Formal Verification Techniques for Model Transformations: A Tridimensional Classification

Moussa AMRANI\(^a\)  Benoît COMBEMALE\(^b\)  Levi LÚCIO\(^c\)

Gehan SELIM\(^d\)  Jürgen DINGEL\(^d\)  Yves LE TRAON\(^a\)

Hans VANGHELUWE\(^{ce}\)  James R. CORDY\(^d\)

\(^a\) University of Luxembourg (Luxembourg)  \(^b\) University of Rennes 1 / INRIA (France)  \(^c\) McGill University (Canada)  \(^d\) Queen’s University (Canada)  \(^e\) University of Antwerp (Belgium)

Abstract In Model Driven Engineering (MDE), models are first-class citizens, and model transformation is MDE’s "heart and soul". Since model transformations are executed for a family of (conforming) models, their validity becomes a crucial issue.

This paper proposes to explore the question of the formal verification of model transformation properties through a tridimensional approach: the transformation involved, the properties of interest addressed, and the formal verification techniques used to establish the properties.

This work is intended for a double audience. For newcomers, it provides a tutorial introduction to the field of formal verification of model transformations. For readers more familiar with formal methods and model transformations, it proposes a literature review (although not systematic) of the contributions of the field.

Overall, this work allows to better understand the evolution, trends and current practice in the domain of model transformation verification. This work opens an interesting research line for building an engineering of model transformation verification guided by the notion of model transformation intent.

Keywords Model-Driven Engineering, Model Transformation, Model Transformation Intent, Formal Verification, Transformation Languages, Classification, Survey, Property of Interest

1 Introduction

Model-Driven Engineering (MDE) promotes models as first class citizens of the software development process. Models are manipulated through model transformations (MTrs), which is considered to be the "heart and soul" of MDE [106]. Naturally, dedicated languages based on different paradigms emerged during the last decade to express MTrs. Since they are executed on a family of (conforming) models, the validity of such model transformations becomes a crucial issue.

From the MDE point of view, a transformation has a dual nature [22]: seen as a transformation model, a transformation can denote a computation, whose expression relies on a particular model of computation (MoC) embedded in a transformation language; and seen as a model transformation, emphasis focuses on the particular artefacts manipulated by a transformation (namely, metamodels for its specification and models for its execution). The computational nature will require that the underlying language guarantees desirable properties, like termination and determinism, needed for the purpose of the transformations, independently of what a particular transformation expresses. On the other hand, manipulating models implies that specific properties of interest will be directly related to the involved models as well as the intrinsic intent behind the transformation: e.g., one may be interested in always producing conforming models by construction, or ensuring that target models still conserve their semantics.

This paper proposes a tridimensional classification of the contributions found in the literature (cf. Figure 1). This classification encompasses all artefacts needed when formally verifying model transformations: the transformation, the property kinds one needs to verify to ensure correctness, and the formal verification technique used to prove those properties. By locating each contribution in this tridimensional space, it becomes possible to reason about the evolution and trends in the domain: for instance, it enables the identification of areas that are over- or under-represented (the properties that interest the community the most, or the verification techniques that are used the less). Validated on a significant corpus of publications, this classification provides an interesting “snapshot” of the current state of the art in the area of MT verification.

The paper focuses on formal Verification & Validation (V&V), understood as the activity of deploying mathematical methods in order to prove the correctness of model transformations in an exhaustive, static way. Other non exhaustive, or dynamic techniques are purposely discarded from this paper’s scope to narrow down the tridimensional space into a meaningful set of properties/tools (cf. e.g. a survey for testing in [105]).

This paper extends a preliminary study that appeared more than two years ago in [4] with the following points:

• a precise description of the characteristics of formal V&V techniques allowed a justification of the techniques that we excluded from our classification;

• an in-depth discussion of the existing classifications of the literature showed their inadequacy regarding the activity of formal verification: they do not allow a proper extraction of the properties of interest one need to prove to ensure transformation correctness;

• an analysis of these classifications weaknesses motivated the introduction of a novel concept for model transformation, the intent, acting as a glue between all dimensions;
Formal Verification Techniques for Model Transformations: A Tridimensional Classification

Figure 1 – Taxonomy overview: the tri-dimensional approach.

• an outlook on the relations between the classification dimensions in the light of this new concept;
• and most importantly, the integration of many contributions that were initially missing in [4] due to space limitations, or that appeared since the first paper was published.

Our study illustrates the dynamism of the field, and the richness of the combinations between these dimensions for solving the multiple challenges related to model transformation correctness. We therefore expect a minimal knowledge in MDE and formal verification: we do not provide introductory material of these fields; instead, we contextualise from each field basic elements necessary to understand our classification.

Our main contribution is the tridimensional classification, validated by its application to over a hundred references. As a consequence, we seek for coverage of each dimension that we try to illustrate with at least one contribution. This paper is neither a systematic literature survey (SLR, as understood by Kitchenbaum [61, 60]) for the reasons we mentioned earlier, nor a study aiming at completeness, since the field evolves fast (as illustrated by the fact that the number of contributions doubled since our original version [4]) and we only wanted to illustrate the diversity and richness of our classification. However, this paper can serve the needs of two different audiences as a side-effect of its main purpose: for newcomers, it provides a tutorial introduction to the field of formal verification of model transformations, by precisely identifying the main notions at play, and proposing an analysis grid for understanding its many variation points; it nevertheless does not replace good introductory material for each specific dimension. For readers more familiar with formal methods and model transformations, it proposes a survey of the field, although not a SLR, spotting the current strengths and weaknesses of existing approaches, and proposing a critique of some of the concepts at play in MDE in the light of the formal verification perspective.

The paper’s outline follows the constituting dimensions of the classification: the transformation involved (cf. Section 2); the property kinds addressed (cf. Section 3); and the verification technique used (cf. Section 4). These dimensions are closely orthogonal but clearly interdependent: Section 5 discusses their pairwise relations. Section 6 discusses the related work dedicated to classifications of model transformation verification and Section 7 summarises and presents future work directions.
2 Dimension 1: Transformations

Figure 2 presents the general idea of model transformation. A model, conforming to a source metamodel, is transformed into another model, itself conform to a target metamodel, by the execution of a transformation specification. The transformation specification is defined at the level of metamodels whereas its execution operates on the model level. Since in MDE models are the primary artefact, both source and target metamodels (as well as the transformation specification) are themselves models, conforming to their respective metamodels: for metamodels, this is the classical notion of meta-metamodel; for transformations, a transformation language (Tl) (or metamodel) allows a sound specification of transformations. Notice here that a transformation can also act on several source and/or target models.

This Section starts by discussing the existing definitions for the concept of transformation; then reviews the main classifications for transformation languages. We then revisit these classifications in the light of formal verification: why is it hard to extract the properties one is interested in from the existing classifications?

2.1 Definition

We quickly discuss several definitions for the concept of model transformation proposed in the literature to highlight their main characteristics from the verification viewpoint.

Historically, one of the first definitions was proposed by the OMG, inspired by the Model-Driven Architecture view. The OMG perceives transformations as "the process of converting one model to another model of the same system" [86].
after, the system-centric view was enlarged by Kleppe et al.: "a model transformation is the automatic generation of a target model from a source model, according to a transformation definition" [62]. This definition shifts from the system-centric view in order to consider general source/target models, insisting on the fact that transformations are mostly perceived as directed and automatic (i.e. without users’ intervention) manipulation of models. Tratt describes a transformation as "a program that mutates one model into another" [115], insisting on the computational aspect of transformations. More recently, two contributions widened the perspective with two important aspects: Mens et al. proposed to see transformations as "the automatic generation of one or multiple target models from one or multiple source models, according to a transformation description" [77], whereas Syriani re-introduced the crucial importance of the specific intention behind transformations ("the automatic manipulation of a model with a specific intention" [112]).

Therefore, as a broader definition, we consider transformations as the automatic manipulation, conform to a description, of (an) input model(s) to produce (an) output model(s) according to a specific intention. This definition clearly embeds the dual nature of model transformation, distinguishing its specification from its execution (cf. Fig. 2), and places the transformation’s intention at its core. When clear from the context, we abuse the word’s meaning and sometimes refer to a transformation specification (instead of our definition’s primary meaning of an execution) to avoid repetitions.

2.2 Languages

When specifying a transformation, a transformation designer expresses a computation using the constructions of a Transformation Language (Tl) syntax that manipulates the concepts of both the source and the target metamodels. Over the years, many languages, as well as many transformation frameworks emerged, with various model transformation purposes and targets. We quickly review the categorisation by [35] on the computing paradigms for model transformations:

**Programming-Based** This category encompasses several ones proposed originally in [35], in which practices already well-known in general-purpose programming languages are applied, or adapted, for transformation specification: popular techniques are visitor-based (used e.g. in JAMDA [18]), template-based (used e.g. in AndroMda [17] or MetaEdit+ [59]), or even the direct manipulation through dedicated APIs (e.g. Java). These techniques are particularly well-suited for model-to-text manipulations.

**Operational** This category, also known as meta-programming [32], shares common features with the previous one, but usually offers more dedicated support for handling model manipulation. This approach consists of enriching metamodelling formalisms (e.g., the OMG MOF) with so-called action languages that are generally object-oriented (two notable examples are Kermeta [79] and Epsilon [63]).

**Declarative** This category relies on rewriting techniques, and contrasts with the previous ones in the declarative computation paradigm (designers specify what the transformation does instead of how it is performed) and in their syntax, usually visual. This category is usually split into two different trends: relational Tls define transformations in terms of mathematical relations among source and
target metamodel elements using constraints; and \textit{Graph-Based Transformations} (GbtTs) represent models and transformations as graphs and graph transformations respectively \cite{103}. The natural visual syntax attached to graphs can be further customised \cite{113, 38} to reflect more adequately the symbols used in particular domains.

This categorisation is not complete: hybrid approaches are common in model transformation languages, especially for declarative TIs: the visual representation adequately represents the manipulation of core concepts (like deletion/creation of class instances or association links), whereas the textual sentences typically define update expressions of values taken by class attributes. Each representation works at a different level, thus helping to take advantage of both worlds.

From a formal verification viewpoint, what is crucial for all TIs is the formal specification of their semantics. Being built upon traditional operational languages, the two first categories directly benefit from the existing formal semantics foundations available for imperative or object-oriented programming languages \cite{122}, and can therefore readapt new advances in formal verification. Declarative TIs also have a clear formal background: logics and constraint programming are the natural semantic targets for relational declarative approaches; whereas the category theory is the underlying formalism for GbtTs \cite{103}.

The computational paradigm of a TI naturally influences both the kind of properties one needs to prove (e.g., determinism for operational approaches is never an issue whereas for declarative TIs, it can be crucial) and the kind of formal verification techniques available (e.g., exhaustive exploration of a transformation’s state space can naturally be computed for declarative approaches, whereas it usually requires a dedicated machinery for operational ones).

2.3 Classification

From a verification viewpoint, it is important to classify model transformations in order to precisely identify which properties have to be checked to guarantee correctness. We review the existing classifications available from the literature and show their limitations on a simple conceptual example.

Two contributions addressed the model transformation classification issue: Czarnecki and Helsen \cite{35} proposed a hierarchical classification of model transformations specification \textit{features}, whereas Mens and Van Gorp \cite{77} focused more on providing a multidimensional taxonomy characterising model transformation \textit{form}.

2.3.1 Model Transformation Features

A first level of classification consists of highlighting the building blocks of a transformation, i.e. which features compose a model transformation. We summarise the features proposed by \cite{35} relevant for the purpose of model transformation formalisation and verification:

\textbf{Transformation Units} are the basic building blocks used to specify how computations are performed during the transformation;

\textbf{Scheduling} specify how transformation units are combined to perform the computation: either implicitly, in which case the transformation designer has no
direct control; or explicitly, using a large set of possibilities ranging from partially user-controlled schedulers to explicitly modelled DSLs for schedule flow specification.

2.3.2 Model Transformation Form

A second level of classification consists of focusing on the form of a transformation, i.e. in which ways a transformation is related to its metamodels for its specification, and to its models for its execution. This classification covers formalisms and tools underlying model transformations, but we summarise here the aspects of Mens and Van Gorp’s multi-dimensional taxonomy [77] relevant to our purpose:

**Heterogeneity** between the source and target metamodel: if they are the same, the transformation is *endogeneous* and expresses a rephrasing intention; otherwise, it is *exogeneous* and conveys a translation intention.

**Abstraction Level** related to the detail level involved into models: if the target metamodel adds or reduces the detail level, the transformation is *vertical*; otherwise, if the abstraction level remains unchanged, it is *horizontal*.

**Transformation Arity** regarding the number of input (respectively, output) models the transformation executes on.

**Input Model Conservation** related to the treatment of the input model: if the transformation directly alters the input model, it is *destructive*, or *in-place*; if another independent model is outputted, it is *conservative*, or *out-place*.

2.4 Discussion: Extracting Properties of Interest

Model transformations differ in the way they are *expressed*, i.e. in the chosen *transformation language* they are specified, but also in what they are to accomplish, i.e. their *intent*. Therefore, the properties of interest for transformations are naturally related to both dimensions.

Unfortunately, the previous classifications do not help for our purpose of facilitating the extraction of the relevant properties to be checked to ensure the correctness of a transformation. Let us see how Mens and Van Gorp’s classification operates on a simple example: consider a Domain-Specific Model (Dsm) for which one needs to define its semantics through a transformation. Two classical approaches exist for that purpose [32], as depicted in Figure 3:

**Translational** This approach expresses the semantics of a Dsm in terms of another *target* metamodel serving as a *semantic domain* [51]: in this case, the transformation is out-place, exogeneous and vertical (with arity 1:1).

**Operational** This approach expresses the semantics of a Dsm directly on the input model, by showing its evolution over time: this makes the transformation in-place, endogeneous and horizontal (also with arity 1:1).

Although both transformations cover the same intent, i.e. providing a semantics, their expression differs radically, as witnessed by the previous classification producing opposite characterisations. However, one would expect to have to prove, for the same transformation intent, the same kind of properties (e.g. proving the correctness) but expressed differently w.r.t. the transformation’s expression: in the first case, the
Figure 3 – Two classical ways for expressing the semantics of a DSML: a *translational* semantics uses a target metamodel as semantic domain, making the transformation out-place, heterogeneous and vertical; an *operational* semantics expresses the behaviour in terms of model changes, making the transformation in-place, endogeneous and horizontal.

Semantic correctness can only be assessed through properties relating both metamodels; whereas in the second case, behavioural properties (also called "dynamic", generally expressed through temporal logic formulæ) will be checked on the transformation's execution.

Consequently, beyond already existing *syntactic* classification, i.e. describing the transformations' expression, it is crucial to have at disposal a *semantic* classification, i.e. describing transformations' intention, in order to relate transformations' meaning with related properties of interest. For our example, whatever form the transformation holds, it expresses in both cases a DSL semantics specification that necessitates to prove its correctness. This work is partially addressed in [3, 72], where a catalogue of the most common intents in MDE are mapped to their characteristic properties that one needs to prove to ensure the correctness of a transformation complying to these intents.

3 Dimension 2: Properties

Expressed in a particular TL, model transformation specifications relate source and target metamodel(s) and execute on models. Considering only conforming models for transformations to be valid is not enough: due to the large number of models transformations can be applied on, one has to ensure their validity by carefully analysing their properties to provide modelers a high degree of trustability when they use automated transformations to perform their daily tasks.

This Section explores properties one may be interested in for delivering proper and valid transformations. Given the large variety of model transformation intents, it was not possible to delve into a detailed classification of properties of interest. Rather, we identify two classes of properties that follow the dual nature of transformations
3.1 Transformation Models: Language-Related Property

From a computational perspective, a transformation specification conforms to a transformation language, which can possess properties of its own suitable either at execution time (referring to the transformation execution) or at design time (addressing the relation between the transformation specification and its language), as described in Figure 4.

At execution time, two properties are crucial and qualify the model transformation execution: termination, which guarantees the existence of target model(s); and determinism, which ensures uniqueness. At design time, typing ensures the well-formedness of transformation specification regarding its defining language, which can be seen as the TL’s static semantics.

Because they hold at the TL’s level, these properties directly impact the execution and design of all transformations. Therefore, formally proving them cannot be done by relying on one particular transformation’s specifics. TLs adopt one of the following strategies for proving execution time properties hold: either the TL is kept as general (and powerful) as possible, making these properties undecidable, but the transformation framework provides capabilities for checking sufficient conditions ensuring them to hold on a particular transformation; or these properties are ensured by construction by the TL, generally by sacrificing its expressive power.
3.1.1 Termination

Termination directly refers to Turing’s halting problem, which is known to be undecidable for sufficiently expressive, i.e. Turing-complete, Tls: Gbt have already been proven to not terminate in the general case [91]; whereas Mpls often use loops and (potentially recursive) operation calls, making them able of simulating Turing machines.

3.1.1.1 Termination Criteria

A large amount of literature for Gbt already exists. In [41], Ehrig et al. introduce layers with injective matches in the rewriting rules that separate deletion from non-deletion steps. In [118], Varró et al. reduce a transformation to a Petri Net, where exhaustion of tokens within the net’s places ensures termination, because the system cannot proceed any more. In [20], Bruggink addressed a more general approach by detecting in the rewriting rules infinite creating chains that are at the source of infinite rewritings. In [66], Küster proposes termination criteria with the same basic idea, but on graph transformations with control conditions. Pattern-based transformations [37] are a new approach for specifying model-to-model bidirectional transformations in a declarative, relational style (cf. Section 2.2), having Triple Graph Grammars (Tgg)s as a formal background. The language makes use of patterns to describe allowed and forbidden relations between two models. Patterns are compiled into Tgg rules to perform forward and backward transformations that can later be executed with classical Gbt engines (e.g., Fujaba [84]). In [37], de Lara and Guerra proved that this compilation scheme is sound, terminating and confluent; the result has been later extended in [46] to handle attribute conditions.

Termination criteria for Mpls directly benefit from the vast and active literature on imperative and object-oriented programming languages. These criteria usually rely on abstract interpretations built on top of low-level programming artefacts (like pointers, variables and call stacks). For example, Spoto et al. detect the finiteness of variable pointers length in [107]; and Berdine et al. use separation logics for detecting effective progress within loops in [12]. Since these techniques are always over-approximations of the Tls’s semantics, they are sound but not complete, and can potentially raise false positives.

3.1.1.2 Expressiveness Reduction

Reducing expressiveness regarding termination generally means avoiding constructs that may be the source of (unbounded) recursion. For example, DslTrans [9] uses layered transformation rules and an in-place style: rules within a layer are executed until they cannot match anymore, which occurs because models contain a finite amount of nodes that are deleted in the process, preventing recursions and forbidding loops syntactically.

3.1.2 Determinism

Determinism ensures that transformations always produce the same result. Generally, this property is only considered up to the interactions with the environment or the users. Considering this, Mpls are considered deterministic since they directly describe the sequence of computations required for the transformation.
3.1.2.1 Determinism Criteria

Determinism directly refers to the notion of confluence (often called the Church-Rosser, or diamond, property) for Gbtl s, which has also been proved as undecidable [92]. Confluence and termination are linked by Newman’s lemma [83], stating that confluence coincides with local confluence if the system terminates. This offers a practical method to prove it by using the so-called critical pairs. The Gbt community is very active in this domain and already published several results. In [52], Heckel et al. formally proved the (local) confluence for Typed Attributed Graph Grammars, and Küster in [66] for graph transformations with control conditions. In [67], Lambers et al. improved the efficiency of critical pairs detection algorithms for transformations with injective matches, but without addressing pairs of deleting rules. More recently, Biermann extended the result to EMF (Eclipse Modelling Framework), thus preserving containment semantics within the transformations in [15]. In another area, Grønmo et al. addresses the conformance issue for aspects in [45], i.e. ensuring that whatever order aspects are woven, it always leads to the same result. The compilation of pattern-based transformations into TGG rules are proven confluent in [37].

3.1.2.2 Expressiveness Reduction

Reducing expressiveness regarding confluence means either suppressing the possibility of applying multiple rules over the same model, or providing a way to control it. In DslTrans for example [9], the TL controls non-determinism occurring within one layer by amalgamating the results before executing the next layer. This ensures confluence at each layer’s execution, and thus for a transformation.

3.1.3 Transformation well-formedness

A crucial challenge for transformation specification is the detection of syntactic errors early in the specification process, to inform designers as early as possible and avoid unnecessary execution that will irremediably fail. This property primarily targets visual modelling languages, since textual modelling already benefits from experience gathered for building IDEs for GPLs, where a type system (usually static) reports errors by tagging the concerned lines. All syntactic errors cannot be detected, but a framework possessing this feature will considerably ease the designers’ work.

To achieve this goal, tools must rely on an explicit modelling of transformations [22]. Kühne et al. studied in [65] the available alternatives for this task and their implications: either using a dedicated metamodel as a basis for deriving a specialised transformation language, or directly using the original metamodel and then modulating the conformance predicates accordingly, for deriving such a language. Studying the second alternative, they proposed the RAM process (Relaxation, Augmentation, Modification) that allows the semi-automatic generation of transformation specification languages. On the other hand, Levendovszky et al. explored in [70] the other alternative by proposing an approach based on Domain-Specific Design Patterns together with a relaxed conformance relation to allow the use of model fragments instead of plain regular models.

A more traditional way of ensuring the well-formedness of transformation rules is to apply traditional static analysis on the transformation specification. Metaprogrammed languages directly benefit from techniques already available for programming languages (like dead code detection, call graph construction, etc.) Several contributions already addressed the case of Gbtl s: Wimmer et al. [121] proposed a taxonomy of rule-based transformation errors targeting various properties (such as rule redundancy, left-hand
3.2 Model Transformations: Transformation-Related Property

From a modelling perspective, a transformation refers to input/output models for which dedicated properties need to be ensured for the transformation to behave correctly.

This Section provides a comprehensive overview of properties of interest separated into two concerns: properties involving transformations’ source or target models in Section 3.2.1; and properties relating (meta-)models at each side of the transformations in Section 3.2.2.

3.2.1 Properties over models and their metamodels

A first concern of property verification addresses the input or output model(s) a transformation refers to, as depicted in Figure 5. The conformance property is historically one of the first addressed formally, because it is generally required by transformations to work properly (cf. Section 2.1). Transformations admitting several models as source and/or target require other kinds of properties, either required for transformations or simply desirable.

3.2.1.1 Conformance & Model Typing

Conformance ensures that a model is valid w.r.t. its metamodel, and is required for a transformation to run properly (usually, a transformation is supposed to be executed solely on conforming input models): this property is represented in Figure 5 by the vertical arrows labelled conforms to between model(s) and metamodel(s).
Usually, *structural* conformance, involving only the model, is distinguished from *constrained* conformance, which is an extended property that includes structural constraints not expressible with the metamodelling language, otherwise referred to as metamodels’ *static semantics* or *well-formedness rules* (see e.g. [19]). Nowadays, this property is well understood and automatically checked within modeling frameworks. However, proving that transformation’s output(s) always conforms to target metamodel(s) is not trivial, especially when using Turing-complete frameworks. Most of the time, an existing procedure for checking conformance is programatically executed after the transformation terminates.

It is nevertheless possible to prove this property in specific contexts. For instance, Baar and Marković [7] proved that UML class diagrams containing OCL constraints are correctly refactored: the refactored diagrams are still valid diagrams, and the OCL constraints are modified according to the refactorings such that they are still valid sentences. Similarly, Schätz [104] proved the conformance of output ECore models using a relational formal specification of metamodels and models; the proof is discharged in most cases automatically by the Isabelle/HOL theorem-prover.

Model Typing [108, 101] extends the notion of type beyond classes, by defining a *subtyping* relation on models. Transformation languages capable of manipulating model types offer better reusability for modelers, because instead of depending on low-level types, transformations become parametrised by whole models and apply to any sub-model. Model Typing has been recently extended by Guy [48, 49] to define a model-oriented type system aimed at overcoming the issues of the conformance relation when reusing the same transformations over similar metamodels.

### 3.2.1.2 N-Ary Transformations Properties

Unsurprisingly, transformations operating on several models at the same time, e.g. model composition, merging, or weaving, require dedicated properties to be checked. These properties are difficult to precisely represent on Figure 5 since they may involve the different models (input and/or output) as well as the respective metamodels.

Concerning **merging**, Chechik *et al.* follow an interesting research line in [29]: they enunciate several properties merge operators should possess: *completeness* means no information is lost during the merge; *non-redundancy* ensures that the merge does not duplicate redundant information spread over source models; *minimality* ensures that the merge produces models with information solely originating from sources; *totality* ensures that the merge can actually be computed on any pair of models; *idempotency*, which ensures that a model merged with itself produces an identical model. These properties are not always desirable at the same time: for example, completeness and minimality become irrelevant for merging involving conflict resolution. Beyond merging, they can potentially characterise other transformations, not necessarily involving several source models.

Concerning **aspect weaving**, Katz identifies in [58] temporal logics to characterise properties of aspects: an aspect is *spectative* if it only changes variable within this aspect without modifying other system variables or the control flow; it is *regulative* if it affects the control flow, either by restricting or delaying an operation; it is *invasive* if it changes system variables. Static analysis techniques enriched with dataflow information or richer type systems are generally used to detect these properties. Despite their textual programming orientation, these properties should apply equally in MDE. In [78], Molderez *et al.* present DELMDSoc, a language for Multi-Dimensional Separation of Concerns implemented in AGG [113]. Ultimately, this framework will allow the
3.2.2 Properties relating input and output models

A second concern for property verification targets the many types of relations that can be established between the input model(s) and the output model(s). We distinguish two application levels: syntactic properties relate the constitutive elements of models (e.g. for Mof, it would be objects, attributes and so on) whereas semantic properties require the often implicit semantics of the metamodels involved, meaning that the relations rather take place between the constitutive elements of the semantic domains. Figure 6 illustrates how the different elements come into play.

3.2.2.1 Syntactic Properties

A model transformation consists in general of a computation that applies repeatedly a set of rewriting rules to a model, where the model represents the structure of a sentence in a given formal language, defined by a metamodel. Because transformation execution is in general a complex computation, the production of a given output model cannot in general be inferred by just looking at the individual transformation rules. It thus becomes important to make sure that certain elements, or structures, of the input model will be transformed into other elements, or structures, of the output model. By abstracting these structural relations between input and output models and expressing them at the level of the graphs defining those model’s languages (or metamodels), it is possible to express relations between all input models of a transformation and their expected outputs.

We call this type of properties syntactic relations, because they relate the shape of a (set of) input model(s) with the shape of a (set of) output model(s). Given that the models we are transforming do not in general include an explicit description of their

---

**Figure 6** – Properties that relate input and output models, either at a syntactic (cf. Section 3.2.2.1) or at a semantic level (cf. Section 3.2.2.2). Semantic properties require at least an explicit, eventually partial, representation of the metamodels’ semantic domains, depicted here symbolically by transitions systems.
Formal Verification Techniques for Model Transformations: A Tridimensional Classification

In [1] Akehurst and Kent formally introduce a set of structural relations between the metamodels of the abstract syntax, concrete syntax and semantics domain of a fragment of the UML. In order to achieve this they create an intermediate structure that relates the elements of both metamodels as well as the elements of the intermediate structure itself. Despite the fact that they only apply it to an academic example, the proposed technique appears to be sufficiently well founded be applied in the general case where one would wish to express structural relations between two metamodels. Narayanan and Karsai also define in [80, 82] a language for defining structural correspondence between metamodels that takes into consideration the attributes of an entity in the metamodel. In particular they apply their approach to verifying the transformation of UML activity diagrams into the CSP (Communicating Sequential Process) formalism. In the same paper they point out the fact that the formalism used to define model transformations in Triple Graph Grammars (TGGs) [6] could also be used to encode structural relation properties between two metamodels. Lúcio and Barroca formally define in [71] a property language to express structural relations between two language’s metamodels and propose a symbolic technique to verify those relations hold, given an input and an output metamodels, and a transformation. Orejas and Wirsing [87] proposed an algebraic framework for representing pattern-based transformations [46], for which they provide a theoretical framework for proving the consistency of a transformation specification with a verification specification, also expressed using patterns. The approach is currently under implementation in Maude [31].

3.2.2.2 Semantic Properties

Beyond structural relationships between source and target models, it may be interesting to relate their meaning. Semantic properties are more difficult to define and prove: although they are defined using elements of the transformation specification and (meta-)models it refers to, they apply on their semantic domains, which requires to dispose of at least a partial explicit representation of the models’ semantics, or a way of computing it. In Figure 6, we have introduced a symbolic representation of the semantic domain of each model(s) involved in the transformation.

Semantic Preservation

Preserving the semantics of models while transforming them can be achieved in many ways. However, since in this case, the transformation may be exogenous, one needs to precisely state when the semantics is considered “equivalent”.

Bisimilarity, a stronger variant of system similarity, states that two systems are able to simulate each other from an observational viewpoint. Proving such a semantic relations requires to establish a relation between the transition systems capturing the execution of both input and output models. When possible, this proof is conducted inductively to ensure that it holds on any input model, or it is specialised for specific inputs.

Narayanan et al in [81] show how to verify that a transformation outputs a statechart bisimilar to an input statechart. Combemale et al. [32] formally prove the weak simulation of xSPoM models transformed into Petri Nets, in the context of the definition of translational semantics of Domain-Specific Languages, thus enabling trustable verification of properties on the target domain. Similarly, Blech, Glesner and Leitner [16] proved the bisimulation between StateChart models and Java-like
generated code. Rahim and Whittle proved in [95] a simplified form of what they called semantic conformance, i.e. the fact that a code generator (from UML State Machines to Java) preserves the semantics of the original metamodel. Semantic considerations are injected as annotations into the generated Java code that is later automatically model-checked by Java PathFinder [120]. This approach has two major drawbacks: one has to inspect the generated code to insert the annotations, and the semantic conformance only covers the properties specified by these annotations.

Another way of capturing the semantic preservation consists of expressing the correctness through an external language over the input and the output models. In [117] Varró and Pataricza use CTL (Computation Tree Logic) [30] as a language to state properties on each side of the transformation. They then prove that the CTL formulæ still hold when transforming StateCharts into Petri Nets. Both metamodels have a close state-based semantics, but in the general case, it seems more difficult to state how formulæ at each side are considered equivalent, especially if the languages at each side are different.

**Behaviour Refinement** Instead of proving preservation, one may be interested in refinement, meaning that every behaviour of the input model is allowed by the output (although the output may present more behaviours). Heckel and Thöne [98] propose a notion of behaviour refinement of GBT models. These models are in fact model transformations specified over source and target metamodels that allow expressing system configurations. The authors provide the formal description of a relation between the labelled transition systems generated two graph-transformation based models such that one is a refinement of the other.

**Safety (Temporal) Properties & Invariants** The fact that two systems are able to simulate each other pertains to the observational behavior of those systems. One may wish to enforce a relation between the actual states of the behavioral input and output models. Several contributions addressed in the recent years the formal verification of temporal properties. The idea consists in representing metamodels, models and transformations in an external formal framework already equipped with verification capabilities, generally delegated to a dedicated tool. Among others, Abstract State Machines provide CTL model-checking by using another mapping into the SPIN model-checker (used in the AsMeta framework [44]), and Maude has been the verification target for several works, providing reachability analysis, theorem-proving by mapping Maude specification into Isabelle/HOL, and LTL model-checking: the approach was used for verifying visual DSLs whose behavioural semantics is expressed using GBT in single/double pushout styles [100, 102], and by Barbosa et al. [8] for proving a relatively simple form of semantic preservation on Java imperative programs by comparing the resulting semantic domains (note that this last work uses a three-lines Java example whose scalability is questionable since none of the challenging Java construction — conditional and loop statements, but also casting and method calls — are handled). Rangel et al. [97] proposed a framework for preserving the behaviour of refactorings in a GBT framework using borrowed contexts that works both at the rule and the transformation levels.

An interesting subset of safety properties are invariants expressed over the set of reachable states of a system. The idea is that certain conditions can never be violated during execution. In this sense, Becker et al are able to prove in [11] that safety properties (expressed as invariants) are preserved during the evolution of a model representing the optimal positioning of a set of public transportation shuttles running on common tracks. Given the evolution of the model is achieved by transformation,
the safety properties will enforce that the shuttles do not collide each other during operation. Padberg et al introduce in [88] a morphism that allows preserving invariants of an Algebraic Petri Net when the net is structurally modified. Although this work was not explicitly created for the purpose of model transformation verification, it could be used to generate a set of model transformations that would preserve invariants in Algebraic Petri Nets by construction. Cabot et al. [23] proposed a technique for proving transformation correctness based on the extraction of behavioural invariants from the transformation rule specification (the technique is applied to Triple Graph Grammars and Qvt). They also propose a catalogue of specialised invariant-based properties characterising both languages, applied at two levels (individual rules and whole transformation) that they can extract and verify. Similarly, Ledang and Dubois [69] formalised the semantics of Gbt in ATL in B and proved rule consistency (i.e. whether they are contradictory) and the correctness of Gbt by formally proving user-defined invariants.

Consistency of Bidirectional Transformations Bidirectional transformations (often coined as “Bx”, echoing the well-known workshop and community working on this topic [21]) consist of coupled transformations that keep two (or more) models consistent: when one model is changed, transformations are triggered to reflect the changes within the other; alternatively, when both models are changed, the transformations try to reconcile the models. Bidirectional transformations are useful in many contexts, including cooperative editing, co-evolution, view/update synchronisation (e.g. a change in a Graphical User Interface modifies the underlying system), reverse engineering and so on: Czarnecki et al. [36] and Stevens [110] propose an interesting survey of Bx transformation characteristics, usage scenarios, challenges and research directions, practices and tools. Decomposing bidirectional transformations as a pair of transformations is not sufficient, since new challenges arise when considering the models involved as closely related: one of the key properties consists of ensuring the consistency between models. Lämmel [73] addresses consistency through different kinds of model reconciliation (degenerated, symmetric and asymmetric) that depend on the final use. Stevens [111] extensively studied the case of bidirectional transformations for the OMG QVT standard. She defines model consistency through a relation that two directional transformations should enforce in a coherent way, meaning that both transformations should be correct, i.e. the transformation actually enforce the relation, hippocratic, i.e. if models are already in relation, there is no modification), but also undoable, i.e. reverting one model should correctly revert the other. The author also proved that transformation coherence is preserved by composition. Triple Graph Grammars are a good candidate for bidirectional transformations based on Gbts. Ehrig, Ehrig, Ermel et al. [40] formalised Triple Graph Grammars in terms of category theory, and defined sufficient conditions for a transformation to be reversible while preserving graph information. König and Schürr [64] on the one hand, and Hildebrandt et al. [54] on the other hand, proposed a survey and a comparative study of the theoretical results and the tool support available for Triple Graph Grammars: they emphasise the necessity of correctness (i.e. models in a triple should be a valid member of the Grammar) and the completeness (i.e. any input model extensible to a consistent triple must be extended for both transformations) for certain classes of Triple Graph Grammars, in order to carry useful information for the user about the manipulated models (i.e. a transformation failing within a tool satisfying completeness says that the model is invalid).
Table 1 – Classification of Contributions according to Property Kinds

### Structural Semantics Properties

Models may have a structural semantics, rather than a behavioral semantics. This is the case of UML class diagrams, which semantics is given by the `instanceOf` relation. In this case, although the behavioral properties mentioned above do not apply, relations between the structural semantics of input and output models may still be established. Massoni et al [76] and Baar and Marković [7, 75] present a set of refactoring transformations that preserve the instantiation semantics of UML class diagrams.

### 3.3 Summary

Table 1 classifies the literature contributions we reviewed according to the property classes they are targeting, based on the dual nature of model transformations. This research emphasised two levels property classes are operating at: at the level of transformation languages, the property classes correspond to those of any computational language; at the level of model transformations, we distinguished two classes: properties between models (input or output) and their metamodels do not involve the transformation beyond the pair of models studied; whereas properties relating the input and output model(s) consider the transformation either at a syntactic or at a semantic level.

Not surprisingly, termination and determinism are broadly explored for graph-based approaches, since they are recurrent issues for this kind of model transformation frameworks.

The way we presented our model transformation classes also reflect the relative difficulty to establish the correspondent correctness proofs: the first class only requires a lightweight possibility for ensuring correctness, since no specific transformation details are involved; the second class includes more details depending on the level it operates at. Obviously, since properties are expressed over the artefacts referred to by the transformation specification, semantic properties require a detour through the semantic domain that is not always explicitly represented, which makes them harder to cover, and therefore less explored. They also reuse existing techniques already explored for general purpose programming languages.
4 Dimension 3: **Formal Verification (FV) Techniques**

We propose to discuss FV techniques according to two criteria: whether a technique is specific to a particular transformation or holds for any transformation written in a given TL; and whether a technique is specific to a particular input or holds for any conforming input model of a transformation. Our classification for FV techniques follows the three different types obtained by combination of these criteria.

This Section starts by reminding what we exactly mean by *formal verification*, then continues by discussing examples of each type of verification technique from the literature.

4.1 Formal Verification in a Nutshell

In this Section, we explain what we mean by *Formal Verification* in the context of model transformation, by dressing a parallel with the formal verification of programs as perceived in the Computer-Aided Verification community. This Section is directly inspired from [34, 2]. We first explain the Fv problem, then identify several characteristics that distinguish Fv from other validation techniques, and finally describe issues and challenges.

4.1.1 The Formal Verification Problem

The Verification problem is a decision problem, i.e. whose answer is either yes or no, that is generally undecidable: it consists of answering non-trivial (and thus, interesting) properties about the possible executions of a computation. The execution of a program, or a transformation, or any piece of software or hardware material, is influenced by several internal or external factors (e.g., the human interactions, or the captors and sensors interacting with the physical world). The so-called concrete syntax captures the values of all possible variables involved in an execution, and can be represented as a variable vector $x(t)$ evolving over time, like a trajectory in physics. In this context, a property of interest designates a set of variable values that characterise erroneous, or undesired, states that the execution should go through.

In Figure 7 (left), these trajectories are described by the numerous lines whereas the erroneous states are represented by the red zones. The verification problem then consists in mathematically proving that the program executions (i.e. the concrete semantics) never reaches these forbidden zones. Unfortunately, this concrete semantics is more a mathematical object than an actual computation: it is often infinite, even for very simple programs, and thus nearly impossible to compute explicitly. The verification problem is undecidable, meaning that it is not always possible to answer this kind of questions completely automatically (i.e. without any human intervention, using finite computer resources, and without any uncertainty).

4.1.2 Characteristics of FV Techniques

FV techniques have several characteristics that make them reliable, but often difficult to employ on real-life software. An FV analysis is offline (or static), meaning that applying such techniques do not require actually executing the analysed program (implying that the analysed program is treated as an input of the analysis rather than instrumenting it to extract relevant data for its correctness); it is exhaustive (or full-covering), meaning that the analysis covers all possible execution paths without exception and entirely, which ensure the absence of errors under all possible circumstances; and sound (or
Figure 7 – The Formal Verification Problem — Trajectories represent possible executions of the system; red zones represent erroneous states. It consists of mathematically proving that no trajectory crosses the red zones (left). To prove that, an abstraction (the green light area on the right), easier to compute than the set of trajectories, overapproximates the trajectories in a sound, exhaustive way. The problem is reduced to prove that the green area does not intersect the red zones.

correctly abstracting), meaning that if the analysis concludes to the absence of errors, it implies there is actually no errors for the range of properties checked.

To achieve exhaustivity and soundness, Fv techniques operate by abstraction: since computing the concrete semantics is not feasible in practice, property checking is performed on an over-approximation of the concrete semantics called abstract semantics. One needs to prove that the abstract semantics is indeed an over-approximation, otherwise, the resulting analysis is not sound: some behaviours, not covered by the abstraction, can be erroneous but will not be detected because they are not covered by the abstraction. Figure 7 (right) depicts a possible sound abstraction in light green.

In practice, because of the undecidability, tools have to balance the different parameters to propose tractable analysis. A first possibility is to relax the answer expected for the decision problem while preserving soundness to obtain a semi-decidable decision algorithm: if no errors are detected, then the system is safe for the set of properties checked; otherwise, the analysis could loop forever and there is no certainty. This possibility is interesting, but has the serious drawback to not allow detecting errors that permit their fixing. Another possibility to overcome undecidability is to require human assistance: this is often the case for theorem-provers like Coq [13] or Isabelle/HOL [85]: the user guides the proof towards the goal when automated procedures fail. One would want to sacrifice even more important criteria, like exhaustivity: the bounded model-checking technique, used e.g. in Alloy [56], assumes the so-called small scope hypothesis (stating that if there is an error, it is likely to appear very early in the execution) to accelerate decision procedures, at the expense of potentially missing late errors.

4.1.3 Issues & Challenges

Fv techniques face several well-known issues that hinder their use and their results. The explosion problem results from the exhaustivity requirement: when not efficiently balanced, the execution representation becomes so large that it makes the analysis itself impossible. Because of the over-approximation requirement, false alarms can appear: by representing more behaviours than actually needed, an analysis can detect an erroneous behaviour that does not correspond to an actual execution of the system, but that is induced by the abstraction itself. Backward traceability is one of the most crucial problem Fv is facing, and more accurately in the context of transformations: when a property is violated, Fv techniques usually produce counterexamples in terms
of the abstract semantics that need to be related to the concrete one in order to interpret the results more appropriately in the formalism the program was written in. However, abstractions are often difficult to reverse, so that programmers have to explicitly understand the abstract semantics to correct the discovered errors.

These issues are intrinsically connected to the nature of Fv and its associated requirements, meaning that they are raised for any analysed artefact including model transformation. In order to enable a wider adoption of these techniques in Mde, it is crucial to address two important challenges:

- A first challenge is the adaptation of Fv techniques, and more generally any analysis approach, to the higher abstraction level of Mde artefacts (namely, models and transformations) that should be taken into account in order to better correspond to the richer data structures and computations manipulated by modelers and transformation designers.

- The second challenge is the extreme versatility of Mde: while traditional Fv techniques assume that the full definition of the execution is already available, Mde proceeds in a more agile way that calls for many development rounds, each of them being potentially submitted to analysis. For example, generating the code corresponding to a model is done iteratively, while it often requires that the full final code becomes available before checking that the transformation implementing the code generation is a correct refinement.

### 4.2 Formal Verification Types

In this Section, we discuss Fv techniques proposed in the literature to prove Mt properties implemented in various Tls. Table 2 captures our classification of Mt verification techniques, which fall into one of three major types: Type I Fv techniques guarantee certain properties for all transformations of a Tl, i.e. they are transformation independent and input independent. Techniques of Type II prove properties for a specific transformation when executed on any input model, i.e. they are transformation dependent and input independent. Techniques of Type III prove properties for a specific transformation when executed on one instance model, i.e. they are transformation dependent and input dependent. When a Fv technique is transformation independent, it implies that no assumption is made on the specific source model: this explains why, in Table 2, the case representing Fv techniques that are transformation-independent and input-dependent is empty.

Although applicable to any transformation, Type I verification techniques are the most challenging to implement since they require expertise and knowledge in formal methods. Type III verification techniques are the easiest to implement and are considered "light-weight" techniques since they do not verify the transformation per se; they verify one transformation execution. Across all the three types of verification techniques, the approaches used often take the form of model checking, formal proofs or static analysis.

Different properties discussed in Section 3 were verified in the literature using different techniques from the three types. Some properties (e.g. termination) were proved only once by construction of the model transformation or the Tl. Proving such properties required less automation and more hand-written mathematical proofs, although some studies used theorem provers to partially automate the verification process. Type I verification techniques were used to prove such properties. Other properties (e.g. model typing and relations between input/output models) were
proved repeatedly for different transformations and for different inputs. Proving such properties required more automated and efficient verification techniques from Type II and Type III techniques.

4.2.1 Type I: Transformation-Independent and Input-Independent

Properties that hold independently of the transformations expressed in a particular Tl are normally proved once and for all. Performing tool-assisted proofs is cumbersome: it requires to reflect the semantics of the underlying Tl directly in the Fv tool, which is a heavy task. For example, formally proving termination for Gbt with a theorem-prover requires to express the semantics of pattern-matching in the theorem-prover’s language. Therefore, these kinds of proofs are usually presented mathematically. Barroca et al. [9] prove termination and confluence of DslTrans inductively, following the layered syntactic construction of the language’s transformations. Ehrig et al. [41] address termination of Gbts inductively, by proving termination of deleting and non-deleting layers separately. In [66], Küster proposes sufficient conditions for termination and confluence of Gbt with control conditions, by formalising the potential sources of problems within the theory of graph rewriting.

Another proof strategy consists of taking advantage of existing machinery in a particular formalism. For example, several techniques for proving termination exist for Petri Nets. The challenge is then to provide a correct translation from the metamodeling framework and the Tl within the Petri Nets technical space. Varró et al. [118] proves termination by translating Gbt rules into Petri Nets and abstracting from the instance models (i.e. the technique becomes input-independent); the proof then uses the Petri Nets algebraic structure to identify a sufficient criterion for termination. Padberg et al. prove in [88] that safety properties, expressed through invariants over Algebraic Petri Nets, transfer to refined Nets if specific modifications of the Nets are followed, thus guaranteeing the preservation of these safety properties. Massoni et al. [76] checks the validity of refactoring rules over Uml class diagrams by translating everything in Alloy to discover inconsistencies in the rules, taking advantage of the instance semantics of Alloy.

Another interesting line of research consists of expressing Mde artefacts in the context of type theory. Taking advantage of the Curry-Howard correspondence (or isomorphism), this theory sees programs (or transformations) as proofs that the modeler and the transformation designer have to discharge; the system then automatically generates a correct-by-construction executable specification. At least two different logics are becoming popular: the Calculus of Inductive Constructions, inspired from typed lambda calculi, at the basis of Coq [13], and the Constructive Type Theory at the foundation of NuPrl [33]. These formalisms have been used in the context of Mde to extract certified transformations: Calegari et al. [25] used Coq for handling ATL transformations, whereas Poernomo [93, 94] used directly NuPrl to formalise MOF. Unfortunately, both contributions are only demonstrated on the famous Class Diagram to Relational Databases classical example. The effort of such a formal specification in logics that are not directly intuitive for Mde engineers seems to hinder its wide applicability.

4.2.2 Type II: Transformation-Dependent and Input-Independent

For this type of verification, classical tool-assisted techniques are generally used: model-checking, static analysis, theorem-proving and abstract interpretation. We briefly recall their characteristics before reviewing the respective contributions.
4.2.2.1 Characteristics

From Section 4.1.2, the verification problem has three components: an abstract semantics that approximates the “real” concrete semantics; a specification of the (set of) properties of interest; and a way to check the latter against the former. Assuming the abstraction correctly defined, classical verification tools aim at automating the checking process, given the other components. They vary however with respect to three elements: which form (sometimes implicit) possess the abstraction, and consequently, which role has the user building it; and which flexibility regarding the range of properties the technique is able to handle. Since only practical considerations are important here, rather than the theoretical considerations, we briefly review each Fv technique in light of these variations points [34].

**Static Analysis** consists in a predefined approximation, generally automatically pre-computed over the analysed entity. Sometimes, the user can parametrise the analysis in order to focus on specific properties. This technique is nowadays well mastered, and accompanies very often development tools for providing assistance and computing relevant information from a program to help developers better understand their programs and correct simple mistakes. Different forms of static analysis have been popularised during the last decades.

Some techniques are very old, because they were studied for optimising compilation of programs (e.g., live variables or dead code detection, for optimising registry allocation and code generation). Other techniques emerged with new programming paradigms like object orientation (e.g., inheritance hierarchy for tracking fields/methods redefinition and overloading; graph calls for better understanding objects interactions, escape and shape analysis for improving data representation).

**Model-Checking** consists in approximating programs into finite dynamic structures (most of the time, labelled transition systems), and providing a formal logic for expressing properties of interest independently from the program. Finite structures obviously result from the fact that some details are lost during the process: for example, for detecting deadlocks, it is often enough to forget about the actual values of variables and focus solely on the interactions executed in parallel.

The user has to provide the finite representation; the model-checker basically automates the checking. Most of the time however, this finite representation is automatically extracted from the program itself, using static analysis techniques, which relieves the user from this burden. A model-checker usually provides a violation trace if the property does not hold: this trace indicates a scenario leading to a violation, which helps the user figuring out what went wrong and where. However, due to the usual semantic gap between program languages and model-checker representations, it is often a real challenge to trace violation paths back to the original program.

**Theorem-Proving** consists of specifying a program by means of inductive properties satisfying verification conditions. Basically, properties of interest have the form of predicates whose truth is checked against the inductive properties representing the program.

The user has to provide such a specification; the theorem-prover basically automates the proof burden, i.e. the fact that these properties are indeed inductive.
However, due to the undecidability issue, the theorem prover sometimes needs guidance to fully discharge the proof. The specification can be partially discovered directly from the program by using static analysis techniques.

**Abstract Interpretation** somehow generalises the previous approaches by allowing any kind of approximation to be defined. The user then has then to prove that the proposed approximation is sound, and the abstract interpreter automates the verification process for the properties. By using predefined abstract domains, the approximation can eventually be automatically computed, but this restricts the range of properties the abstract interpreter is capable of checking.

As previously stated, we did not found any contributions making use of abstract interpretation as a \textit{Fv} technique.

### 4.2.2.2 Techniques

**Static Analysis** Becker et al. [11] proposed a static analysis technique to check whether a model transformation (formalized as graph rewriting) preserved constraints expressed as (conditional) forbidden patterns in the output model. The study proved that the structural adaptation does not transform a safe system state to an unsafe one by verifying that the backward application of each rule to each forbidden pattern cannot result in a safe state. Vieira and Ramalho [119] built a Java Api for automatic inspection of ATL transformations: by calling appropriate methods in the Api, transformation designers could retrieve the relations and dependencies of any artefact involved in a transformation (e.g., which rules are impacted by the firing of another one). Planas et al. [90] proposed to compute rule weak executability (i.e. whether a rule can be safely fired without breaking input/output models’ integrity) and rule set covering (i.e. whether a set of rules cover all elements of the source/target metamodels) by static analysis on an ATL specification.

**Model-Checking** Rensink et al. compared in [99] two approaches for the model-checking of GBTS. The first approach used the CheckVML Tool to transform a GBT system to a Promela model, further verified using SPIN. The second approach used the Groove Tool to simulate GBT rules and build a state space of graphs for model-checking. The second approach was found more suited for dynamic and symmetric problems. Lucio et al. [71] implemented a model-checker for the DSLTrans Tool that builds a state space for a transformation where each state is a possible combination of the transformation rules of a given layer, combined with all states of the previous ones. The generated state space is then used to prove if properties hold for all input models of the transformation. Varró and Pataricza [117] used model checking to prove that dynamic consistency properties were preserved in a model transformation from statecharts to Petri Nets. Rahim and Whittle [95] used the Java PathFinder model-checker [120] to check the semantic conformance of the Java code produced by two industrial code generators (Rhapsody and Visual Paradigm) with respect to the semantics of UML State Machines: they annotated the output Java program with property specifications that the model-checker can then handle automatically.

**Theorem Proving** Asztalos et al. [74] proposed deduction rules that can be applied to model transformation rules (formalized as graph rewriting) to prove or disprove a property. The deduction rules were implemented as a verification framework in VmTS (Visual Modeling and Transformation System) and was used to verify a refactoring
transformation on business process models. Paige et al. [89] compared two approaches for the verification of model conformance checking and multi-view consistency checking (Mvcc): with PVS, a popular theorem prover based on set theory; and with Eiffel, an object-oriented language. Nevertheless, performing Mvcc checking requires actually executing the generated Eiffel code. Giese et al. [55] proposed formalizing Model-to-Code transformations using TGGs in Fujaba, further verified within Isabelle/HOL. Again with Isabelle/HOL, Blech, Glesner and Leitner [16] proved that a code-generation transformation preserves the bisimulation between StateChart models and Java-like code. Isabelle/HOL was also used by Schätz [104] to prove the conformance of output models. Marković and Baar [7, 75] proved that a Java implementation of some of their refactoring rules for UML class diagrams containing OCL constraints preserve the instantiation semantic of class diagrams (although the reimplementation in Java of the refactoring rules can be seen as a non-direct proof of the rules’ OCL implementation). Lano and Kolahdouz-Rahimi [68] used the B Method to prove several properties over UML-RSDS (syntactic correctness, uniqueness, and transformation confluence), from which trustable Java code is generated automatically. B was also used by Ledang and Dubois [69] to prove that user-defined invariants hold over GBTs. Barbosa et al. [8] used the Maude’s Interactive Theorem Prover to prove a relatively simple form of semantic preservation.

4.2.3 Type III: Transformation-Dependent and Input-Dependent

The following techniques are close to testing, since they hold for one particular model. However, these techniques still cover all possible execution paths and can be exploited offline.

4.2.3.1 Using Traceability Links

To prove that a specific transformation preserved certain properties for a specific input model, some studies proved that input-output relationships are maintained for a transformation instance. Narayanan and Karsai used GREAT for both structural and semantic relationships between source and target models. In [80, 82], they generate crosslinks between source and target models to check structural correspondence between source and target models. In [81], they check state reachability in a transformation between StateCharts to Extended Hybrid Automata, by checking the existence of a bisimulation with the help of crosslinks between source and corresponding target models.

4.2.3.2 Using Petri Net Analysis

Lara and Vangheluwe [39] formalized the operational semantics of a visual TL using graph rewriting. The transformations and the manipulated models were transformed into Petri Nets to benefit from existing FV techniques. The study further proposed extending graph rewriting rules with timing information and transforming them into timed Petri Nets for formal verification.

4.2.3.3 Using Constraint Solvers

Anastasakis et al. [5] used Alloy for simulation and assertion checking. Source and target metamodels, as well as transformations, are represented as Alloy models. The Alloy Analyzer then generates possible instances of the source metamodel and the transformation; the Analyzer is then used to check if the corresponding target model satisfies assertions. If no instance is found, it reveals inconsistencies in the
transformation specification. Cabot et al. [23] used CSP solvers to prove that invariants automatically extracted from QVT and Triple Graph Grammars rules hold, by reusing an existing transformation from UML [24].

4.2.3.4 Using Contracts

Design by Contracts is a well-known technique that consists of defining precise and verifiable interface specifications for software components (e.g., in object-oriented GPLs, these components could be classes, operations, or bigger combinations) through invariants and pre-/post-conditions [14]. Strictly speaking, design by contracts is not a FV technique because it requires executing the transformation to discover contract violations. However, we mention it because contracts can be formally exploited in two ways: translating the contracts into a theorem-prover or a model-checker, assuming that there is a way of resolving contracts underspecifications, typically by human assistance (cf. e.g., the case of globally asynchronous, locally synchronous systems relying on Promela and the Scade Verifier [50]); or proving that an implementation respects the contract specification (cf. Eiffel itself [14] or the GNATProve Project for Ada [114]). However, this still need more investigations in order to apply to current metaprogrammed transformation languages.

Cariou and his colleagues [28, 27] studied the use of OCL for the specification of contracts for model transformations: they associate preconditions over the input model for being eligible to transformation, postconditions on the output model for being considered as a valid result, and other constraints expressed over the pair of input/output models that track how model elements evolve during transformations. They also applied their technique for the specification of DSL semantics (following the operational approach of Figure 3) and have demonstrated the feasibility of the approach on UML state machine executions [27]. Based on OCL, the authors noticed limitations due to the mono-context nature of OCL and the difficulty of applying such approach on exogeneous transformations because it requires the manipulation of an amalgamated metamodel (i.e., a metamodel that includes both the source and the target). Guerra, de Lara, Wimmer et al. [47] proposed PaMoMo, a platform for declaratively specifying requirements independently of the language used to perform the transformation. The visual contracts specifying the requirements can express pre-/post-conditions (i.e., conditions for a model to be eligible for transformations, and patterns in an output models to capture required or forbidden configurations for the result) and invariants for the input/output pair of models, and can be automatically translated into QVT-R for checking that a given transformation (expressed using GBT) respects them. Being formally defined, the PaMoMo contract language allows an enhanced diagnosis process when contracts are violated, pointing precisely to model elements involved in the violation, as well as reasoning on the consistency of the contracts by detecting redundancies, contradictions and pattern satisfactions and assessing metamodel coverage.

4.3 Summary

Table 2 classifies the literature contributions according to the last dimension, namely the formal verification technique employed for ensuring transformation correctness. This research emphasised two cross-cutting levels: whether a verification technique depends on the transformation at hand, or applies to all possible transformation expressible within the transformation framework; and whether it depends on a particular input
Formal Verification Techniques for Model Transformations: A Tridimensional Classification

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Independent</th>
<th>Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I:</td>
<td>[9][88][76][109][41] [118][66][25][93][94]</td>
<td>[99][71][11][74][89] [55][69][117][98][95] [119][16][7][75][8]</td>
</tr>
<tr>
<td>Type III:</td>
<td>[81][39][82][5][23][28][27]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 – Classification of Contributions according to Formal Verification techniques.

model, or for all possible conforming input models. This classification offers a graduated evaluation of the effort needed for the verification process: the more specific it is (i.e. specific to a transformation and an input model), the easier and “lightweight” the technique can be performed. At one extreme, transformation-dependent and input-dependent techniques are very close to testing, but with the fundamental difference that they are still offline.

Another result of this research is the large spectrum of techniques already implemented for verifying model transformations. With the notable exception of Abstract Interpretation, all other techniques are at least implemented by one contribution, the most represented being model-checking. Beyond its relative popularity, model-checking is an attractive technique for two reasons: it is fully automatic, and the state explosion problem can be sometimes circumvented within specific contexts, when the models involved in a transformation are sufficiently small.

5 Discussion

Our tridimensional classification captures all variation points influencing the activity of formally verifying model transformations. In this Section, we first revisit the tridimensional classification, then elaborate on the relation between model transformation intent and characteristic properties.

5.1 Revisiting the Tridimensional Classification

We now revisit our tridimensional classification by studying the relations between each pair of dimensions, extracting lessons from our survey.

5.1.1 Property Kind / Fv Technique (PK/FVT)

This relation is the best explored within this paper, and directly relates to the contributions made by the Computer-Aided Verification community: on the one hand, we distinguished two property kinds that follow the dual nature of model transformations [22] to obtain language-related and transformation-related properties; on the other hand, we identified three different types of Fv techniques that depend or not on the transformation and the input.
Transformation
Verification
Technique
Property (kinds)
PK/FVT
T/PK
T/FVT
Clockwise
Scenario
Anticlockwise
Scenario

Figure 8 – Closing the loop: the relation between each pair of transformation dimension; and the two possible scenarios for model transformation verification.

From the literature, we showed that language-related properties, such as termination or determinism, are often proved mathematically: when it is possible to establish such properties for all transformations expressible in a given \( Tl \), the proof is discharged mathematically once and for all; otherwise, sufficient criteria (also mathematically proved) are integrated into \( Tls \) to help transformation designers establish those kind of properties for each transformation. The difficulty is then to ensure that the \( Tl \)'s implementation adequately follows the mathematical proofs.

Classical FV techniques, such as static analysis, model-checking, theorem-proving, are mostly employed for transformation-related properties that are related to the transformation’s semantics, and that have to be verified on all possible inputs. We note that abstract interpretation is largely absent from the reviewed contributions. Two explanations can justify this fact: the underlying difficulty of the mathematical underground, and the lack of general-purpose tools.

It is sometimes interesting to prove some properties of interest on a specific input only, for example when the transformation is used on a limited number of input models that need to be deployed within an application (e.g. a transformation expressing the behaviour of a Dsl). In this context, the classical techniques remain applicable but lightweight approaches become also interesting because they are generally easier to deploy.

An interesting trend in the literature is the use of model syntactic correspondences. Although the idea already exists for programming languages, it reaches another level of complexity for model manipulations. Capturing the similarity of the input and output models through patterns or contracts expressed on each side avoids diving into the complexity of the semantic layer. This trend has interesting results for structural models, but can only provide a quick check for more complex models that integrate behaviour.

5.1.2 Transformation / FV Technique (T/FVT)

This relation remains largely unexplored in our work as well as in the literature. It seems natural that the underlying paradigm of \( Tls \) would influence the spectrum of FV techniques that are usable for proving some property kinds. The respective research communities (of programming languages on the one hand, and of verification on the other hand) could provide interesting knowledge about the core principles
governing this relation, and to which extent it is possible to adapt this knowledge to the manipulation of models.

From the literature, we noticed however an interesting trend. Instead of developing specific techniques for model transformations, some contributions took the opposite approach: they express the full semantics of a TL within a verification domain that is usually a general-purpose programming language already equipped with analysis capabilities. For example, Maude, a powerful algebraic specifications rewriting engine allowing simulation, model-checking and theorem-proving within the same framework [31], has been chosen as a semantic and verification domain for various TLs belonging to different styles: GHT in simple/double pushout style [102]; object-oriented metaprogramming in Kermeta [2]; and hybrid transformations in ATL [116]. This approach is interesting because it is often perceived as easier than developing an analysis engine from scratch. However, it is limited by two factors: first, the spectrum of verifiable property kinds is limited by the verification domain itself; second, it forces to have a minimal knowledge of the verification domain in order to interpret the verification answers, since the backward translation is almost never implemented.

The previous trend requires a deep understanding of both the TL’s and the verification domain’s semantics in order to adequately express the first into the other. For reaching better results, and overcoming the limitations that may hinder the verification process, we believe that the MDE community should invest into two research directions for enabling formal verification scalable in an industrial context. First, investigating the development of verification engines that are fully aware of the specificities of models and their manipulation could bring a significant improvement both in terms of usability (thus overcoming the second limiting factor), as well as performance and scalability (because translations to verification domains are avoided). Second, deriving models specifically for the verification process could be interesting: those models would contain additional information for “guiding” verification algorithms or for decorating them with information computed during verification: e.g., annotating operations with call sites to accelerate call graph construction1. Of course, such a practice raises the usual questions: how is it possible to maximise the automation of the derived models creation, and how to keep both models synchronised. We expect the community to focus on those topics in the upcoming years, when MDE will gain more audience in embedded and critical systems.

5.1.3 Transformation / Property Kind (T/PK)

We showed on a simple example that the current classifications for model transformation are not sufficient to derive the properties one needs to prove to ensure transformation correctness. What really matters is the intent of a transformation: by capturing the purpose of a transformation instead of the form of its expression, one can define precisely what is the appropriate notion of correctness attached to this transformation. We extracted several kinds of properties of model transformations from the contributions found in the literature; however, a more systematic study of both dimensions and how they relate to each other will enable a better engineering of model transformation verification.

1We thank the participants of the Volt 2013 Workshop for this suggestion and their feedback on industrial applications of MDE verification.
5.2 Gluing all dimensions: towards an Intents/Properties Mapping

How can a transformation designer be guided through the process of formally verifying model transformations, especially when those transformations are used in sensitive applications like safety-critical and embedded systems? We have already discussed the drawbacks of first selecting a transformation engine natively equipped with predefined verification capabilities, which corresponds to the anticlockwise scenario in Figure 8. In our opinion, another possibility is more desirable: the clockwise scenario of Figure 8 makes a central place to the notion of model transformation intent by gluing together all three dimensions.

Figure 9 depicts our proposal for an intent-based verification engineering. Our three dimensions are represented in the bottom, greyed layer: a formal verification tool makes use of both the transformation (specification) and the various properties. We motivated in Section 2.4 the fact that finding the appropriate properties for a given transformation is difficult based on a syntactic characterisation of transformations. Instead of working at a syntactic level, the clockwise scenario suggests to work at a more “semantic” level: an intent captures the purpose of a set of transformations independently of the language they are expressed in, and independently of their syntactic features. Some intents will by nature fix specific syntactic aspects: e.g., defining a DSL’s semantics operationally or refactoring a transformation specification always happens endogenously. Providing a catalogue of such intents for identifying the appropriate intent of a transformation should be based on the identification of recurrent problems of transformation specification, and the documentation of their solution based on the best practices of the engineering experience, just as it happens for design patterns (for object-oriented programming [42], enterprise application, security applications, etc.)

In this approach, transformations are related to intent properties that capture the notion of correctness appropriate for each intent. Intent properties always include what
we have called transformation-related properties (Section 3.2) for describing the intent computation type. Furthermore, we can characterise intent properties by the level they are operating at: either syntactic, referring to the transformation specification; or semantic, referring to the transformation execution. We found properties kinds that share the same mathematical form, although acting at different levels. A (mathematical) relation maps either syntactic elements (i.e. objects, attributes and so on) through structural correspondences [1, 80, 82], or semantic elements (i.e. states from the semantic domains) through (bi-)simulations [81, 32, 16]. A preservation expresses patterns at a syntactic level [71, 87, 46], or formulae at a semantic level [117] that need to be proved equivalent (in a given sense) through the transformation. Behavioural properties use safety properties that should hold on the transformation execution [44, 100, 102, 8, 97, 11, 88, 23, 69]. Figure 10 places some of the contributions reviewed in Section 3.2 according to their mathematical form and application level. New forms should be expected when studying new intents.

We are currently working on an Intent/Property Framework whose purpose is relate each possible intent with the set of properties that characterise it, providing an efficient, practical way to guide the engineers’ work towards the verification of their transformation. As an illustration, let us apply these ideas to our small example of Section 2.4. For a Dsl semantics defined operationally, the transformation correctness should be assessed through the proof of behavioural, temporal properties, and depending on the nature of the Dsl, termination and/or determinism may not hold: e.g., a security system embedded in a car should not terminate by itself (until the engine is stopped). When defined translationally, the correctness is assessed by establishing correctness relations between each side of the transformation: structural correspondences between the source and target model or, if an explicit representation of each semantic domain is available, a bisimulation.

These ideas have already been investigated in [3, 72]: we have built an intent catalogue that documents more than 20 transformation purposes; and we have formally described properties at various abstraction levels. We established these intent/property mapping for five of the most common intents found in the literature, corresponding to the higher level of Figure 9, and describe the application of such mappings for concrete transformations. Naturally, practical experimentation as well as further description of other intents will allow us to validate and/or improve this Framework.

6 Related Work

The interest of the MDE community in the formal verification of model transformations grew up significantly in the recent years leading to a large number of different approaches. This Section discusses previous reviews of model transformation validation [26, 96, 43] that complete the tridimensional classification presented in this paper.

Calegari and Szasz [26] borrowed our tridimensional classification (originally presented in [4]) to extract a state-of-the-art from a systematic literature review. Beyond being systematic, and written more than a year later, thus covering more contributions, their work enunciates two additional property kinds: preservation of execution semantics at the language-related level, and functional behaviour at the transformation related level. The former is defined as the fact that “the transformation execution must behave as expected according to the definition of the transformation language semantics”; with the vocabulary of Figure 2, it means that transformation executions should comply, or respect, the transformation specification’s semantics. For us, this
is not a property kind *per se*: preserving the execution’s semantics cannot be the
transformation designer’s responsibility, who simply has the role of a user of the
transformation engine, but rather the responsibility of the transformation language
itself to ensure that sentences of the language execute correctly with respect to their
expected semantics. Besides, at a lower level of abstraction, this is the minimal
expectation for a transformation language. The latter property kind is not general
but only applies for graph-based, model-to-model transformations, as described in the
framework studied in the work [23] from which it has been borrowed. In contrast,
our classification is agnostic of the transformation language’s paradigm and its imple-
mentation. Furthermore, the authors also consider *testing* as a verification technique:
this directly contradicts our full covering and static criteria (cf. Section 4.1.2) for an
analysis technique for being considered as formal.

Rahim and Whittle [96] proposed a large survey using as a primary entry point the
validation techniques employed, extending their study to informal approaches as well:
testing, generated code inspection and metrics. Surprisingly, they also consider “graph
transformation” as a specialised possible entry, whereas we showed that most of the
techniques can be employed independently of the transformation language’s paradigm.
For us, it is not a technique, but rather simply a transformation language’s paradigm
(cf. Dimension 1 in Section 2) for which elements from other dimensions (property
kinds and Fv techniques) can indistinctly apply: as a matter of fact, we showed that
model-checking and theorem-proving, among others, are already available for GtBrts.
Interestingly, they cover certification techniques, i.e. techniques generating checkable
evidence — either by a human or by another machine — that the verification process
went well; this is only partially covered by our work. We did not consider certification
as a primary technique since it always subsumes some sort of analysis performed ahead
of the certification process. Similarly to our Fv types in Section 4.2, they distinguish
between indirect/direct techniques (corresponding to our input (in-)dependent type).
At a finer-grained level, they included two additional points that are, from our point
of view, less objective: the input/output complexity, i.e. the amount of data to inject
to make things work; and the tooling, i.e. whether tool support exists for a reviewed
approach. We do not address those points at all due to their subjectivity, but also because they are subject to evolution in the future.

Gabmeyer, Brosch and Seidl [43] proposed a feature-based classification of only five contributions specifically targeting the model-checking Fv technique. Not surprisingly, many of their features cross components of our dimensions: for example, their verification goal is largely covered by the property kind dimension; their domain representation corresponds to our transformation dimension. Contrasting to our classification that remains general, they adopt a finer-grained classification for model-checking specificities: the state space representation and the precise model-checked property. Although preliminary, their feature-based classification can be complementary to ours if it is extended to all Fv techniques: with finer-grained features, comparing literature contributions becomes easier, at the likely expense of complicated the classification grid.

7 Conclusion

This paper proposed to analyse the question of formal verification of properties for MDE applications according to a tridimensional classification, based on the core ingredients involved in verification: transformations, properties and formal verification techniques. Several contributions can be identified in our work: (i) it broadened the conceptual definition of model transformation and discussed the necessity of a classification based on transformations’ intents; (ii) it proposed a comprehensive taxonomy of property kinds, supported by many contributions from the literature; (iii) it proposed a review of the available verification techniques supported by the contributions in the literature to ensure such properties. This study also discussed in detail how the relations between each dimension can bring more insight about the current practice in this field. Note that this study is complementary to another popular validation technique, testing, which was largely explored and surveyed in the recent years (cf. [10, 105, 57]).

Future work should identify how it is possible to take advantage of each verification technique and overcome their respective drawbacks [123]. As a concrete continuation of this work, we would like to propose the community to contribute for a comprehensive benchmark for Fv of transformations: it consists of storing pairs constituted by a transformation together with its properties of interest. This benchmark can help researchers as well as practitioners, and could provide a common reference for playing with verification of transformations by easily targeting a technique, a property kind among those identified in this paper, and comparing efficiency and scalability of approaches.

References


[34] Patrick Cousot and Radhia Cousot. A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation. In Javier Esparza, Orna


About the authors

Moussa Amrani Moussa.Amrani@uni.lu, or visit URL.

Benoît Combemale Benoît.Combemale@irisa.fr, or visit URL.

Levi Lúcio Levi@cs.mcgill.ca, or visit URL.

Gehan Selim Gehan@cs.queensu.ca, or visit URL.

Jürgen Dingel Dingel@cs.queensu.ca, or visit URL.
Yves Le Traon Yves.LeTraon@uni.lu, or visit URL.

Hans Vangheluwe Hans.Vangheluwe@ua.ac.be, or visit URL.

James R. Cordy Cordy@cs.queensu.ca, or visit URL.

Acknowledgments

The authors warmly thank the anonymous reviewers who suggested many improvements that helped clarifying the paper, and several contributions that were missing.

This work is partially supported by the Luxemburgish Fonds National de la Recherche (Fnr), the Natural Sciences and Engineering Research Council of Canada (Nserc), the IBM Canada Center for Advanced Studies (CAS), and by the French Agence Nationale pour la Recherche (ANR) INS Project GEMOC (ANR-12-INSE-0011).