation	[106, 107]	Java	2021
WODEL [161]	Automated generation of model mutants.	[46, 54]	Java	2020

7.1 Tools

The available tool support is typically a good indicator of the maturity of a research field. To this end, we studied the tools reported in the empirical evaluations of the primary studies. Specifically, 86 out of the 140 primary studies mentioned some kind of tool support, although many of them were not publicly available at the time this survey was conducted (links to the available tools are provided on the companion website [254]).

Table 6 shows the tools presented in the primary studies that are available at the time of writing this survey. For each tool, the table shows its name (if any), brief description, primary studies using them, and the year of the last update (if available). As illustrated, the most popular tools are USE [159]—a modeling tool for system specification and validation using a subset of UML and OCL, and AnATLyzer [141]—a static analysis tool for ATL transformations, referenced in 7 and 6 primary studies, respectively. Conversely, 10 tools were referenced only once. Interestingly, only two tools, USE and Eclipse Epsilon, were used by different groups of authors. This suggests that existing tool support is scattered and mostly exploited by the groups where they were developed. A vast majority of the tools are written in Java, which places it as the dominant programming language. We managed to access the source code of all the tools except of one. Only 7 tools had been updated in the 18 months prior to writing this survey.

7.2 Case studies

We have identified 153 different case studies, out of which 24 come from industry, 123 do not, and we could not identify the nature of the remaining 6. For the 24 cases coming from industry, we must remark that the study was always performed outside the industrial setting.

Table 7 presents the case studies that have been used in at least three different studies. For each case study, we show the primary studies in which it has been used, the number of studies (column *# Studies* for readability purposes), and the languages in which it has been implemented. It is worth noting that none of these frequently-used case studies comes from industry, that all transformation scenarios focus on out-place M2M transformations and that most of them target the ATL language. Several case studies address similar domains with slight differences. In particular, the database domain is the most widely targeted, which includes the case studies: *Class2RDBMS*, *Class2Relational*, *UML2ER*, *SimplyUML2Rel*, and *JavaSource2Table*.

Apart from the 13 case studies presented in Table 7, we identified another 13 case studies which are each used in two papers (*ER2ReL, Simplified StateChart MM, extlibrary MM, KM32EMF, Ant2Maven,*

Table 7. Most-used case studies

Case study	Studies	# Studies	Languages	Size
BibTex2DocBook	[2, 18, 21, 22, 45, 52, 61, 75, 87, 100, 122]	11	ATL	9 rules, 4 helpers
Class2RDBMS	[5, 9, 41, 65, 80–83, 109, 110]	10	ATL, Kermeta	1 rule, 6 helpers (ATL)
Class2Relational	[34, 49, 53, 64, 72, 100, 121, 123, 124]	9	ATL	7 rules
UML2ER	[2, 18, 19, 34, 53, 75, 92, 102, 122]	9	ATL	8 rules
Families2Persons	[3, 18, 19, 22, 44, 45, 61, 100, 127]	9	ATL	2 rules, 2 helpers
CPL2SPL	[2, 18, 53, 98, 114, 122]	6	ATL	19 rules
HSM2FSM	[26, 27, 53, 55, 116]	5	XMU	6 rules
PetriNet2PNML	[34, 98, 120, 124]	4	ATL	4 rules
Ecore2Maude	[2, 18, 75, 122]	4	ATL	39 rules
Book2Publication	[3, 28, 29, 100]	4	ATL, QVT-O	1 rule, 3 helpers (ATL)
Families2Persons Extended	[34, 89, 124]	3	ATL	10 rules
SimplyUML2Rel	[128–130]	3	Tefkat	6 rules
JavaSource2Table	[57, 100, 140]	3	ATL	4 rules, 2 helpers

Maven2Ant, Gaspard transformation chain, SAMM2KLAPER, BPMN2PetriNets, UML2BPMN, UML2Java, Class2Rel and *Ecore*). The remaining case studies were used only in one paper.

With respect to industrial case studies, we observed that only 7 out of 140 primary studies contain at least one industrial case study and only one (*UML2Java*) was re-used in two papers [27, 134].

8 CHALLENGES (RQ4)

In this section, we derive several open research challenges in the field of MT testing and debugging as synthesis from our survey results. For each challenge, we provide a name, a description of what is currently missing, and potential directions with concrete action points to improve this situation.

Challenge 1: Exemplars for MT Testing and Debugging. Currently, there are several transformations reused across different languages and studies—as those suggested at events such as [171]—for performing research on MT testing and debugging (cf. previous section). However, the reused information mostly concerns the pure MT implementation itself. Most often, input models, transformation specifications, and potential transformations errors are redeveloped from scratch for each study, which is of course a major obstacle for comparison and future studies as also reported in other specialized fields of testing, such as compiler testing [183].

Action Points. In addition to existing MT collections such as the ATL Transformation Zoo, the community may establish additional repositories where transformation testing and debugging packages are available. This is successfully done in related fields such as for general-purpose programming languages, e.g., see Siemens Suite[2] and Defects4j[3]. These types of collections typically include buggy programs with well-documented real-world bugs. Based on such collections, existing approaches may be more systematically compared according to their ability to detect and/or locate these bugs. As a starting point, one may collect the cases used in the previous studies discussed in this survey and provide them in a consolidated way.

Challenge 2: Generalization of existing Approaches to MT Testing and Debugging. Current research conducted in this field is mostly language-specific. As we have seen in the results, there are prominent MT languages in the previous studies for which dedicated support is developed. However, it is still unclear which concepts and techniques may be reused for other languages as well and what their concrete performances are.

Action Points. Further secondary studies are required to reason about more general concepts which may be applied for all MT languages or at least for a certain category of MTs such as in-place or out-place transformations. While metamodel-based techniques such as black-box test generation seem reusable out-of-the-box based on standardized languages for metamodeling, transformation-based techniques such as white-box test generation seem more challenging to abstract without having a common transformation formalism. Finally, at least guidelines would be important for

[2]http://sir.csc.ncsu.edu/portal/bios/tcas.php#siemens

[3]https://github.com/rjust/defects4j

the community for making decisions about how to realize a testing and debugging approach for a particular MT language. Our feature model for evaluating existing approaches may be a first starting point, but a more explicit decision model with some reusable technologies would be interesting to speed-up the development of MT testing and debugging frameworks—see also Challenge 10.

Challenge 3: Going beyond Functional Tests. Although the community is starting to pay attention to issues beyond functionality (e.g., [204, 205]), research on functional testing is dominant. However, we also found 5 primary studies [56–58, 86, 140], four from the same group of authors, which consider performance tests. Testing and debugging non-functional properties of MTs may be of importance for future work as the performance of MTs has been identified by the community as one of the reasons preventing the adoption of MT languages [176]. Interestingly, there is not a single work on testing other quality characteristics such as usability or interoperability of MTs. Finally, it is also worth mentioning that although the parallel execution of model transformations is supported by emerging approaches [240, 251], not a single work on testing and debugging considers the parallel execution of MTs, which is in contrast to the general field of software testing [198].

Action Points. Going beyond functional tests for sequential model transformation executions requires novel approaches for MT testing and debugging potentially providing new test model generators—or at least strategies—as well as new adequacy criteria and contract languages. The same is true for locating issues in MTs, here enhanced and integrated tool support is needed, e.g., execution profilers for MT engines when it comes to performance testing.

Challenge 4: Going beyond M2M Transformations. Only 11 of our primary studies address the testing of M2T transformations [14, 33, 42, 66, 91, 96, 113, 117–119, 134], and only one can be applied for T2M transformations [134]. This must be partly due to the assumption that everything is explicitly modeled, i.e., having injector/extractors between models and text-based artefacts—but still, these components would have to be tested as well.

Action Points. The challenge here is not only to provide effective support to testing and debugging M2T and T2M transformations, but also to improve the languages to write these MTs as they are often realized with imperative MT languages such as template-based languages which are consuming or producing simple text. Moreover, more specific MT languages are used in model management such as in the Epsilon language family [144], e.g., for model comparison or merging. This is a clear application niche in which MTs are a piece of a wider and more ambitious architecture where, based on our survey results, testing and debugging is under-explored despite the specific nature of domain-specific MTs may allow for dedicated support due to their specific scope.

Challenge 5: Test Case Generation and Test Process Optimization. Tools that generate input models for MTs are of major importance. There are several approaches for input model generation that manage to cover major parts of the input metamodel and MT. Many of them also accept invariants and pre/postconditions. However, most of them apply constraint solvers to obtain the models, which do not scale well. At the same time, approaches based on search-based techniques mostly depart from already available models. Finally, many model generation approaches are general techniques, even if they are discussed for particular MT types and languages. On the one hand, this allows us to reuse them in many settings, but on the other hand, they are not deeply integrated into the test process, e.g., to optimize the generated models for specific testing objectives.

Action Points. We believe that there is an opportunity for further research related to search-based model generators. With the proliferation of many-objective algorithms [209], the generation of models may be driven by the optimization of many objectives. These could additionally overcome the performance limitations of constraint solvers, e.g., aiming for hybrid model generation approaches, and finally, also deal with test case selection, prioritization, and minimization [198, 261] as known from the general field of software testing, which is also an important future research line to better support regression testing of MTs. Finally, further research may be needed on how to link test case

generation with the test process such as generating only models which find new issues in MTs or, at least, do not produce the same issues again and again.

Challenge 6: Usability and Expressiveness of Contract Languages. As already discussed in a previous study [244], the contract languages used for MTs are of high importance. In our study, we could identify the reuse of already existing languages such as OCL or graph patterns—for instance, 14 primary studies reuse OCL as the main language for defining oracle functions.

Action Points. While the reuse of existing approaches enables the use of expressive languages which may be already supported by different tools, the question is if they provide the right level of usability for testing and debugging of MTs. We could not find empirical studies about the usage of these languages by transformation developers going beyond the research teams proposing the dedicated approaches. Also, the combination of graph patterns and OCL seems important in the future to allow developers to explicitly express certain model patterns as well as to define complex constraints for the oracle functions which may require the usage of a text-based language such as OCL. Further comparison studies are needed to shed more light on these practical aspects.

Challenge 7: From Testing to Debugging and Back Again. Current approaches are either focusing on MT testing or debugging, but the link between these two phases is mostly not explored. How testing may further help locating a bug by generating more specific input models and how debugging may help for regression testing has not been subject for extensive research concerning MTs. For instance, we did not find a single approach for prioritizing bugs in MTs, a topic which clearly falls in the intersection of testing and debugging.

Action Points. Dedicated interfaces between testing and debugging processes may be required to stimulate the information exchange between these two phases. Especially, the combination of spectrum-based approaches and test model generation strategies seem promising to be combined. It may also stimulate further dedicated approaches which go into the direction of automated bug fixing in MTs which currently have been only sparsely considered in this area by one work [35].

Challenge 8: Reusable and Realistic Evaluation Methods. As concluded from Section 7, we consider the evaluation of approaches for MT testing and debugging as another challenge that must be addressed with appropriate methods. In this respect, we have seen that only a few case studies (27 out of 159) come from industry, and most of them are small-scale. Besides, most approaches use only one case study to validate their approach. We consider as an important challenge the use of realistic case studies, instead of small-scale academic examples. Mutation seems to be the major evaluation technique, since at least 17 primary studies apply it in their evaluation [5, 8, 9, 18, 26, 27, 34, 48, 55, 57, 64, 70, 83, 110, 111, 124, 131].

Action Points. We foresee more research on tools for automating the whole evaluation process, i.e., generating mutants and calculating the mutation score of a set of test cases. The mutation operators proposed by Mottu et al. [80] have been used in the evaluation of several approaches. These operators are generic and not described in the context of any particular MT language, so they are often not automated. This means that mutants must be (manually) obtained for specific MT languages. However, mutations may be again considered as MTs for automation purposes. This is provided for ATL by the work of Sánchez-Cuadrado et al. [53, 98–100] which automate the generation of a large set of mutants by using several different mutation operators [70, 80, 120]. Having dedicated guidelines how to conduct such mutations and their usage for evaluation purposes for MT languages in general may be of high interest—especially combined with the availability of the exemplars as discussed in Challenge 1.

Challenge 9: Baseline Technology Infrastructure. The MT community benefits from stable MDE baseline technologies, e.g., see USE and anATLyzer as two very positive examples which are used in several research works. Such tools are of high relevance to bootstrap other techniques for testing and debugging as they provide base support which can be reused. Moreover, this challenge

also relates to baseline support in existing MT engines for testing and debugging. Currently, MT engines are running the whole MTs at once to produce the output models from the input models which is not handy for fine-grained testing, e.g., for testing explicit rules for particular behavior.

Action Points. With our set of primary studies, we see that authors tend to use already available infrastructures. Indeed, some approaches can be helpful to evaluate some other approaches. For instance, it is useful to use MT mutants for evaluating the efficiency of a test model generator. Similarly, an input model generator is very important for evaluating test oracles. Therefore, making mature tools available and sharing them with other researchers and practitioners of MT testing and debugging is another challenge ahead. Furthermore, having open tools which employ some standards for representing input and output artefacts are beneficial, e.g., for building bridges between different tools and formats required for covering larger portions of the testing and debugging processes. Finally, extending existing MT engines with dedicated interfaces for testing and debugging is a must in order to support more fine-grained capabilities in testing and debugging frameworks for MTs in the future, see the following challenge.

Challenge 10: Comprehensive Testing and Debugging Frameworks. There is a lack of actual automated testing and debugging frameworks for MT as they are available for general-purpose programming languages. Such frameworks should define a simple way to write test cases, run them, and generate test results reports. For instance, JUnit can be easily integrated with any approach for the generation of test cases or test oracles. It would be important to have similar frameworks in the context of MTs, perhaps extending JUnit (as REST Assured[4] in the context of Web APIs).

Action Points. A future line which seems interesting to explore is to reuse ideas from executable modeling language engineering which provide not only the execution engine for a given language but also additional tools such as a debugger, logger, etc. Based on such meta-frameworks, MT languages may be recreated in order to provide out-of-the-box tool support for testing and debugging as well as dedicated interfaces for additional tool support [147, 173, 219].

Challenge 11: MT Testing and Debugging Unit. Most of the existing approaches surveyed in this paper consider the full transformation as one unit when it comes to testing and debugging. For instance, oracles are developed for the full transformation independently from the transformation implementation. As a consequence, additional approaches for finding the links between oracles and transformation rules are required when it comes to debugging. We only found very few approaches for testing transformations which deal with unit tests on the transformation rule level. A few other approaches consider integration tests of full transformation units when running a chain of transformations, i.e., a sequence of different transformations which are feeding each other, which is a more course-grained understanding of the unit concept.

Action Points. For the future, language extensions for developing more fine-grained unit tests for MTs are a promising target. In addition, best practices to test rules in isolation and having integration tests for different rules up to systems tests considering transformation chains are needed. Such best practices in combination with enhanced MT engines, cf. Challenge 9, allow for additional testing strategies known from software testing going beyond testing for success, i.e., intended output has been produced for valid input. For instance, testing for failure is currently only supported by checking contracts before running a transformation as an additional step, but these contracts may be injected to the transformation implementation as assertions to have them manifested in the transformation. Moreover, ideas from bi-directional transformations [248] may be employed even for uni-directional transformations to support testing for sanity. Currently, developing complete bi-directional transformations is considered expensive, but for certain rules

[4]https://rest-assured.io/

which are considered important it may be affordable to define a bi-directional contract which can be used to check a forward transformation rule as well as a backward transformation rule.

9 CONCLUSION

In this paper, we have surveyed the state of the art in MT testing and debugging. We have studied 140 primary studies, classified them, and investigated their experimental evaluation methods and subjects. When revisiting the challenges outlined more than a decade ago [168], we can conclude that there has been much progress on generating test input models, dedicated oracle languages for MTs, and also some progress on test adequacy criteria. However, we also identified challenges which still have to be tackled in the future, e.g., tool support in testing and debugging is still scattered and mostly simplistic case studies are used for evaluations.

In order to reach the next generation of testing and debugging tools for MTs, we have identified several promising research lines. For instance, we believe it is interesting to count on exemplars for MT testing and debugging with dedicated packages available on open repositories. It is also important to develop techniques applicable to different MT languages, to go beyond M2M transformations and to consider non-functional properties. MT testing and debugging tools need to be more powerful, i.e., they should allow developers to move from testing to debugging and back again, provide more support for test case generation and prioritization, and be able to reuse realistic evaluation methods. Finally, additional studies to contrast the state of the art in testing and debugging of MTs with the state of the art in the general field of software testing is another interesting follow-up study potentially based on the corpus provided by the study presented in this paper.

PRIMARY STUDIES

[1] A. S. Al-Sibahi, A. S. Dimovski, and A. Wąsowski. 2016. Symbolic Execution of High-level Transformations. In *Proc. of SLE*. 207–220.

[2] B. Alkhazi, C. Abid, M. Kessentini, D. Leroy, and M. Wimmer. 2020. Multi-criteria test cases selection for model transformations. *Automated Software Engineering* (2020), 91–118.

[3] J. M. Almendros-Jiménez and A. Becerra-Terón. 2016. Automatic Generation of Ecore Models for Testing ATL Transformations. In *Proc. of MEDI*. 16–30.

[4] K. Anastasakis, B. Bordbar, and J. Küster. 2007. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVA*. 47–56.

[5] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser. 2015. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing Verification and Reliability* (2015), 653–683.

[6] V. Aranega, J.-M. Mottu, A. Etien, and J. Dekeyser. 2009. Traceability mechanism for error localization in model transformation. In *Proc. of ICSOFT*. 66–73.

[7] V. Aranega, J.-M. Mottu, A. Etien, and J. Dekeyser. 2009. Using Trace to Situate Errors in Model Transformations. In *Proc. of ICSOFT*. 137–149.

[8] V. Aranega, J.-M. Mottu, A. Etien, and J. Dekeyser. 2010. Using Traceability to Enhance Mutation Analysis Dedicated to Model Transformation. In *Proc. of MoDeVVA*. 1–6.

[9] V. Aranega, J.-M. Mottu, A. Etien, and J. Dekeyser. 2011. Traceability for Mutation Analysis in Model Transformation. In *Proc. of MODELS*. 259–273.

[10] S. Arifulina, C. Soltenborn, and G. Engels. 2012. Coverage Criteria for Testing DMM Specifications. In *Proc. of GTVMT*.

[11] E. Batot and H. Sahraoui. 2016. A generic framework for model-set selection for the unification of testing and learning MDE tasks. In *Proc. of MODELS*. 374–384.

[12] E. Bauer and J. M. Küster. 2011. Combining Specification-Based and Code-Based Coverage for Model Transformation Chains. In *Proc. of ICMT*. 78–92.

[13] E. Bauer, J. M. Küster, and G. Engels. 2011. Test Suite Quality for Model Transformation Chains. In *Proc. of TOOLS*. 3–19.

[14] S. Bonfanti, A. Gargantini, and A. Mashkoor. 2020. Design and validation of a C++ code generator from Abstract State Machines specifications. *Journal of Software: Evolution and Process* 32 (2020), 1–27. Issue 2.

[15] C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim. 2011. On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts. In *Proc. of SBMF*. 108–123.

[16] C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim. 2012. Transformation contracts in practice. *IET Software* (2012), 16–32.

[17] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. 2006. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *Proc. of ISSRE*. 85–94.
[18] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. 2015. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering* (2015), 490–506.
[19] L. Burgueño, M. Wimmer, and A. Vallecillo. 2012. Towards Tracking "Guilty" Transformation Rules: A Requirements Perspective. In *Proc. of AMT*. 27–32.
[20] L. Burgueño, J. Cabot, R. Clarisó, and M. Gogolla. 2019. A Systematic Approach to Generate Diverse Instantiations for Conceptual Schemas. In *Proc. of ER*. 513–521.
[21] L. Burgueño, F. Hilken, A. Vallecillo, and M. Gogolla. 2016. Generating effective test suites for model transformations using classifying terms. In *Proc. of PAME/VOLT*. 48–57.
[22] L. Burgueño, F. Hilken, A. Vallecillo, and M. Gogolla. 2017. Testing Transformation Models Using Classifying Terms. In *Proc. of ICMT*. 69–85.
[23] D. Calegari and A. Delgado. 2013. Rule Chains Coverage for Testing QVT-Relations Transformations. In *Proc. of AMT*. 1–10.
[24] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. 2009. OCL contracts for the verification of model transformations. In *Proc. of OCL*, Vol. 24. 1–15.
[25] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. 2004. OCL for the Specification of Model Transformation Contracts. In *Proc. of OCL-MDE*. 1–15.
[26] Z. Cheng and M. Tisi. 2017. A Deductive Approach for Fault Localization in ATL Model Transformations. In *Proc. of FASE*. 300–317.
[27] Z. Cheng and M. Tisi. 2018. Slicing ATL model transformations for scalable deductive verification and fault localization. *International Journal on Software Tools for Technology Transfer* (2018), 645–663.
[28] A. Ciancone, A. Filieri, and R. Mirandola. 2010. MANTra: Towards Model Transformation Testing. In *Proc. of QUATIC*. 97–105.
[29] A. Ciancone, A. Filieri, and R. Mirandola. 2014. Testing operational transformations in model-driven engineering. *Innovations in Systems and Software Engineering* (2014), 19–32.
[30] J. Corley. 2014. Exploring Omniscient Debugging for Model Transformations. In *Proc. of PSRC*. 63–68.
[31] J. Corley, B. P. Eddy, E. Syriani, and J. Gray. 2017. Efficient and scalable omniscient debugging for model transformations. *Software Quality Journal* (2017), 7–48.
[32] A. Darabos, A. Pataricza, and D. Varró. 2008. Towards Testing the Implementation of Graph Transformations. In *Proc. of GT-VMT*. 75–85.
[33] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. 2010. Debugging Model-Transformation Failures Using Dynamic Tainting. In *Proc. of ECOOP*. 26–51.
[34] K. Du, M. Jiang, Z. Ding, H. Huang, and T. Shu. 2020. Metamorphic testing in fault localization of model transformations. In *Proc. of SOFL+MSVL*. 299–314.
[35] F. Ege and M. Tichy. 2019. A Proposal of Features to Support Analysis and Debugging of Declarative Model Transformations with Graphical Syntax by Embedded Visualizations. In *Proc. of MODELS Companion*. 326–330.
[36] K. Ehrig, J. M. Kuester, and G. Taentzer. 2009. Generating instance models from meta models. *Software and Systems Modeling* (2009), 479–500.
[37] K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann. 2006. Generating Instance Models from Meta Models. In *Proc. of FMOODS*. 156–170.
[38] O. Finot, J.-M. Mottu, G. Sunyé, and C. Attiogbé. 2013. Partial Test Oracle in Model Transformation Testing. In *Proc. of ICMT*. 189–204.
[39] O. Finot, J.-M. Mottu, G. Sunyé, and T. Degueule. 2013. Using meta-model coverage to qualify test oracles. In *Proc. of AMT*. 1–10.
[40] C. Fiorentini, A. Momigliano, M. Ornaghi, and I. Poernomo. 2010. A Constructive Approach to Testing Model Transformations. In *Proc. of ICMT*. 77–92.
[41] F. Fleurey, B. Baudry, P. Muller, and Y. Le Traon. 2009. Qualifying input test data for model transformations. *Software and Systems Modeling* (2009), 185–203.
[42] J. Garcia, M. Azanza, A. Irastorza, and O. Diaz. 2014. Testing MOFScript Transformations with HandyMOF. In *Proc. of ICMT*. 42–56.
[43] M. Gogolla, J. Bohling, and M. Richters. 2005. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling* (2005), 386–398.
[44] M. Gogolla and A. Vallecillo. 2011. Tractable Model Transformation Testing. In *Proc. of ECMFA*. 221–235.
[45] M. Gogolla, A. Vallecillo, L. Burgueño, and F. Hilken. 2017. Employing Classifying Terms for Testing Model Transformations. In *Proc. of MODELS*. 312–321.

[46] P. Gómez-Abajo, E. Guerra, and J. de Lara. 2016. Wodel: A Domain-Specific Language for Model Mutation. In *Proc. of SAC*. 1968–1973.
[47] C. A. González and J. Cabot. 2012. ATLTest: A White-box Test Generation Approach for ATL Transformations. In *Proc. of MODELS*. 449–464.
[48] E. Guerra. 2012. Specification-Driven Test Generation for Model Transformations. In *Proc. of ICMT*. 40–55.
[49] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. 2010. A Visual Specification Language for Model-to-Model Transformations. In *Proc. of VLHCC*. 119–126.
[50] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. 2010. transML: A Family of Languages to Model Model Transformations. In *Proc. of MODELS*. 106–120.
[51] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. 2013. Automated verification of model transformations based on visual contracts. *Automated Software Engineering* (2013), 5–46.
[52] E. Guerra and M. Soeken. 2015. Specification-driven model transformation testing. *Software and Systems Modeling* (2015), 623–644.
[53] E. Guerra, J. Sánchez-Cuadrado, and J. de Lara. 2019. Towards Effective Mutation Testing for ATL. In *Proc. of MODELS*. 78–88.
[54] P. Gómez-Abajo, E. Guerra, and J. de Lara. 2017. A Domain-specific Language for Model Mutation and Its Application to the Automated Generation of Exercises. *Computer Languages, Systems & Structures* (2017), 152–173.
[55] X. He, X. Chen, S. Cai, Y. Zhang, and G. Huang. 2018. Testing bidirectional model transformation using metamorphic testing. *Information and Software Technology* (2018), 109–129.
[56] X. He, W. Li, T. Zhang, and Y. Liu. 2016. Towards Parallel Model Generation for Random Performance Testing of Model-Oriented Operations. In *Proc. of TASE*. 57–64.
[57] X. He, T. Zhang, C. Hu, Z. Ma, and W. Shao. 2016. An MDE performance testing framework based on random model generation. *Journal of Systems and Software* (2016), 247 – 264.
[58] X. He, T. Zhang, M. Pan, Z. Ma, and C. Hu. 2019. Template-based Model Generation. *Software and Systems Modeling* (2019), 2051–2092.
[59] R. Heckel, T. A. Khan, and R. Machado. 2011. Towards Test Coverage Criteria for Visual Contracts. In *Proc. of GTVMT*.
[60] M. Hibberd, M. Lawley, and K. Raymond. 2007. Forensic Debugging of Model Transformations. In *Proc. of MODELS*. 589–604.
[61] F. Hilken, M. Gogolla, L. Burgueño, and A. Vallecillo. 2018. Testing models and model transformations using classifying terms. *Software and Systems Modeling* (2018), 885–912.
[62] C. T. M. Hue, D. D. Hanh, and N. N. Binh. 2018. A Transformation-Based Method for Test Case Automatic Generation from Use Cases. In *Proc. of KSE*. 252–257.
[63] S. Jahanbin and B. Zamani. 2018. Test Model Generation Using Equivalence Partitioning. In *Proc. of ICCKE*. 98–103.
[64] M. Jiang, T. Y. Chen, F. Kuo, Z. Zhou, and Z. Ding. 2014. Testing Model Transformation Programs using Metamorphic Testing. In *Proc. of SEKE*. 94–99.
[65] A. A. Jilani, M. Z. Iqbal, and M. U. Khan. 2014. A Search Based Test Data Generation Approach for Model Transformations. In *Proc. of ICMT*. 17–24.
[66] S. Jörges and B. Steffen. 2014. Back-To-Back Testing of Model-Based Code Generators. In *Proc. of ISoLA*. 425–444.
[67] M. Jukss, C. Verbrugge, and H. Vangheluwe. 2017. Transformations Debugging Transformations. In *Proc. of MODELS*. 449–454.
[68] M. Kessentini, H. Sahraoui, and M. Boukadoum. 2010. Sequence Diagram to Colored Petri Nets Transformation Testing: An Immune System Metaphor. In *Proc. of CASCON*. 72–85.
[69] M. Kessentini, H. Sahraoui, and M. Boukadoum. 2011. Example-based model-transformation testing. *Automated Software Engineering* (2011), 199–224.
[70] Y. Khan and J. Hassine. 2013. Mutation Operators for the Atlas Transformation Language. In *Proc. of ICSTW Workshops*. 43–52.
[71] D. S. Kolovos. 2009. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proc. of ECMDA-FA*. 146–157.
[72] M. Lahrouni, E. Cariou, and A. E. Fazziki. 2019. A black-box and contract-based verification of model transformations. *The International Arab Journal of Information Technology* (2019), 651–660.
[73] M. Lamari. 2007. Towards an Automated Test Generation for the Verification of Model Transformations. In *Proc. of SAC*. 998–1005.
[74] L. Lengyel and H. Charaf. 2015. Test-driven verification/validation of model transformations. *Frontiers of Information Technology and Electronic Engineering* (2015), 85–97.
[75] P. Li, M. Jiang, and Z. Ding. 2020. Fault Localization With Weighted Test Model in Model Transformations. *IEEE Access* (2020), 14054–14064.

[76] Y. Lin, J. Zhang, and J. Gray. 2005. A testing framework for model transformations. In *Model Driven Software Development*. 219–236.
[77] N. D. Matragkas, D. S. Kolovos, R. F. Paige, and A. Zolotas. 2013. A Traceability-Driven Approach to Model Transformation Testing. In *Proc. of AMT*. 1–10.
[78] S. Mazanek and C. Rutetzki. 2011. On the importance of model comparison tools for the automatic evaluation of the correctness of model transformations. In *Proc. of IWMCP*. 12–15.
[79] J. A. McQuillan and J. Power. 2009. White-Box Coverage Criteria for Model Transformations. In *Proc. of AMT*. 63–77.
[80] J.-M. Mottu, B. Baudry, and Y. Le Traon. 2006. Mutation Analysis Testing for Model Transformations. In *Proc. of ECMDA-FA*. 376–390.
[81] J.-M. Mottu, B. Baudry, and Y. L. Traon. 2008. Model transformation testing: oracle issue. In *Proc. of MoDeVVA*. 105–112.
[82] J.-M. Mottu, S. Sen, J. Cadavid, and B. Baudry. 2015. Discovering model transformation pre-conditions using automatically generated test models. In *Proc. of ISSRE*. 88–99.
[83] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. 2012. Static Analysis of Model Transformations for Effective Test Generation. In *Proc. of ISSRE*. 291–300.
[84] T. Mészáros, P. Fehér, and L. Lengyel. 2013. Visual debugging support for graph rewriting-based model transformations. In *Proc. of EUROCON*. 482–488.
[85] A. Narayanan and G. Karsai. 2008. Verifying Model Transformations by Structural Correspondence. In *Proc. of GT-VMT*, Vol. 10. 1–15.
[86] N. Nassar, J. Kosiol, T. Kehrer, and G. Taentzer. 2020. Generating large EMF models efficiently: A rule-based, configurable approach. In *Proc. of FASE*. 224–244.
[87] T. Nguyen and D. Dang. 2018. An approach for testing model transformations. In *Proc. of KSE*. 264–269.
[88] T. H. Nguyen, D. H. Dang, and Q. T. Nguyen. 2019. On Analyzing Rule-Dependencies to Generate Test Cases for Model Transformations. In *Proc. of KSE*. 181–186.
[89] B. J. Oakes, L. Lucio, C. Verbrugge, and H. Vangheluwe. 2018. Debugging of Model Transformations and Contracts in SyVOLT. In *Proc. of MODELS Workshops*. 532–537.
[90] S. Popoola, D. S. Kolovos, and H. H. Rodriguez. 2016. EMG: A Domain-Specific Transformation Language for Synthetic Model Generation. In *Proc. of ICMT*. 36–51.
[91] A. C. Rajeev, P. Sampath, K. C. Shashidhar, and S. Ramesh. 2010. CoGenTe: A Tool for Code Generator Testing. In *Proc. of ASE*. 349–350.
[92] R. Rodríguez-Echeverría, F. Macías, and A. Rutle. 2016. On reducing model transformation testing overhead. In *Proc. of PAME/VOLT*. 58–67.
[93] L. M. Rose and S. Poulding. 2013. Efficient probabilistic testing of model transformations using search. In *Proc. of CMSBSE*. 16–21.
[94] O. Runge, T. A. Khan, and R. Heckel. 2013. Test Case Generation Using Visual Contracts. In *Proc. of GTVMT*.
[95] D. Sahin, M. Kessentini, M. Wimmer, and K. Deb. 2015. Model transformation testing: A bi-level search-based software engineering approach. *Journal of Software: Evolution and Process* (2015), 821–837.
[96] P. Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. 2008. Behaviour Directed Testing of Auto-code Generators. In *Proc. of SEFM*. 191–200.
[97] J. Sánchez-Cuadrado. 2020. Towards Interactive, Test-driven Development of Model Transformations. *Journal of Object Technology* 19, 3 (2020), 1–12.
[98] J. Sánchez-Cuadrado, E. Guerra, and J. de Lara. 2015. Quick fixing ATL model transformations. In *Proc. of MODELS*. 146–155.
[99] J. Sánchez-Cuadrado, E. Guerra, and J. de Lara. 2017. Static Analysis of Model Transformations. *IEEE Transactions on Software Engineering* (2017), 868–897.
[100] J. Sánchez-Cuadrado, E. Guerra, and J. de Lara. 2018. Quick fixing ATL transformations with speculative analysis. *Software and Systems Modeling* (2018), 779–813.
[101] M. Scheidgen. 2015. Generation of Large Random Models for Benchmarking. In *Proc. of BigMDE*. 1–10.
[102] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer. 2010. Catch Me if You Can – Debugging Support for Model Transformations. In *Proc. of MODELS*. 5–20.
[103] J. Schonbock, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger. 2013. TETRABox - A Generic White-Box Testing Framework for Model Transformations. In *Proc. of APSEC*. 75–82.
[104] J. Schönböck, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger, and W. Schwinger. 2012. Debugging Model-to-Model Transformations. In *Proc. of APSEC*. 164–173.
[105] G. M. K. Selim, F. Büttner, J. R. Cordy, J. Dingel, and S. Wang. 2013. Automated Verification of Model Transformations in the Automotive Industry. In *Proc. of MODELS*. 690–706.

[106] O. Semeráth, A. A. Babikian, A. Li, K. Marussy, and D. Varró. 2020. Automated Generation of Consistent Models with Structural and Attribute Constraints. In *Proc. of MODELS*. 187–199.
[107] O. Semeráth, A. S. Nagy, and D. Varró. 2018. A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. In *Proc. of ICSE*. 969–980.
[108] S. Sen and B. Baudry. 2006. Mutation-based Model Synthesis in Model Driven Engineering. In *Proc. of Mutation Workshop*. 1–10.
[109] S. Sen, B. Baudry, and J.-M. Mottu. 2008. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. In *Proc. of ICSE*. 328–337.
[110] S. Sen, B. Baudry, and J.-M. Mottu. 2009. Automatic Model Generation Strategies for Model Transformation Testing. In *Proc. of ICMT*. 148–164.
[111] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. 2012. Using models of partial knowledge to test model transformations. In *Proc. of ICMT*. 24–39.
[112] J. Shelburg, M. Kessentini, and D. R. Tauritz. 2013. Regression Testing for Model Transformations: A Multi-objective Approach. In *Proc. of SSBSE*. 209–223.
[113] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper. 2007. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering* 33, 9 (2007), 622–634.
[114] J. Sánchez-Cuadrado, E. Guerra, and J. de Lara. 2014. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *Proc. of ISSRE*. 34–44.
[115] J. Sánchez-Cuadrado, E. Guerra, and J. de Lara. 2018. AnATLyzer: An Advanced IDE for ATL Model Transformations. In *Proc. of ICSE Companion*. 85–88.
[116] J. Sánchez-Cuadrado, E. Guerra, J. de Lara, R. Clarisó, and J. Cabot. 2017. Translating Target to Source Constraints in Model-to-Model Transformations. In *Proc. of MODELS*. 12–22.
[117] A. Tiso, G. Reggio, and M. Leotta. 2012. Early Experiences on Model Transformation Testing. In *Proc. of AMT*. 15–20.
[118] A. Tiso, G. Reggio, and M. Leotta. 2013. A Method for Testing Model to Text Transformations. In *Proc. of AMT*, Vol. 1077. 1–10.
[119] A. Tiso, G. Reggio, and M. Leotta. 2014. Unit Testing of Model to Text Transformations. In *Proc. of AMT*. 14–23.
[120] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer. 2015. Towards systematic mutations for and with ATL model transformations. In *Proc. of ICSTW Worshops*. 1–10.
[121] J. Troya, S. Segura, J. Parejo, and A. Ruiz-Cortés. 2017. An approach for debugging model transformations applying spectrum-based fault localization. In *Proc. of JISBD*. 1–4.
[122] J. Troya, S. Segura, J. Parejo, and A. Ruiz-Cortés. 2018. Spectrum-based fault localization in model transformations. *ACM Transactions on Software Engineering and Methodology* (2018), 1–50.
[123] J. Troya, S. Segura, and A. Ruiz-Cortés. 2016. Towards the automation of metamorphic testing in model transformations. In *Proc. of JISBD*. 281–284.
[124] J. Troya, S. Segura, and A. Ruiz-Cortés. 2018. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software* (2018), 188–208.
[125] Z. Ujhelyi, Horváth, and D. Varró. 2011. Towards dynamic backward slicing of model transformations. In *Proc. of ASE*. 404–407.
[126] Z. Ujhelyi, Horváth, and D. Varró. 2012. Dynamic Backward Slicing of Model Transformations. In *Proc. of ICST*. 1–10.
[127] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. 2012. Formal Specification and Testing of Model Transformations. In *Proc. of SMF*. 399–437.
[128] J. Wang, S. Kim, and D. Carrington. 2006. Verifying metamodel coverage of model transformations. In *Proc. of ASWEC*. 270–282.
[129] J. Wang, S. Kim, and D. Carrington. 2008. Automatic Generation of Test Models for Model Transformations. In *Proc. of ASWEC*. 432–440.
[130] W. Wang, M. Kessentini, and W. Jiang. 2013. Test cases generation for model transformations from structural information. In *Proc. of MDEBE*. 42–51.
[131] M. Wieber, A. Anjorin, and A. Schürr. 2014. On the Usage of TGGs for Automated Model Transformation Testing. In *Proc. of ICMT*. 1–16.
[132] M. Wieber and A. Schürr. 2012. Gray Box Coverage Criteria for Testing Graph Pattern Matching. In *Proc. of GraBaTs*, Vol. 52. 1–12.
[133] M. Wieber and A. Schürr. 2013. Systematic Testing of Graph Transformations: A Practical Approach Based on Graph Patterns. In *Proc. of ICMT*. 205–220.
[134] M. Wimmer and L. Burgueño. 2013. Testing M2T/T2M Transformations. In *Proc. of MODELS*. 203–219.
[135] M. Wimmer, G. Kappel, J. Schoenboeck, A. Kusel, W. Retschitzegger, and W. Schwinger. 2009. A Petri Net Based Debugging Environment for QVT Relations. In *Proc. of ASE*. 3–14.

[136] M. Wimmer, A. Kusel, T. Reiter, W. Retschitzegger, W. Schwinger, and G. Kappel. 2009. Lost in Translation? Transformation Nets to the Rescue!. In *Proc. of UNISCON*. 315–327.
[137] M. Wimmer, A. Kusel, J. Schoenboeck, G. Kappel, W. Retschitzegger, and W. Schwinger. 2009. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets. In *Proc. of MODELS*. 727–732.
[138] H. Wu. 2016. Generating metamodel instances satisfying coverage criteria via SMT solving. In *Proc. of MODELSWARD*. 40–51.
[139] H. Wu, R. Monahan, and J. F. Power. 2013. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *Proc. of TASE*. 175–182.
[140] H. Xiao, Z. Tian, M. Zhiyi, and S. Weizhong. 2014. Randomized Model Generation for Performance Testing of Model Transformations. In *Proc. of COMPSAC*. 11–20.

REFERENCES

[141] [n.d.]. AnATLyzer. https://anatlyzer.github.io/. Accessed February 2022.
[142] [n.d.]. ATL Zoo. http://www.eclipse.org/atl/atlTransformations. Accessed May 2021.
[143] [n.d.]. DSLTrans. https://github.com/mbeddr/language_verification. Accessed February 2022.
[144] [n.d.]. Eclipse Epsilon. https://www.eclipse.org/epsilon/. Accessed February 2022.
[145] [n.d.]. EMF Model Generator. https://github.com/RuleBasedApproach/EMFModelGenerator. Accessed February 2022.
[146] [n.d.]. EMFtoCSP. https://github.com/atlanmod/EMFtoCSP. Accessed February 2022.
[147] [n.d.]. GEMOC Initiative. http://gemoc.org/. Accessed May 2021.
[148] [n.d.]. MDE Testing. https://github.com/jdelara/MDETesting. Accessed February 2022.
[149] [n.d.]. MRs4MTgenerator. https://gestionproyectos.us.es/projects/curso-ice-2016-rest-fp-coordinacioo/wiki. Accessed February 2022.
[150] [n.d.]. PaMoMo. http://miso.es/tools/transML/main.htm. Accessed February 2022.
[151] [n.d.]. PRAMANA. https://www.irisa.fr/triskell/Software/pramana/index.html. Accessed February 2022.
[152] [n.d.]. RandomEMF. http://github.com/markus1978/RandomEMF. Accessed February 2022.
[153] [n.d.]. SBFL_MT. https://github.com/javitroya/SBFL_MT. Accessed February 2022.
[154] [n.d.]. SymexTRON. https://github.com/models-team/SymexTRON. Accessed February 2022.
[155] [n.d.]. TracsTool. http://atenea.lcc.uma.es/projects/FaultLocMT.html. Accessed February 2022.
[156] [n.d.]. Unnamed tool. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Mutations. Accessed February 2022.
[157] [n.d.]. Unnamed tool. https://github.com/SOM-Research/constraint-mutation. Accessed February 2022.
[158] [n.d.]. Unnamed tool. https://bitbucket.org/ustbmde/model-generation/wiki/Home. Accessed February 2022.
[159] [n.d.]. USE. https://sourceforge.net/projects/useocl/. Accessed February 2022.
[160] [n.d.]. VIATRA Generator. https://github.com/viatra/VIATRA-Generator. Accessed February 2022.
[161] [n.d.]. WODEL. http://gomezabajo.github.io/Wodel/. Accessed February 2022.
[162] A. Abade., F. Ferrari., and D. Lucrédio. 2015. Testing M2T Transformations - A Systematic Literature Review. In *Proc. of ICEIS*. 177–187.
[163] L. Addazi, A. Cicchetti, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. 2016. Semantic-based Model Matching with EMFCompare. In *Proc. of ME*. 40–49.
[164] M. Amrani, B. Combemale, L. Lúcio, G. Selim, J. Dingel, Y. Le Traon, H. Vangheluwe, and J. R. Cordy. 2015. Formal Verification Techniques for Model Transformations: A Tridimensional Classification. *JOT Journal* (2015), 1:1–43.
[165] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. 2010. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. of MODELS*. 121–135.
[166] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.
[167] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. 2006. Model Transformation Testing Challenges. In *Proc. of IMDT*. 1–10.
[168] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. 2010. Barriers to Systematic Model Transformation Testing. *Commun. ACM* (2010), 139–143.
[169] D. Benavides, S. Segura, and A. R. Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.
[170] H. Benouda, M. Azizi, R. Esbai, and M. Moussaoui. 2016. Code generation approach for mobile application using acceleo. *International Review on Computers and Software* (2016), 160–166.
[171] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. 2005. Model Transformations in Practice Workshop. In *Proc. of MODELS Satellite Events*. 120–127.
[172] K. Birken. 2014. Building Code Generators for DSLs Using a Partial Evaluator for the Xtend Language. In *Proc. of ISOLA*. 407–424.
[173] E. Bousse, M. Wimmer, W. Schwinger, and E. Kapsammer. 2016. On Leveraging Executable Language Engineering for Domain-Specific Transformation Languages. In *Proc. of EM*. 41–43.

[174] M. Brambilla, J. Cabot, and M. Wimmer. 2017. *Model-Driven Software Engineering in Practice (2nd edition).*
[175] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. 2010. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proc. of ASE*. 173–174.
[176] L. Burgueño, J. Cabot, and S. Gérard. 2019. The Future of Model Transformation Languages: An Open Community Discussion. *Journal of Object Technology* 7 (2019), 1–11.
[177] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. 2012. Verification of ATL Transformations Using Transformation Models and Model Finders. In *Proc. of ICFEM*. 198–213.
[178] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. 2005. Model Transformations in Practice Workshop. In *Proc. of MODELS Satellite Events*. 120–127.
[179] J. Cabot, R. Clarisó, and D. Riera. 2014. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software* 93 (2014), 1–23.
[180] J. Cabot and M. Gogolla. 2012. Object Constraint Language (OCL): A Definitive Guide. In *Proc. of SFM*. 58–90.
[181] J. J. Cadavid, B. Combemale, and B. Baudry. 2015. An Analysis of Metamodeling Practices for MOF and OCL. *Computer Languages, Systems and Structures* (2015), 42–65.
[182] D. Calegari and N. Szasz. 2013. Verification of Model Transformations: A Survey of the State-of-the-Art. *Electronic Notes in Theoretical Computer Science* 292 (2013), 5–25.
[183] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36.
[184] T. Y. Chen, F. Kuo, H. Liu, P. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1 (2018), 4:1–4:27.
[185] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. 2011. JTL: A Bidirectional and Change Propagating Transformation Language. In *Proc. of SLE*. 183–202.
[186] F. Ciccozzi, I. Malavolta, and B. Selic. 2019. Execution of UML models: a systematic review of research and practice. *Software and Systems Modeling* (2019), 2313–2360.
[187] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. 2007. *All About Maude – A High-Performance Logical Framework.*
[188] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. 2002. VIATRA - visual automated transformations for formal verification and validation of UML models. In *Proc. of ASE*. 267–270.
[189] K. Czarnecki and S. Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal* (2006), 621–646.
[190] A. R. da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* (2015), 139–155.
[191] J. de Lara and H. Vangheluwe. 2002. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proc. of FASE*. Vol. 2306. 174–188.
[192] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of TACAS*. 337–340.
[193] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* (2002), 182–197.
[194] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. 2005. Termination Criteria for Model Transformation. In *Proc. of FASE*. 49–63.
[195] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (Eds.). 1999. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools.*
[196] M. Fleck, J. Troya, and M. Wimmer. 2015. Marrying Search-based Optimization and Model Transformation Technology. In *Proc. of NasBASE*. 1–16.
[197] M. Fleck, J. Troya, and M. Wimmer. 2016. Search-Based Model Transformations. *Journal of Software: Evolution and Process* (2016), 1081–1117.
[198] G. Fraser and J. M. Rojas. 2019. Software Testing. In *Handbook of Software Engineering*. 123–192.
[199] L. Gammaitoni, P. Kelsen, and Q. Ma. 2018. Agile validation of model transformations using compound F-Alloy specifications. *Science of Computer Programming* (2018), 55–75.
[200] V. Garousi, M. Felderer, and M. V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Inf. Softw. Technol.* 106 (2019), 101–121.
[201] M. Gogolla, F. Büttner, and M. Richters. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming* (2007), 27–34.
[202] C. A. González and J. Cabot. 2014. Formal verification of static software models in MDE: A systematic review. *Information and Software Technology* (2014), 821–838.
[203] J. Greenyer and E. Kindler. 2010. Comparing relational model transformation technologies: implementing Query/View/-Transformation with Triple Graph Grammars. *Software and Systems Modeling* (2010), 21–46.

[204] R. Groner, L. Beaucamp, M. Tichy, and S. Becker. 2020. An Exploratory Study on Performance Engineering in Model Transformations. In *Proc. of MODELS*. 308–319.
[205] R. Groner, S. Gylstorff, and M. Tichy. 2020. *A Profiler for the Matching Process of Henshin*.
[206] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. [n.d.]. A Visual Specification Language for Model-to-Model Transformations. In *Proc. of VLHCC*. 119–126.
[207] R. Heckel and G. Taentzer. 2020. *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*.
[208] A. Hettab, E. Kerkouche, and A. Chaoui. 2015. A Graph Transformation Approach for Automatic Test Cases Generation from UML Activity Diagrams. In *Proc. of C3S2E*. 88–97.
[209] H. Ishibuchi, N. Tsukamoto, and Y. Nojima. 2008. Evolutionary many-objective optimization. In *Proc. of GEFS*. 47–52.
[210] ISO 29119-4 2015. *ISO/IEC/IEEE International Standard - Software and systems engineering–Software testing–Part 1: Concepts and Definitions*. Standard.
[211] ISO/IEC/IEEE. 2013. ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)* (2013), 1–64. https://doi.org/10.1109/IEEESTD.2013.6588537
[212] D. Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* (2002), 256–290.
[213] A. Z. Javed, P. A. Strooper, and G. N. Watson. 2007. Automated Generation of Test Cases Using Model-Driven Architecture. In *Proc. of AST*. 3–3.
[214] J. Jézéquel, O. Barais, and F. Fleurey. 2011. Model Driven Language Engineering with Kermeta. In *Proc. of GTTSE*. 201–221.
[215] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* (2011), 649–678.
[216] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. 2006. ATL: A QVT-like Transformation Language. In *Proc. of OOPSLA Companion*. 719–720.
[217] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró. 2019. Survey and classification of model transformation tools. *Software and Systems Modeling* (2019), 2361–2397.
[218] A. Kalaee and V. Rafe. 2019. Model-based test suite generation for graph transformation system using model simulation and search-based techniques. *Information and Software Technology* (2019), 1–29.
[219] F. Khorram, E. Bousse, J. Mottu, and G. Sunyé. 2021. Adapting TDL to Provide Testing Support for Executable DSLs. *J. Object Technol.* 20, 3 (2021), 6:1–15.
[220] B. Kitchenham. 2004. Procedures for Performing Systematic Reviews. Joint Technical Report, Keele University TR/SE-0401 and NICTA 0400011T.1.
[221] A. Kleppe, J. Warmer, and S. Cook. 1999. Informal Formality? The Object Constraint Language and Its Application in the UML Metamodel. In *Proc. of UML*. 148–161.
[222] T. Kühne. 2006. Matters of (Meta-) Modeling. *Software and Systems Modeling* (2006), 369–385.
[223] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf. 2018. A survey of model transformation design patterns in practice. *Journal of Systems and Software* (2018), 48–73.
[224] M. Lawley, K. Duddy, A. Gerber, and K. Raymond. 2004. Language features for re-use and maintainability of MDA transformations. In *Proc. of OOPSLA Workshops*.
[225] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer. 2014. Model Transformation Intents and Their Properties. *Software and Systems Modeling* (2014), 1–35.
[226] J. Ludewig. 2003. Models in software engineering – an introduction. *Software and Systems Modeling* (2003), 5–14.
[227] R. Mannadiar. 2012. *A multi-paradigm modelling approach to the foundations of domain-specific modelling*. Ph.D. Dissertation. MgGill University.
[228] P. McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
[229] S. J. Mellor, K. Scott, A. Uhl, D. Weise, and R. M. Soley. 2004. *MDA distilled: principles of model-driven architecture*.
[230] T. Mens and P. V. Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* (2006), 125–142.
[231] B. J. Oakes, J. Troya, L. Lúcio, and M. Wimmer. 2018. Full contract verification for ATL using symbolic execution. *Software and Systems Modeling* (2018), 815–849.
[232] J. Offutt and W. Xu. 2004. Generating Test Cases for Web Services Using Data Perturbation. *SIGSOFT Software Engineering Notes* 29, 5 (2004), 1–10.
[233] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A. Berre. 2005. Toward standardised model to text transformations. In *Proc. of ECMFA*. 239–253.
[234] S. Poulding and J. A. Clark. 2010. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Transactions on Software Engineering* (2010), 763–777.

[235] L. A. Rahim and J. Whittle. 2015. A survey of approaches for verifying model transformations. *Software and Systems Modeling* (2015), 1003–1028.
[236] J. E. Rivera, F. Duran, and A. Vallecillo. 2009. A Graphical Approach for Modeling Time-dependent Behavior of DSLs. In *Proc. of VL/HCC*. 51–55.
[237] L. M. Rose, N. Matragkas, D. S. Kolovos, and R. F. Paige. 2012. A Feature Model for Model-to-text Transformation Languages. In *Proc. of MiSE*. 57–63.
[238] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. 2008. The epsilon generation language. In *Proc. of ECMFA*. 1–16.
[239] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb. 2014. Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology* 24, 6 (2014), 1–44. Issue 1.
[240] J. Sanchez Cuadrado, L. Burgueno, M. Wimmer, and A. Vallecillo. 2020. Efficient execution of ATL model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering* (2020), 1–1.
[241] A. Schürr. 1995. Specification of graph translators with triple graph grammars. In *Proc. of WG*. 151–163.
[242] S. Segura, G. Fraser, A. Sánchez, and A. Ruiz-Cortes. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* (2016), 805–824.
[243] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Software* (2020), 46–53.
[244] G. M. K. Selim, J. R. Cordy, and J. Dingel. 2012. Model Transformation Testing: The State of the Art. In *Proc. of AMT*. 21–26.
[245] G. M. K. Selim, J. R. Cordy, J. Dingel, L. Lucio, and B. J. Oakes. 2015. Finding and Fixing Bugs in Model Transformations with Formal Verification: An Experience Report. In *Proc. of AMT*. 26–35.
[246] S. Sendall and W. Kozaczynski. 2003. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software* (2003), 42–45.
[247] L. Shan and H. Zhu. 2009. Generating Structurally Complex Test Cases By Data Mutation: A Case Study Of Testing An Automated Modelling Tool. *Comput. J.* 52, 5 (2009), 571–588.
[248] P. Stevens. 2007. A Landscape of Bidirectional Model Transformations. In *Proc. of GTTSE*. 408–424.
[249] G. Taentzer. 2003. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proc. of AGTIVE*. 446–453.
[250] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. 2009. On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA*. 18–33.
[251] M. Tisi, S. M. Perez, and H. Choura. 2013. Parallel Execution of ATL Transformation Rules. In *Proc. of MODELS*. 656–672.
[252] E. Torlak and D. Jackson. 2007. Kodkod: A Relational Model Finder. In *Proc. of TACAS*. 632–647.
[253] J. Troya, N. Moreno, M. F. Bertoa, and A. Vallecillo. 2021. Uncertainty representation in software models: a survey. *Software and Systems Modeling* (2021), 1–31.
[254] J. Troya, S. Segura, L. Burgueño, and M. Wimmer. 2021. Model Transformation Testing and Debugging: A Survey – Companion website. http://atenea.lcc.uma.es/projects/MTTestingDebuggingSurvey.html
[255] J. Troya and A. Vallecillo. 2011. A Rewriting Logic Semantics for ATL. *Journal of Object Technology* (2011), 5:1–29.
[256] D. Varró and A. Pataricza. 2003. Automated Formal Verification of Model Transformations. In *Proc. of CSDUML*. 63–78.
[257] J. Webster and R. T. Watson. 2002. Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly* 26, 2 (2002).
[258] E. J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* (1982), 465–470.
[259] C. Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proc. of EASE*.
[260] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[261] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* (2012), 67–120.
[262] R. Yousefian, V. Rafe, and M. Rahmani. 2014. A heuristic solution for model checking graph transformation systems. *Applied Software Computing* (2014), 169–180.
[263] H. Zhu, P. A. V. Hall, and J. H. R. May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (1997), 366–427.

A TRENDS

This appendix presents some interesting statistics out of the 140 primary studies related to publication trends, authors, venues, and types of papers.

A.1 Number of publications

Fig. 12 illustrates the number of publications on MT testing and debugging published between 2004 and 2020. We can see a variation in the number of papers depending on the year, but certainly a good activity between 2009 and 2018, with at least 7 publications each year and reaching a maximum of 17 publications in a single year (2013). We can also see the publication types, having a majority of conference papers. However, from 2014 we can see that the number of journal papers in the field is also significant, representing between 9% an 57% of the total of publications on each year. This can mean a certain degree of maturity in the context of MT testing and debugging.

We also related the geographical origin of each primary study to the affiliation country of its first co-author at the time of publishing the study. All 140 primary studies were originated from 22 different countries, with Spain and France ahead, as presented in Table 8. By continents, 73% of the papers are from Europe, 12% from America, 11% from Asia, 3% from Oceania and 1% from Africa.

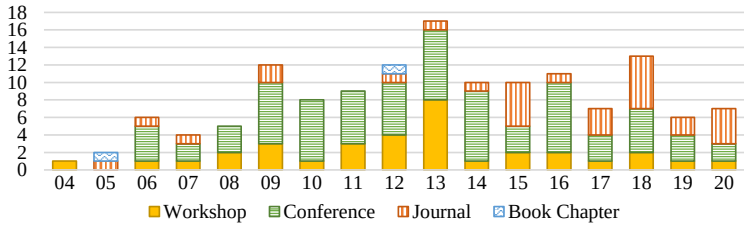

Fig. 12. Number of publications (2004-2020)

A.2 Researchers and organizations

We identified 227 distinct co-authors in the 140 primary studies under review, each paper having an average of 3.43 co-authors.

Table 9 presents the top authors on MT testing and debugging, their affiliations and country and the average number of co-authors. As a metric of impact, it also shows, for the papers considered in this study, the year in which their first paper was published and the total number of citations to their papers. The number of citations was collected from Google Scholar on May 4, 2021.

Fig. 13 shows a node per each author with at least five publications and a vertex with their co-authors who also have at least five publications. The graph easily lets us see collaborations among authors who are active in this field.

A.3 Venues

Table 10 displays the venues where more papers have been published. We can see that the recently extinguished *International Conference on Model Transformation (ICMT)* and the *International Conference on Model Driven Engineering Languages and Systems (MODELS)* are the leading conferences, with 12 and 11 papers published, respectively. Regarding journals, the one with the highest number of publications is the—well-known by the modeling community—*International Journal on Software and Systems Modeling (SoSyM)*, with 7 papers. As for workshops, the one with more publications is the *Workshop on the Analysis of Model Transformations (AMT)*, co-located in some editions of the MODELS conference, with 7 papers. It is noteworthy that this topic is of interest for the software

Table 8. Publications per country

Country	Spain	France	Germany	UK	USA	Austria	China	Italy	Hungary	Canada	Others
Papers	29 (20.7%)	23 (16.4%)	15 (10.7%)	8 (5.7%)	8 (5.7%)	7 (5%)	7 (5%)	7 (5%)	7 (5%)	6 (4.3%)	23 (16.5%)

Table 9. Top Authors on MT Testing

Author	Institution	Country	Papers	Avg co-authors	1^{st} paper	# citations
E. Guerra	Autonomous University of Madrid	Spain	14	2.50	2010	508
J.-M. Mottu	University of Nantes	France	14	2.86	2006	592
M. Wimmer	JKU Linz (bf. TU Wien)	Austria	14	4.14	2009	467
J. de Lara	Autonomous University of Madrid	Spain	12	2.83	2010	401
L. Burgueño	UOC (bf. Univ. of Malaga and CEA LIST)	Spain	10	2.80	2012	268
B. Baudry	KTH (bf. INRIA)	France	9	2.67	2006	752
M. Gogolla	University of Bremen	Germany	8	2.75	2005	475
A. Vallecillo	University of Malaga	Spain	8	2.75	2011	279

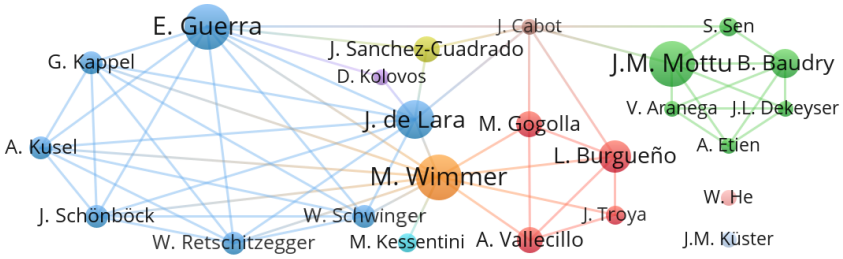

Fig. 13. Co-authors

Table 10. Top venues

Venue	ICMT	MODELS	AMT	SoSyM	ASE	ISSRE	MoDeVVa	KSE
Venue type	Conference	Conference	Workshop	Journal	Conference	Conference	Workshop	Conference
Papers	12	11	7	7	5	4	4	3

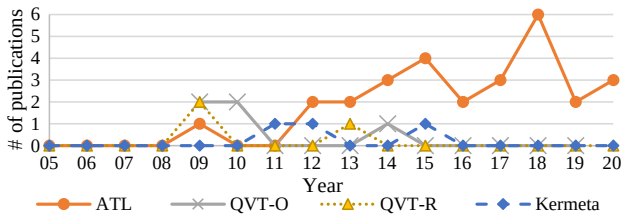

Fig. 14. MT languages most used

Table 11. Top MT Languages

Language	Papers
ATL	36
QVT-O	5
QVT-R	5
Kermeta	5
Acceleo	3
Tefkat	3

engineering community, as we observe by the number of papers published in venues such as the *International Conference on Automated Software Engineering (ASE)* or the *International Symposium on Software Reliability Engineering (ISSRE)*.

A.4 Model Transformation languages

A very interesting aspect to look into is the MT languages used in the different approaches. Certainly, there have been many different transformation languages proposed.

We have analyzed how the different languages have been used for MT testing and debugging along the years. The results are summarized in Fig. 14. We have also counted the total amount of works using each language which is shown in Table 11. ATL is by far the MT language that has been more frequently used in the context of MT testing and debugging, with a total amount of 36 studies. Besides, from 2012 we can see how its use has been intensified. ATL is followed by QVT languages, specifically QVT-O and QVT-R, having Kermeta in fourth position.