A Tridimensional Approach for Studying the Formal Verification of Model Transformations

Moussa AMRANI[†], Levi LÚCIO[‡], Gehan SELIM[§],
Benoît COMBEMALE[¶], Jürgen DINGEL[§], Hans VANGHELUWE[‡‖], Yves LE TRAON[†], and James R. CORDY[§]

[†] University of Luxembourg (Luxembourg), Moussa.Amrani, Yves.LeTraon}@uni.lu
[‡] McGill University (Canada), {Levi, hv}@cs.mcgill.ca
[§] Queen's University (Canada), {Gehan, Dingel, Cordy}@cs.queensu.ca
[¶] University of Rennes / IRISA (France), Benoit.Combemale@irisa.fr
[‖] University of Antwerp (Belgium), Hans.Vangheluwe@ua.ac.be

***Abstract*—In Model Driven Engineering (MDE), models are first-class citizens, and model transformation is MDE's "heart and soul" [1]. 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 tri-dimensional approach: the transformation involved, the properties of interest addressed, and the formal verification techniques used to establish the properties. This work allows a better understanding of the expected properties for a particular transformation, and facilitates the identification of the suitable tools and techniques for enabling their verification.**

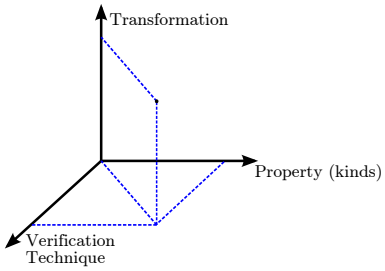

Fig. 1. Taxonomy overview: the tri-dimensional approach.

I. INTRODUCTION

Model-Driven Engineering (MDE) promotes models as first class citizens of the software development process. Models are manipulated through model transformations (MTs), which is considered to be the "*heart and soul*" of MDE [1]. Naturally, dedicated languages based on different paradigms emerged during the last decade to express MTs. 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 [2]: 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 ensures 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 intention behind the transformation: e.g., one may be interested in always producing conform models by construction, or ensuring that target models still conserve their semantics.

This paper proposes a preliminary classification of MT verifiable properties. Applied 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, thus enabling reasoning about the evolution and trends of this research domain. In order to provide a comprehensive interpretation, we propose a tri-dimensional approach (cf. Fig. 1) that allows locating each contribution regarding the following characteristics: the *transformation* involved (cf. Sec. II); the *property kinds* addressed III; and the *verification technique* used IV. These dimensions are closely related but clearly interdependent. This work allows a better understanding of the expected properties for a particular transformation, and facilitates the identification of the suitable tools and techniques for enabling their verification.

II. 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 actual 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, everything is modelled, both source and target metamodels, as well as the transformation specification, are themselves models, conform 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 acts on several source and/or target models.

This section starts by discussing existing definitions for the concept of transformation; and then reviews essential characteristics of transformation languages; and ends by recalling basic elements for classifying transformations.

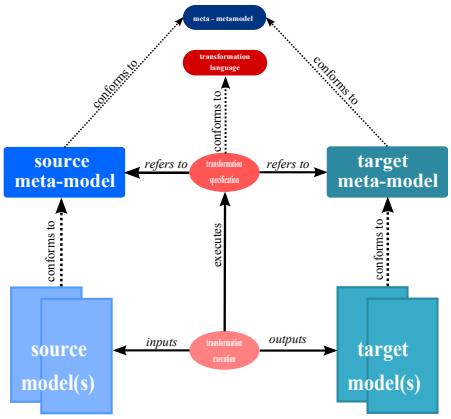

Fig. 2. Model Transformation: the big picture (adapted from [3])

A. 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 definition was proposed by the OMG, in straight line with the Model-Driven Architecture view. The OMG perceives transformations as "*the process of converting one model to another model of the same system*" [4]. Immediately after, the system-centric view was enlarge by Kleppe *et al.*: "*a model transformation is the automatic generation of a target model from a source model, according to a transformation definition*" [5]. 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*" [6], 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*" [7], whereas Syriani re-introduced the crucial importance of the specific intention behind transformations ("*the automatic manipulation of a model with a specific intention*" [3]).

Therefore, as a broader definition, we consider transformations as *the* ***automatic*** *manipulation, conform to a* ***description****, of* ***(a) source model(s)*** *to produce* ***(a) target 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.

B. Languages

TLs allow designers specify their transformations. Given the plethora of TLs, tools and frameworks, it is impossible to review them exhaustively in such limited space. We only highlight some hints interesting for our purpose.

Graph-Based Transformation Languages (GBT(L)s) rely technically on graph rewriting and formally on Category Theory [8]. Despite their long existence, some native MDE constructs like inheritance and containment have been integrated only recently [9]. GBTLs provide a natural visual syntax due to the use of graphs, and can be further customised [10][11], or can be used with textual syntax (e.g., [12]). MDE-dedicated verification techniques were only recently addressed, but GBT share with Petri Nets a common formal background, making verification techniques and tools easily adaptable [13].

Meta-Programming Languages (MPLs) [14] provide an Action Language (AL) for directly manipulating metamodels' concepts. They are by nature operational, meaning that each computation step is precisely described by using the AL's constructions. Formally, they make use of existing semantic foundations available for imperative or object-oriented programming languages [15] and thus directly benefit from advances in formal verification. Among others, we refer the readers to Kermeta [16], which weaves an object-oriented AL within the MOF, or the Action Semantics for MOF [17] and UML [18], or the xOCL [19].

C. Classification

Model transformations differ in *expression*, i.e. in the chosen *transformation language* they are specified, but also in *nature*, i.e. according to the *intention* behind the model manipulation. Therefore, the properties of interest for transformations are naturally related to both dimensions.

Two works [7][20] addressed the model transformation classification issue. Czarnecki and Helsen [20] proposed a hierarchical classification of the model transformations specification (mainly centered on graph transformations), whereas Mens and Van Gorp [7] focused more on providing a multi-dimensional taxonomy, thus covering as well formalisms and tools underlying model transformations. The latter helps capturing the syntactic aspects behind transformations' intentions (readers should refer to the original work for complementary information and examples). The authors identified several dimensions: the *heterogeneity* between the source and target metamodel, qualifying a transformation as *endogeneous* if it refers to the same metamodels, and *exogeneous* otherwise; the *abstraction level* of the source and target models involved, qualifying as *vertical* a transformation that adds or reduces the detail level, and *horizontal* if it remains the same; the *transformation arity* refers directly to the respective numbers of input / output models; and the *source model conservation*, qualifying as *in-place* a transformation that directly alters the source model, and *out-place* otherwise.

However, this classification does not help for our purpose: a classification based on the transformations' intentions is more helpful for identifying which properties should be verified. As an example, consider a Domain-Specific Model (DSM) for which one needs to define its semantics through a transformation. If the designer chooses a *translational* approach, in which the semantics is expressed by translating DSM concepts into a target semantic domain, the transformation is out-place,

exogeneous and vertical (with arity 1:1). On the other hand, if a *behavioral* approach is chosen, the semantics is expressed directly on the source model, by showing its evolution over time, which makes the transformation in-place, endogeneous and horizontal. Although the transformations cover the same intention, i.e. providing a semantics, their expression differs radically. However, the associated properties of interest will be conceptually the same, e.g. proving the correctness, but will be expressed differently w.r.t. the transformation's expression: in the first case, the 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 *syntatic* 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 this paper: Sec. III documents the properties one needs to verify, given a transformation's intention; and Sec. IV reviews the verification techniques, which rely on the TL one uses to express the transformation.

III. 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. Following the dual nature of transformations, we identified two classes of properties: the first class in Sec. III-A relates to the computational nature of transformations and targets properties of TLs; whereas the second class in Sec. III-B deals with the modelling nature where models plays a prominent role. Table I summarises the reviewed papers according to the property kinds identified hereby.

LANGUAGE-RELATED		TRANSFORMATION-RELATED	
Termination	[21], [22] [23], [24] [25], [26] [27]	**Source/Target**	[28], [29] [30], [31]
Determinism	[24], [27] [32], [33] [34]	**Syntactic Rel.**	[35], [36] [37], [38]
Typing	[39], [40]	**Semantic Rel.**	[41], [42] [43], [44] [45]

TABLE I
CLASSIFICATION OF CONTRIBUTIONS ACCORDING TO PROPERTY KINDS

A. Transformation Models: Language-Related Property

From a *computational* perspective, a transformation specification conforms to a *transformation language* (cf. Fig. 2), which can possess properties on its own. In the MDE context, two properties are interesting at execution time: *termination*, which guarantees the existence of target model(s); and *determinism*, which ensures uniqueness. These properties qualify the execution of transformations written in such languages. Another property, namely *typing*, relates to design time: it ensures the well-formedness of transformation specification, and 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.

1) Termination: Termination directly refers to Turing's halting problem, which is known to be undecidable for sufficiently expressive, i.e. Turing-complete, TLs: GBTs have already been proven to not terminate in the general case [46]; whereas MPLs often use loops and (potentially recursive) operation calls, making them able of simulating Turing machines.

a) Termination Criteria: A large literature for GBT already exists, which makes exhauste coverage beyond this paper's scope. In [21], Ehrig *et al.* introduce layers with injective matches in the rewriting rules that separate deletion from non-deletion steps. In [22], Varrò *et al.* reduce a transformation to a Petri Net, where exhaustion of tokens within the net's places ensures termination, because the system cannot pursue any more. In [23], Bruggink addressed a more general approach by detecting in the rewriting rules infinite creating chains that are at the source of infinite rewritings. In [24], Küster proposes termination criteria with the same base idea, but on graph transformations with control conditions.

Termination Criteria for MPLs directly benefit from the large and active literature on imperative and object-oriented programming languages. They 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 [25]; and Berdine *et al.* use separation logics for detecting effective progress within loops in [26]. Since these techniques are always over-approximations of the TL's semantics, they are sound but not complete, and can potentially raise false positives.

b) Expressiveness Reduction: Reducing expressivity regarding termination generally means avoiding constructs that

may be the source of (unbounded) recursion. For example, DSLTrans [27] 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.

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.

a) Determinism Criteria: Determinism directly refers to the notion of *confluence* (often called the *Church-Rosser*, or *diamond*, property) for GBTLs, which has also been proved as undecidable [47]. Confluence and termination are linked by Newman's lemma [48], stating that confluence coincides with local confluence if the system terminates. This offers a practical method to proove it by using the so-called *critical pairs*. The GBT community is very active and already published several results. In [49], Heckel *et al.* formally proved the (local) confluence for Typed Attributed Graph Grammars, and Küster in [24] for graph transformations with control conditions. In [32], 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 [33]. In another area, Grønmo *et al.* addresses the conformance issue for aspects in [34], i.e. ensuring that whatever order aspects are woven, it always leads to the same result.

b) Expressive Reduction: Reducing expressivity 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 [27], the TL controls non-determinism occuring 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) Typing: 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 [2]. Kühne *et al.* studied in [39] 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 checkings 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 [40] 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.

B. Model Transformations: Transformation-Related Property

From a *modelling* perspective, a transformation refers to source/target models (cf. Fig. 2) for which dedicated properties need to be ensured for the transformation to behave correctly. Of course, the properties of interest range over a large scale of nature, precision, and complexity.

This section provides a comprehensive overview of properties of interest based on their kinds: properties involving transformations's source and/or target models were historically the first to be considered; then, syntax-guided properties, which relate models' syntaxes (i.e. their metamodels) provide a first level of analysis; and finally, properties involving the underlying semantics of models are discussed.

1) On the Source/Target Model(s): A first level of properties verification concerns the source and/or targetmodel(s) a transformation refers to. The *conformance* property is historically one of the first addressed formally, because it is generally required by transformations to work properly (cf. Sec. II-A). Transformations admitting several models as source and target require other kinds of properties, either required for transformations or simply desirable.

a) 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, *structural* conformance, involving only the model, is distinguished from *constrained* conformance, which is an extended property that includes structural constraints, otherwise referred to as metamodels' *static semantics* or *well-formedness rules* (see e.g. [31]). Nowadays, this property is well understood and automatically checked within modeling frameworks. However, proving that a transformation always outputs conform target model(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. Model Typing [50] extends the notion of type beyond classes, by defining a *subtyping* relation on models. This enables better reuse for modelers: transformations defined for particular models also works for any sub-model.

b) 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.

Concerning *merging*, Chechik *et al.* follow an interesting research line in [28]: 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 [29] 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 [30], Molderez *et al.* present DELMDSOC, a language for Multi-Dimensional Separation of Concerns implemented in AGG [10]. Ultimately, this framework will allow the detection of conflicts between aspects by model-checking, typically when multiple advices must be executed on the same joinpoints.

2) Model Syntax Relations: A model transformation consists in general of a computation that applies repeatedly a set of rewriting rules to a models, 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 *model syntax 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 own semantics, these structural relations regard the actual meaning (formally called *semantics*) of those models in an implicit fashion. In [35] 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 [36] 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) [37] could also be used to encode structural relation properties between two metamodels. Lúcio and Barroca formally define in [38] 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.

3) Model Semantics Relations: Beyond structural relationships between source and target models, it may be interesting to relate their meaning, which implies to dispose of at least a partial explicit representation of the models' semantics, or a way of computing it.

An example of such a semantics relation property could the fact that a statechart is transformed into a *bisimilar* statechart. In this case the relation between the semantics of these two behavioral models is *bisimulation* – which is a strong variant of a *simulation* relation – where both systems are able to simulate each other from an observational point of view. In order to prove automatically such a relation is enforced by a transformation it is necessary to build, explicitly or implicitly, the labelled transition systems corresponding the semantics of both the input and the output model.

Narayanan *et al* in [41] show how to verify that a transformation outputs a statechart bisimilar to an input statechart. Combemale *et al.* [14] formally prove the weak simulation of xSPEM 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.

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. In [42] Varro and Pataricza are able to prove that CTL (Computation Tree Logic) properties are preserved when transforming statecharts into Petri Nets. 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, Gargantini *et al.* use Abstract State Machines within the ASMeta to perform CTL verification using SPIN; Rivera *et al.* [51] use the Maude rewriting system and its embedded LTL model-checker to verify semantic properties of Domain-Specific Languages.

An interesting subset of CTL are *safety* properties, which are expressed as invariants over the reachable states of the system.

The idea is that certain conditions can never be violated during execution. In this sense, Becker *et al* are able to prove in [43] 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 crash into each other during operation. Padberg *et al* introduce in [44] 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.

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* present in [45] a set of refactoring transformations that preserve the semantics of UML Class diagrams.

IV. Formal Verification (Fv) Techniques

In this section, we discuss Fv techniques proposed in the literature to prove MT properties implemented in various TLs. Table II 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 II, 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 III 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. In the following subsections, we discuss examples of each type of verification technique from the literature.

	Transformation Independent	**Transformation Dependent**
Input Independent	Type I: [27], [44], [45], [52], [21], [22], [24]	Type II: [53], [38], [43], [54], [55], [56], [42]
Input Dependent		Type III: [41], [57], [36], [58]

TABLE II
CLASSIFICATION OF VERIFICATION TECHNIQUES.

A. Type I: Transformation-Independent and Input-Independent

Some studies verified that any transformation preserved certain properties for any input model by proposing a TL that preserves the properties by default. Other studies proposed approaches to follow to develop any transformation that preserves certain properties for any input model.

By Construction of the TL*:* Barroca *et al.* [27] proposed DSLTrans and formalized its syntax and semantics using typed graphs. DSLTrans guarantees termination and confluence for any MT by construction.

By Construction of the Model Transformation: Several studies used graph rewriting to formalize MT and verify that they preserve certain properties by construction. Ehrig *et al.* [21] and Küster [24] proposed criteria for building GBT rules to ensure that a GBT terminates and is confluent by construction.

Other studies used Petri Nets to formalize model transformations and to reason about their properties. Padberg *et al.* [44] proposed morphisms for Algebraic Petri Nets that preserved by construction safety properties expressed through invariants, by transferring them from the source to the target model. Rule-based modification was integrated with the proposed morphisms resulting in safety-preserving rule-based refinement. In [22], GBT are formalized as Petri Nets; and termination is verified if the corresponding Petri Net runs out of tokens in finite steps.

Massoni *et al.* [45] proposed an approach to develop model refactorings for UML class diagrams that preserve semantics by construction. A set of basic, semantic-preserving transformation laws were used to compose more complicated model refactorings which would be, in effect, semantics-preserving. The laws were verified by translating them and the class diagrams using Alloy to reason about their soundness.

B. Type II: Transformation-Dependent and Input-Independent

For this type of verification, usual tool-assisted techniques are generally used: model-checking, static analysis and theorem-proving.

1) Model-Checking: Rensink *et al.* compared in [53] 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.* [38] 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. Várro and Pataricza [42] used model checking to prove that dynamic consistency properties were preserved in a model transformation from statecharts to Petri Nets.

2) Static Analysis: Becker *et al.* [43] 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.

3) Theorem Proving: Asztalos *et al.* [54] 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 Visual Modeling and transformation System and was used to verify a refactoring transformation on business process models. Paige *et al.* [55] 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. In Eiffel, the verification requires an actual execution: Eiffel specifications are generated from class diagrams whereas Eiffel unit tests are generated from dynamic diagrams, making an executable Eiffel code that allows performing the MVCC verification. Giese *et al.* [56] proposed formalizing Model to Code transformations using Triple Graph Grammars in the Fujaba tool suite, verified with the help of the Isabelle/HOL theorem-prover.

C. Type III: Transformation-Dependent and Input-Dependent

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 [36], they generate crosslinks between source and target models to check structural correspondence between source and target models. In [41], 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.

2) Using Petri Net Analysis: Lara and Vangheluwe [57] 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.

3) Using Constraint Solvers: Anastasakis *et al.* [58] 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.

V. DISCUSSION AND CONCLUSION

This paper proposed to analyse the question of formal verification of properties for MDE applications according to a tri-dimensional approach, 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 transformation concept definition and discussed the necessity of a classification based on transformations' intention; (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. By performing this study, we make a first attempt at identifying how our three dimensions relate when a transformation is to be verified. In the paragraphs that follow we revisit our work by relating the three dimensions we've studied in this paper. In order to be precise in what has been achieved we will, at a time, discuss the relation between each two of the three dimensions:

It is to be expected that a *transformations' intention* indicates which *properties* one may want to verify about it. However, given the fact that a proper classification of the *intentions* of model transformations is currently unavailable, the work of relating these two dimensions is unfinished. We were however able to identify in section III a number of properties of transformations associated to specific transformation intentions, as found in the literature.

In what concerns the relation between *transformations* and FV techniques, it seems likely that the type of transformation language used (graph based or meta-programming) will have an influence on the pragmatically usable proof techniques. The form of the transformation (*endogenous* vs. *exogenous*, *inplace* vs. *outplace*, etc) may also be relevant for such a choice. This topic remains largely to be explored in future work.

The best explored relation in this paper is the one between *properties* and FV Techniques. We were able to identify that formal (mathematical) proofs are typically more related to properties that hold for all the transformations expressible in a given TL, such as termination or confluence. *Model checking*, *static analysis* and *theorem proving* are used when

a transformation is to be verified for all its possible inputs. When both the transformation and its input is fixed, lighter techniques such as *traceability links*, translation into *petri nets* or *constraint solvers* can be used.

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 referential 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.

Acknowledgements. This work is partially supported by the Luxemburgish Fonds National de la Recherche (FNR), the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the IBM Canada Center for Advanced Studies (CAS).

REFERENCES

[1] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart And Soul Of Model-Driven Software Development," IEEE *Software*, vol. 20, no. 5, pp. 42–45, 2003.

[2] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, "Model Transformations? Transformation Models!" in MODELS, 2006.

[3] E. Syriani, "A Multi-Paradigm Foundation for Model Transformation Language Engineering," Ph.D. dissertation, McGill University, 2011.

[4] O. M. Group, "MDA Guide (version 1.0.1)," Tech. Report, June 2003.

[5] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained — The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[6] L. Tratt, "Model Transformation And Tool Integration," SOSYM, vol. **4**(2), pp. 112–122, 2005.

[7] T. Mens and P. Van Gorp, "A Taxonomy Of Model Transformation," *Electronic Notes in Theoretical Computer Science (*ENTCS*)*, vol. 152, pp. 125–142, 2006.

[8] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing, 1997, vol. I.

[9] S. Jurack and G. Taentzer, "A Component Concept for Typed Graphs With Inheritance and Containment Structures," in ICGT, 2010.

[10] G. Taentzer, "AGG: A Tool Environment for Algebraic Graph Transformation," in AGTIVE, vol. 1779, 2000, pp. 333–341.

[11] J. de Lara and H. Vangheluwe, "Using ATOM^3 as a Meta-CASE Tool," in ICEIS, 2002, pp. 642–649.

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *J. Sc. Comp. Prog.*, vol. **72**(1–2), pp. 31–39, 2008.

[13] M. Maximova, H. Ehrig, and C. Ermel, "Formal Relationship between Petri Net and Graph Transformation Systems Based on Functors between M-Adhesive Categories," in PN-GT, vol. 40, 2010, pp. 23–40.

[14] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux, "Essay On Semantics Definition in MDE – An Instrumented Approach for Model Verification," *Journal of Software*, vol. 4, no. 9, pp. 943–958, 2009.

[15] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction (Foundations of Computing)*. MIT Press, 1993.

[16] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-Oriented Meta-Languages," in MODELS, 2005, pp. 264–278.

[17] R. Paige, D. Kolovos, and F. , Polack, "Adding an Action Semantics to MOF 2.0," in SAC, 2006.

[18] M. Yang, G. J. Michaelson, and R. J. Pooley, "Formal Action Semantics for a UML Action Language," J.UCS, vol. **14**(21), pp. 3608–3624, 2008.

[19] T. Clark, P. Sammut, and J. Willans, *Superlanguages: Developping Languages and Applications with* XMF. Ceteva, 2008.

[20] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," IBM *Systems J.*, vol. **45**(3), pp. 621–645, 2006.

[21] H.-K. Ehrig, G. Taentzer, J. de Lara, D. Varró, and S. Varró-Gyapai, "Termination Criteria for Model Transformation," in FASE, 2005.

[22] D. Varró, S. Varró-Gyapai, H. Ehrig, U. Prange, and G. Taentzer, "Termination Analysis of Model Transformations by Petri Nets," in ICGT, vol. 4178, 2006, pp. 260–274.

[23] H. S. Bruggink, "Towards a Systematic Method for Proving Termination of Graph Transformation Systems," ENTCS, vol. **213**(1), 2008.

[24] J. M. Küster, "Definition and Validation of Model Transformations," SOSYM, vol. **5**(3), pp. 233–259, 2006.

[25] F. Spoto, P. M. Hill, and E. Payet, "Path-Length Analysis of Object-Oriented Programs," in EAAI, 2006.

[26] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn, "Automatic Termination Proofs for Programs with Shape-Shifting Heaps," in *Computer-Aided Verification (*CAV*)*, ser. LNCS, vol. 4144, 2006, pp. 386–400.

[27] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa, "DSLTrans: A Turing-Incomplete Transformation Language," in SLE, 2010.

[28] M. Chechik, S. Nejati, and M. Sabetzadeh, "A Relationship-Based Approach to Model Integration," ISSE, vol. 7, 2011.

[29] S. Katz, "Aspect Categories and Classes of Temporal Properties," TAOSD, vol. 3880, pp. 106–134, 2006.

[30] T. Molderez, H. Schippers, D. Janssens, H. Michael, and R. Hirschfeld, "A Platform for Experimenting with Language Constructs for Modularizing Crosscutting Concerns," in WASDETT, 2010.

[31] A. Boronat, "MOMENT: A Formal Framework for MOdel manageMENT," Ph.D. dissertation, University of Valencia, 2007.

[32] L. Lambers, H. Ehrig, and F. Orejas, "Efficient Detection of Conflicts in Graph-based Model Transformation," ENTCS, vol. 152, 2006.

[33] E. Biermann, "Local Confluence Analysis of Consistent EMF Transformations," ECEASST, vol. 38, pp. 68–84, 2011.

[34] R. Grønmo, R. Runde, and B. Møller-Pedersen, "Confluence of Aspects For Sequence Diagrams," SOSYM, pp. 1–36, Sep. 2011.

[35] D. Akehurst, S. Kent, and O. Patrascoiu, "A Relational Approach to Defining and Implementing Transformations in Metamodels," SOSYM, vol. **2**(4), pp. 215–239, 2003.

[36] A. Narayanan and G. Karsai, "Verifying Model Transformation By Structural Correspondence," ECEASST, vol. 10, pp. 15–29, 2008.

[37] A. Schürr and F. Klar, "15 Years of Triple Graph Grammars," in ICGT, 2008, pp. 411–425.

[38] L. Lúcio, B. Barroca, and V. Amaral, "A Technique for Automatic Validation of Model Transformations," in MODELS, 2010.

[39] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Systematic Transformation Development," ECEASST, vol. 21, 2009.

[40] T. Levendovszky, L. Lengyel, and T. Mészáros, "Supporting Domain-Specific Model Patterns With Metamodeling," SOSYM, vol. **8**(4), 2009.

[41] A. Narayanan and G. Karsai, "Towards Verifying Model Transformations," ENTCS, vol. 211, pp. 191–200, April 2008.

[42] D. Varró and A. Pataricza, "Automated Formal Verification of Model Transformations," in CSDUML, 2003, pp. 63–78.

[43] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling, "Symbolic Invariant Verification For Systems With Dynamic Structural Adaptation," in ICSE, 2006.

[44] J. Padberg, M. Gajewsky, and C. Ermel, "Refinement *versus* Verification: Compatibility of Net Invariants and Stepwise Development of High-Level Petri Nets," Technische Universität Berlin, Tech. Rep., 1997.

[45] T. Massoni, R. Gheyi, and P. Borba, "Formal Refactoring for UML Class Diagrams," in BSSE, 2005, pp. 152–167.

[46] D. Plump, "Termination of Graph Rewriting is Undecidable," *Fundamenta Informaticæ*, vol. 33, no. 2, pp. 201–209, 1998.

[47] ——, "Confluence of Graph Transformation Revisited," in *Processes, Terms and Cycles: Steps on the Road to Infinity*, vol. 3838, 2005.

[48] M. H. A. Newman, "On Theories With a Combinatorial Definition of "Equivalence"," *Annals of Mathematics*, vol. **43**(2), pp. 223–243, 1942.

[49] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems," in ICGT, 2002.

[50] J. Steel and J. Jézéquel, "On Model Typing," SOSYM, vol. **6**(4), 2007.

[51] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal Specification and Analysis of Domain-Specific Models Using Maude," *Simulation*, vol. 85, no. 11–12, pp. 778–792, 2009.

[52] K. Stenzel, N. Moebius, and W. Reif, "Formal Verification of QVT Transformations for Code Generation," in MODELS, 2011.

[53] A. Rensink, À. Schmidt, and D. Varró, "Model Checking Graph Transformations: A Comparison of Two Approaches," in ICGT, 2004.

[54] M. Asztalos, L. Lengyel, and T. Levendovszky, "Towards Automated, Formal Verification of Model Transformations," in ICST, 2010.

[55] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-Based Model Conformance and Multi-View Consistency Checking," ACM TOSEM, vol. 16, no. 3, pp. 1–48, 2007.

[56] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner, "Towards Verified Model Transformations," in MODEVVA, 2006, pp. 78–93.

[57] J. de Lara and H. Vangheluwe, "Automating the Transformation-Based Analysis of Visual Languages," FAC, vol. **22**(3-4), pp. 297–326, 2010.

[58] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of Model Transformations *via* Alloy," in MODEVVA, 2007, pp. 47–56.