Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 292 (2013) 5–25

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Verification of Model Transformations A Survey of the State-of-the-Art

Daniel Calegari[1]

Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Nora Szasz[2]

Facultad de Ingeniería
Universidad ORT Uruguay
Montevideo, Uruguay

Abstract

Within the Model-Driven Engineering paradigm, software development is based on the definition of models providing different views of the system to be constructed and model transformations supporting a (semi)automatic development process. The verification of models and model transformations is crucial in order to improve the quality and the reliability of the products developed using this paradigm. In this context, the verification of a model transformation has three main components: the transformation itself, the properties of interest addressed, and the verification techniques used to establish the properties. In this paper we present an exhaustive review of the literature on the verification of model transformations analyzing these three components. We also take a problem-based approach exemplifying those aspects of interest that could be verified on a model transformation and show how this can be done. Finally, we conclude the need of an integrated environment for addressing the heterogeneous verification of model transformations.

Keywords: Model-Driven Engineering, model transformations, formal verification.

1 Introduction

Every traditional software development life-cycle is supported by a number of artifacts (e.g. requirements specifications, analysis and design documents, test suites, source code) which are mostly used as guides for the development as well as communication tools with the stakeholders.

The Model-Driven Engineering (MDE,[49]) paradigm pushes this view to its limits by envisioning a software development life-cycle driven by artifacts which are

[1] Email: dcalegar@fing.edu.uy
[2] Email: szasz@ort.edu.uy

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2013.02.002

models representing different views of the system to be constructed. Its feasibility is based on the existence of a (semi)automatic construction process driven by model transformations, starting from abstract models of the system and transforming them until an executable model is generated. In consequence, the quality of the whole process strongly depends on the quality of the models and model transformations.

We are concerned with model transformations and particularly with their verification. In this sense, the minimal requirement to be verified on a model transformation is that the transformation and the source and target models are well-formed. However, there are multiple other properties that could be verified and there is a plethora of verification approaches to do so. This topic is analyzed in [2] as a tri-dimensional problem consisting of: the transformation involved, the properties of interest addressed, and the formal verification techniques used to establish the properties.

The aim of this paper is to present a comprehensive review of the literature on the verification of model transformations extending the work in [2]. Particularly, we introduce the first dimension without going deeper, since there are well-known works [67,26] addressing this subject, and we extend the second and third dimensions with other aspects not addressed in [2]. We also follow a problem-based approach exemplifying by a case study those aspects of interest that could be verified on a model transformation and how they can be verified. Finally, we conclude the need of an integrated environment for addressing the heterogeneous verification of model transformations.

The remainder of the paper is structured as follows. We first detail the review process followed in Section 2. Then, in Section 3 we take a quick look at model transformations and define a running example. In Section 4 we introduce the different aspects of a transformation that must be verified, and in Section 5 we review how these aspects are verified in the literature. In Section 6 we use the running example to exemplify verification properties and discuss how they can be verified. Finally, in Section 7 we present some concluding remarks on this topic and guidelines for future work.

2 On the Literature Review

The literature review was conducted following a process related to the Systematic Review method [50]. We focused on answering the question: Which strategies have been used to deal with verification of model transformations and what kind of problems tried them to solve?

The process was performed in two steps: (a) a web search within electronic databases, and (b) a parallel search for authors, conferences, and references enforcing the web search results.

The first step started by identifying the keywords “model transformation” (with synonym “transformation”) and “verification” (with synonyms: “verification”, “formal verification”, “validation”, “certification”) and constructing a search query which was used within some selected electronic databases: SCOPUS, ScienceDi-

rect, Springer, IEEE Digital Library, and ACM Digital Library, through the Timbó portal [3]. The search was applied to the titles, abstracts and keywords of the papers in those databases, or all search fields in those cases in which we were not allowed to restrict the search to those fields.

The second step was a reinforcement of the first one by looking for those papers referenced in [2], as well as identifying those authors and conferences related to the subject. With the authors list we searched within their personal web pages as well as in the DBLP database. This second step helped us finding some technical reports and minor conferences material, as well as allowed us follow the evolution of some authors' work.

The inclusion criterion was based on the review of the title, abstracts and keywords of the papers found, evaluating whether they answered the initial question in some way. We considered both papers written in English and in Spanish. This initial set of papers was refined by reading their full text. Although some papers could not be considered of high quality since they were not published after a strict review process, we privileged their content in favor of answering the initial question.

For space reasons we do not include here the complete literature review. An extended version of this work with a description of each paper can be found at [18].

3 A Quick Look at Model Transformations

In the MDE ecosystem everything is a model, even the code is considered as a model. In this context, a model is an abstraction of the system or its environment. Every model *conforms* to a metamodel, i.e. a model which introduces the syntax and semantics of certain kind of models. In the same way, a metamodel conforms to some metametamodel. A metametamodel is usually self-defined, which means that it can be specified by means of its own semantics.

Metamodels are usually defined using UML Class Diagrams [36]. However, there are several other specific languages for this purpose, e.g. the MetaObject Facility (MOF, [35]), the Ecore metametamodel defined for the Eclipse Modeling Framework (EMF, [31]), and the Kernel MetaMetaModel (KM3, [5]). Besides a metamodel defines a modeling language which usually has a concrete syntax, it is possible to represent a model using the same languages as for metamodels. Moreover, for representing model instances, as well as for models which are a kind of "instance" of a metamodel, there is the graphical representation provided by UML object diagrams. Finally, every element in the hierarchy could be represented using XML Metadata Interchange (XMI, [39]). In some cases, there are conditions (called invariants) that cannot be captured by the structural rules of these languages, in which case modeling languages are supplemented with another logical language, e.g. the Object Constraint Language (OCL, [38]).

Let us introduce as a running example the well-known Class to Relational model transformation, originally introduced in [10], which became the de-facto standard

[3] Timbó. http://www.timbo.org.uy/

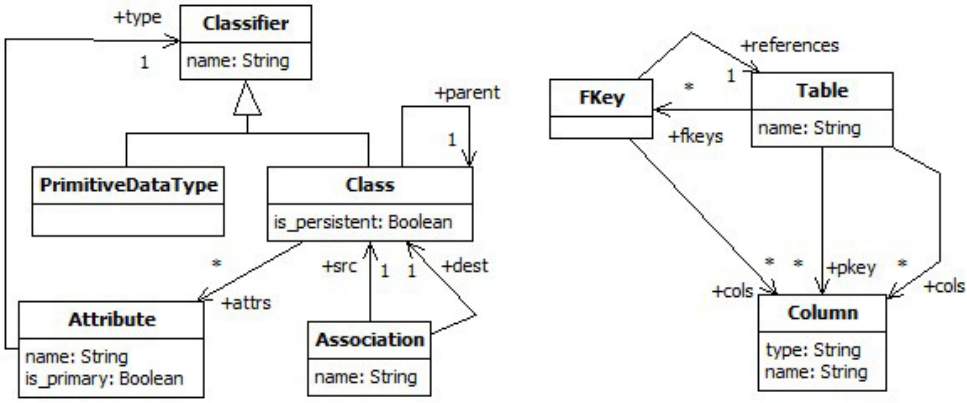


Fig. 1. Class and relational metamodel

example for model transformations. We will use the transformation as presented in [47]. The metamodel on the left side of Figure 1 defines UML class diagrams, where classes can contain one or more attributes, can belong to a class hierarchy and may be declared as persistent. Each attribute has a type that can be another class or a primitive datatype (string, boolean, integer, etc.). Attributes may be defined as primary. Associations are defined between classes with a direction from source to destination. An additional constraint is imposed that every class must have at least one attribute and at least one primary attribute (they may be inherited).

On the other side, relational models conform to the metamodel on the right side of Figure 1. Every model contains a number of tables with a number of columns. Some of these columns are primary keys of the corresponding table. A table may be associated to zero or more foreign keys. Each foreign key refers to a table and is associated with a number of columns that constitute the key.

The second building block of the MDE paradigm is the model transformation, which can also be considered as a model. As pointed out in [26], model transformation is closely related to program transformation. "Their differences occur in the mindsets and traditions of their respective transformation communities, the subjects being transformed, and the sets of requirements being considered. While program transformation systems are typically based on mathematically oriented concepts such as term rewriting, attribute grammars, and functional programming, model transformation systems usually adopt an object-oriented approach for representing and manipulating their subject models."

Kleppe et al. [51] define a model transformation as follows: "A *transformation* is the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

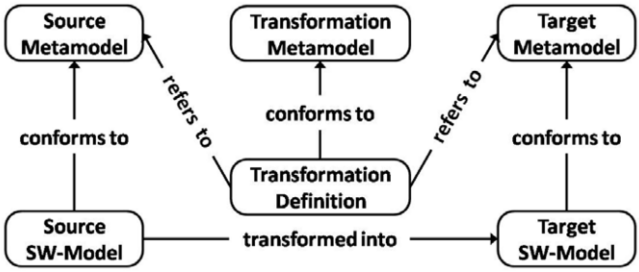


Fig. 2. An overview of model transformation

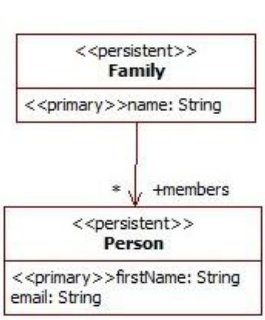
As summarized in Figure 2, extracted from [6], a model transformation basically takes as input a model *Ma* conforming to a given source metamodel *MMa* and produces as output another model *Mb* conforming to a given target metamodel *MMb*. The model transformation can be defined as well as a model *Mt* which itself conforms to a model transformation metamodel *MMt*. This last metamodel, along with the *MMa* and *MMb* metamodels, must conform to a metametamodel (such as MOF or Ecore). The transformation definition is executed by a transformation engine.

This schema defines model-to-model transformations. There are also model-to-text and text-to-model transformations where the target and source models, respectively, are just strings not conforming to any specific metamodel. Without loss of generality we will only consider model-to-model transformations (from now just transformations, or model transformations).

As pointed out in [67,66], "this definition is very general, and covers a wide range of activities for which model transformation can be used: automatic code generation, model synthesis, model evolution, model simulation, model execution, model quality improvement (e.g. through model refactoring), model translation, model-based testing, model checking, model verification."

However, this schema can be extended as is exhaustively studied in [26]. Leaving aside the details, the authors identify multiple variabilities on a model transformation, e.g. it can be bidirectional, it can take more than one source model as input and/or produce multiple target models as output, its rule application strategy can be deterministic, non-deterministic or interactive, the source and target models could be at different abstraction levels or not (horizontal versus vertical transformations), and the source and target models could conform to the same metamodel or not (endogenous versus exogenous transformations).

Beyond these aspects, there are several approaches for defining and executing model transformations, from **direct-manipulation** in which the transformations are usually developed in a programming language accessing an in-memory representation of models (e.g. Java Metadata Interface [27]), to **relational** (a.k.a. declarative) which consists of defining transformation rules as mathematical relations between source and target elements (e.g. Query/View/Transformation (QVT) Relations [37]), via **graph-transformation-based** which consists of considering models as typed attributed labeled graphs and applying graph transformations techniques




```
table Family {
  primary column name : String;
}

table Person {
  primary column firstName : String;
  column email : String;
}

table Family_members {
  column familyName : String;
  column personFirstName : String;
}
```

Fig. 3. Source and target models for the Class to Relational transformation

(e.g. Attributed Graph Grammar [83]).

In our running example, the transformation basically describes how persistent classes are transformed into tables. Attributes and associations of a class are transformed into columns of the corresponding table, and the primary and foreign keys are also set appropriately. Primary keys are defined for attributes defined as primary, as well as foreign keys for associations with other classes, including those in the class hierarchy. Below we show the persistent class to table rule definition using ATL declarative notation. The transformation uses a pre-processing step which flattens the features (either attributes or outgoing associations) of classes in a hierarchy. With this intermediate structure it is easier to define the rules, as the following rule which maps persistent classes to tables.

```
rule PersistentClass2Table{
  from
    c : SimpleClass!Class (c.is_persistent and c.parent.oclIsUndefined())
  to
    t : SimpleRDBMS!Table (
      name <- c.name,
      cols <- c.flFeatures->select(f | not f.isForeignKey)->collect(ft | ft.trace),
      pkey <- c.flFeatures->select(f | f.isPrimary)->collect(ft | ft.trace),
      fkeys <- c.flFeatures->select(f | f.isForeignKey))
}
```

In Figure 3 there is an example of a source model (in UML class diagram notation with stereotypes) and its corresponding target model (in KM3-like notation).

4 What to verify?

In this section we will focus on the second dimension introduced in [2]: the properties of interest addressed by the verification of a model transformation. There are also other works [56,84] which introduce the problem of verification by defining the set of properties to be addressed. However, the contents of these proposals are mostly included in the former one. We thus present the categories of properties identified in [2]: language-related and transformation-related properties. Next we explain each category without deeping inside those aspects already presented in [2], and extend them when necessary for adding more (sub)categories. We also include properties addressed in other works with a standardized nomenclature.

4.1 Language-Related Properties

This category refers to the computational nature of transformations and target properties of transformation languages. As introduced in [2], a transformation specification conforms to a transformation language which can possess properties on its own. In this context there are four properties of interest.

The first two properties are identified as execution-time properties. They are the **Termination** property which guarantees the existence of a target model, i.e. that the transformation execution finishes for any well-formed transformation specification, and the **Determinism** (a.k.a. Confluence) property which ensures uniqueness of the target model for a given source model and transformation specification. These properties are related to undecidable problems for sufficiently expressive (i.e. Turing-complete) transformation languages. In these cases, "[...] formally proving them cannot be done by relying on one particular transformation's specifics. [...] either the TL [Transformation Language] is kept as general (and powerful) as possible, making these properties undecidable, but the transformation framework provides capabilities for checking sufficient conditions ensuring them to hold on a particular transformation; or these properties are ensured by construction by the TL, generally by sacrificing its expressive power" [2]. According to this, in the first case, the properties could also be identified as a transformation-related properties (as defined in the next section) when proved for a specific transformation specification.

The third property, identified as a design-time property, is **Typing**, i.e. ensuring the well-formedness of the transformation specification w.r.t. its transformation language. The process of type checking may occur either at compile or run-time. Since model transformations are models, and models have metamodels (defining the transformation language), solutions to this problem are strongly related to Conformance and Model Typing as will be introduced in the next section.

Finally, we introduce a fourth property, not mentioned in [2], the **Preservation of Execution Semantics** property. This execution-time property states that the transformation execution must behave as expected according to the definition of the transformation language semantics. Related to this, and in strong contact with the Typing property, there are consistency needs between transformation rules which must also hold. For example, some languages do not allow an element of the input model to be matched more than once (redundancy problem). If this property does not hold, contradictory rules may be applied, e.g. two rules applied to the same element implying different things. Moreover, it is possible that a rule applied to an element of a hierarchy may be more restrictive than another one applied to an element in a lower level of the same hierarchy. In this case, there will be some models not matched by the second rule.

4.2 Transformation-Related Properties

This category refers to the modeling nature of transformations. As introduced in [2], a transformation refers to source/target models for which dedicated properties need to be ensured for the transformation to behave correctly.

In this context there is a first step on verification that strictly concerns the source and/or target model(s) a transformation refers to. This subcategory of properties is known as **On the Source/Target Model(s)**. As pointed out in [84] "the minimal requirement is to assure syntactic correctness, i.e., to guarantee that the generated model is a syntactically well-formed instance of the target language [w.r.t. structural and non-structural constraints]". This introduces a first group of properties known as **Conformance and Model Typing**. Conformance is nowadays well understood and automatically checked within modeling frameworks. There is a second group known as **N-Ary Transformations Properties**: transformations operating on several models at the same time, e.g. model composition, merging, or weaving, require dedicated properties to be checked.

But verification interests go beyond this kind of problems. When verifying a model transformation we want to consider its elements as a whole and not individually. In this sense, some authors, as in [16], use the notion of a transformation model, i.e. a model composed by the source and target metamodel, the transformation specification and the well-formedness rules. This transformation model could be implicit, i.e. we assume that every element is connected, or explicit, i.e. we really construct a unified structure with different purposes (e.g. tracing or verification). This model states how elements are related, and these relations introduce some syntactic and semantic questions.

In this sense, there are properties known as **Model Syntax Relations** that relate metamodel elements of the source and the target metamodels trying to ensure that certain elements or structures of any input model will be transformed into other elements or structures of the output model. This problem arises when these relations cannot be inferred by just looking at the individual transformation rules, or when the transformation language does not allow expressing some relations, and another constraint language must be used. This is also known as preservation of transformation invariants or structural correspondence.

Beyond structural relationships between source and target models, there are semantic properties that must be preserved, known as **Model Semantics Relations** and also as *semantic correctness* or *dynamic consistency* [84]. These properties generally depend on the metamodels semantics or on the kind of transformation. Some properties of interest are semantic equivalence, (weak) bisimilarity and preservation of properties, temporal properties, refactoring, and refinement.

Finally, we add a fourth category called **Functional Behavior**, not considered in [2], which refers to determining whether a transformation behaves as a mathematical function. In particular, it is possible that a transformation may be injective, surjective, bijective, or at least, executable (i.e. there exists a valid pair of source and target models that satisfy the transformation). It is also possible to analyze these properties considering individual rules within a transformation. These properties are introduced in [16]. Moreover, there is a specific property known as **Syntactic Completeness** which refers to the need (in some cases) of completely covering the source/target metamodel by transformation rules. This is also presented in [56] as *metamodel coverage* introducing the problem that if the transformation does not

cover the entire metamodel, then this leads to some input models which cannot be transformed. From a functional point of view, syntactic completeness means that the transformation is a total function. When considered for a specific transformation, determinism is also a functional property. In fact, as introduced in [16], when a transformation is *total* and *deterministic*, it is called *functional*.

4.3 Concluding Remarks

We have seen a classification of the properties of interest addressed by the verification of a model transformation. This classification, formerly introduced in [2], identifies language-related and transformation-related properties, the first ones referring to the computational nature of transformations and target properties of transformation languages, and the second ones referring to the modeling nature of transformations. We extended this classification by adding two subcategories addressing properties based on other related works.

When following a MDE-based software project, formal verification is mostly focused on the second category of properties (transformation-related), whilst those within the first category (language-related) are in general assumed to be somehow automatically verified by the development tools.

A summary of the properties addressed in the literature can be found in Table 1.

5 How to verify?

As pointed out in [30], a property can be either verified or validated, leading to the well-known distinction between verification and validation. Formally, verification is addressed to "determine whether the products [...] satisfy the conditions imposed" whilst validation is addressed to "determine whether it [the product] satisfies specified requirements" [45]. In other words, verification is the process of proving that we are building the product in the right way, while validation is the process of proving that we are building the right product.

We are focused on verification, and in particular on formal verification, i.e. in the act of verifying using formal methods. Although formal verification techniques may be expensive, they can be helpful in guaranteeing the correctness of critical applications where no other verification technique is acceptable. In contrast to formal verification, there are other techniques which may detect errors or improve confidence, but they cannot prove any property in a definite way.

In this section we will focus on the third dimension introduced in [2]: the formal verification techniques used to establish the properties. We extend the former work by defining two new categories for verification techniques.

5.1 Inference, model checking, testing, static analysis or by construction

We introduce this first category which refers to the kind of technique used for verification. *Logical inference* (a.k.a. *theorem proving*) consists of using a mathematical

Table 1
Summary of properties addressed in the literature

Language-Related Properties		
Termination	[8][15][28][55][58][85][89]	
Determinism	[8][16][43][41][55][57][58][89]	
Typing	[52][62],[81]	
Preservation of Exec. Sem.	[3][16][40][60][75][89]	
Transformation-Related Properties		
Source/Target	Conformance	[3][1][4][19][17][32][54] [60][61][58][59][76][79][82][89]
	N-Ary	[22][48][68]
Syntax Relations	[1][4][19][17][32][25][40][63] [60][61][58][70][73][76][79][80]	
Semantics Relations	General	[17][59][76][79][82]
	Sem. Eq.	[7][12][23][33][44][58] [71][74][78][84]
	Temporal	[9][11][25][30]
	Refactoring	[42]
	Refinement	[19][65][77]
Functional Behavior	General	[16]
	Synt. Comp.	[25][40][56][58][75][87]

representation of a system and the properties that must be verified, as well as a logic in that semantic domain which allows reasoning about that representation, leading from premises to conclusions. This process is usually carried out using theorem proving software and it is usually only partially automated. *Model checking* also consists of using a mathematical representation of a system, and proofs consist of a systematic exhaustive exploration of the mathematical model. With the first approach there is usually a high verification cost, whilst with the second there are well-known limitations such as the state-explosion problem. On the other hand, *testing* relies on the construction of test strategies for a property including subsequent execution of (either parts or all of) the system according to these strategies. Although testing is usually considered a validation strategy, it could be used for verification purposes. However, as it is well known, testing can only show the presence of errors and not their absence. Finally, we can find strategies based on *static*

analysis, i.e. on the analysis of a model transformation that is performed without actually executing it. Static analysis typically consists on semi-decision techniques. In this sense, they are efficient but they cannot assure the overall correctness of the design. For matter of completeness we also consider the satisfaction of properties that hold by construction of the transformation, e.g. those achieved by using special transformation languages such as DSLTrans [8].

5.2 Metamodel or model level

This category consists of the abstraction level w.r.t. the elements involved in the transformation, and it is also referred as *offline and online* verification [4], and as *input independent and input dependent* verification [2]. Metamodel-level verification uses the metamodel information for verifying properties for any well-formed model instance while model-level verification uses arbitrary source models. As pointed out in [84], the first level typically requires the use of sophisticated theorem proving techniques and tools with a huge verification cost. For this reason, the second is in many cases a practical and valuable aid, but it cannot ensure the zero-fault level of quality since it checks a finite number of specific cases. However, as model-level verification takes place on a lower level of abstraction, the range of properties that can be validated is much greater than when using metamodel-level verification.

5.3 Specification or implementation

As introduced in [30], verification can either be done on the model (specification) level or on the implementation level. *Specification-level* verification involves only the specification of the transformation in some transformation language, and in consequence the semantics defined for that transformation language. In contrast, *implementation-level* verification means also considering the way a transformation is executed by a transformation engine. As far as we know, verification techniques found in the literature are of the first type, since it is assumed that any transformation engine conforms with the semantics of the transformation language and properties do not depend on how exactly the transformation is executed, including the case of Determinism, Termination and Preservation of Execution Semantics properties.

5.4 Transformation independent or dependent

This dimension is introduced in [2]. *Transformation independent* techniques are those techniques which prove properties for any transformation, and in consequence they assure that no assumption is made on the specific source model. In contrast, *transformation dependent* techniques rely on a specific model transformation. Transformation independency is achieved either by a transformation language that preserves the properties by default, or by ensuring a property by construction of the transformation.

5.5 Concluding Remarks

We have shown how verification techniques can be classified in different categories referring to: (a) the kind of technique used for verification, (b) the abstraction level w.r.t. the elements involved in the transformation, (c) the abstraction level w.r.t. the implementation of the transformation, and (d) the dependency/independency w.r.t the transformation specification. It is worth saying that these categories are orthogonal, i.e. there are verification techniques which correspond to more than one category. A summary of the verification techniques addressed in the literature within this categorization can be found in Table 2.

6 Verification by Example

In this section we go back to the Class to Relational example introduced in Section 3 to illustrate several verification properties and discuss how the verification could be addressed.

6.1 Conformance and Model Typing

Beyond the basic conformance needs, there are usually invariants that cannot be captured by the structural rules of the modeling language. Invariants are well-formedness rules that must hold at all time for any model conforming to a metamodel. Invariants can be defined on metametamodels, metamodels and models. In the example the following invariants must hold on models:

- All associations have distinct names (the same for classes)
- The owned attributes of a class are uniquely named within it owner class and the classes it inherits
- There are no cycles of inheritance within the parent relation in classes
- If a class is persistent so are all of its superclasses
- If a class is persistent it has at least one attribute marked as primary
- All tables have distinct names
- All columns have distinct names within a table
- Every table must have at least one primary column

Moreover, possible invariants on models are for example:

- All families have distinct names
- Every person has a distinct first name within a family

Invariants can be expressed using a constraint language like the OCL, and as it was said in Section 4, this conformance checking is nowadays automatically addressed within modeling frameworks using automated checkers. These checkers can be based on SAT solvers or model-checking, as in [3,32]. This verification is at a model level, but there are other alternatives, for example performing the verification using logical inference. In this case, we can formalize metamodels, models and

Table 2
Summary of verification techniques addressed in the literature

	inf	mod chk	sta ana	tes	con	met	mod	spe	imp	tra ind	tra dep
[3,9]	√						√	√			√
[4,61,33]	√					√		√			√
[15,21,17] [58,60,76] [79,82]	√					√		√		√	
[1,22]	√						√	√		√	
[77]	√	√		√			√	√			√
[16,57,28] [68,43,41] [73,85,89]		√				√		√		√	
[63,84]		√				√		√			√
[7,11,19,25] [30,40,32]		√					√	√		√	
[44]		√						√		√	
[42,65,78]		√				√	√	√		√	√
[54,55]		√	√			√		√		√	
[71,70]		√				√	√	√			√
[48,86]			√					√		√	
[75]			√			√		√		√	
[87,56]				√			√	√		√	
[12,8,52] [62,74,81]					√	√		√		√	
[23]					√	√		√			√

invariants in some formal language, e.g. in first-order logic, and then use a proof assistant as done in [60,58,79,82].

Moreover, using this strategy, one could perform the verification at a metamodel level, forgetting models and considering transformations and proving the postcon-

ditions assuming that both the pre-conditions and the transformation rules hold, as done in [17]. For our running example, a simple property that can be proven is that the length of the `Columns` within a `Table` must be greater than zero. This property holds by the fact that every `Attribute` is transformed into a `Column` and because every `Class` has at least one `Attribute`. This information is given in the transformation rules and in the source invariants, respectively.

A step further of this approach is the one presented in [76] where giving a representation of models, metamodels and transformations in the Calculus of Inductive Constructions [24], they can extract a correct transformation. This requires specifying a transformation as types of the form

$$\forall x \ : \ Pil.I(x) \ \rightarrow \ (\exists y \ : \ Psl.O(x, y))$$

where Pil and Psl are source and target metamodel types, $I(x)$ specifies a precondition on the source model x for the transformation to be applied, and $O(x, y)$ specifies required properties of the output model y. A proof of this expression allows the automatic construction of a function f such that, given any x satisfying the precondition $I(x)$, then the postcondition $O(x, fx)$ will be satisfied.

Finally, a complementary approach in [61] proposes a language for assertions based on first-order logic that describes some characteristics of a model under transformation. Then, they can derive how an assertion evolves when applying transformation rules using SWIProlog [88] as an inference system. If the assertions to be verified could be derived from the final assertion, thus they hold in the target model.

6.2 Model Syntax Relations

A transformation model gives useful information about the relation between the elements connected by a transformation. With this, some other relations could be inferred which are not evident by just looking at the individual elements.

In our example we can illustrate this with the following property: if `c` is a subclass of `d`, all columns of `c`'s table are included in `d`'s table. Since the rule `PersistentClass2Table` uses the flattening of features of the source metamodel, this property cannot be trivially inferred.

There are many alternatives for proving this property. First, we can use a formal language to state it and a proof assistant to prove it. We can also use the language for assertions and prove that this property can be derived from the final assertion. We explored these alternatives when discussed conformance and model typing.

Another option is to define a transformation contract stating the pre and post conditions of a transformation (or an individual rule), and check whether this contract holds. This contract could be written in OCL and then verified using an OCL checker or some other model-checker, as in [19,32]. It can also be written using a dedicated tool and then verified using some specific algorithm, as in [25,40].

6.3 Functional Behavior

Our example has some "functional" needs. As defined in [16], it is possible to check whether some properties hold considering either a specific rule or the whole model transformation. As an example, the rule `PersistentClass2Table` is executable since there exists a valid pair of models (those in Figure 3) that satisfy it. Since this rule is the top rule in the transformation, we can derive that the whole transformation is also executable.

Moreover, this rule is not injective. Consider that the transformation maps persistent classes that are roots of a hierarchy to tables, whilst derived classes are flattened and their attributes mapped to columns of the former table. Then, we can find a counterexample which has the same target (a table with two columns) where one is produced from a class with two attributes, and the other from two classes related through inheritance with one attribute each. Following the same ideas, neither the rule nor the transformation is total (Syntactic Completeness) since they apply only to persistent classes and then non-persistent classes will never be transformed. In this case it is clear that syntactic completeness is not desirable.

These properties could be verified by encoding them as UML/OCL consistency problems on the transformation model, as defined in [16]. Then, an OCL-checker could be used. Another alternative is to verify them by static analysis of the transformation rules and the underlying metamodels, as in [75].

6.4 Determinism and Termination

As we already said in the previous section, these two properties are the hardest to prove since there are related to undecidable problems. One alternative to achieve them is to use some language which guarantees both by construction, as introduced in [8]. However, this option clearly reduces expressive power.

Other alternatives are representing the transformation model in some formal language to perform logical inference, as in [58], or using static analysis, as in [55]. However, the most referred alternative is to express the transformation as a graph-rewriting problem which allows performing critical pair analysis, as in [15,28,43,41,57,85].

6.5 Preservation of Execution Semantics

The preservation of the execution semantics is matter of verification during the development of a transformation engine. However, when defining a model transformation there are consistency needs that must be addressed. As an example, we may not want redundant rules, and indeed our example does not have redundancy. An example of a redundant rule would be one mapping attributes to columns but applicable only to persistent classes.

As before, we can encode these needs as a consistency problem on the transformation model and use a model-checker, as in [3], or use static analysis, as in [75]. Moreover we can use logical inference as in [60].

Another possible approach is the followed in [89], where the transformation is expressed as a Colored Petri Net (CPN, [46]) which allows the formal exploration of CPN properties. In particular, the authors can verify whether there are transitions which are never enabled during execution, so called Dead Transition Instances or L0-Liveness.

Only two properties are not considered within the example: Typing and Model Semantics Relations. As we said before, Typing is strongly related to Conformance and Model Typing. Also, since we are working with structural models, we do not have dynamic properties which are the main source of semantic properties. We might as well force some semantic property, but we decided to let the reader refer to [18] for more examples.

7 Conclusions and Future Work

We conducted a comprehensive literature review on the verification of model transformations which was structured following the three dimensions presented in [2]. We extended the former review and followed a problem-based approach exemplifying those aspects of interest that could be verified on a transformation and discussing how they can be verified. For space reasons we left some details, which can be found at [18].

At this point it is clear that there exist several alternatives, not only for the specification of a transformation but also for its formal verification, which depend on those properties that must be addressed in each specific case. This problem increases when considering bidirectional, higher-order, and multi-model transformations. In this sense, there is some parallelism between MDE-built systems and traditional software systems: heterogeneous multi-logic specifications are needed, since different systems have different aspects that are best specified in different semantic domains. An example of this was introduced in the last section, where a small-size example introduces different problems, each of them best addressed by several strategies.

To cope with this situation it is usually proposed a separation of duties between software developers. On the one side there are those experts in the MDE domain, and on the other, those in formal verification. This gives rise to different technological spaces [53], i.e. working contexts with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. In general terms, MDE experts define models and transformations, while formal verification experts conduct the verification process, often aided by some (semi)automatic generation process which translates the MDE elements to their formal representation in the semantic domain used by the experts for verification purposes.

Related to this, the most common transformation approaches referred in the literature are the relational and graph-based approaches. The point here is that those approaches closely related with traditional programming languages (direct manipulation, operational, etc.) introduce verification problems that are carried out by traditional code verification approaches. Moreover, both relational and graph-based

approaches define elements that are most easily translated into formal domains preserving their semantics.

The landscape before us leads us to consider a heterogeneous strategy to verification, closely related to the ideas in [20,69]. In these works the authors define an environment where specification languages are described in their "natural" semantics, and the relations between languages are expressed by appropriate translations. Moreover, different semantic domains have associated different tools for formal reasoning, and since there are formal translations between languages, it is possible to "share" tools. These ideas are based in the Theory of Institutions [34].

Working on this idea, we could express metamodels and transformations as so called *institutions* in some consistent and interdependent way, as well as transformation properties in different logics (also institutions), and their translations (formally, institution (co)morphisms) into several logics with the purpose of proving transformation properties. Models could be both represented as institutions or as sentences within the institution of metamodels. For example, considering our running example, we can represent models in XMI, metamodels in MOF, and transformations in QVT. These languages could be defined as institutions and there could be translations from them to different logics (also specified as institutions), e.g. first-order logic, rewriting logic, modal logic, etc. If there is a conformance need specified in OCL, it could be possible to translate the different elements to rewriting logic and perform the verification as defined in [13].

To put these ideas into practice we can use the Heterogenous Tool Set (Hets, [69]) which is meant to support heterogeneous multi-logic specifications. Hets is a parsing, static analysis and proof management tool combining various such tools for individual specification languages. Nowadays the tool supports many logics (e.g. FOL, rewriting logic and modal logic) and tools (e.g. IsabelleHOL [72] and Maude [64]). Moreover, it provides proof management capabilities for monitoring the overall correctness of a heterogeneous specification whereas different parts of it are verified using, possibly different, proof systems.

Anyway, it is worth pointing out that the instantiation of this framework is not as direct as it seems. We need to formally specify firstly every MDE building block within the Theory of Institutions and secondly any possible and useful translation to those logics we need. These representations may neither be direct nor even possible within this theory.

Besides, any new logic in Hets may support formal verification in its own semantic domain. However, this requires the definition of a sound proof system. In some cases where there is no such proof system, its definition can be very expensive. For example, in [61] the authors introduce a language for assertions and a semi-automated reasoning system which is not formally specified as a sound proof system.

References

[1] Akehurst, D., S. Kent and O. Patrascoiu, *A relational approach to defining and implementing transformations between metamodels*, Software & Systems Modeling **2** (2003), pp. 215–239.

[2] Amrani, M., L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. traon and J. Cordy, *A tridimensional approach for studying the formal verification of model transformations*, in: *Verification and validation Of model Transformations (VOLT)* (2012).

[3] Anastasakis, K., B. Bordbar, G. Georg and I. Ray, *On challenges of model transformation from UML to Alloy*, Software & Systems Modeling **9** (2010), pp. 69–86.

[4] Asztalos, M., L. Lengyel and T. Levendovszky, *Towards automated, formal verification of model transformations*, in: *ICST* (2010), pp. 15–24.

[5] ATLAS, "KM3: Kernel MetaMetaModel," LINA & INRIA, Manual v0.3 edition (2005).

[6] ATLAS, "ATL: Atlas Transformation Language," LINA & INRIA, User Manual v0.7 edition (2006).

[7] Baresi, L., K. Ehrig and R. Heckel, *Verification of model transformations: A case study with BPEL*, in: U. Montanari, D. Sannella and R. Bruni, editors, *TGC*, Lecture Notes in Computer Science **4661** (2006), pp. 183–199.

[8] Barroca, B., L. Lucio, V. Amaral, R. Félix and V. Sousa, *DSLTrans: A turing incomplete transformation language*, in: B. A. Malloy, S. Staab and M. van den Brand, editors, *SLE*, Lecture Notes in Computer Science **6563** (2010), pp. 296–305.

[9] Becker, B., D. Beyer, H. Giese, F. Klein and D. Schilling, *Symbolic invariant verification for systems with dynamic structural adaptation* (2006), p. 72.

[10] Bézivin, J., B. Rumpe, A. Schürr and L. Tratt, *Model transformations in practice workshop*, in: Bruel [14], pp. 120–127.

[11] Boronat, A., R. Heckel and J. Meseguer, *Rewriting logic semantics and verification of model transformations*, in: *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)* (2009), pp. 18–33.

[12] Boronat, A., A. Knapp, J. Meseguer and M. Wirsing, *What is a multi-modeling language?*, in: A. Corradini and U. Montanari, editors, *WADT*, Lecture Notes in Computer Science **5486** (2008), pp. 71–87.

[13] Boronat, A. and J. Meseguer, *Algebraic semantics of OCL-constrained metamodel specifications*, in: M. Oriol, B. Meyer, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw and C. Szyperski, editors, *Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing **33**, Springer, 2009 pp. 96–115.

[14] Bruel, J.-M., editor, "Satellite Events at the MoDELS 2005 Conference, Jamaica. Revised Selected Papers," Lecture Notes in Computer Science **3844**, Springer, 2006.

[15] Bruggink, H. S., *Towards a systematic method for proving termination of graph transformation systems*, Electronic Notes in Theoretical Computer Science **213** (2008), pp. 23–38.

[16] Cabot, J., R. Clarisó, E. Guerra and J. de Lara, *Verification and validation of declarative model-to-model transformations through invariants*, Journal of Systems and Software (2009).

[17] Calegari, D., C. Luna, N. Szasz and A. Tasistro, *A type-theoretic framework for certified model transformations*, in: J. Davies, L. Silva and A. da Silva Simão, editors, *SBMF*, Lecture Notes in Computer Science **6527** (2010), pp. 112–127.

[18] Calegari, D. and N. Szasz, *Verification of model transformations: A survey of the state-of-the-art (extended version)*, Technical Report RT12-05, InCo-PEDECIBA (2012).

[19] Cariou, E., N. Belloir, F. Barbier and N. Djemam, *OCL contracts for the verification of model transformations*, OCL workshop of MoDELS 2009 (2009).

[20] Cengarle, M. V., A. Knapp, A. Tarlecki and M. Wirsing, "A Heterogeneous Approach to UML Semantics," Lecture Notes in Computer Science **5065**, Springer Berlin Heidelberg, 2008 pp. 383–402.

[21] Chan, K., *Formal proofs for QoS-oriented transformations*, in: *EDOC Workshops* (2006), p. 41.

[22] Chechik, M., S. Nejati and M. Sabetzadeh, *A relationship-based approach to model integration*, Innovations in Systems and Software Engineering **8** (2012), pp. 3–18.

[23] Combemale, B., X. Cragut, P. Garoche and X. Thirioux, *Essay on semantics definition in mde - an instrumented approach for model verification*, Journal of Software **4** (2009).

[24] Coquand, T. and C. Paulin, *Inductively defined types*, in: *COLOG-88: Proceedings of the international conference on Computer logic* (1990), pp. 50–66.

[25] Csertán, G., G. Huszerl, I. Majzik, Z. Pap, A. Pataricza and D. Varró, *VIATRA - visual automated transformations for formal verification and validation of uml models*, in: *ASE* (2002), pp. 267–270.

[26] Czarnecki, K. and S. Helsen, *Feature-based survey of model transformation approaches*, IBM Systems Journal **45** (2006), pp. 621–645.

[27] Dirckze, R., *Java Metadata Interface (JMI) specification*, Technical Report JSR 040, Java Community Process (2002).

[28] Ehrig, H., K. Ehrig, J. de Lara, G. Taentzer, D. Varró and S. Varró-Gyapay, *Termination criteria for model transformation*, in: M. Cerioli, editor, *FASE*, Lecture Notes in Computer Science **3442** (2005), pp. 49–63.

[29] Ehrig, H., R. Heckel, G. Rozenberg and G. Taentzer, editors, "Graph Transformations, 4th International Conference (ICGT 2008), UK. Proceedings," Lecture Notes in Computer Science **5214**, Springer, 2008.

[30] Engels, G., J. Küster, R. Heckel and M. Lohmann, *Model-based verification and validation of properties*, Electr. Notes Theor. Comput. Sci. **82** (2003).

[31] F.Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. Grose, "Eclipse Modeling Framework," Addison-Wesley Professional, 2003.

[32] Garcia, M. and R. Möller, *Certification of transformations algorithms in model-driven software development*, in: W.-G. Bleek, J. Räsch and H. Züllighoven, editors, *Software Engineering 2007*, GI-Edition Lecture Notes in Informatics **105**, 2007, pp. 107–118.

[33] Giese, H., S. Glesner, J. Leitner, W. Schäfer and R. Wagner, *Towards verified model transformations*, in: D. Hearnden, J. G. Süß, B. Baudry and N. Rapin, editors, *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification, Italy* (2006), pp. 78–93.

[34] Goguen, J. A. and R. M. Burstall, "Introducing institutions," Lecture Notes in Computer Science **164**, Springer Berlin Heidelberg, 1984 pp. 221–256.

[35] Group, O. M., *Meta Object Facility (MOF) 2.0 Core Specification*, Specification Version 2.0, Object Management Group (2003).

[36] Group, O. M., *Unified Modeling Language: Superstructure*, Specification Version 2.0, Object Management Group (2005).

[37] Group, O. M., *Meta Object Facility (MOF) 2.0 Query/View/Transformation*, Final Adopted Specification Version 1.1, Object Management Group (2009).

[38] Group, O. M., *Object Constraint Language*, Formal Specification Version 2.2, Object Management Group (2010).

[39] Group, O. M., *OMG MOF 2 XMI Mapping Specification*, Specification Version 2.4.1, Object Management Group (2011).

[40] Guerra, E., J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schonbock and W. Schwinger, *Automated verification of model transformations based on visual contracts*, Automated Software Engineering (2012), pp. 1–42.

[41] Heckel, R., J. M. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *ICGT*, Lecture Notes in Computer Science **2505** (2002), pp. 161–176.

[42] Heckel, R. and S. Thne, *Behavioral refinement of graph transformation-based models*, Electronic Notes in Theoretical Computer Science **127** (2005), pp. 101–111.

[43] Hermann, F., H. Ehrig, F. Orejas and U. Golas, *Formal analysis of functional behaviour for model transformations based on triple graph grammars*, in: H. Ehrig, A. Rensink, G. Rozenberg and A. Schürr, editors, *ICGT*, Lecture Notes in Computer Science **6372** (2010), pp. 155–170.

[44] Hermann, F., M. Hülsbusch and B. König, *Specification and verification of model transformations*, ECEASST **30** (2010).

[45] IEEE, *IEEE standard glossary of software engineering terminology*, IEEE Std 610.12-1990 (1990), p. 1. URL standards.ieee.org/findstds/standard/610.12-1990.html

[46] Jensen, K. and L. M. Kristensen, "Coloured Petri Nets - Modelling and Validation of Concurrent Systems," Springer Berlin Heidelberg, 2009 .

[47] Jouault, F. and I. Kurtev, *Transforming models with ATL*, in: Bruel [14], pp. 128–138.

[48] Katz, S., "Aspect Categories and Classes of Temporal Properties," Lecture Notes in Computer Science **3880**, Springer Berlin Heidelberg, 2006 pp. 106–134.

[49] Kent, S., *Model-driven engineering*, in: M. Butler, L. Petre and K. Sere, editors, *IFM*, Lecture Notes in Computer Science **2335** (2002), pp. 286–298.

[50] Kitchenham, B., *Procedures for performing systematic reviews*, Keele university. technical report tr/se-0401, Department of Computer Science, Keele University, UK (2004).

[51] Kleppe, A., J. Warmer and W. Bast, "MDA Explained; The Model Driven Architecture: Practice and Promise," Addison Wesley, 2003 .

[52] Kühne, T., G. Mezei, E. Syriani, H. Vangheluwe and M. Wimmer, *Systematic transformation development*, ECEASST **21** (2009).

[53] Kurtev, I., J. Bezivin and M. Aksit, *Technological spaces: An initial appraisal*, in: *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002.

[54] Küster, J. M., *Systematic validation of model transformations*, in: *Proceedings of WiSME'04 (associated to UML'04)*, 2004.

[55] Kuster, J. M., *Definition and validation of model transformations*, Software & Systems Modeling **5** (2006), pp. 233–259.

[56] Küster, J. M. and M. Abd-El-Razik, *Validation of model transformations - first experiences using a white box approach*, in: T. Kühne, editor, *MoDELS Workshops*, Lecture Notes in Computer Science **4364** (2006), pp. 193–204.

[57] Lambers, L., H. Ehrig and F. Orejas, *Efficient detection of conflicts in graph-based model transformation*, Electronic Notes in Theoretical Computer Science **152** (2006), pp. 97–109.

[58] Lano, K. and S. K. Rahimi, *Specification and verification of model transformations using UML-RSDS*, in: D. Méry and S. Merz, editors, *IFM*, Lecture Notes in Computer Science **6396** (2010), pp. 199–214.

[59] Lano, K. and S. K. Rahimi, *Model-driven development of model transformations*, in: J. Cabot and E. Visser, editors, *ICMT*, Lecture Notes in Computer Science **6707** (2011), pp. 47–61.

[60] Ledang, H. and H. Dubois, *Proving model transformations*, in: J. Liu, D. Peled, B.-Y. Wang and F. Wang, editors, *TASE* (2010), pp. 35–44.

[61] Lengyel, L., I. Madari, M. Asztalos and T. Levendovszky, *Validating Query/View/Transformation relations* (2010), pp. 7–12.

[62] Levendovszky, T., L. Lengyel and T. Aztalos, *Supporting domain-specific model patterns with metamodeling*, Software & Systems Modeling **8** (2009), pp. 501–520.

[63] Lucio, L., B. Barroca and V. Amaral, *A technique for automatic validation of model transformations*, in: D. C. Petriu, N. Rouquette and Ø. Haugen, editors, *MoDELS (1)*, Lecture Notes in Computer Science **6394** (2010), pp. 136–150.

[64] M. Clavel, P. L., S. Eker and J. Meseguer, *Principles of Maude*, , **4** (2000).

[65] Massoni, T., R. Gheyi and P. Borba, *Formal refactoring for UML class diagrams*, in: *19th brazilian Symposium on Software Engineering (SBES)*, 2005, pp. 152–167.

[66] Mens, T., *Model transformation: A survey of the state-of-the-art*, in: S. Gerard, J.-P. Babau and J. Champeau, editors, *Model Driven Engineering for Distributed Real-Time Embedded Systems*, Wiley - ISTE, 2010 .

[67] Mens, T., K. Czarnecki and P. V. Gorp, *04101 discussion - a taxonomy of model transformations*, in: J. Bézivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings **04101** (2004).

[68] Molderez, T., H. Schippers, D. Janssens, M. Haupt and R. Hirschfeld, *A platform for experimenting with language constructs for modularizing crosscutting concerns*, in: *Proceedings of the 3rd International Workshop on Academic Software Development Tools and Techniques (WASDeTT)*, 2010.

[69] Mossakowski, T., *Heterogeneous specification and the heterogeneous tool set*, Technical report, Universitaet Bremen (2005), habilitation thesis.

[70] Narayanan, A. and G. Karsai, *Specifying the correctness properties of model transformations* (2008), p. 45.

[71] Narayanan, A. and G. Karsai, *Towards verifying model transformations*, Electronic Notes in Theoretical Computer Science **211** (2008), pp. 191–200.

[72] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL - A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[73] Orejas, F. and M. Wirsing, *On the specification and verification of model transformations*, in: J. Palsberg, editor, *Semantics and Algebraic Specification*, Lecture Notes in Computer Science **5700**, Springer, 2009 pp. 140–161.

[74] Padberg, J., M. Gajewsky and C. Ermel, *Refinement versus verification: Compatibility of net-invariants and stepwise development of high-level petri nets*, Technical Report 97-22, Technical University Berlin (1997).

[75] Planas, E., J. Cabot and C. Gómez, *Two basic correctness properties for ATL transformations: Executability and coverage*, in: *3rd International Workshop on Model Transformation with ATL*, Zurich, Switzerland, 2011.

[76] Poernomo, I., *Proofs-as-model-transformations*, in: A. Vallecillo, J. Gray and A. Pierantonio, editors, *ICMT*, Lecture Notes in Computer Science **5063** (2008), pp. 214–228.

[77] Pons, C. and D. Garcia, *A lightweight approach for the semantic validation of model refinements*, Electronic Notes in Theoretical Computer Science (2008).

[78] Rangel, G., L. Lambers, B. König, H. Ehrig and P. Baldan, *Behavior preservation in model refactoring using dpo transformations with borrowed contexts*, in: Ehrig et al. [29], pp. 242–256.

[79] Schätz, B., *Verification of model transformations*, ECEASST **29** (2010).

[80] Schürr, A. and F. Klar, *15 years of triple graph grammars*, in: Ehrig et al. [29], pp. 411–425.

[81] Steel, J. and J.-M. Jezequel, *On model typing*, Software & Systems Modeling **6** (2007), pp. 401–413.

[82] Stenzel, K., N. Moebius and W. Reif, *Formal verification of QVT transformations for code generation*, in: J. Whittle, T. Clark and T. Kühne, editors, *MoDELS*, Lecture Notes in Computer Science **6981** (2011), pp. 533–547.

[83] Taentzer, G., *AGG: A graph transformation environment for modeling and validation of software*, in: J. L. Pfaltz, M. Nagl and B. Böhlen, editors, *AGTIVE*, Lecture Notes in Computer Science **3062** (2003), pp. 446–453.

[84] Varró, D. and A. Pataricza, *Automated formal verification of model transformations*, in: J. Jürjens, B. Rumpe, R. France and E. B. Fernandez, editors, *Critical Systems Development in UML (CSDUML 2003), Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report (2003), pp. 63–78.

[85] Varró, D., S. Varró-Gyapay, H. Ehrig, U. Prange and G. Taentzer, *Termination analysis of model transformations by petri nets*, in: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro and G. Rozenberg, editors, *ICGT*, Lecture Notes in Computer Science **4178** (2006), pp. 260–274.

[86] Vieira, A. and F. Ramalho, *A static analyzer for model transformations*, in: *3rd International Workshop on Model Transformation with ATL*, Zurich, Switzerland, 2011.

[87] Wang, J., S.-K. Kim and D. A. Carrington, *Verifying metamodel coverage of model transformations*, in: *ASWEC* (2006), pp. 270–282.

[88] Wielemaker, J., T. Schrijvers, M. Triska and T. Lager, *SWI-Prolog*, CoRR **abs/1011.5332** (2010).

[89] Wimmer, M., G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck and W. Schwinger, *Right or wrong? - verification of model transformations using colored petri nets*, in: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, 2009.