Softw Syst Model (2015) 14:1003–1028
DOI 10.1007/s10270-013-0358-0

REGULAR PAPER

A survey of approaches for verifying model transformations

Lukman Ab. Rahim · Jon Whittle

Received: 22 January 2012 / Revised: 20 March 2013 / Accepted: 28 May 2013 / Published online: 13 June 2013

Abstract As with other software development artifacts, model transformations are not bug-free and so must be systematically verified. Their nature, however, means that transformations require specialist verification techniques. This paper brings together current research on model transformation verification by classifying existing approaches along two dimensions. Firstly, we present a coarse-grained classification based on the technical details of the approach (e.g., testing, theorem proving, model checking). Secondly, we present a finer-grained classification which categorizes approaches according to criteria such as level of formality, transformation language, properties verified. The purpose of the survey is to bring together research in model transformation verification to act as a resource for the community. Furthermore, based on the survey, we identify a number of trends in current and past research on model transformation verification.

Keywords Model transformations · Verification · Survey

Communicated by Dr. Jeff Gray.

The research described in this paper was part of a Ph.D. research project sponsored by Universiti Teknologi PETRONAS, Lancaster University and an ORSAS grant (Overseas Research Students Award Scheme).

L. Ab. Rahim (✉)
Department of Computer and Information Sciences,
Universiti Teknologi PETRONAS, Bandar Seri Iskandar,
31750 Tronoh, Perak, Malaysia
e-mail: lukmanrahim@petronas.com.my

J. Whittle
School of Computing and Communications, InfoLab21,
Lancaster University, Lancaster LA1 4WA, UK
e-mail: j.n.whittle@lancaster.ac.uk

1 Introduction

In model-driven development (MDD), instead of focussing their efforts on constructing code, developers build models and, in particular, create model transformations that transform these models into new models or code. As with other software development artifacts, however, model transformations are not free from bugs and, thus, they must be verified and validated.

Verifying a transformation that transforms one artifact into another is fundamentally more complex than verifying an individual artifact itself [11,38]. This is true whether the artifact is a model or a program.[1] To illustrate why transformation verification is more difficult than artifact verification, consider the case when the artifact is Java code and the transformation is from a UML model to Java code. Now, consider the challenges in verifying the artifact (i.e., a program) versus verifying the transformation. We will consider the special case when verification is done via testing. When testing a program, inputs and outputs are typically values or sequences of events. Although there are many difficult challenges when testing a program, checking whether a test output matches the expected output is usually relatively straightforward because it is just a case of comparing values or event sequences. In testing a model-to-code transformation, however, the output is a program, not a value or event sequence. A program is a complex data structure with behavior and so comparing the output program with an expected program is non-trivial because one has to show that the output and

[1] Arguably, a program is a type of model, and, in fact, we would agree with this view; the difference lies in the level of abstraction. Here, we make the distinction only to emphasize that there are two separate research communities—on model transformation and program transformation—but that they both face similar challenges in terms of verification.

expected programs have the same semantics. The situation is similar when verifying model-to-model transformations.

Many researchers therefore argue that specialist techniques are required [11,57] to verify model transformations due to challenges that are specific to transformations. For example,

1. In the case of testing model-to-model transformations, there are two key difficulties. Firstly, the input models for test cases must be described according to the metamodel of the source modeling language. This makes it challenging to generate test cases automatically since state-of-practice metamodels are highly complex. Furthermore, a naive algorithm to generate test cases according to a given source metamodel will produce many irrelevant or inconsistent test models; therefore, a test case generator must be highly intelligent. Secondly, it is challenging to compare the generated target model for a given test case with the oracle. Both the oracle and the target are models. Comparing the oracle and target can be done either syntactically—in which case, the comparison algorithm must compare two graphs (since a metamodel instance can be seen as a graph), which boils down to the graph isomorphism problem, which is NP-complete—or semantically—in which case, the comparison algorithm must have deep knowledge of the semantics of the target language.
2. In the case of more formal verification approaches, such as formally checking a specification of the rules of a model transformation, the challenge is scalability since an industrial model transformation will include hundreds of rules. In this case, the problem is analogous to compiler verification, which is a long-standing problem in computer science (cf. Hoare's [41] characterization of a verifying compiler as a grand challenge for computing research).
3. In addition to these challenges, the diversity of model transformation languages that are used in practice brings further issues. There is no universally accepted language for writing model transformations and so a verification approach that can be generally applied needs to be independent of the model transformation language used in particular cases. This is clearly difficult for formal approaches that verify transformation rules.

Given the challenges of verifying model transformations, recent years have seen a concerted effort to provide new approaches that are tailored to the specifics of model transformation verification. This paper surveys current approaches for verifying model transformations and classifies them along a number of dimensions. We present both a coarse-grained classification based on the technical approach (e.g., testing, theorem proving, model checking) and a finer-grained classification which categorizes approaches according to criteria such as level of formality, transformation language, properties verified. The purpose of the survey is to bring together research in model transformation verification to act as a resource for the community and to identify a number of trends in current and past research on model transformation verification.

Before continuing, we wish to clarify terminology used in the paper. First of all, there is some confusion in the literature over the definitions of verification and validation. We follow Boehm [14] and consider verification as building the thing right and validation as building the right thing. Validation, therefore, is concerned with getting a transformation's requirements correct, which is a general software engineering issue. Verification, on the other hand, is concerned with checking the properties of model transformations (here, we include property preservation from source to target). The survey covers only those approaches which address transformation verification, according to this definition. We note, however, that we have included approaches that follow our definition of verification even if their authors have called their approach validation.

Secondly, we make a distinction in the paper between formal and informal approaches to transformation verification. Although we do not claim that there is a clear boundary between these terms, generally speaking, we refer to approaches as 'formal' if they aim to prove properties of a model transformation using the formal methods of mathematics and aim to guarantee that the properties hold in all cases. On the other hand, we refer to an approach as 'informal' if a property can only be guaranteed to hold in a sample of all possible cases. In practice, this means that approaches based on theorem proving and model checking are classified as formal, whereas approaches based on testing, metrics, and inspection are classified as informal. Note that there is a recent survey on formal verification of model transformations [4], and this survey only discusses three aspects of transformation verification, i.e., verification techniques, types of transformations, and types of properties. This paper covers more aspects (e.g., complexity and tooling) and also includes informal verification.

Thirdly, discussion of transformation verification relies on a notion of what it means for a transformation to be 'correct.' In the software engineering field, a program is correct if it meets its requirements. Since a transformation is normally implemented as a program, this definition also applies to model transformations. However, different model transformations have different requirements thus creating a sense that there are different aspects of correctness, and different verification approaches have chosen to focus on different aspects depending on the requirements the transformation is supposed to fulfill. In this paper, we therefore distinguish between different notions of correctness. We say that a transformation satisfies the *type-correctness* property if its target models conform to the abstract syntax of the target language. A transformation is said to be correct with respect to *static semantics* if the target models satisfy the well-formedness constraints of the target metamodel. A transformation is correct with respect to *dynamic semantics* if the target models preserve a given property of the source model (these could be domain properties of the source model such as security, application-specific properties, or properties relating to the semantics of the source modeling language, e.g., run-to-completion semantics for UML state machines). A transformation can also be deemed correct if the target model contains the expected target elements corresponding to the source elements in the source model (e.g., for a UML class diagram to entity relationship diagram transformation, the transformation can be said to be correct if it can generate an entity for each class in the class diagram). We refer to this aspect of correctness as *correspondence correctness*. We also acknowledge that some approaches focus on key properties of the transformation itself—such as termination, confluence of transformation rules, and executability. We call these *semantics of model transformation* properties. These definitions of correctness properties will be revisited in the classification in Sect. 3. Note that these definitions come from the literature on model transformation—in our survey of existing verification approaches, each approach defined correctness in one or more of these ways.

In terms of scope, the authors strictly limit the survey to verification of transformations and, in particular, exclude research concerned only with the verification of source and target artifacts (e.g., models), but which does not consider transformations of those artifacts. Work on both model-to-model transformations and model-to-code transformations is included. The authors, however, do not include code-to-code transformation because including this type of transformation implies a much broader remit than intended by this paper.[2]

The paper is organized as follows. The next section discusses state-of-the-art approaches in verifying model transformations. The approaches are categorized in broad terms, according to whether they are testing-based approaches or apply theorem proving or model checking. Section 3 classifies the approaches using finer-grained criteria such as the level of formality, the transformation languages supported, the transformation paradigms supported, the properties verified. This section also identifies common trends in research based on this classification. The paper concludes in Sect. 4 where all the approaches mentioned in Sect. 2 are summarized and gaps in the research are highlighted.

[2] Again, program transformation is a separate research community, and it would be too complex to include all research on program transformation in this paper.

2 Description of existing research

This section describes existing approaches to model transformation verification. The approaches are first classified according to a broad-brush criterion, namely the main technical approach applied: testing, theorem proving, model checking, and graph transformations. Section 3 will present a much finer-grained classification. To make it easier to cross-reference each work, we label each approach in the text using a simple tag. These tags are given in bold and in square brackets.

2.1 Testing model transformations

There are two significant challenges when trying to apply testing approaches to model transformations [11]. Firstly, testing generally tries to automate the generation of test cases because of the complexity involved. For model transformations, however, test cases (henceforth, test models) are complex structures with data and behavior, which must conform to constraints defined in the source metamodel. Generating realistic test models automatically and efficiently is non-trivial. Secondly, comparing test oracles to test outputs is also challenging because it requires a comparison of two models. Sophisticated comparison techniques are required because simple ones will fail if the models are syntactically different even though they may be semantically equivalent.

2.1.1 Generating test models

Approaches to the automatic generation of test models adapt standard testing techniques, which try to generate test cases with guaranteed coverage. For test models, an appropriate notion of coverage is metamodel coverage—that is, each source metaclass should be instantiated at least once in at least one test model and, furthermore, properties of metaclasses (e.g., meta-attributes) should take several representative values. Wang et al. [103] propose a definition of metamodel coverage based on MOF **[WangRules]**. The intuition is that since all modeling languages used in model-driven architecture (MDA) are defined in MOF, metamodel coverage can be defined at the MOF level. In particular, metamodel coverage is defined in terms of core MOF structural concepts—feature, inheritance, and association. Hence, metamodel coverage is achieved if feature, inheritance, and association coverage are also achieved. A similar observation is made in [34] (where, in fact, EMOF is used) **[FleureyEMOF]**. The same technique to achieve metamodel coverage can also be performed on other metamodels as demonstrated in [35].

Well-known techniques for achieving coverage, e.g., equivalence partitioning, can be applied in the context of model transformations. For example, Fleurey et al. [35] apply category-partition testing to decompose a source metamodel into equivalence classes and then choose representative test models from each equivalence class **[FleureyCBT]**. Category partitioning is applied by manually identifying equivalence classes of all possible values of a meta-attribute. A tool will then create a test model for each equivalence class and select a representative value from the equivalence class as the value for the meta-attribute.

Partitioning is also used in an approach by Wang et al. [104] **[WangRules]** and Stürmer et al. [90] **[StürmerFramework]**. Wang provides tool support for testing transformation rules written in Tefkat [67] using category-partition testing based on ideas from [35]. Stürmer follows a similar approach but uses a related partition testing technique called the classification-tree method.

A simple application of equivalence partitioning is not without its problems. Firstly, it will generate a very large number of test models. Secondly, as pointed out in [35], a transformation may be intended to work only on a subfragment of the source metamodel. In such cases, it is ineffective to generate test models from the entire metamodel. Fleurey et al. [35] therefore introduce the notion of an 'effective metamodel,' i.e., the fragment of the source metamodel actually acted on by a transformation **[FleuryCBT]**. An effective metamodel is the domain of a partial transformation on the whole source metamodel. It is shown that the effective metamodel can be computed automatically by including every type, attribute, and association referred to by the transformation specification (i.e., metamodel, OCL constraints, and transformation pre-conditions). The effective metamodel can be computed either by examining the transformation specification (in [35], OCL preconditions in transformation specifications are used for this purpose) or by statically analyzing the implementation code of the transformation (an extension of type checking). Another similar approach is proposed by Lamari [60]. In [60], the effective metamodel is computed from transformation specifications written in a specialized transformation language, MTSpecL **[LamariSpec]**. Zelenov [108], on the other hand, uses category partitioning (inspired by the **[WangRules]** approach) to generate effective metamodels **[ZelenovEffMeta]**.

While the notion of effective metamodel does indeed reduce the size and number of test models generated, it is sensitive to the correctness of the transformation implementation. For example, if the transformation implementer forgets to handle hierarchical states, test models for hierarchical states may not be generated. One further problem with generating test models in this way is that the test models may not be comprehensible by testers since they merely respect the coverage criteria and will not generally correspond to intuitively meaningful models. Fleurey et al. [35] therefore argue that an interactive, rather than fully automatic, approach to test model generation is preferable. In this case, testers first provide an initial—intuitive—test model. This is then perturbed by a tool to generate a number of similar models that are both comprehensible and guarantee coverage. These perturbations are provided by applying mutation operators (using mutation-testing techniques).

Mottu et al. [70] define a set of mutation operators which are used in the Omogen tool to evaluate metamodel coverage in this way [18,71] **[MottuMutation]**. Mutation operators modify a transformation definition by introducing faults. If a set of test models cannot detect the fault introduced, then the test models are incomplete and should be updated. *Collection filtering change with deletion* is one of the Mottu's mutation operators. It applies to a transformation rule that filters a collection and purposely deletes filtering, so that the rule applies to the entire collection. For example, a rule might apply only to simple states. The mutant rule would instead apply to any kind of state. If a test model set cannot detect the error introduced by the mutant rule, then coverage is insufficient.

An interesting application of this mutation-based approach is presented in [35], where a bacteriologic technique is used to optimize the generation of test models **[FleuryCBT]**. The approach consists of an iterative algorithm which starts with an initial set of test models and a fitness function (a set of rules specifying which meta-elements and meta-attributes need to be covered during verification) based on metamodel coverage. The current set of test models is mutated and then re-evaluated according to the fitness function. This process is repeated until an 'optimal' set of test models has been generated. Other approaches that use mutations are [29] and [59]. Darabos [29] relies on mutation operators derived from a fault model of typical faults encountered when writing graph transformations **[DarabosFramework]**. Küster et al. [59] mutate test models created from declarative specifications of transformations. A given transformation specification (similar to a QVT relational specification) is used to generate a number of test models by replacing child metaclasses by their siblings **[KüsterWhiteBox]**. For example, a transformation specification acting on instances of state would lead to test models for state and Pseudo-state.

Sen et al. [85] introduce a somewhat unique way of generating test models based on the Alloy constraint solver **[SenAlloy]**. Four sources of knowledge are taken into account—the source metamodel (in Ecore and OCL), transformation pre-conditions (in OCL), partitions on the input domain (expressed in a dedicated language), and transformation post-conditions (post-conditions are called test model objectives in [85] and they are expressed in Alloy). The approach translates and integrates each of these knowledge sources into a common formalism—Alloy's first-order relational logic—and then relies on Alloy to generate test models based on these four knowledge sources. The Alloy tool has been shown to be effective at generating instances of a relational specification by translating the specification into a Boolean satisfiability problem which can be solved using a SAT solver. Sen et al. [86] also experiment with partition analysis to improve the result of their approach. The scalability of Alloy remains a question, as does, therefore, the scalability of Sen's tool, Cartier.

To summarize this section, generation of test models has received a lot of interest in the literature. Many techniques have been applied to automatically generate test models, and these techniques primarily focus on producing meaningful test models that provide extensive test model coverage.

2.1.2 Developing test oracles

A second major challenge in testing model transformations concerns test oracles. There are two issues. Firstly, Where do the oracles come from? Secondly, How to compare the result of a transformation with the oracle?

Mottu et al. [72] provide a summary of types of oracles that could be used in testing transformations. Each requires a different level of input from the user and has been implemented on a sample transformation as a point of comparison **[MottuOracle]**. These categories provide a good overview of alternatives when choosing an oracle: (1) a reference transformation, wherein a known-to-be correct transformation is referred to which has the same functionality as the transformation under test (e.g., a declarative version of it); (2) an inverse transformation, whereby $t \circ t^{-1}$ (for a transformation under test t) is checked to be the identity transformation; (3) expected output model, wherein the user (manually) provides the expected result of the transformation; and (4) constraints (see below).

The most common of these in the literature are (3) and (4). In the latter case, researchers have looked at checking transformation outputs against pre-existing sources of knowledge given as constraints, e.g., post-conditions of transformations or invariants on the output language. This is the approach taken, for example, in [11], which uses a modified OCL to specify constraints **[BaudryContracts]**. Checking post-conditions of transformations is self-explanatory. As for invariants on the output model language, these could be simply the well-formedness constraints of the target metamodel but, more generally, can be additional invariants that restrict the output metamodel in some way. For example, Whittle and Gajanovic [105] discuss such an approach in the context of generating code from NASA-relevant domain-specific models. In this case, there were known properties regarding the expected structure of the generated code. By capturing these properties as OCL output invariants (which go beyond simply well-formedness constraints), outputs of the code generator can easily be checked for compliance to an expected structure. A similar approach is taken in [54] except that a dedicated language—the Epsilon Comparison Language (ECL)—is used to specify constraints between a source model and a target model **[KolovosECL]**. For example, an ECL rule could be written to check that a state instance in the source model corresponds to a Java class with the same name in the generated code. The use of ECL allows specification of constraints across two different metamodels. This is also supported in Cariou's work [22] **[CariouContracts]**. Lano and Clark [63] use a similar approach. They propose a set of constraints to verify syntactic and semantic correctness of model transformations **[LanoConditions]**.

The constraints mentioned in the approaches above are expressed in the form of rules, specified in OCL or other rule-based languages such as ECL. Another way of expressing constraints is by using graph patterns that define expected combinations of metamodel instances. Model transformations can then be verified by checking whether a target model matches certain expected patterns. In standard graph rewriting approaches, graph rules are defined using patterns: patterns on the left-hand side of a rule contain variables which match concrete graph instances; if there is a match, the matched instance is rewritten to the pattern on the right-hand side of the rule. This type of oracle has been used by Orejas and Wirsing [76] **[OrejasPattern]**. Properties to be verified can be expressed as patterns (containing variables) to match against in the source and target models. For example, a pattern might specify that every persistent class, if it does not have a parent class, must be transformed into a table with the same name as the class. Essentially, this is very similar to the ECL approach above since it defines a declarative constraint between source and target which can be checked. The difference is in the expressiveness of the language used to define the constraints; graph patterns are typically very expressive and can, for example, include negative application conditions specifying the absence of certain metamodel instances.

To summarize, there are two major issues with checking outputs using constraint-based oracles. Firstly, constraints only support partial verification since they can only verify that the target model meets certain conditions or patterns. The language to write the conditions and patterns is typically not expressive enough to verify properties such as preservation of semantic properties. Secondly, the developer must provide the constraints, which themselves may be erroneous. A more direct approach is for the developer to produce expected output models (oracle type (3) above). This approach requires even greater effort from the developer as an expected output model must be given for each test model. Furthermore, there needs to be a sophisticated comparison algorithm to check the transformation's result against the expected output for both syntactic and semantic equivalence.

2.1.3 Testing frameworks

For completeness, we briefly mention work on testing frameworks, i.e., infrastructure designed to support the testing approaches described earlier. Frameworks for testing model transformations have been developed by Giner and Pelechano [37], Darabos et al. [29], Lin et al. [68], and Stürmer [90]. The frameworks in [29] **[DarabosFramework]** and [68] **[LinFramework]** are very similar. Both have three main components: test case constructor, testing engine, and test analyzer.

The framework by Lin et al. requires a tester to provide a test specification (i.e., a document specifying the transformation rules being tested, the input and output models, and the oracle). A similar approach is taken by Giner and Pelechano [37] where the difference is in the transformation engine used (Lin et al. uses the C-SAW model transformation engine, while Giner and Pelechano use EPSILON [55]). The verification framework proposed by Giner and Pelechano also requires the tester to provide a specification document that includes a description of the test model (in HUTN [55]) and the oracle (in EVL [55]). The EPSILON tool [55] takes the HUTN description to generate test models and the EVL script to verify the output model generated by the model transformation **[GinerTDD]**. The weakness of both frameworks is that the frameworks do not consider coverage in generating test cases; thus, there is no guarantee that the frameworks are able to detect all the errors in the model transformation. Both frameworks only test the model transformation based on the test models provided by the testers.

The framework by Darabos et al. provides more coverage support than the frameworks by Giner and Pelechano, and Lin et al. When generating test models, the framework by Darabos et al. uses mutations to ensure wider test coverage (this method is explained in Sect. 2.1.1). With regard to the errors that can be detected, all three frameworks are able to identify missing elements (i.e., the generated model does not have elements that are expected), additional elements (i.e., the generated model has additional elements that should not have been generated), and value differences (i.e., attributes in the target model have different values than the ones expected). The method used by the testing engine to detect errors is the same for all three frameworks, which is to compare the generated model with the expected model. **[LinFramework]** provides a mature test analyzer where the test results are highlighted in the target model. **[DarabosFramework]** and **[GinerTDD]** do not discuss their test analyzer in great detail; thus, the authors assume that the test analyzers are still being researched.

A somewhat different approach is taken by Stürmer. Instead of comparing the result of the transformation with an expected output, this framework compares a simulation of the test model with the results of executing the generated model **[StürmerFramework]**. This implies, of course, that the framework only works when the target models are executable. But it does have the advantage that the framework can detect dynamic errors (i.e., errors that can be found only by executing the model). In contrast, the frameworks by Giner and Pelechano, Lin et al., and Darabos et al. only consider static errors.

As a summary, research on testing frameworks focuses on end-to-end tooling for testing transformations. However, a stable end-to-end tooling framework for testing model transformations does not yet exist.

2.2 Theorem proving

In general, testing provides only partial guarantees of correctness. This is why a long-standing challenge in computer science has been to formally prove the correctness of compilers. Despite notable efforts toward this goal (e.g., [19]), formal verification of compilers has not scaled to modern-day compilers which perform complex optimizations. Moreover, whereas the traditional challenge was to prove the equivalence of source code and generated machine code, for model transformations, the challenge is harder still, since the task is to show equivalence of models generated from other models. For realistic modeling languages such as UML, there may be significant ambiguity about the semantics of the source and target languages, which makes formal proof impossible.

Nevertheless, the challenge of formally proving the correctness of model transformations is still an important one since code generated from models will inevitably be used in mission- and safety-critical applications. Code for these kinds of applications must undergo thorough certification processes (e.g., DO178B [82]) to ensure that the code can be trusted. Current certification authorities only allow the use of code generators if the generators themselves have been qualified according to the same processes. The authors know of only one model-to-code generator currently qualified according to such standards (the KCG code generator in the SCADE suite [23]).

There is a body of work in the literature which specifically aims at achieving formal proofs for model transformations. These proofs may not necessarily yield full behavioral correctness but do, in many cases, provide limited formal guarantees which can complement testing-based approaches. In general, theorem provers are used to verify transformation rules or to verify the model transformation by proving properties of the generated output models. The first technique is called *direct* verification, and the second technique is called *indirect* verification. Direct verification approaches verify a transformation's rules expressed in some language. In contrast, indirect verification approaches verify the output of applying the transformation. Both approaches have advantages and disadvantages. For indirect verification, the transformation can never be guaranteed to be fully correct; it is only guaranteed correct for cases where the transformation has been executed. However, this is usually sufficient because a transformation itself is rarely deployed; rather, it is the output of a transformation that is deployed. In addition, it is an order of magnitude less complex to verify the output of a transformation rather than the transformation itself because the transformation rules must take care of all possible ways of generating output models. On the other hand, indirect verification means that a transformation cannot be packaged and then applied without an accompanying verification of the output. Direct verification does provide such a possibility, but at the expense of increased complexity of verification. Furthermore, the type of properties that can be proved using direct and indirect approaches differ—for example, confluence can only be proven from the transformation rules directly while reusability of the target model can only be determined by analyzing the target model. These two methods of verification therefore complement each other.

Both direct and indirect approaches can be further classified into certification and non-certification approaches. Certification approaches not only verify model transformations but also provide checkable evidence that the verification was performed correctly. Typically, this evidence is a human-readable or machine-checkable proof. Not all methods that use theorem provers for verification are certification approaches. We only classify an approach under certification if a proof is produced by the theorem prover that is in a form suitable for independent validation. This means that the proof must be either human readable or machine checkable by a formally verified proof checker. We also note at this stage that the definitions of indirect versus direct and certification versus non-certification also apply to model checking–based approaches. Such approaches are discussed in Sect. 2.4.

2.2.1 Direct approaches without certification

We first discuss direct, non-certification approaches. Recall that these are approaches that apply theorem proving to analyze the rules that define a transformation, but do so in a way that does not explicitly generate human- or machine-checkable evidence that the verification was performed correctly.

Calegari et al. [21] formalize ATL [51] transformations and some operations of the ATL transformation language (e.g., the *allInstances* operation) into a Calculus of Inductive Construction (CIC) specification **[CalegariProof]**. The CIC specification is proven using the Coq theorem prover, and this will verify that the model transformation is able to generate a target model that meets certain invariants.[3]

[3] Calegari et al. [21] mention that their approach is for certifying model transformations. However, we classify this approach as non-certification because there is no discussion about proof certificate generation in. Our definition of certification requires more than simply using a theorem prover (Coq in this case), since the output of a theorem prover is not necessarily independently checkable.

Similarly, Stenzel et al. [88] propose a framework formalizing the QVT operational transformation and the QVT language into a specification that can be processed by the KIV system [8] **[StenzelProof]**. The framework is applied to verify Java code generators for security properties and syntactic correctness. However, the framework requires the source model to be transformed to an intermediate Java model (a model conforming to a Java metamodel in ECore). This means that the framework is rather restricted in terms of the source modeling language. Jackson et al. [47] verify syntactic correctness of model transformations by formalizing model transformations as constraint logic programming (CLP) logic programs [49] **[JacksonProof]**. The FORMULA [48] tool verifies syntactic correctness by proving that the type of each model element in the target model is a meta-element of the target metamodel. A notable contribution of this approach is that it also formalizes the MOF model. The approach can also be used in automatic test model generation, similar to the **[SenAlloy]** approach.

Asztalos et al. [6] and Cabot et al. [20] formalize invariants obtained from model transformation specifications and use theorem provers to prove properties of the model transformation. These invariants are constraints on the metamodel, conditions in the transformation (the **where** and **when** clause in QVT), pre-conditions and post-conditions of the model transformation. Cabot et al. synthesize the invariants into a specification that can be checked with a theorem prover[4] **[CabotOCL]**. The user then specifies a set of predicates (denoting a set of properties) that refers to the translated invariants. The theorem prover checks these predicates to prove properties of the model transformation. Asztalos et al. [6] **[AsztalosADL]** propose a similar technique but create a new language called the assertion description language (ADL) to write these assertions. The ADL invariants are identified automatically using the VMTS tool. Asztalos et al. also propose a few deduction rules to derive new invariants from the initial invariants using the SWI-Prolog tool. These invariants are added into the control-flow graph of the model transformation, which is created by VMTS, and checked using the same tool.

The approach by Cabot et al. verifies many properties of the model transformation such as termination, determinism, and executability (the reader is referred to [20] for the full list of properties and their definitions). The expressiveness of the ADL also allows the **[AsztalosADL]** approach to verify the same properties as the **[CabotOCL]** approach.

In both approaches **[CabotOCL]** and **[AsztalosADL]**, the model transformation specification is written in a textual language such as OCL. An approach by Lano and Rahimi [64] uses UML models and OCL statements (that is, a combina-

[4] Cabot et al. mention that constraint solvers and SAT solvers can also be used with their approach.

tion of a graphical and textual language) to create a specification of a model transformation. Existing methods are then used to translate the UML models and OCL statements into a formal B specification. UML class diagrams are used in specifying transformation rules where a set of transformation rules is modeled as a class with common data used by the rules as attributes. The conditions and order of execution of transformation rules are modeled using an activity diagram. OCL is used to specify the pre- and post-conditions of each transformation rule and constraints on the model transformation as a whole. The translation of UML models and OCL statements to B is performed automatically using a tool, UML Reactive System Development Support (UML-RSDS) **[LanoB]**. This approach is used to verify syntactic and semantic correctness, uniqueness, and confluence (readers are referred to [64] for the definition of these properties).

The research described above verifies a formalized declarative specification of a model transformation. This is of course an advantage because it simplifies the task of formalizing the transformation. However, a potential criticism is that the declarative specification may not match the operational definition of the transformation. Hence, these approaches require a further verification task to be completed, which is to verify that the operational and declarative definitions match. None of these approaches address this additional verification task.

As a summary, direct verification approaches using theorem proving have applied many types of theorem provers to verify model transformations written in both declarative and operational languages. These approaches also require the model transformation to be translated into a formalism that can be proved by the theorem prover. So far, none of the approaches discuss the verification of the translation mechanism.

2.2.2 *Direct approaches with certification*

This subsection discusses direct, certification approaches, that is, approaches which verify the transformation rules directly but also generate checkable evidence that verification was carried out correctly. Certification approaches are different in applicability and cost than non-certifying approaches. This kind of certification is important in practice because safety-critical application developers will not necessarily trust a verification task just because a theorem prover was applied; after all, there are other elements in the verification that cannot be trusted such as the formalization of the language, and theorem provers can contain bugs too. By producing evidence that can easily be checked by either a human expert or an external tool, the confidence in the verification task is increased. However, this confidence comes with a price because all elements of the approach (e.g., the theorem prover and the translation mechanism) will also need to generate a checkable evidence.

One example of a direct, certification approach is the technique known as proofs-as-programs [43]. In proofs-as-programs, a high-level specification of a program is formulated as an existential formula. A proof of this formula must provide a 'witness' or instantiation of the existential variable. This witness is interpreted as a program and, hence, the formula's proof provides not only a program but also a proof that the program satisfies the specification. Hence, proofs-as-programs can be used to transform high-level models into programs: a model is expressed as an existential formula and the technique returns both a proof of the formula and a program which instantiates the existential variable. The proof therefore acts both as the target program and also a proof that the model and generated program are equivalent. We classify proofs-as-programs as a direct approach because proof rules are used to encode transformation rules, so the transformation rules are formal by definition. Typically, a theorem proving environment is used to specify the proof rules, and this environment will provide some correctness guarantees for the rules, such as well-formedness. In addition, the semantics of the rules can be formally proven within the theorem proving environment because the proof rules are lemmas. We also classify proofs-as-programs as a certification approach because the approach provides a proof which can be independently checked by another theorem prover. Crucially, this is different than the approaches described in Sect. 2.2.1 where a formal proof was produced for a formalization of the transformation not the transformation itself; proofs-as-programs therefore do not require an additional verification task of proving the equivalence of a transformation and its formalization.

Over the years, proofs-as-programs have been scaled up to larger programs using the power of state-of-the-art provers such as Coq [46]. Still, however, proofs-as-programs do not scale in general to industrial size problems. This is because automated theorem provers need to be used to generate the proofs; interactive or manual theorem proving is too costly. In focussed domains, however, proofs-as-programs have been demonstrated to work in practice. For proof-as-programs to succeed, the domain must be well-structured, well-understood, and tightly constrained. Otherwise, the automated theorem prover will have difficulty in generating the proof since complex domains will have too many rules, and hence, the size of the search space grows exponentially.

Researchers at NASA Ames Research Center showed the practicality of the approach if applied to domain-specific modeling languages. They developed Amphion [98], which relies on proofs-as-programs but provide an intuitive graphical interface for code generation from models of space geometry problems **[Amphion]**. In Amphion, all theorem proving activities are hidden from the user and users are not required to have knowledge of formal proof. The programs generated are corrected by construction and, in fact, human-readable explanations of proofs are available to be provided as evidence to certification authorities [106].

The concept of proofs-as-programs has been developed further for model transformations with proofs-as-model transformations [78] **[PoernomoProof]**. Generally, proofs-as-model transformations are similar to proofs-as-programs. A specification of a model transformation is given as a constraint specifying the pre-conditions and post-conditions of the transformation. This can be formally specified as a ∀∃ term, and the term is formally proven using formal tools such as PVS or Coq. From this term, a function representing the model transformation is extracted. However, the scope of [78] is limited to a single transformation rule, whereas model transformations normally consist of many transformation rules. Another work by the same author addresses this concern using partial-order specification to link rules together [79]. The partial-order specification is defined based on the order of executing the transformation rules, and where the transformation rules can be executed in different orders or when the order is not defined, the approach selects an arbitrary root element from the source metamodel and starts with the transformation rules that transform that element.

Before applying the **[PoernomoProof]** approach, one needs to formalize the source and target metamodels. In [78], the source and target metamodels are formalized into a mathematical formalism called constructive type theory [78], while in [79], the metamodels are formalized using the Calculus of Inductive Construction [21]. Therefore, the success of the **[PoernomoProof]** approach depends on the correctness of the formalized source and target metamodels. Since proofs-as-model transformations are based on proofs-as-programs, it is a valid assumption to assume that proofs-as-model transformations have the same scalability issue.

To summarize, direct approaches with certification are limited in applicability to domains that are well defined and properly structured. Research is limited to the generation of evidence for verification tasks performed by theorem provers. Research on producing evidence for the translation mechanism is still in its infancy.

2.2.3 Indirect approaches without certification

This subsection describes indirect, non-certification approaches. Recall that indirect verification approaches do not verify the transformation rules directly; rather, they verify properties of the target model generated by the transformation.

Egea and Rusu [32] verify whether a model transformation can generate well-formed target models **[EgeaProof]**. Well-formedness is verified by checking whether the target model conforms to OCL constraints of the target metamodel. In [32], OCL constraints on the target metamodel and the target model itself are translated into membership equational logic (MEL) [16] specifications. Then, using the ITP/OCL tool [26] in MAUDE, the target model is proven to be well formed if:

1. the MEL specification denoting the target model is an instance of the MEL specification denoting the target metamodel.
2. the MEL specification denoting the OCL constraints is true in the MEL specification denoting the target model.

Conformance to constraints has also been used by Baar and Marković [7] to verify model transformations **[Baar-Constraint]**. The difference between this approach and the **[EgeaProof]** approach is where the approach is applied and how properties are verified. Baar and Marković [7] apply their approach to verify semantic preservation of refactoring transformations. They achieve this by checking whether an instance of the target model conforms to the constraints on the target model. The verification is performed using an evaluation function implemented as a graph transformation.

The above approaches focus only on the target model. Some indirect approaches, however, compare the source and target models; these are still classified as indirect as they do not formalize the transformation rules themselves but formalize the semantics of the source and target language and then verify semantic preservation. Blech [13], for example, has done this for a statechart-to-Java transformation formalized within Isabelle/HOL, although the semantics of the languages has been simplified **[BlechProof]**.

Barbosa et al. [9] propose a more novel approach to verifying model transformations. Their approach extends the MDA architecture to include a semantic metamodel and a semantic model. The approach uses these models to verify semantic preservation of model transformations **[BarbosaSemModel]**. A semantic metamodel is a model describing a language to specify the semantics of a model (e.g., a metamodel of the OCL language). It also includes operational semantics—rewrite rules specifying behaviors of the model. The semantic metamodel is created manually. A semantic model is an instance of the semantic metamodel that is specific to the source/target model (e.g., OCL constraints on the source/target model). In this approach, the semantic model is generated automatically using a model transformation called a semantic equation. The approach uses a theorem prover to verify semantic preservation of model transformations by proving equivalence between a source semantic model and a target semantic model. This approach has also been used with a model checker to verify preservation of safety and liveness properties [10]. Safety and liveness properties are checked from the operational semantics of the source and target model that are generated using the rewrite rules.

To summarize, indirect approaches using theorem proving are currently focused on verifying model-to-model transformations, and these approaches require the translation of the target model into a formalism that can be handled by a theorem prover. The difference between these approaches is in how they use the theorem provers. Theorem provers are either used to prove that the model transformations can generate target models with certain properties or compare the target models with oracles.

2.2.4 Indirect approaches with certification

Indirect approaches with certification extend indirect approaches by providing checkable evidence that verification was done correctly. Typically, an automated prover is applied every time an artifact is generated from a model, and this is done in such a way that the proof can be independently checked either by a certification authority or by a trusted machine-based proof checker. In the latter case, the proof checker is typically a small kernel proof checker only a few hundred lines long, which has been formally verified. The approach reduces the complexity of verifying the transformation because the size of code generated is much smaller than the size of the code needed to implement the transformation system.

Three notable approaches along these lines are Autofilter [31], AutoBayes [84], and AutoCert [30]. Autobayes and Autofilter are similar but apply to different domains—geometric state estimation in the case of Autofilter, and data analysis problems in the case of Autobayes **[Autofilter]**. Both Autofilter and Autobayes are domain-specific model-based code generators. In each case, an automated theorem prover is applied to check properties of the generated code. In general, however, automated theorem provers do not scale. Therefore, the key idea in Autofilter and Autobayes is to use the transformation system to automatically insert annotations into the generated code which allow automated theorem proving to scale. Since the transformation system is domain specific, it therefore contains domain knowledge about the structure and behavior of the generated code. This domain knowledge can be used to simplify the theorem proving process. As a simple example, Autobayes can generate loop invariants which can then be both checked by the theorem prover but also used when checking the correctness of the generated code. Such an approach is more efficient than trying to prove the correctness of the code without the aid of the loop invariants since this would necessarily involve loop invariant discovery, which is an unsolved problem. Hence, the approach in Autobayes and Autofilter can be seen as the transformation system inserting annotations into the code which act as hints to the theorem prover and direct its search for a proof. AutoCert [30] **[AutoCert]** advocates a model-driven approach for developing the annotations.

As a summary, indirect certification approaches such as these have only been applied to model-to-code transformations. One reason for this is because certification authorities are more interested in certifying the final product (i.e., the code) rather than the intermediate products (i.e., models).

2.3 Graph theory

Since models can be represented as graphs and model transformations can also be specified using graph-based languages, a common choice to formalize transformations is to use graph theory in constructing a formal model of the model transformations. Several researchers have managed to prove properties of graph transformations by manually constructing mathematical proofs using graph theory.

Küster et al. [58] prove that a graph transformation is terminating and confluent by checking whether the graph transformation meets certain criteria **[KüsterCriteria]**, namely

1. a graph transformation is terminating if the elements in the source pattern are finite, and there are no transformation rules that add these elements in such a way that leads to infinite recursive execution.
2. a graph transformation is confluent if it contains rules that are parallel independent.

Ehrig et al. [33] prove that a bidirectional graph transformation is information preserving (identifiers of elements in the source model are preserved in the target model and vice versa) by proving that there exists an inverse of the graph transformation that can produce the source model **[EhrigInfoPreserving]**. Their approach is different than the inverse oracle proposed in [72] where this approach only needs to prove that the inverse transformation exists, while, in [72], the inverse oracle needs to be created.

Hermann et al. [40] propose an approach related to **[EhrigInfoPreserving]**. The approach proves invertibility of bidirectional model transformations. A bidirectional model transformation is invertible if the forward transformation is an inverse of the backward transformation and vice versa. The main contribution of this work is a theorem (and its proof) that defines conditions of an invertible graph transformation. The conditions are (1) the model transformation must be deterministic and (2) the model transformation must be tight (the model transformation contains transformation rules that modify elements in the source and target models) **[HermannInvertibility]**.

Although proving properties of graph transformation using this technique has been successful, Strecker [89] argues that using graph theory alone is insufficient to verify whether the target model corresponds to the source model. His argument is that, when using graph theory, complex reasoning about graph morphism is needed when the target model graph is compared with the source model graph. Therefore, he suggests that the graph patterns be translated into formulae in first-order logic and uses theorem provers (e.g., Isabelle/HOL) to prove their properties **[StreckerProof]**. This work clearly specifies the limitations of graph theory.

To summarize, graph theory approaches base their verification on the notion of a model transformation as a graph transformation. This supports the verification of properties, such as invertibility, that have not been considered by other approaches.

2.4 Model checking

Other than using theorem provers, model checkers have also been used to verify model transformations.

2.4.1 Indirect approaches

Model checking using indirect approaches means verifying model transformations by model checking target models. Varró and Pataricza [99] first transform source and target models into equivalent transition systems **[VarroMC]**. Properties which should be preserved by the transformation are translated from a source-specific representation to a target-specific representation (manually or using a model transformation) and then model checked on the both source and target transition system.[5] The approach is restricted to behavioral models which have an appropriate translation to transition systems. This work also does not address the scalability issues of model checking.

A similar approach has been taken by Staats and Heimdahl [87], who have shown reasonable scalability to around 12,000 lines of code when using model checking to verify the preservation of properties in the Simulink-to-C code generator **[StaatsMC]**. Ab. Rahim and Whittle [1] also use a similar approach to verify UML state machines-to-Java code generators. The novelty of the approach is that it can verify model transformations developed by a third party where the transformation rules are not available **[AbRahimMC]**. The approach includes an informal static analysis task that requires the user to study the generated code in order to understand how the code generator works. The result of this analysis is then used by the user to formulate assertions that would later be added into the generated code using another model transformation. These assertions capture the semantics that the code generator is supposed to preserve. A model checker is used to check the assertions in the generated code. This approach also addresses the scalability issue by using heuristics, proposed in [2], to select a state space reduction technique, which can be applied in the generated code, based on design patterns in the source model.

The approach proposed by Narayanan and Karsai [74] verifies preservation of reachability using two methods: (1) checking bisimilarity of source and target models using structural correspondence (for every element in the source model, there exists a corresponding element in the target model) and (2) model checking the target model **[NarayananMC]**.

Summarizing this section, indirect verification using model checking has focused on verifying model-to-code transformations. These approaches have also started addressing the state space explosion problem.

2.4.2 Direct approaches

The opposite of indirect approaches is direct approaches where model transformations are verified by model checking the model transformation rules or a translation of the model transformation rules into some formal language. Direct approaches using model checking have been proposed by Garcia and Möller [36], Boronat et al. [15], Varró et al. [100], Wimmer et al. [107], and Lúcio et al. [69].

Garcia and Möller [36] translate an EMOF model, OCL and the transformation rules into a +CAL specification [62] and use the TLC model checker [61] to check for well-formedness of the output model and termination of the model transformation **[GarciaMC]**. Boronat et al. [15] formalize source model, target model, and transformation rules into graph rewriting logic and model check using a model checker in MAUDE **[BoronatMC]**.

Varró et al. [100] and Wimmer et al. [107] use Petri Nets to analyze termination of model transformations. Wimmer et al. [107] verify model transformations written in the Transformation Net language, a domain specific language based on Colored Petri Nets (CPN). The model transformation is translated into CPN, and using an analyzer tool, the CPN specification is checked for termination, confluence, and correctness **[WimmerCPN]**. Termination is verified by checking whether there is a loop in the transformation that creates new elements in the source model. Confluence is checked by detecting whether there are two transformation rules that process the same element in the source model at the same time. Correctness is verified by comparing the generated target model and the expected target model where the latter is entered into the analyzer tool.

Varró et al. [100] derive a Petri Net model (how this task is performed is not clearly explained in the paper) from the model transformation. The Petri Net model is an abstraction of the model transformation. The model transformation is abstracted using the cone of influence reduction technique where only certain model elements are included in the abstraction. The model transformation terminates if it

[5] The target-specific representation of the properties being verified should be verified by domain experts.

reduces the number of elements in the Petri Net after executing a finite number of steps **[VarroPN]**. This is similar to the criteria used by Küster et al. [58].

Lúcio et al. [69] take a more traditional approach in using model checking to verify model transformations. They develop a symbolic model checker that verifies model transformations written in the DSLTrans language **[LucioMC]**. The model checker constructs a state space of the model transformation, which consists of states that correspond to a set of possible execution combinations of transformation rules. The model checker verifies a property by traversing the state space and checking whether the property holds in each path. If a property does not hold, the model checker produces a counter example in the form of a sequence of transformation rules leading to the wrong transformation result. The model checker has been successfully used in verifying a model transformation with an order of magnitude of 10^4 states. Unfortunately, the model checker used in this approach can only be used for DSLTrans model transformations. Model transformations in the DSLTrans language are structured as a graph. The language has a built-in feature where the language requires transformation rules to be encapsulated in boxes called 'layers,' and these layers are connected with each other. These layers are like nodes in a graph and the connections are the edges. The graph of connected layers is the state space being checked by the model checker. Therefore, a model checker can be developed for this language because of how the transformation rules are structured.

As a summary, the solution taken by most researchers to directly verify model transformations using model checking is similar to the one taken by researchers who use theorem proving, which is to translate the model transformations into models that can be verified by a model checker.

2.4.3 Certifying approaches

There have been some attempts to provide evidence for certification when model checking transformations.

Chaki et al. [24] use a combination of a technique called Proof-Carrying Code (PCC) [75] and model checking to generate certified software component binaries from UML statechart specifications **[ChakiPCC-MC]**. PCC is a method which instruments binaries with formal proofs of their correctness. It has been used primarily to support safe execution of untrusted code: a code receiver can check the proof before running the code. In Chaki's work, model checking is used at the specification level to derive invariants from a set of safety requirements (expressed as assertions over UML states). These invariants are then passed to a PCC-based tool which generates proof certificates. The approach is novel in that it addresses both UML-to-C and C-to-assembly transformations. However, the derivation of the specification invariants using model checking is not fully automatic and requires manual assistance in the form of explicit annotations.

A more innovative approach is to replace the theorem prover in the PCC approach with a model checker. The CVT [77] tool does exactly this for C code generation from Statemate specifications **[CVTPneuli]**. It uses a BDD-based decision procedure, called TLV, to formally check the verification conditions generated for individual C programs produced by the transformation. The verification conditions are large logical implications which, if proven true, imply a correct refinement. The approach has been shown to scale to realistic problems of a few thousands lines of code. A significant part of CVT is in appropriately decomposing the verification conditions, so that TLV can prove them. This is handled by a series of automatic decomposition procedures as well as by applying a number of standard abstract interpretation techniques (e.g., to replace integer and float variables by symbols).

One issue with using a model checker as the proof engine is how to convince a certification authority that the model checker itself has not made an error. When using a theorem prover as proof engine, this problem is solved by applying an alternative formally verified kernel proof checker that checks the proof provided by the theorem prover. Model checkers, however, are not capable of outputting a checkable certificate—a model checker either returns a counterexample in the case that a bug is found, or simply 'yes.' The notion of a certifying model checker resolves this issue by modifying model checkers to produce, in the case of success, a deductive proof which can then be independently checked **[CertifyMC]** [73].

In another line of work, Karsai and Narayanan [53] proposed two methods of certifying model transformations **[KarsaiMC]**. The first method is by establishing links between the elements in the target model to the elements in the source model. These links will then be checked using a bisimilarity checker tool to prove that the target model is a bisimulation of the source model. Other than using the bisimilarity checker, the target model is also checked for required properties using a model checker. The links and the results from the bisimilarity checker and the model checker are the proof certificate.

The second method is through 'semantic anchoring,' which requires the translation of the source and target model to an equivalent formal model that is written in the same formal language. The formal models will then be checked for bisimulation. One issue with this method is that in cases where the source and target languages contain different elements, the method can only check for weak bisimulation (i.e., checking bisimulation only for elements of the source and target languages that are semantically equal). For example, when verifying a UML class diagram to entity-relational diagram (ERD) transformation, bisimilarity between the relationships in the class diagram and the relationships in the ERD will not be checked because they are semantically not equal. Another problem with this method is the extra verification of the translation of the source and target model, although this can be performed using the first method.

Verifying and certifying model transformations using model checking have advanced to also include model-to-model transformations. However, similar to theorem proving, these approaches only cover certification of the model checking. Certification of other parts of the approach (if any) has not been considered.

2.5 Inspection and metrics

The authors conclude this part of the review by briefly mentioning work on the inspection of generated code and on metrics for model transformation. The argument made in [91] is that the inspection of auto-generated code is made easier because of the availability of models **[StürmerInspect]**. A process is presented in which it is easier to distinguish between errors produced by the code generator and errors introduced by an incorrect model—if there is confidence in the correctness of the model, then any errors in the generated code must come from the generator.

Metrics could potentially be useful in assessing non-functional qualities of model transformations, such as understandability, modifiability, modularity, consistency, and completeness [94]. For metrics to be useful in verification, these metrics are mapped to these qualities/properties. Amstel et al. [96] define these qualities as internal qualities and assessing these qualities should be performed using direct assessment (i.e., assessing the quality by looking at the model transformations and not at the quality of the generated model, which Amstel defines as indirect assessment).

Amstel has defined several metrics for model transformations such as the number of fan-ins and fan-outs of a transformation function that is used to measure the dependency between transformation functions. These metrics are used to assess ASF+SDF [5], ATL [95], and QVT [93]. For these metrics to be useful, Amstel et al. [5] relate these metrics with quality attributes based on evidence from an empirical study in which model transformation developers were asked to assess a set of model transformations, answer a questionnaire relating to the model transformations and participate in a semi-structured interview **[AmstelMetrics]**. The empirical study was conducted for six real-world model transformations by four experienced ASF+SDF users.

Vignaga [102] has also proposed a set of metrics for ATL, and there are similarities between these metrics and the metrics for ATL transformations proposed by Amstel et al. [95] **[VignagaMetrics]**. The important difference between [102] and [95] is in how they relate their metrics with model transformation qualities. Amstel et al. use empirical assessment techniques to establish this relation, while Vignaga uses intuition.

Both Vignaga and Amstel do not clearly mention how they produced the metrics. Work by Kolahdouz-Rahimi and Lano [65,80] uses the Goal–Question–Metric (GQM) [66] method to identify a set of metrics for measuring comprehensibility of model transformations **[RahimiMetrics]**. This work also investigates how model transformation languages influence the comprehensibility of a model transformation by measuring the comprehensibility of the same model tranformation written in five different languages.

Saeki et al. [83] present a preliminary approach for model transformation metrics, which is based on applying metrics to the source and target models **[SaekiMetrics]**. Using this technique, for example, it could be discovered that a transformation has a significantly negative impact on the modularity of the source model, which may imply that the modular structure of the source is not preserved in the target.

The works discussed above have proposed some general metrics and then established the relationship between these metrics and the qualities of model transformations. There are works that focus specifically on certain model transformation qualities. Kapová et al. [52] use metrics to assess the maintainability of QVT relational model transformations **[KapovaMetrics]**. The metrics are taken or modified from the metrics presented in [5,39,56,81]. The thresholds for these metrics, indicating how the value of a metric should be interpreted, have not yet been identified.

Amstel et al. [97] use metrics along with dependency analysis and metamodel coverage analysis to address the problem of maintaining model transformations. These analyses are used to understand the model transformations and raise maintenance issues such as dependency and modularity **[AmstelAnalysis]**. This paper uses two toy-like examples for its experiments. A more realistic result may be obtained by using real-world model transformations. Amstel et al. [93] have also used metrics to assess the performance of ATL, QVT relational, and QVT operational model transformations. This study not only assesses and compares the performance of the model transformation engines of the three languages but also how different input model structures and how the language constructs of the modeling languages influence performance **[AmstelPerformance]**. The threshold for measuring performance using metrics has not been identified.

It is also worth mentioning how these approaches are implemented. Amstel et al. [5,93,95,97] use tools to get the measurement value for the metrics. They use a parser to generate a model of the model transformation and develop a model transformation to get the value of the metrics. Kapová et al. [52] and Vignaga [101] also use the same method in collecting the value of some of their metrics (some metrics have to be collected manually). For the empirical study, Amstel et al. use Kendall's τ_b rank correlation test, but it is not clear whether this test is performed manually or using a tool. For the dependency analysis and metamodel coverage analysis, Amstel et al. use ExtraVis [27] and TreeComparer [42] to visualize the dependency between model transformation functions and the relationship between source model elements and the model transformation functions that operate on the model elements. Saeki et al. [83], and Kolahdouz-Rahimi and Lano [80] do not mention the tools they used in their assessment.

To summarize, research on metrics for assessing properties of model transformations has covered many properties, such as maintainability and performance, and tools have been provided to calculate values for the metrics. Interestingly, the research has also involved practitioners through empirical studies.

3 Classification

The authors now classify the approaches described according to a number of finer-grained criteria. This classification helps in identifying some trends in the research community, commonality in the proposed approaches along with their similarities and differences. The criteria used in the classification are:

1. **Technique**: whether a transformation is verified *directly* (verifying the model transformation rules) or *indirectly* (verifying properties of the generated output).
2. **Formality:** whether they are *formal* (proving properties using formal methods such as theorem proving or model checking), *semi-formal* (a combination of informal and formal methods), or *informal* (checking properties using testing, inspection, or metrics).
3. **Effort**: whether the approaches require a *high* (difficult to be achieved without tools), *medium* (easier/faster if tools exist) or *low* (can be carried out manually) degree of effort to apply.
4. **Tooling**: whether tool support is currently available—*no* (not supported), *yes* (fully supported), *not yet* (tools under development), *partial* (partially supported), or *n/a* (no tools needed).
5. **Properties**: which properties are verified for a transformation—*type correctness* (TC—elements in the target model are instances of elements in the target metamodel), *preservation of static semantics of models* (SSM—the target model conforms to well-formedness constraints of the target metamodel), *preservation of dynamic semantics of models* (DSM—the transformation preserves semantic properties), *correspondence between source and target* (C—the target model contains elements that correspond to elements in the source model), or *semantics of model transformations* (SMT—termination, confluence, executability, etc).
6. **Transformation Language**: for which transformation languages the technique is applicable.
7. **Transformation Type**: for which types of transformation the technique is applicable—*model-to-code* (M2C), *model-to-model* (M2M), or *both*.
8. **Transformation Paradigm**: for which paradigm of transformations an approach applies to, based on the categorization in [28] (e.g., operational, relational, or template-based).
9. **Input Complexity**: the amount of input data that must be provided by the user for the approach to work—*complex* requires many different types of input with complex structures and behaviors (e.g., large, complex metamodels); *simple* requires relatively simple input data.
10. **Output Complexity**: how detailed the output generated by the approaches is—some approaches give explanations along with their output (*detail*); others give just the output (*simple*).

Tables 1 and 2 classify the approaches discussed in Sect. 2 according to these criteria. It is not always possible to fully categorize approaches: a hyphen denotes that a criterion is not appropriate for a particular approach; N/S stands for not specified. Obviously, some of the criteria above require a subjective decision to classify approaches. Such decisions have been made by the authors after careful and detailed reading of the literature describing these approaches.

3.1 Trends

In this subsection, we comment on several trends in the research community which can be identified from our survey. The trends are presented according to the technical approach classification from Sect. 2, i.e., testing, theorem proving, model checking, graph theory, and inspection and metrics. Before that, there are certain global trends that appear to be true regardless of the technical approach taken.

3.1.1 Overall trends

Our first observation is that the effort needed to carry out verification tasks—in any category—is generally high. Although our rating of effort (low, medium or high) is largely subjective, the rating was obvious in all cases, and so we can be confident that all verification approaches tend toward high effort. This is not surprising for formal methods such as theorem proving or model checking; however, it is perhaps more surprising for testing-based approaches. The 'high' effort rating illustrates that, for model transformations, even testing is a labor-intensive task due to the complexity and scale of model transformations and the number of tasks involved:

Table 1 Categorization by technique, formality, effort, tooling & properties

Approach	Technique	Formality	Effort	Tooling	Properties
[FleureyCBT]	–	Informal	High	Yes	–
[FleureyEMOF]	–	Informal	High	Yes	–
[MottuMutation]	–	Informal	High	No	–
[WangRules]	–	Informal	High	Yes	–
[ZelenovEffMeta]	–	Informal	High	Yes	–
[LamariSpec]	–	Informal	High	Partial	–
[KüsterWhiteBox]	–	Semi-formal	High	No	–
[SenAlloy]	–	Semi-formal	High	Yes	–
[MottuOracle]	Direct	Informal	High	No	C
[BaudryContracts]	Indirect	Informal	High	No	C
[KolovosECL]	Indirect	Informal	High	Yes	C
[CariouContracts]	Indirect	Informal	Medium	Not yet	C
[LanoCondition]	Indirect	Informal	Medium	Yes	TC, DSM
[OrejasPattern]	Direct	Formal	High	Yes	C
[LinFramework]	Indirect	Informal	High	Partial	TC
[StürmerFramework]	Indirect	Semi-formal	High	Yes	DSM
[DarabosFramework]	Indirect	Semi-formal	High	No	TC
[GinerTDD]	Indirect	Informal	Medium	Yes	C
[Amphion]	Direct	Formal	High	Yes	DSM
[PoernomoProof]	Direct	Formal	High	Yes	TC, SSM
[BlechProof]	Indirect	Formal	High	Yes	DSM
[AutoFilter]	Indirect	Formal	High	Yes	DSM
[AutoCert]	Indirect	Formal	High	Yes	DSM
[CalegariProof]	Direct	Formal	High	Yes	TC
[StenzelProof]	Direct	Formal	High	Yes	DSM
[JacksonProof]	Direct	Formal	High	Yes	TC
[EgeaProof]	Indirect	Formal	High	Yes	SSM
[BaarConstraint]	Indirect	Formal	Medium	N/S	DSM
[AsztalosADL]	Direct	Formal	High	Yes	TC, SMT
[CabotOCL]	Direct	Formal	High	Yes	SMT
[LanoB]	Direct	Formal	High	Yes	TC, SMT, SSM
[KüsterCriteria]	Direct	Formal	High	No	SMT
[EhrigInfoPreserving]	Direct	Formal	High	No	C
[HermannInvertibility]	Direct	Formal	High	Yes	SMT
[StreckerProof]	Direct	Formal	High	Yes	C
[ChakiPCC-MC]	Indirect	Formal	High	Partial	DSM
[CVTPnueli]	Indirect	Formal	High	Yes	DSM
[CertifyMC]	Indirect	Formal	High	Yes	DSM
[KarsaiMC]	Indirect	Formal	High	Yes	DSM
[VarroMC]	Indirect	Formal	High	Yes	DSM
[StaatsMC]	Indirect	Formal	High	Yes	DSM
[AbRahimMC]	Indirect	Formal	High	Yes	DSM
[NarayananMC]	Indirect	Formal	High	Yes	DSM
[GarciaMC]	Direct	Formal	High	Yes	SSM
[BoronatMC]	Direct	Formal	High	Yes	DSM
[WimmerCPN]	Direct	Formal	High	Yes	SMT

Table 1 continued

Approach	Technique	Formality	Effort	Tooling	Properties
[VarroPN]	Direct	Formal	High	N/S	SMT
[LucioMC]	Direct	Formal	High	Yes	C
[BarbosaSemModel]	Indirect	Formal	High	Yes	DSM
[StürmerInspect]	Indirect	Informal	High	No	–
[AmstelMetrics]	Direct	Informal	Medium	Yes	SMT
[AmstelAnalysis]	Direct	Informal	Medium	Yes	SMT
[AmstelPerformance]	Direct	Informal	Medium	Yes	SMT
[SaekiMetrics]	Direct	Informal	Medium	No	DSM
[VignagaMetrics]	Direct	Informal	Medium	No	DSM
[KapovaMetrics]	Direct	Informal	Medium	Partial	DSM
[RahimiMetrics]	Direct	Informal	Medium	No	DSM

generating test models, preparing oracles, executing the test cases, etc. One possible consequence of this for the research community is to focus more attention on test automation and specifically on reducing the amount of effort in testing transformations. To date, most research has focussed on getting the 'basics' of testing right: test models, oracle checking, etc. These are necessary first steps and much have been achieved; now is perhaps the time to specifically address the level of effort required. Level of effort is usually only considered implicitly or as an after thought. In particular, although some aspects of testing model transformations are always going to be labor intensive, there could be more work on reusing test models and on test model prioritization. These are well-studied areas in testing more generally but have not yet been fully researched in the context of transformations.

A similar story can be seen when looking at the inputs required by the verification approaches; in all cases, we have classified them as 'complex.' This is perhaps to be expected since the artifact being verified is a model transformation, thus the inputs are either models (source models and meta-models), the model transformation itself (including its specification and constraints), or both. One interesting analysis to consider in the future would be to unpick where are the points of essential and accidental complexity when verifying model transformations. Brooks [17] makes the distinction between complexity which is inherent and unavoidable (essential) and complexity which is introduced as a result of the software engineering approach. Arguably, approaches to verifying model transformations could get a better handle on how to manage complexity if the points of essential and accidental complexity were explicitly highlighted.

In general, there appears to be no clear trend as to which transformation languages and paradigms are preferred in model transformation verification. A wide variety of transformation languages and paradigms have been considered, and there is no obvious dominant language or paradigm around which the research community has coalesced. This may speak to the relative immaturity of the field, in that no single transformation language has received universal acceptance, or it may simply speak to the fact there is no one-size-fits-all solution in terms of transformation language/paradigm. This level of diversity is consistent with a recent study on industrial use of model-driven engineering (MDE), where over 100 MDE tools were seen to be in use in industry [44,45]. One can identify a trend within the technical approaches, however. Generally speaking, the direct theorem proving approaches consider declarative transformation languages. This is because of the inherent complexity of verifying operational languages. In contrast, testing approaches have been applied to both declarative and operational paradigms.

Interestingly, a majority of the research reported in this paper has addressed model-to-model transformations. Indeed, 68 % of the approaches verify model-to-model transformations (only four of the approaches address both model-to-model and model-to-code). This is a pleasing observation as it shows that the modeling community has taken on the challenge of how best to transfer existing verification methods to the case when the target is a model.

3.1.2 Trends in testing approaches

The general approach taken by researchers in testing model transformations is to transfer standard testing techniques from program testing to transformation testing. This makes sense and is the reason why we see significant efforts toward generation of test cases, test case coverage, and oracle checking. As noted above, however, not all areas of testing have yet been transferred to transformations. Test case reuse and prioritization are notable exceptions.

There have been two key challenges that have been addressed in testing model transformations: automation of test case generation and oracle checking. For the former, the

Table 2 Categorization by language, paradigm, type, input & output

Approach	Language	Paradigm	Type	Input	Output
[FleureyCBT]	N/S	N/S	M2M	Complex	–
[FleureyEMOF]	N/S	N/S	M2M	Complex	Simple
[MottuMutation]	Tefkat, Java & Kermeta	Operational & relational	M2M	Complex	–
[WangRules]	Tefkat	Relational	M2M	Complex	Detail
[ZelenovEffMeta]	N/S	GT	M2C	Complex	–
[LamariSpec]	N/S	N/S	M2M	Complex	–
[KüsterWhiteBox]	Java	Operational	M2M	Complex	Simple
[SenAlloy]	N/S	N/S	M2M	Complex	–
[MottuOracle]	N/S	N/S	M2M	Complex	Simple
[BaudryContracts]	N/S	N/S	M2M	Complex	Simple
[KolovosECL]	ECL	Relational	M2M	Complex	Simple
[CariouContracts]	N/S	N/S	Both	–	–
[LanoCondition]	QVT	Relational	M2M	Complex	Simple
[OrejasPattern]	Triple algebras	GT	M2M	Complex	Simple
[LinFramework]	CSAW	Operational	M2M	–	–
[StürmerFramework]	N/S	GT	M2C	Complex	–
[DarabosFramework]	N/S	GT	M2M	Complex	–
[GinerTDD]	Epsilon	Relational	M2M	Complex	Simple
[Amphion]	N/S	N/S	M2C	Complex	Detail
[PoernomoProof]	N/S	N/S	M2M	Complex	Detail
[BlechProof]	N/S	N/S	M2C	Complex	Detail
[AutoFilter]	Prolog	Template	M2C	Complex	Detail
[AutoCert]	N/S	N/S	M2C	Complex	Detail
[CalegariProof]	ATL	Relational	M2M	Complex	Detail
[StenzelProof]	QVT	Operational	M2C	Complex	Detail
[JacksonProof]	N/S	N/S	M2M	Complex	Detail
[EgeaProof]	N/S	N/S	N/S	Complex	Detail
[BaarConstraint]	N/S	GT	M2M	Complex	Simple
[AsztalosADL]	VMTS	N/S	M2M	Complex	Simple
[CabotOCL]	QVT/AGG	Relational/ GT	M2M	Complex	Detail
[BarbosaSemModel]	ATL	Declarative	M2M	Complex	Detail
[LanoB]	UML-RSDS	Declarative	M2M	Complex	Detail
[KüsterCriteria]	AGG	GT	M2M	Complex	Detail
[EhrigInfoPreserving]	TGG	GT	M2M	Complex	Detail
[HermannInvertibility]	TGG	GT	M2M	Complex	Detail
[StreckerProof]	AGG	GT	M2M	Complex	Detail
[ChakiPCC-MC]	N/S	N/S	M2C	Complex	Detail
[CVTPnueli]	N/S	N/S	M2C	Complex	Detail
[CertifyMC]	N/S	N/S	M2C	Complex	Detail
[KarsaiMC]	GReAT	GT	Both	Simple	Simple
[VarroMC]	VIATRA	GT	M2M	Complex	Detail
[StaatsMC]	N/S	N/S	M2C	Complex	Detail
[AbRahimMC]	Any	Any	M2C	Complex	Detail
[NarayananMC]	GReAT	GT	M2M	Complex	Detail
[GarciaMC]	N/S	N/S	M2M	Complex	Simple
[BoronatMC]	QVT	Relational	M2M	Complex	Detail
[WimmerCPN]	TN	Relational/ operational	M2M	Complex	Detail

Table 2 continued

Approach	Language	Paradigm	Type	Input	Output
[VarroPN]	N/S	GT	M2M	Complex	Simple
[LucioMC]	DSLTrans	Relational	M2M	Complex	Detail
[StürmerInspect]	N/S	N/S	M2C	Complex	Detail
[AmstelMetrics]	ASF+SDF, ATL, QVT	Relational/ operational	M2M	Complex	Simple
[AmstelAnalysis]	ATL, QVT, Xtend	Operational/ template	Both	Complex	Complex
[AmstelPerformance]	ATL, QVT	Relational/ operational	M2M	Complex	Complex
[SaekiMetrics]	N/S	N/S	Both	Complex	Simple
[VignagaMetrics]	ATL	Declarative/ imperative	M2M	Complex	Simple
[KapovaMetrics]	QVT	Relational	M2M	Complex	Simple
[RahimiMetrics]	QVT, Kermeta, VIATRA, ATL, UML-RSDS	Relational/ operational	M2M	Complex	Simple

key advances have been in (1) generating intuitively meaningful test models and (2) defining coverage criteria based on metamodel coverage. In particular, the notion of an effective metamodel in coverage seems to be highly effective. In terms of oracle checking, the approaches have principally been partial—in the sense that most approaches check constraints on the target model. These constraints are necessarily partial and usually only capture static properties. These limitations arise due to the complexity of assessing whether a target model is indeed the expected one, especially when dynamic properties are to be considered. This does appear to be a major research gap and an area where further investigation is required. One notable case where dynamic properties have been considered is that of **[StürmerFramework]**, which stands alone at focusing on the case where target models are executable. Strangely, one rather obvious approach to testing target models seems to have been largely ignored—rather than simply checking static constraints of the target model, one could develop a test suite whose test cases could be applied directly to the target model. Perhaps this solution is overly complex due to the need to have a separate test suite for every target model generated, but there may be ways of reusing parts of the test suites given that there is inherent similarity between target models generated by the same transformation.

A clear trend in testing approaches is toward verifying model-to-model transformations. Although from a research perspective, this is to be expected, it is interesting to note that a recent survey of industrial practice found that most transformations used in industry are still model-to-code [44,45].

To summarize, there are three main gaps in existing research on testing model transformations. Firstly, approaches are very specific to the transformation language that a transformation is written in. As illustrated by Table 2, there is no concerted effort among researchers to concentrate on one or a small number of transformation languages; rather, each researcher applies his/her approach to his/her own preferred language. This is problematic as it limits learning in the field as a whole. In particular, there should be more research on testing approaches that are independent of the transformation language—that is, they can be applied to a transformation written in any transformation language. Although some of the approaches surveyed in this paper may indeed satisfy this criterion, the research is always described in the context of a particular transformation language, and little attempt is made to discuss the generality of the approach to other transformation languages. This is a clear gap which should be filled to consolidate the knowledge in the area.

Secondly, although a number of tools and, in particular, testing frameworks have been developed, there is as yet no 'complete' testing infrastructure available. Even the most advanced testing frameworks do not include key features considered by other researchers. For example, none of the testing frameworks described in Sect. 2.1.3 does everything: some do not include explicit coverage criteria, some address only static properties, whereas others consider dynamic properties. This is not a criticism; we simply mean to point out that the area of testing model transformations is relatively mature compared to the other areas considered in this survey, and so perhaps now is the time to consider the development of an end-to-end testing tool that brings all aspects of the research discussed in Sect. 2.1 together.

Finally, as has already been noted, the majority of testing approaches to date have considered static properties primarily. This is because the approach of using constraints as test oracles has proved to be popular, but constraints are limited to static properties. We therefore see consideration of dynamic properties as a growth area for testing model transformations.

3.1.3 Trends in theorem proving approaches

Section 2.2 considered attempts to apply theorem proving to the problem of verifying model transformations. Recall that the literature was classified according to direct versus indirect and certification versus non-certification approaches. As with testing, the general approach has been to transfer standard methods from formal methods to the new problem of verifying model transformations. Typically, direct approaches focus on verifying transformations given as declarative specifications. In these cases, a declarative specification language is translated into a formal language, and a theorem prover is then used to prove properties. What is missing here is the fact that to be executed transformations usually need to have an operational specification. These approaches typically do not prove equivalence of the operational and declarative specifications. Indirect approaches do not directly apply theorem proving to the transformation rules but, rather, they apply it to the target models generated. These approaches have largely been applied in the context of model-to-code transformations because it is easier to apply existing theorem proving tools to programs—theorem proving tools are complex and to develop new ones for modeling languages is highly nontrivial.

Based on our survey, we have identified three fruitful areas for future research. The first relates to certification approaches. We would argue that, given the complexity of applying theorem proving in practice, this is only worthwhile if the objective is certification in safety- or mission-critical domains. This implies therefore that theorem proving approaches are mostly only relevant for model-to-code transformations because certification authorities are normally more interested in certifying the final product, i.e., the code, rather than intermediate products. Achieving a full end-to-end certification framework, however, is extremely difficult in practice because in such an end-to-end approach, every aspect of the framework must be verified and evidence of each verification must be given. To illustrate this point, it is not enough to simply apply a theorem prover to a generated program because the theorem prover itself must be verified. One solution to this in our survey has been to check the proofs generated by the theorem prover using a small kernel proof checker, which may only be a few hundred lines long but has been formally verified (formally verifying most non-kernel provers is not practical). This is just one example where there are additional verification tasks needed to ensure an end-to-end certification approach. But all the approaches in our survey suffer from this problem. Another example is where researchers translate transformation rules into a formal language and then prove properties of the transformation using a prover for this formal language. Researchers tend to ignore the issue of whether the translation itself is correct. We therefore argue that certification of model transformations will not be taken seriously until research is undertaken to define a framework within which all pieces of the framework are verified and evidence is prepared. Even if doing so is not feasible in practice, researchers should still define such a framework and clearly state which parts of the framework are certified, so that there can be a greater degree of confidence in how close we are to achieving an end-to-end solution for certification.

Secondly, indirect approaches appear at first sight to be very appealing because they verify the target model rather than the transformation rules, which is an order of magnitude less complex as discussed in Sect. 2.2.3. However, a key point to understand is that indirect approaches are not applicable in all contexts. Since indirect approaches verify only the output model, they cannot be used to provide wholesale guarantees of a transformation: each output must be individually verified on a case-by-case basis. There are certain contexts where such an approach is insufficient. Consider, for example, the recent trend toward models@run.time [12]. Models@run.time is a term used to describe the use case where models are runtime artifacts. In particular, one case of models@run.time is where model transformations are also run-time artifacts: that is, a model is maintained at runtime and is used to generate new models or code in real time which is then deployed automatically. Such techniques have application in context aware or self-adaptive systems [25]. Although it is not impossible to use indirect approaches in such a scenario, it clearly would be difficult because every time the runtime infrastructure generates an artifact, it would have to be verified on-the-fly. Hence, not only the models would be @run.time but so would be the model transformation verification! In this scenario, therefore, direct approaches to model transformation would be preferable. The point is that the choice of direct versus indirect is dependent on context and neither approach trumps the other in all cases.

Finally, as has already been noted, there are additional verification tasks that are necessary but not considered by many of the works reviewed in our survey. One special case of this, as mentioned above, is equivalence of a declarative specification of a transformation and its operational implementation. Checking for such an equivalence relationship, although difficult, is an under-researched area, and we therefore suggest it as an additional avenue for further research.

3.1.4 Trends in graph theory approaches

Graph theory approaches are very mature and have a distinguished history in terms of their application to model transformation. Despite a rich and mature theoretical background, however, graph transformations have received limited acceptance in practice. There are a number of possible reasons for this. One might be that there is a lack of tools that provide both intuitive and practical modeling environments but also incorporate the theoretical underpinnings. Although there have been recent attempts to provide such tools (e.g., Moflon [3]), graph transformations are still very much an academic rather than industrial pursuit. However, the importance of graph theory applications to model transformations should not be neglected. Typically, those working on graph transformations lead the way in considering important issues, which may then prompt other researchers to consider these issues in the future. A good example of this is bidirectional transformations. Of all the work in our survey, it was only when looking at graph-theoretic approaches that we saw explicit consideration of bidirectional transformations. This suggests an area for future research, therefore, both for those based in graph theory but also for those pursuing other technical approaches such as those based on testing.

Another gap appears to be in the fact that some standard analyses from rule-based systems have not been widely applied to verifying model transformations. Although our survey provides ample evidence of research efforts looking at properties such as termination and confluence (our so-called semantic model transformation properties), other techniques such as critical pair analysis (CPA) do not seem to have been widely considered. CPA is a technique for computing conflicting rule matches statically. It has not received much attention when verifying model transformations. Part of the reason for this could be that most graph transformation tools do not implement CPA. (A notable exception is the AGG tool, which was used by Jayaraman et al. [50] in applying CPA to detect conflicts between model transformations when composing models of features in a software product line).

3.1.5 Trends in model checking approaches

The application of model checking in model transformation verification is relatively immature compared to the use of theorem proving. It has mainly been used in one of two ways—either to check the equivalence of formalizations of the source and target models or within an indirect approach to model check the target model. Most approaches focus on verifying dynamic semantics of the models, which makes sense because this is what model checkers are typically used for.

We can identify two interesting trends that are perhaps under-researched and deserve further attention from the community. Firstly, the approach by Ab. Rahim **[AbRahimMC]** seems novel (we are, of course, biased) in the way that it uses information available in the source model to assist in verifying the target model. Recall that the approach extracts pertinent facts from the source model and adds them as annotations in generated code. These annotations are in this case assertions which are then used to assist the model checker in directing its search. This is an approach that has been applied before but only when the formal engine for verification is a theorem prover not a model checker. In addition, **[AbRahimMC]** addresses the state space explosion in model checking by extracting properties of the source model to reduce the state space in the target: in this case, if certain design patterns are used in the source, these can be used to decide between standard state space reduction techniques in the target.

More generally, we feel that further research could be undertaken where information from the source model is taken into account when verifying the target model. Indeed, this illustrates an advantage of verification that model transformation has—normally, when undertaking verification, one only has the artifact-to-be-verified to analyze. However, for model transformations, there are multiple artifacts, including both the target and the source. And the source model may contain important information (e.g., about design decisions that have been applied as in the case above) which can be used to simplify and possibly scale up the verification of the target.

Another area that deserves further attention is that of certifying model checkers. Although one approach to this has been included in our survey, there has been only limited attention in developing model checkers that can output a checkable witness, which can be used to verify that the model checker is operating correctly. If model checkers are to be used more widely in certification approaches in the future, such certifying model checkers will be crucial.

3.1.6 Trends in metrics and inspection approaches

Research in verification using metrics and inspection is still at an early stage where the research focuses on establishing the metrics and mapping these metrics to quality attributes of model transformations. Threshold values for these metrics have not been established; thus, there is no standard rule to say that a model transformation is good or bad with respect to certain qualities. Furthermore, this area of research is still at the stage where the approaches are not well supported by tools. Availability of tools is important but not as crucial as in other approaches because the effort needed to use the approach is only 'medium.' The importance of tools will increase when model transformations become more complex. Another area where further research is needed is in empirical studies. These metrics-based approaches work by trying to declare connections between values of metrics and well-known qualities (e.g., maintainability). But making these connections must be grounded in real-world experience and, to date, they have been made either only on intuition or based on rather limited empirical studies.

The kind of properties of model transformations checked in this approach are quite different from the other approaches. That is, the properties are typically quality attributes such as maintainability, testability, and performance. In this sense, metrics-based verification is geared more toward good software engineering principles than the other verification approaches. This is a welcome development and shows that, in the future, it is likely that there will be more consideration of the question of how to develop a good transformation in an efficient manner. Another obvious trend is that metrics and inspection approaches are mostly used to verify model-to-model transformations.

One interesting avenue for future research is illustrated by **[StürmerInspect]**, where the observation is made that the inspection of auto-generated code is made easier because of the availability of models. More generally, one could argue that inspection of the target is made easier because of availability of the source. This is an insight similar to that made in the previous subsection: that there is information available in the source model that should not be ignored when verifying the target. The same seems to hold true for inspection, which suggests, more broadly, that there should be further research that takes source model information into account in a whole range of transformation-related activities.

4 Conclusion and outlook

This paper has presented current approaches for verifying model transformations. These approaches were described according to the techniques they used to verify model transformations: testing, theorem proving, model checking, metrics or graph theory. The paper also gave a fine-grained classification of the approaches according to several criteria, and as a result, the authors have identified several trends in research in model transformation, as well as a number of research gaps and future directions for research.

Testing model transformations has reached a level of maturity where tools that adapt well-known program testing techniques to model transformation verification can be practically applied. For theorem proving and model checking–based techniques, many of the approaches use existing automated theorem provers and model checkers. Consequently, they translate the transformation rules into a formalism that can be verified by the theorem prover or the model checker. These approaches should also include a process to verify the translation tool but this is a step that is usually ignored. If the translation tool is a model transformation itself then existing approaches to verify model transformations could potentially be used, particularly approaches that verify correspondence between source and target models. Another path to consider is to reduce the gap between the transformation and formal specification languages by proposing a set of libraries that formally define the instructions in these languages. Several transformation languages, especially the OCL-like languages (e.g., ATL and ETL), have similar instructions (e.g., creating a subset by selecting elements of a larger set that meet certain rules); thus, the libraries are applicable to many languages and could be reused.

Not all theorem proving approaches require the formalization of the model transformation languages. Several approaches such as **[CabotOCL]** and **[AsztalosADL]** only formalize the model transformation specification, and this is an advantage because it removes the difficult task of formalizing the transformation language. However, one can argue that the formal theorem being proved is the transformation specification and not the actual transformation.

The issue with translating model transformations also plagues several model checking approaches. An exception to this is the **[LucioMC]** approach that uses a specially developed model checker that can verify model transformations in the DSLTrans language. The key to model checking DSLTrans model transformations is that the transformation rules are organized into a tree structure. Therefore, a model checker for different languages can be developed as long as the transformation rules can be structured as a tree. This resolves the issue of translating model transformations for model checking approaches.

The problem of state space explosion is not generally addressed by the approaches that use model checking, although this problem is an important issue. Model checking approaches should address this issue by either using symbolic or bounded model checkers (such as the one used in [69]) or using state space reduction techniques (such as the techniques described in [2]). The techniques proposed in [2] can be extended for verifying model transformations using direct techniques where state space reduction techniques are selected based on design patterns appearing in the model transformation. For reduction techniques to be identified from design patterns, the patterns for model transformations must first be identified. Researchers may also want to consider modular designs for model transformations that allow model transformations to be decomposed into components that can be verified separately.

Another issue with the current testing, model checking, and theorem proving approaches is that most of them are language dependent (i.e., they are used to verify model transformations specific to certain languages). To resolve this issue for model checking approaches, a model checker that can execute different transformation rules must be developed. To resolve this issue for theorem proving approaches, the idea of developing a set of libraries for similar language constructs of different languages could be used. By having these libraries, the approach can work with different languages that share similar language constructs.

Finally, a possible research work for metrics is in identifying the threshold value for each metric for developers to know what is a high value, what is a low value, and what is an acceptable value. Furthermore, the threshold value can also be used to determine whether a model transformation has certain quality attributes associated with the metrics. One method to identify the threshold value is to perform empirical studies. Researchers should also consider combining the metrics for declarative and imperative languages because model transformations often contain both declarative and imperative elements.

To summarize, there has been a huge amount of attention in verifying model transformations in recent years. This has led to some level of maturity, although, of course, further avenues for research still exist. Based on our survey, we suggest that the following are the most important lines of future research:

- **Modularity and Reuse.** There has been limited attention looking at reusing verification information or tasks when verifying model transformations. This issue arises in testing where there has been a lack of research on test case reuse or test case prioritization. It also arises in formal approaches where developing a transformation in a more modular way would allow techniques for formal modular verification to be applied.
- **Transformation Language Independence.** Almost all approaches are applied within the context of a specific transformation language. Given industry practice, it may not be realistic to assume that everyone uses a single standard transformation language. But this implies that more research is needed into approaches that are in some sense independent of the transformation language: this could be either by identifying similar constructs common to many languages and developing generic approaches for those constructs, proposing techniques that can be applied to different language-specific verification approaches, or by focusing on indirect approaches which do not necessarily need to know anything about the transformation language used.
- **End-to-end tooling.** As with most research areas, there is a distinct lack of tools that address all aspects of the verification process from start to finish. There is no single testing tool or framework which integrates all the research represented in this survey. In formal certification approaches, there is a need to specify a framework which clearly identifies the various verification tasks and states clearly which parts of a verification process have been certified and which have not. It is unlikely that we will get to the point of an end-to-end certified verification process, but researchers should do a better job of being up-front about the weak points of their process.
- **Make More Use of Source Model Information.** When verifying a target model, only a few approaches take into account information in the source model. This seems like a missed opportunity. For example, a source model may contain important design information (such as the use of design patterns) which can be used to tell a verifier about the modular structure of the model; if this modular structure is preserved in the target, it can be used in modular verification.
- **The Software Engineering of Model Transformations.** Work on metrics-based approaches seems to be the first to consider software engineering questions for model transformations: that is, how to build a good transformation which is maintainable, high quality, and reusable. This is a fruitful area for further research, especially as it relates to the verifiability of model transformations; if transformations are written in a modular way, for example, they will be easier to verify.
- **Future Kinds of Systems.** The field of software and system engineering is changing rapidly, and research in model transformation verification needs to respond to these changes. One particular trend we have highlighted in this paper is the use of models and models at runtime. These approaches may require specialized verification methods for transformations. We have seen no research on this topic to date.
- **Be Grounded in Industrial Practice.** The vast majority of research in model transformation verification is very academic in nature. Researchers do not have a good understanding of how transformations are developed and applied in practice, and without such knowledge, it is impossible to prioritize research in the area. Therefore, we argue for a concerted effort toward a greater understanding of how industry uses transformations. Preliminary work toward this goal has been reported in [44,45].
- **Higher-Order Transformations.** Higher-order transformations (HOTs) [92] are transformations that generate transformations. The fundamental concepts of HOTs are that model transformations are also models, and therefore, existing model transformation verification techniques may be applicable to HOTs. However, verifying HOTs may be more challenging, for example, when using an indirect approach since the argument that verifying the target model is less complex than the transformation cannot be applied for HOTs.

Model transformation verification research is an active field. There have been a significant number of important advances in recent years, and we would expect activity to grow in this area. We offer the eight open questions above as areas for future research; these are areas that have received relatively little attention but which are crucial to the success of model transformation verification as a research field.

References

1. Ab. Rahim, L., Whittle, J.: Verifying semantic conformance of state machine-to-java code generators. In: Model Driven Engineering Languages and Systems—13th International Conference, MODELS 2010, Proceedings, Part I, vol. 6394 of Lecture Notes in Computer Science, pp. 166–180, Oslo, Norway, October 2010. Springer, Berlin (2010)

2. Ab. Rahim, L., Whittle, J.: Identifying state space reduction techniques from behavioural design patterns. In: Proceedings of the Third Workshop on Behavioural Modelling, BM-FA '11, pp. 49–55. ACM, New York (2011)
3. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based tool integration with Moflon. In: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pp. 807–810. ACM, New York (2008)
4. Amrani, M., Lucio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 921–928. IEEE Computer Society, Washington (2012)
5. Amstel, M.F., Lange, C.F., Brand, M.G.: Using metrics for assessing the quality of ASF+SDF model transformations. In: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, ICMT '09, pp. 239–248. Springer, Berlin (2009)
6. Asztalos, M., Lengyel, L., Levendovszky, T.: A formalism for describing modeling transformations for verification. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, pp. 1–10. ACM, New York (2009)
7. Baar, T., Marković, S.: A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In: Perspectives of Systems Informatics, vol. 4378 of Lecture Notes in Computer Science, pp. 70–83. Springer, Berlin (2007)
8. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) Fundamental Approaches to Software Engineering, vol. 1783 of Lecture Notes in Computer Science, pp. 363–366. Springer, Berlin (2000)
9. Barbosa, P.E.S., Ramalho, F., de Figueiredo, J.C.A., dos S. Junior, A.D.: An extended MDA architecture for ensuring semantics-preserving transformations. In: 32nd Annual IEEE Software Engineering, Workshop, pp. 33–42, October (2008)
10. Barbosa, P.E.S., Ramalho, F., de Figueiredo, J.C.A., dos S. Junior, A.D., Costa, A., Gomes, L.: Checking semantics equivalence of MDA transformations in concurrent systems. J. Univers. Comput. Sci. **15**(11), 2196–2224 (2009)
11. Baudry, B., Dinh-trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing (2006)
12. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer **42**, 22–27 (2009)
13. Blech, J.O., Glesner, S., Leitner, J.: Formal verification of java code generation from UML models. In: 3rd International Fujaba Days 2005-MDD, in Practice, pp. 49–56 (2005)
14. Boehm, B.: Verifying and validating software requirements and design specifications. IEEE Softw. **1**(1), 75–88 (1984)
15. Boronat, A., Heckel, R., Meseguer, J.: Rewriting logic semantics and verification of model transformations. In: Fundamental Approaches to Software Engineering (FASE), pp. 18–33 (2009)
16. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**(12), 35–132 (2000)
17. Brooks Jr, F.P.: The Mythical Man-Month, Anniversary edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
18. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: 17th International Symposium on Software Reliability Engineering (ISSRE '06), pp. 85–94. IEEE (2006)
19. Buth, B., Buth, K., Franzle, M., Karger, B., Lakhneche, Y., Langmaack, H., Muller-Olm, M.: Provably correct compiler development and implementation. In: Compiler Construction, pp. 141–155. Springer, Berlin (1992)
20. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. **83**, 283–302 (2009)
21. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: Davies, J., Silva, L., Simao, A. (eds.) Formal Methods: Foundations and Applications, vol. 6527 of Lecture Notes in Computer Science, pp. 112–127. Springer, Berlin (2011)
22. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML'04) (2004)
23. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., Niebert, P.: From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES '03), pp. 153–162. ACM (2003)
24. Chaki, S., Ivers, J., Lee, P., Wallnau, K., Zeillberger, N.: Model-driven construction of certified binaries. In: 10th International Conference, MODELS 2007, pp. 666–681. Springer, Berlin (2007)
25. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo, Serugendo G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer, Berlin (2009)
26. Clavel, M., Egea, M.: ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In: Johnson, M., Vene, V. (eds.) Algebraic Methodology and Software Technology, vol. 4019 of Lecture Notes in Computer Science, pp. 368–373. Springer, Berlin (2006)
27. Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J.J., van Deursen, A.: Understanding execution traces using massive sequence and circular bundle views. In: Proceedings of the 15th IEEE International Conference on Program Comprehension, ICPC '07, pp. 49–58. IEEE Computer Society, Washington (2007)
28. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
29. Darabos, A., Pataricza, A., Varró, D.: Towards testing the implementation of graph transformations. In: Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques, pp. 69–80. Elsevier (2006)
30. Denney, E., Fischer, B.: Generating customized verifiers for automatically generated code. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '08), pp. 77–88. ACM (2008)
31. Denney, E., Fischer, B., Schumann, J., Richardson, J.: Automatic certification of Kalman filters for reliable code generation. In: IEEE Aerospace Conference, pp. 1–10. IEEE (2005)
32. Egea, M., Rusu, V.: Formal executable semantics for conformance in the MDE framework. Innov. Syst. Softw. Eng. **6**(1–2), 73–81 (2010)
33. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Fundamental Approaches to Software Engineering (FASE), pp. 72–86 (2007)
34. Fleurey, F., Baudry, B., Muller, P.-A., Le Traon, Y.: Towards dependable model transformations: qualifying input test data. In: Software and System Modeling. Springer, Berlin (2007)

35. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Proceedings of the 1st International Workshop on Model, Design and Validation, pp. 29–40. IEEE (2004)
36. García, M., Möller, R.: Certification of transformation algorithms in model-driven software development. In: Software Engineering, pp. 107–118 (2007)
37. Giner, P., Pelechano, V.: Test-driven development of model transformations. In: 12th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 748–752 (2009)
38. Goos, G.: Compiler verification and compiler architecture. Electron. Notes Theor. Comput. Sci. **65**(2), 1 (2002). COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002)
39. Harrison, R., Samaraweera, L., Dobie, M., Lewis, P.: Estimating the quality of functional programs: an empirical investigation. Inf. Softw. Technol. **37**(12), 701–707 (1995)
40. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of model synchronization based on triple graph grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems, vol. 6981 of Lecture Notes in Computer Science, pp. 668–682. Springer, Berlin (2011)
41. Hoare, T.: The verifying compiler: a grand challenge for computing research. In: Hedin, G. (ed.) Compiler Construction, vol. 2622 of Lecture Notes in Computer Science, pp. 262–272. Springer, Berlin (2003)
42. Holten, D., van Wijk, J.J.: Visual comparison of hierarchically organized data. Comput. Graph. Forum **27**(3), 759–766 (2008)
43. Howard W. (1980) To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism, chapter The Formulae-as-types Notion of Construction, pp. 479–490. Academic Press.
44. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 633–642. ACM, New York (2011)
45. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 471–480. ACM, New York (2011)
46. Izerrouken, N., Thirioux, X., Pantel, M., Strecker, M.: Certifying an automated code generator using formal tools: preliminary experiments in the GeneAuto project. In: Electronic Proceedings of 4th European Congress in Real-Time Sofware (ERTS'08) (2008)
47. Jackson, E., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems, vol. 6981 of Lecture Notes in Computer Science, pp. 653–667. Springer, Berlin (2011)
48. Jackson, E.K., Kang, E., Dahlweid, M., Seifert, D., Santen, T.: Components, platforms and possibilities: towards generic automation for MDA. In: Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT '10, pp. 39–48. ACM, New York (2010)
49. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: The semantics of constraint logic programs. J. Log. Program. **37**(13), 1–46 (1998)
50. Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D., Weil, F. (eds.) Model Driven Engineering Languages and Systems, vol. 4735 of Lecture Notes in Computer Science, pp. 151–165. Springer, Berlin (2007)
51. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(12), 31–39 (2008)
52. Kapová, L., Goldschmidt, T., Becker, S., Henss, J.: Evaluating maintainability with code metrics for model-to-model transformations. In: Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice–Reality and Gaps, QoSA'10, pp. 151–166. Springer, Berlin (2010)
53. Karsai, G., Narayanan, A.: On the correctness of model transformations in the development of embedded systems. In: Kordon, F., Sokolsky, O. (eds.) Composition of Embedded Systems. Scientific and Industrial Issues, vol. 4888 of Lecture Notes in Computer Science, pp. 1–18. Springer, Berlin (2007)
54. Kolovos, D., Paige, R., Polack, F.: Model comparison: a foundation for model composition and model transformation testing. In: Proceedings of the 1st International Workshop on Global Integrated Model Management (G@MMA'06), pp. 13–20. ACM (2006)
55. Kolovos, D., Paige, R., Rose, L., Polack, F.: The Epsilon Book. University of York, York (2009)
56. Kübler, J., Goldschmidt, T.: A pattern mining approach using QVT. In: Proceedings of the 5th European Conference on Model Driven Architecture-Foundations and Applications, ECMDA-FA '09, pp. 50–65. Springer, Berlin (2009)
57. Küster, J., Heckel, R., Engels, G.: Defining and validating transformations of UML models. In: IEEE Symposium on Human Centric Computing Languages and Environments, pp. 145–152. IEEE (2003)
58. Küster, J.M.: Definition and validation of model transformations. Softw. Syst. Model. **5**(3), 233–259 (2006)
59. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations first experiences using a white box approach. In: Models in Software Engineering, pp. 193–204 (2007)
60. Lamari, M.: Towards an automated test generation for the verification of model transformations. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07), pp. 998–1005. ACM (2007)
61. Lamport, L.: Checking a multithreaded algorithm with $^{+}$CAL. In: Proceedings of 20th International Symposium on Distributed Computing, pp. 151–163. Springer, Berlin, Stockholm, September (2006)
62. Lamport, L.: The PlusCal algorithm language. In: Proceedings of 6th International Colloquium on Theoretical Aspects of Computing, pp. 36–60, Kuala Lumpur, Malaysia, August (2009)
63. Lano, K., Clark, D.: Model transformation specification and verification. In: The Eighth International Conference on Quality Software (QSIC '08), pp. 45–54. ACM (2008)
64. Lano, K., Kolahdouz-Rahimi, S.: Specification and verification of model transformations using UML-RSDS. In: Méry, D., Merz, S. (eds.) Integrated Formal Methods, vol. 6396 of Lecture Notes in Computer, pp. 199–214. Springer, Berlin (2010)
65. Lano, K., Kolahdouz-Rahimi, S.: Model-Driven Development of Model Transformations. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations (ICMT), vol. 6707 of Lecture Notes in Computer Science, pp. 47–61. Springer, Berlin (2011)
66. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, Germany (2006)
67. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Bruel, J.-M. (ed.) Satellite Events at the MoDELS 2005 Conference, vol. 3844 of Lecture Notes in Computer Science, pp. 139–150. Springer, Berlin (2006)
68. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Model-Driven Software Development-Research and Practice in Software Engineering, pp. 219–236. Springer, Berlin (2005)
69. Lúcio, L., Barroca, B., Amaral, V.: A technique for automatic validation of model transformations. In: Petriu, D., Rouquette,

N., Haugen, O. (eds.) Model Driven Engineering Languages and Systems, vol. 6394 of Lecture Notes in Computer Science, pp. 136–150. Springer, Berlin (2010)

70. Mottu, J.-M., Baudry, B., Le Traon, Y.: Mutation analysis testing for model transformations. In: Model Driven Architecture–Foundations and Applications, Second European Conference, ECMDA-FA 2006, pp. 376–390. Springer, Berlin (2006)
71. Mottu, J.-M., Baudry, B., Le Traon, Y.: Reusable MDA components: a testing-for-trust approach. In: Proceedings of the MoDELS/UML 2006, pp. 589–603. Springer, Berlin (2006)
72. Mottu, J.-M., Baudry, B., Le Traon, Y.: Model transformation testing: oracle issue. In: IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08), pp. 105–112. IEEE (2008)
73. Namjoshi, K.S.: Certifying model checkers. In: Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01), pp. 2–13. Springer, Berlin (2001)
74. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electron. Notes Theor. Comput. Sci. **211**, 191–200 (2008)
75. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), pp. 106–119. ACM (1997)
76. Orejas, F., Wirsing, M.: On the specification and verification of model transformations. In: Palsberg, J. (eds) Semantics and Algebraic Specification, vol. 5700 of Lecture Notes in Computer Science, pp. 140–161. Springer, Berlin (2009)
77. Pnueli, A., Shtrichman, O., Siegel, M.: The code validation tool CVT: automatic verification of a compilation process. Softw. Tools Technol. Transf. **2**, 192–201 (1998)
78. Poernomo, I.: Proofs-as-model-transformations. In: Proceedings of the 1st international conference on Theory and Practice of Model Transformations, ICMT '08, pp. 214–228. Springer, Berlin (2008)
79. Poernomo, I., Terrell, J.: Correct-by-construction model transformations from partially ordered specifications in Coq. In: Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, ICFEM'10, pp. 56–73. Springer, Berlin (2010)
80. Rahimi, S., Lano, K.: Integrating goal-oriented measurement for evaluation of model transformation. In: International Symposium on Computer Science and Software Engineering (CSSE), pp. 129–134. IEEE (2011)
81. Reynoso, L., Genero, M., Piattini, M., Manso, E.: Assessing the impact of coupling on the understandability and modifiability of OCL expressions within UML/OCL combined models. In: Software Metrics, 2005. 11th IEEE International, Symposium, pp. 10–14, September (2005)
82. RTCA: DO-178B, Software Consideration in Airborne Systems and Equipment Certification. Technical report, RTCA Inc (1992)
83. Saeki, M., Kaiya, H.: Measuring model transformation in model driven development. In: Proceedings of the International Conference on Advanced Information, Systems Engineering (CAiSE'07), pp. 77–80 (2007)
84. Schumann, J., Fischer, B., Whalen, M., Whittle, J.: Certification support for automatically generated programs. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences, pp. 1–10. IEEE (2003)
85. Sen, S., Baudry, B., Mottu, J.-M.: On combining multi-formalism knowledge to select models for model transformation testing. In: 1st International Conference on Software Testing, Verification, and Validation, pp. 328–337. IEEE (2008)
86. Sen, S., Baudry, B., Mottu, J.-M.: Automatic model generation strategies for model transformation testing. In: Paige, R. (ed.) Theory and Practice of Model Transformations, vol. 5563 of Lecture Notes in Computer Science, pp. 148–164. Springer, Berlin (2009)
87. Staats, M., Heimdahl, M.: Partial translation verification for untrusted code-generators. In: International Conference on Formal Engineering Methods (ICFEM'08), pp. 226–237. Springer, Berlin (2008)
88. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems, vol. 6981 of Lecture Notes in Computer Science, pp. 533–547. Springer, Berlin (2011)
89. Strecker, M.: Modeling and verifying graph transformations in proof assistants. Electron. Notes Theor. Comput. Sci. **203**(1), 135–148 (2008)
90. Stürmer, I., Conrad, M., Doerr, H., Pepper, P.: Systematic testing of model-based code generators. IEEE Trans. Softw. Eng. **33**(9), 622–634 (2007)
91. Stürmer, I., Conrad, M., Fey, I., Dörr, H.: Experiences with model and autocode reviews in model-based software development. In: Proceedings of the 2006 International Workshop on Software Engineering for Automotive systems (SEAS '06), pp. 45–52. ACM (2006)
92. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bzivin, J.: On the use of higher-order model transformations. In: Paige, R., Hartman, A., Rensink, A. (eds.) Model Driven Architecture—Foundations and Applications, vol. 5562 of Lecture Notes in Computer Science, pp. 18–33. Springer, Berlin (2009)
93. van Amstel, M., Bosems, S., Kurtev, I.: Performance in model transformations: experiments with ATL and QVT. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations, vol. 6707 of Lecture Notes in Computer Science, pp. 198–212. Springer, Berlin (2011)
94. van Amstel, M., Lange, C., van den Brand, M.: Metrics for analyzing the quality of model transformations. In: Proceedings of the 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE08), pp. 41–51. Paphos, Cyprus (2008)
95. van Amstel, M., van den Brand, M.: Quality Assessment of ATL Model Transformations using Metrics, Technical Report. Department of Mathematics and Computer Science, Eidhoven University of Technology (2010)
96. van Amstel, M.F.: The right tool for the right job: assessing model transformation quality. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, COMPSACW '10, pp. 69–74. IEEE Computer Society, Washington (2010)
97. Van Amstel, M.F., Van Den Brand, M.G.J.: Model Transformation Analysis: Staying ahead of the maintenance nightmare. In: Proceedings of the 4th International Conference on Theory and Practice of Model Transformations, ICMT'11, pp. 108–122. Springer, Berlin (2011)
98. Van Baalen, J., Robinson, P., Lowry, M., Pressburger, T.: Explaining synthesized software. In: Proceedings of 13th IEEE International Conference on Automated Software Engineering, 1998, pp. 240–248 (1998)
99. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop, number TUM-I0323 in Technical Report, pp. 63–78. Technische Universität München, September (2003)
100. Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination analysis of model transformations by Petri nets. In: ICGT, pp. 260–274 (2006)
101. Vignaga, A.: Measuring Atl Transformations. Technical report, MaTE. Department of Computer Science, Universidad de Chile (2009)

102. Vignaga, A.: Metrics for Measuring ATL Model Transformations. Technical report. MaTE, Department of Computer Science, Universidad de Chile (2009)
103. Wang, J., Kim, S.-K., Carrington, D.: Verifying metamodel coverage of model transformations. In: Australian Software Engineering Conference, pp. 270–282. IEEE (2006)
104. Wang, J., Kim, S.-K., Carrington, D.: Automatic generation of test models for model transformations. In: 19th Australian Conference on Software Engineering (ASWEC'08), pp. 432–440. IEEE (2008)
105. Whittle, J., Gajanovic, B.: Model transformations should be more than just model generators. In: Satellite Events at the MoDELS 2005 Conference, pp. 32–38. Springer, Berlin (2005)
106. Whittle, J., Van Baalen, J., Schumann, J., Robinson, P., Pressburger, T., Penix, J., Oh, P., Lowry, M., Brat, G.: Amphion/NAV: deductive synthesis of state estimation software. In: Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE'01), pp. 395–399. IEEE (2001)
107. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: Right or wrong?—verification of model transformations using colored Petri nets. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modelling (2009)
108. Zelenov, S., Silakov, D., Petrenko, A., Conrad, M., Fey, I.: Automatic test generation for model-based code generators. In: 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 75–81. IEEE (2006)

Author Biographies

Lukman Ab. Rahim is a lecturer at Universiti Teknologi PETRONAS. He obtained his B.Sc in Computer Science at Universiti Teknologi Malaysia and his M.Sc in Software Engineering at University of York, UK. He obtained his Ph.D. from Lancaster University, UK. His research interests are in Model-Driven Engineering, Software Modelling, and architecture and formal verification. Currently, he is carrying out research on formally verifying model transformations and using model checking in verifying cloud computing infrastructure and middleware.

Jon Whittle is full Professor and Chair of Software Engineering at Lancaster University. He is also a Royal Society Wolfson Merit Award Scholar. He has been working with Model-Driven Engineering for over ten years, including stints at NASA Ames Research Center (California), George Mason University (Virginia), IIT Kanpur (India), and more recently at Lancaster University. He has been intimately involved with the MDE research community during this time, serving as the Chair of the Steering Committee of the MODELS conference from 2006 to 2008 and as PC Chair in New Zealand in 2011. He also serves on the editorial board of the Journal of Software and System Modeling. Jon's current interests are in empirically investigating what factors lead to success or failure with MDE in industry.