Analysis of Model Transformations

Gehan Selim, James R. Cordy, Juergen Dingel

{gehan, cordy, dingel}@cs.queensu.ca

Technical Report 2012-592

School of Computing, Queen's University

Kingston, Ontario, Canada, K7L3N6

August 2012

©2012 Gehan Selim, James R. Cordy, and Juergen Dingel

Contents

1 Introduction 1

2 Background 2

2.1 Graph Rewriting Systems . . . 2

2.2 Triple Graph Grammars (TGGs) . . . 3

2.3 Petri Nets . . . 3

2.4 Alloy . . . 4

2.5 Maude . . . 5

3 A Taxonomy of Model Transformation Analysis Techniques 5

4 Static Analysis Techniques 7

4.1 Formal Methods . . . 7

4.1.1 Type I Formal Methods . . . 8

4.1.2 Type II Formal Methods . . . 13

5 Dynamic Analysis Techniques 17

5.1 Type III Formal Methods . . . 17

5.2 Model Checking . . . 19

5.3 Design Space Exploration (DSE) . . . 22

5.4 Instrumentation . . . 25

5.5 Testing . . . 26

5.5.1 Challenges of Model Transformation Testing . . . 27

5.5.2 Phase I: Test Case Generation . . . 29

5.5.3 Phase II: Test Suite Assessment . . . 35

5.5.4 Phase III: Building the Oracle Function . . . 37

6 Discussion 41

6.1 Overview of the Surveyed Studies with Respect to the Three Dimensions . . . 41

6.2 First Dimension: The Analysis Technique Used . . . 43

6.2.1 A Comparison Between the Three Types of Formal Methods 43
6.3 Second Dimension: Property Analyzed 44
6.4 Third Dimension: The Formalization Used to Specify the Model Transformation of Interest 45
6.5 Relations Between the Three Dimensions 45

7 Conclusion 46

List of Figures

1 Taxonomy overview of model transformation analysis techniques . . 6

List of Tables

1 Classification of approaches to model transformation analysis based on formal methods. 8
2 Projection of the three dimensions for static analysis techniques . . 41
3 Projection of the three dimensions for dynamic analysis techniques . 42

1 Introduction

Model Driven Architecture (MDA) [78] is a standardization effort led by the Object Management Group (OMG) for developing systems using platform-independent models or software abstractions. The application of MDA to software systems is referred to as Model Driven Development (MDD). MDD is a relatively new software development methodology that uses *models* as the basic building blocks in software development. In MDD, software development starts off with abstract models that are successively *transformed* into detailed models and finally into code. Thus, *model transformations* are extensively used in MDD.

A model transformation is a program that maps one or more input models (conforming to a source metamodel) to one or more output models (conforming to a target metamodel). Mens and Van Gorp [57] proposed a multi-dimensional taxonomy of model transformations where several factors were used to classify model transformations. For instance, the heterogeneity of the manipulated metamodels qualifies a transformation as exogenous or endogenous. A transformation is exogenous if it manipulates different source and target metamodels, and is endogenous otherwise. The abstraction levels of the manipulated models qualify a transformation as a horizontal or a vertical transformation. A transformation is horizontal if the input and output models are at the same abstraction level, and is vertical otherwise. Conservation of the input model qualifies a transformation as an in-place or an out-place transformation. A transformation is in-place if it directly alters the input model, and is out-place otherwise. The transformation's arity or the number of models manipulated by a transformation is another factor that can be used to classify model transformations. The study further discussed model transformation characteristics, formalizations and tools proposed in the literature.

To verify and reason about the correctness of transformations, it is important to develop effective techniques for model transformation analysis [4]. This paper has three main objectives. First, we propose a taxonomy of model transformation analysis techniques (Section 3). Second, we survey the current state of the art in model transformation analysis techniques and classify them according to the proposed taxonomy (Section 4 and Section 5). Third, following [3], we identify three dimensions to the model transformation analysis problem:

1. First Dimension: The analysis technique used
2. Second Dimension: The property being analyzed
3. Third Dimension: The formalization used to specify the model transformation of interest

Accordingly, for each surveyed study, we highlight the three dimensions of the proposed approach (Section 7).

This paper is organised as follows: Section 2 briefly summarizes the formalizations that can be used to specify model transformations; Section 3 demonstrates the proposed taxonomy of model transformation analysis techniques; Section 4 details static analysis techniques; Section 5 details dynamic analysis techniques; Section 6 discusses the proposed taxonomy from several perspectives; and finally Section 7 concludes the paper.

2 Background

Many model transformation analysis techniques use formalizations such as graph rewriting systems and Petri Nets to define and formalize a model transformation and its manipulated metamodels and models. Such formalizations are widely used since they have well-established analysis techniques that can be leveraged to analyze model transformations. In this section, we briefly overview the formalizations used by the studies surveyed in this paper.

2.1 Graph Rewriting Systems

A graph rewriting system [75, 17] is a formal model in which the static states of a system are represented as graphs, and the behaviour and evolution of a system are represented as graph rewriting rules on those graphs [73]. The manipulated graphs can be typed and attributed, where the graph rewriting rules must be type-preserving and must handle rewriting the attributes. The terms *graph rewriting systems* and *graph transformation systems* were used interchangeably in the literature. In this paper we use the term *graph rewriting systems* to unify the used terminology.

A graph rewriting system [75, 17] is composed of one or more graph rewriting rules. A graph rewriting rule has a left-hand side (LHS), a right-hand side (RHS) and optional negative application conditions (NACs). The LHS is a graph pattern to look for in the host graph, the RHS is the graph pattern to create if the LHS was found in the host graph, and the NACs are graph constraints which prohibit the existence of certain patterns in the host graph. The execution of a graph rewriting rule r on a host graph G is carried out as follows: (i) find a match for the LHS of r in G, (ii) check whether the match found satisfies the NACs of r, (iii) remove all the graph elements from G which have an image in the LHS but not in the RHS, and (iv) create new graph elements in G for all elements that have an image in the RHS but not in the LHS. Thus, the LHS and the NACs are the preconditions of the graph rewriting rule while the RHS is the postcondition of the graph rewriting rule. Two graph rewriting approaches were proposed in the literature: the *Double Pushout (DPO)* approach and the *Single Pushout (SPO)* approach [75].

A graph rewriting system and an initial host graph is referred to as a *graph grammar* [75]. A *graph transition system* generated from a graph grammar is the state space of the graph grammar where nodes represent intermediate graphs, and transitions represent possible graph rewriting rule applications [70].

Graph rewriting systems have been traditionally used to design terminating and confluent transformations. Graph rewriting systems are considered to have limited analysis capabilities in comparison to other formalizations especially if they manipulate attributed graphs [74].

2.2 Triple Graph Grammars (TGGs)

One major disadvantage of graph rewriting systems is that they are usually restricted to performing in-place graph rewriting between graph instances conforming to the same graph. Thus, they cannot be easily used to track traceability links between input and output graph instance elements. A *Triple Graph Grammar (TGG)* [77] specification is a declarative definition of a bidirectional graph rewriting system. TGGs overcome the disadvantage of graph rewriting systems by using correspondence graphs or metamodels that maintain m-to-n relationships between the input and output graph instance elements, hence maintaining consistency between them. One TGG rule is composed of three rule components; a left rule, a right rule and a correspondence rule, each responsible for matching and rewriting the corresponding graphs. Each rule component has a LHS and a RHS. Thus, the notion of *triple* captures the input, output and correspondence graph instances that are rewritten in parallel in any TGG rule.

Using one TGG rule, one can perform source-to-target transformation, target-to-source transformation, or correspondence analysis (i.e., analyzing the relationships between the input and output graph instance elements) without having to define three separate graph rewriting rules for each operation.

2.3 Petri Nets

Petri Nets [66] are formal models of information flow and control in systems, especially asynchronous and concurrent systems. Petri Nets are composed of four basic elements: *places, transitions, edges* and *(optional) weights* [81].

Places are represented graphically as hollow circles and are used to represent states of a system. Places may contain one or more *tokens*, represented graphically by black solid dots within the places. The distribution of tokens in different places of a Petri Net is called the *marking* of the Petri Net. The *initial marking* of a Petri Net refers to the initial distribution of tokens in a Petri Net. A transition is represented graphically as a horizontal bar and represents a possible change in

the system state. A transition can be connected to one or more input places by incoming edges. Similarly, a transition can be connected to one or more output places by outgoing edges. A transition is said to be *enabled* (i.e., the transition may fire) if each of its input places contains at least as many tokens as specified by the *weights* associated with the incoming edges. If no weights are specified, then a weight of one is assumed. If a transition is enabled and fires, the marking of a Petri Net changes. This is done by removing tokens from the input places of the transition and producing tokens in the output places of the transition. The number of tokens removed and produced by the firing of a transition is determined by the weights associated with the edges. Timed Transition Petri Nets (TTPNs) are one variant of Petri Nets that were used by one of the surveyed studies [26]. TTPNs are similar to regular Petri Nets with the exception of associating a delay with each transition, such that the transition has to be enabled for a number of time units equal to the delay before firing.

Many studies transformed formal models such as graph rewriting systems to some variant of Petri Nets due to their strong support for analysis (e.g., correctness analysis, dependability analysis, performance analysis [31] and termination analysis [81]). For example, Varró *et al.* [81] abstracted graph rewriting systems into Petri Nets to perform termination analysis. Node and edge types were represented as places; rules were represented as transitions; the LHS and the RHS of a rule were represented as weighted arcs between the place of the corresponding elements type and the transition of the corresponding rule; and the input graph instance determined the initial marking of the Petri Net.

2.4 Alloy

Alloy [42] is a declarative modelling language based on first order relational logic with well-defined semantics. *Signatures*, *relations*, *facts* and *predicates* are used to specify a model in Alloy. Signatures represent the entities of a system and relations represent the relations between such entities. Facts and predicates specify constraints on signatures and relations.

Alloy comes with an analyzer, the Alloy analyzer [41], that uses constraint solvers to automatically analyze Alloy models. The Alloy analyzer supports two kinds of analysis: consistency checking and assertion checking. Consistency checking verifies that the specified Alloy model is consistent by generating a random model instance that conforms to the Alloy model specification. Assertion checking ensures that the Alloy model satisfies some assertions or constraints. If assertion checking fails, the Alloy analyzer generates a counter example or an instance model that violates one or more assertions. Although Alloy is a modelling language, it has been used to specify model transformations (e.g., [54, 4]), thus allowing researchers to take advantage of the Alloy analyzer.

2.5 Maude

Maude [25] is a language and an engine that supports Membership Equational Logic (MEL) [22] specifications and rewriting logic specifications. A Maude specification of a system can have *functional modules* and *system modules*. A functional module specifies membership equational theories that describe possible states of the modeled system. A functional module also uses equations as simplification rules to find unique and simplified forms of terms conforming to the membership equational theories. System modules specify rewrite theories that rewrite terms conforming to membership equational theories.

Maude supports three kinds of analysis: simulation, reachability analysis and model checking linear temporal logic (LTL) properties. Simulation is the execution of a Maude system specification. Reachability analysis builds the state space of a Maude system specification to look for deadlocks (i.e., states on which no further rewrite may take place) and to prove or disprove system invariants by generating counter examples. Model checking LTL properties involves checking whether every possible behavior of a Maude system specification, starting from an initial model, satisfies a LTL property. LTL properties (i.e., properties that can be specified in linear temporal logic) take the form of safety properties (i.e., ensuring that something bad never happens) or liveness properties (ensuring that something good eventually happens)[1] [25]. If an LTL property is violated, Maude generates a counter example too.

Maude has been used to represent model transformations and their manipulated metamodels and models due to several reasons besides its support for analysis. Maude's rewriting logic preserves conformance of output models to metamodel constraints. Maude also manipulates models in a consistent way, e.g., deleting orphan nodes and dangling edges.

3 A Taxonomy of Model Transformation Analysis Techniques

In this section, we propose and briefly discuss a taxonomy of model transformation analysis techniques investigated in the literature. Figure 1 shows the proposed taxonomy.

Model transformation analysis techniques can be categorized as static analysis techniques or dynamic analysis techniques. Static analysis techniques do not require executing the model transformation of interest, unlike dynamic analysis techniques. Other informal analysis techniques include output model walkthroughs

[1] In the rest of this paper, we refer to safety properties and liveness properties as LTL properties.

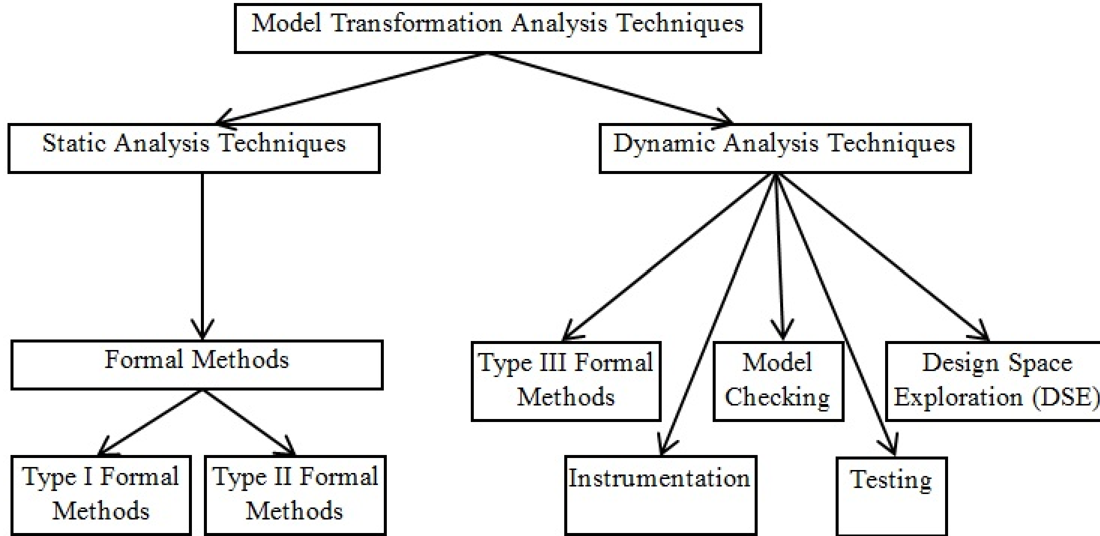

Figure 1: Taxonomy overview of model transformation analysis techniques

and inspection of code and documentation. Such informal analysis techniques are out of the scope of this paper, and hence, will not be discussed further.

Static analysis techniques encompass formal methods; Type I formal methods or Type II formal methods. Dynamic analysis techniques encompass Type III formal methods, model checking, design space exploration, instrumentation and testing. We explain static analysis techniques and dynamic analysis techniques in more detail in Sections 4 and 5. Some studies used more than one technique to analyze model transformations. For example, Narayanan and Karsai [61] used both Type III formal methods (Section 5.1) and model checking (Section 5.2) to prove bisimilarity and eventually analyze reachability. Other studies compared between techniques belonging to the same class in the proposed taxonomy. For example, Paige *et al.* [65] proposed and compared two different approaches of Type III formal methods (Section 5.1).

In the dynamic analysis techniques, model transformation testing has been attracting more interest due its usefulness in uncovering bugs while maintaining a low computational complexity [37]. Some characteristics specific to model transformations make their testing a challenging problem [11, 12, 33, 51]. Thus, several challenges related to model transformation testing have been identified and researched. We investigate these challenges and their proposed solutions in Section 5.5 of this paper.

4 Static Analysis Techniques

Static analysis techniques analyze model transformations without executing them. Formal methods have been explored by many studies in the literature to statically analyze model transformations. Other static analysis techniques developed originally for analyzing code (e.g., examining dependency graphs) have also been adapted for model transformations but are not extensively explored in the literature. For example, Küster [45] discusses basic static analysis of model transformations. The study proposed automatically checking that the variables used in a model transformation occur in the source metamodel, and using non-terminal dependency graphs to ensure that all rules are reachable and that non-terminals created by a rule are deleted by later rule applications. The study discussed how to perform the suggested static analysis in theory, but did not detail how it can be carried out in practice.

4.1 Formal Methods

Formal methods use formalizations such as Petri Nets [66] (Section 2) to define and formalize model transformations and their input and output domains. The formalized model transformations can then be used to analyze certain transformation properties (e.g., termination) or output model properties (e.g., conformance to the target metamodel). It has been argued that formal methods are computationally complex and hence are not scalable to complex transformations and input models [37].

We differentiate between three types of formal methods used to analyze model transformations. We previously discussed the differences between the three types in [3] and we summarize them in Table 1. Type I formal methods analyze certain properties for all transformations when executed on any input model, i.e., they are transformation-independent and input-independent. Type II formal methods analyze certain properties for a specific transformation when executed on any input model, i.e., they are transformation-dependent and input-independent. Type III formal methods analyze certain properties for a specific transformation when executed on one instance model, i.e., they are transformation-dependent and input-dependent. When a formal method is transformation-independent, it implies that no assumptions are made about the input model. This explains the empty category in Table 1 representing formal methods that are transformation-independent and input-dependent.

Type I and Type II formal methods are discussed in this section since they are input-independent, and hence are classified as static analysis techniques. Type III formal methods are classified as dynamic analysis techniques and will be discussed in Section 5.

	Transformation-Independent	**Transformation-Dependent**
Input-Independent	Type I: [54], [31], [26], [19], [81], [67], [79], [7], [20], [50], [10]	Type II: [8], [36], [15], [53]
Input-Dependent	-	Type III: [62], [9], [16], [4], [65]

Table 1: Classification of approaches to model transformation analysis based on formal methods.

4.1.1 Type I Formal Methods

Type I formal methods analyze certain properties for any model transformation (i.e., transformation-independent) when run on any input model (i.e., input-independent). Studies that fall into this category take one of two forms: proposing an approach or criteria that can be used to build model transformations that preserve certain properties by construction of the transformation (e.g., [54, 31, 26, 19, 81, 67, 79, 7, 20, 50]); or proposing a model transformation language that preserves certain properties by construction of the language (e.g., [10]). In the former class of studies, the proposed criteria can also be used to analyze properties of a model transformation after building the transformation. We structure our discussion in this section around the analyzed properties.

Analyzing algebraic properties: Algebraic properties include all properties of model transformations or their output that can be expressed using algebraic specifications. Examples of such properties include type consistency and semantical properties. Stenzel *et al.* [79] used the interactive theorem prover KIV to perform two kinds of analysis: analyzing semantical properties of operational Query/Views/Transformations (QVT) [6] transformations[2] and analyzing type consistency and semantical properties of their generated output[3]. The approach was demonstrated on a model-to-text QVT transformation that transforms UML class diagrams to Java Abstract Syntax Trees (JASTs) and eventually to Java classes. To analyze properties of QVT transformations, the study proposed and implemented an algebraic formalization of a subset of the expressions and operations of operational QVT in KIV. The model transformation properties of interest were also formalized. KIV was then used to prove or disprove the formulated properties using the KIV-compatible format of the QVT transformation. To reason about properties generated output of the transformation, output JAST

[2]An example of a semantical property of a model-to-code transformation is that each UML class is transformed to a Java class.

[3]An example of a semantical property of a transformation's output code is that calling a setter then the corresponding getter returns the setter's argument.

models were exported into a KIV-compatible format using an Eclipse plug-in. KIV was then used to transform the JAST models in the KIV compatible format into an abstract syntax tree with defined properties to analyze properties of the Java code. The proposed approach can be used to analyze any property of operational QVT transformations that can be expressed using the proposed algebraic formalization.

Analyzing semantics-preservation: A model-transformation is *semantics-preserving* if the output model of the transformation preserves the semantics of the input model. If the transformation manipulates structural models (e.g., class diagrams), then the *structural semantics* must be preserved by the transformation. On the other hand, if the transformation manipulates behavioural models (e.g., statecharts), then the *behavioural semantics* must be preserved by the transformation. Massoni *et al.* [54] presented an approach to develop a model refactoring that preserves structural semantics of UML class diagrams with OCL [83] constraints[4]. The study defined an equivalence relation between UML class diagrams with OCL constraints that can be used to identify the equivalence of two class diagrams. The equivalence relation was based on an *alphabet* and a *mapping function*. The alphabet denoted the set of common elements in the input and output domain that directly map to each other. The mapping function was used to define equivalence between elements that are not common to the input and output domains. The new equivalence relation was used to define a set of basic refactoring rules that are semantics-preserving. The study proposed using such semantics-preserving, basic refactoring rules to compose more complicated refactorings, which would be, in effect, semantics-preserving too. To analyze the implemented model refactoring, the study proposed a translational semantics for UML class diagrams and the basic refactoring rules to Alloy. The generated Alloy models were used to argue that the refactoring was semantics-preserving with respect to the defined equivalence relation.

Analyzing Confluence: A confluent model transformation is a transformation that has a deterministic, unique output for every unique input.

Plump [67] analyzed confluence of hypergraph rewriting systems using *critical pair analysis*. Hypergraph rewriting systems are generalizations of graph rewriting systems where an edge can connect more than two nodes [75]. In critical pair analysis, all pairs of rules with a common left-hand side and which delete an element to be used by the other rule are computed. Such rule pairs are referred to as critical pairs since they are in conflict (i.e., both rules can be executed since they have a

[4]OCL [83] or the *Object Constraint Language* is a language originally developed to define constraints on UML models. The language was later adapted to define constraints on model transformations too.

common LHS but execution of one rule inhibits the other). If each computed critical pair is *joinable* (i.e., both rules in the pair reduce to a common hypergraph) then the model transformation is confluent. Critical pair analysis needs only to be performed for rules with a nondeterministic execution order. A hypergraph rewriting system without critical pairs or with a deterministic execution order is confluent. AGG [80] is a development environment for graph rewriting systems that supports consistency checking and critical pair analysis. The AGG tool was used in [31] and [26]to prove confluence of graph rewriting systems using critical pair analysis. Critical pair analysis was used in other contexts too. For example, Bottoni [21] used critical pair analysis as a prerequisite to parsing visual languages.

Assmann [7] used *stratification* to guarantee confluence of graph rewriting systems. Stratification is carried out by ordering rules into a list of rule sets; the *strata*, based on the rule dependency graph. Rules within a stratum are forced to execute in an order that fulfills certain conditions. Such conditions guarantee the generation of a unique output per stratum. Given that every stratum has a unique output and the list of all strata are computed in their stratification order, then the graph rewriting system has a unique output too and hence, is confluent.

Analyzing Termination: A terminating model transformation is a transformation that always stops executing after a finite number of steps. Although it is undecidable in general whether a double pushout (DPO)-based (Section 2.1), graph rewriting system is terminating [68], some studies proposed sufficient termination criteria.Other studies proposed languages that guarantee termination for any model transformation implemented using that language.

Varró *et al.* [81] proposed a sufficient termination criterion for DPO-based, graph rewriting systems. The study proposed abstracting graph rewriting systems into Petri Nets by tracking the number of objects in the graph rewriting system for each node and edge type, regardless of the structure of the instance graph. The abstraction was carried out as follows: node and edge types were represented as places; negative application conditions (NACs) were represented as *permission places*; rules were represented as transitions; the LHS and the RHS of a rule were represented as weighted arcs between the place of the corresponding elements type and the transition of the corresponding rule; and the input graph instance determined the initial marking of the Petri Net. A permission place corresponding to a NAC gets a token from a preceding rule, if the rule creates an element in the *permission pattern* of the NAC corresponding to the permission place. Thus, the Petri Net captured the causality between different rules. After generating the abstracted Petri Net, algebraic techniques were used to solve a matrix inequality to determine whether the Petri Net runs out of tokens in a finite number of steps, and hence whether the corresponding graph rewriting system terminates. Levendovszky *et al.* [50] proposed sufficient termination criteria for DPO-based, graph rewriting

systems with NACs that manipulate typed, attributed graphs. The study assumed injective matches between rules and the host graph; i.e., each element in a rule is matched to one element in the host graph. The criterion states that if the RHS of a rule requires an extension to be mapped to the LHS of another rule,then the rule terminates. For cases where the RHS of a rule can be mapped to the LHS of another rule without extensions, then for infinite rule applications if each graph appears finitely many times, then the rule eventually terminates. The criteria provide a theoretical basis to prove termination and cannot be easily applied in an algorithmic manner. The study used the proposed criteria to prove termination of two model transformation examples. Assmann [7] proposed two concrete, non-generalizable, criteria for termination, since the main focus of the study was to use graph rewriting systems for program optimization and not to analyze termination. The first criterion assumes that a graph rewriting system only adds edges and that between any two nodes, only one edge of a certain label is allowed. Thus, if a graph rewriting system only adds edges while checking that the edges added do not already exist, then the graph rewriting system will eventually terminate since there is a finite number of edges that can be created between any two nodes. The second criterion assumes that a graph rewriting system only deletes nodes and edges. Thus, elements are subtracted from the host graph until the graph rewriting system terminates.

Bottoni *et al.* [19] proposed a property that a measurement function has to satisfy to be a valid, sufficient termination criterion of DPO-based, high level replacement units (HLRUs) without NACs. HLRUs are a generalised form of graph rewriting systems (i.e., they allow manipulating different kinds of graphs) with a control mechanism for rule applications, e.g., allowing sequential application of rules, or applying just a single rule. The measurement function must measure some commodity that decreases with every rule application, i.e., the value of the measurement function for the LHS of the rule is greater than the value of the measurement function for the RHS of the rule. A HLRU terminates if any rule that is to be applied until its application is no longer possible has a valid termination criterion that it satisfies. The proposed properties and proofs were extended to attributed, graph rewriting systems. Accordingly, termination criteria that satisfy the proposed property were defined. The proposed criteria required that the number of nodes and edges on the LHS of a rule is greater than the number of nodes and edges on the RHS of the rule. A model refactoring composed of one rule was then shown to satisfy the proposed termination criteria. Bottoni and Parisi-Presicce [20] later improved the workin [19] and proposed a sufficient termination criterion for the repeated application of a single, non-deleting rule with one NAC. The study discussed how to extend the criterion for a rule with multiple NACs and for a sequence of rules with NACs. Instead of measuring the consumption of some element as in [19], the study measured the distance between the LHS and the NAC by measuring whether the number of matches

increased or decreased after a rule application. This measurement was achieved by constructing a labeled transition system where states correspond to matches of a rule with all possible intermediate graphs between the LHS and the NAC of the rule. Transitions in the labeled transition system represented rule applications that moved the graph from one state to another state representing a graph instance closer to the NAC. On each transition, the number of rule matches were measured before and after applying a rule to determine whether the number of matches decreased with rule applications, and hence whether the rule terminates. The approach was demonstrated on simple model transformation examples. However, no case study was carried out to rigorously evaluate the proposed criterion.

Ehrig *et al.* [31] proposed an approach to build terminating, DPO-based graph rewriting systems with NACs that manipulate typed, attributed graphs. The approach is based on formulating a graph rewriting system as layers of graph rewriting rules with deletion and non-deletion layers. Each rule and manipulated model element is assigned to a layer. Deletion layers contain rules that delete at least one element. Non-deletion layers contain rules that do not delete elements, cannot be applied twice to the same match and cannot use a newly created item for the match. The layers of the transformation must obey some *layering conditions* to terminate. For example, for a deletion layer, the last creation of a node of a specific type should precede the first deletion of the node of the same type. This ensures that the graph rewriting system does not get into an infinite loop of deleting and recreating any element type. The study formally proved that using the proposed layering conditions and the assignment of rules and model elements to layers guarantees termination of deletion and non-deletion layers of a graph rewriting system. Hence, the entire graph rewriting system is guaranteed to terminate too. The approach was demonstrated on a transformation from state charts to Petri Nets and it was proved that the transformation terminates. Applicability of the approach to other transformation examples was also demonstrated. However, the proposed sufficient termination criteria where found not to apply to model transformations where rules are causally dependent on themselves [81].

de Lara and Taentzer [26] proved termination of a model transformation that transforms process interaction models to timed transition Petri Nets (TTPNs). An intermediate metamodel composed of the source and target metamodels and auxillary entities and links was used to represent the intermediate outputs of a transformation. The transformation from process interaction models to TTPNs was implemented using four layers of DPO-based graph rewriting rules. The study informally discussed how the model transformation terminates and how layering conditions [31] can be adopted to prove termination of the transformation. Critical pair analysis was used to prove confluence of the model transformation where only rules within the same layer were analyzed for conflicts. The study informally discussed how the transformation was also found to produce output

models that preserved syntactic consistency with respect to the target meta-model and behavioural equivalence with the input models. Syntactic consistency was proved by demonstrating how elements present in the intermediate metamodel but not in the target metamodel were consistently removed by the transformation rules and how the transformation rules created elements conforming to the target metamodel. Behavioural equivalence was argued by discussing several example process interaction models and showing how they behave similar to their corresponding TTPNs. A prototype of the approach was implemented using AToM3 and AGG tools. AToM3 is a tool that can capture models and metamodels as graphs and can execute graph rewriting systems. AGG was used to perform consistency checking and to prove confluence using critical pair analysis. AToM3 was used to implement the transformation instead of AGG since AToM3 is a multi-formalization metamodelling tool and the aim of the study was to transform complex systems expressed in multiple formalizations into a formalization that facilitates analysis.

Barroca *et al.* [10] proposed a Turing-incomplete, visual model transformation language, DSLTrans, that guarantees termination and confluence for any model transformation by construction. In DSLTrans, a model transformation is composed of a set of ordered layers that are executed in sequence. Each layer contains one or more transformation rules that are executed in a non-deterministic order. Each rule is expressed as a (match, apply) pattern where a match pattern is a pattern of the source metamodel and an apply pattern is a pattern of the target metamodel. The syntax and the semantics of DSLTrans were formalized using concepts from graph rewriting systems. The study proved that any model transformation in DSLTrans terminates by requiring that the input model is acyclic and using the fact that a transformation can only have a finite number of layers and each layer can have a finite number of rules. Moreover, since DSLTrans does not support recursion or loops, the limited expressiveness of the language further guarantees termination. The study also proved that any model transformation in DSLTrans is confluent albeit the fact that rules in a layer are applied in a non-deterministic order. The non-determinism within one layer is controlled by amalgamating the output of each rule in a layer to the final output using graph union, which is commutative. This ensures that the only place where non-determinism exists in the language produces a deterministic output.

4.1.2 Type II Formal Methods

Type II formal methods analyze certain properties for a specific model transformation (i.e., transformation-dependent) when run on any input model (i.e., input-independent).

Analyzing properties expressed as first-order logic: Asztalos *et al.* [8] analyzed properties of model transformations by formulating transformation rules and the property of interest as assertions in first-order logic. Formal deduction rules were then used to deduce the assertion representing the property of interest from the set of assertions representing the transformation rules. The more deduction rules are defined and the more sophisticated they are, the stronger is the deduction capability of the approach. Thus, the proposed approach can be used to prove any property of a model transformation as long as the property is expressible in first-order logic, such as a rule application deletes all edges of a certain type. The approach was realized as a verification framework in the Visual Modeling and Transformation System (VMTS) using SWI-Prolog to implement the deduction rules. VMTS can automatically generate assertions representing the individual transformation rules.VMTS also parses constraints and imperative code attached to transformation rules and produces equivalent pattern attribute constraints. The verification framework was used to prove a property for a refactoring of business process models. The study claimed that their approach and verification engine are extensible to different model transformation frameworks and can be used to prove different properties. The study also discussed disadvantages of the approach which include its inefficiency if complicated deduction rules were defined or if a large base of assertions was used.

Analyzing property-preservation: Giese *et al.* [36] used the automated theorem prover Isabelle/HOL to verify that a model-to-code transformation specified using triple graph grammars (TGGs) [77] in the Fujaba tool suite[5] preserves some safety property. In other words, assuming that a safety property was verified on the input model, Isabelle/HOL was used to check whether the transformation of interest preserves the safety property in the generated code. To analyze a transformation using Isabelle/HOL, Isabelle/HOL algebraic representations for metamodels were derived from the TGG structures in Fujaba. Safety properties were then defined for such algebraic representations within Isabelle/HOL. TGG rules were also formalized using Isabelle/HOL. The mapping from metamodels, safety properties and TGG rules to Isabelle/HOL structures was performed manually. Since TGG rules rewrite input and output models in parallel (Section 2), Isabelle/HOL was used to prove that the code generated by a TGG rule did not violate the equivalence of the input and output models with respect to the defined safety property.

Becker *et al.* [15] proposed an approach to verify whether a model transformation preserved constraints expressed as (conditional) forbidden patterns in output models. The study formalized systems with dynamic structural adaptation as graph rewriting systems where structural adaptations were formalized as graph

[5]http://www.fujaba.de/

rewriting rules and system safety requirements were formalized as graph patterns. To verify that a system will not violate any safety requirement, one can check whether the graph rewriting rules can transform a safe system state to an unsafe one. However, since the initial system state is unknown and the reachable set of states can be unbounded, the study checked whether the backward application of each rule to each forbidden pattern can result in a safe state. Since the number of rules and forbidden patterns are bounded, this check is plausible. If the backward application of a rule to a forbidden pattern would result in a safe state, then the safe state and the rule are generated as a counter example. However, the study did not address checking whether the safe state in a counter example is reachable in the first place. The approach was demonstrated on a system of autonomous shuttles running on rail tracks with dynamic structural adaptation. A tool was implemented using traditional data structures for graph manipulations and using symbolic representations that can be run on engines optimized for symbolic computations (e.g., SAT solvers). Both implementations were integrated into the Fujaba tool and were compared with model checking using GROOVE [71]. The results showed that GROOVE and the explicit implementation of graph rewriting systems were suitable for small examples. The symbolic implementation scaled better to larger examples with much shorter execution times.

Analyzing preservation of syntactic relations: Lúcio *et al.* [53] implemented a model transformation checker for model transformations formalized as graph rewriting systems and specified in the DSLTrans language [10]. While traditional model checkers build a state space of models resulting from possible rule applications, a model transformation checker builds a state space of possible rule applications regardless the initial and intermediate graphs generated. Properties and rules of a model transformation are expressed in DSLTrans as (match, apply) patterns. The model checker builds the state space where each state is a possible combination of the transformation rules in a given layer, combined with all states of the previous layers. Hence, each state is a combination of (match, apply) patterns, and the set of match patterns in a state are patterns that should exist in an input model in order to reach that state. Using the generated state space, the model transformation checker can be used to prove a *satisfiable* property, produce a counter example for an *unsatisfiable* property or render a property as *unprovable*. The model transformation checker proves that a property is *satisfiable* by traversing the state space and finding a state that satisfies the match pattern of the property and a subsequent state on the same path that satisfies the apply pattern of the property. The model transformation checker proves that a property is *unsatisfiable* by providing a counter example or finding at least one path where the match pattern of the property exists but the apply pattern does not exist in any subsequent state on the same path. *Unprovable* properties are properties that the model transformation checker cannot prove since the match pattern of the property does

not exist in the generated state space. The model transformation checker was used to analyze a simple model transformation example and the study reported on the size of the generated state space. However, the results were not described in detail.

5 Dynamic Analysis Techniques

5.1 Type III Formal Methods

Type III formal methods analyze certain properties for a specific model transformation (i.e., transformation-dependent) when run on a specific input model (i.e., input-dependent). We structure our discussion in this section around the analyzed properties.

Analyzing structural correspondence between input and output models: Narayan and Karsai [62] analyzed structural correspondence between the input and output models of a model transformation to prove that the transformation achieved its intended mapping. Analyzing structural correspondence was achieved in three steps. First, a composite metamodel that contains the source and target metamodels and structural correspondence rules defined between elements of the two metamodels was generated. Second, the model transformation of interest was extended to generate relations or *crosslinks* between input model elements and their corresponding output model elements. Third, the crosslinks were used to evaluate the structural correspondence rules for a specific pair of input and output model instances. The GReAT model transformation framework [2] was used to implement the approach. In GReAT, transformations are specified as graph rewriting systems and crosslinks are automatically generated between input and output model elements. Moreover, GReAT can be used to specify structural correspondence nodes and vertices in a composite metamodel to define structural correspondence rules. The approach was demonstrated on a transformation from UML activity diagrams to communicating sequential process models. However, no rigorous case study was carried out to assess the proposed approach.

Analyzing semantics-preservation: Baar and Marković [9] proposed an approach to prove that a refactoring of UML class diagrams with OCL constraints preserved static semantics. According to the study, a model refactoring that transforms a class diagram with OCL constraints and a set of conforming object diagrams to a new class diagram with new OCL constraints and a new set of object diagrams is said to preserve static semantics if evaluating the initial constraints on the initial set of object diagrams produces the same results as evaluating the refactored constraints on the set of refactored object diagrams. The refactorings of class diagrams, object diagrams and OCL constraints were formalized as separate graph rewriting rules of typed graphs. The evaluation of OCL constraints was also formalized as graph rewriting rules. The study demonstrated how a sample model refactoring was found to preserve static semantics according to the proposed approach.

Analyzing preservation of type consistency and multi-view consistency: A model transformation preserves *type consistency* if it generates output models that are well-formed with respect to the target metamodel and the constraints on the target metamodel. A model preserves *multi-view consistency* if multiple views of the generated model do not contradict each other.

Becker *et al.* [16] proposed an approach to verify that refactoring a metamodel of a modelling language preserved *type consistency* with respect to well-formedness constraints of the modeling language that cannot be specified as a (conditional) forbidden patterns, e.g., two methods in the same class cannot have the same signature. Model refactoring was formalized as a graph rewriting system. The source metamodel was extended with predicate structures and indirect well formedness constraints were specified as graph constraints that manipulate the predefined predicate structures. *Maintenance rules* were evaluated after every refactoring rule to add predicates to the model if the model has a forbidden pattern. If the refactored model overlapped with a forbidden pattern, a counter example was generated by executing the inverse of the rule on the overlap. The approach was demonstrated on two refactorings of the java language metamodel that were proven to be consistency preserving. The study further discovered two bugs in a refactoring that was used by Eclipse, one of which was only recently fixed.

Anastasakis *et al.* [4] formalized a model transformation and its manipulated metamodels as Alloy models that were analyzed using the Alloy analyzer. A tool called UML2Alloy was used to translate the metamodels and their OCL constraints to Alloy models. The model transformation was transformed to an Alloy model by expressing the transformation rules in first-order logic and introducing a *mapping relation* in Alloy to keep track of the mapping between input and output model elements. The study discussed how the Alloy analyzer can be used to analyze the model transformation. Using consistency checking, instances of the source metamodel and the model transformation can be generated and used to produce an output model. Failure to produce an instance of the model transformation signals inconsistencies in the transformation specification which can be resolved using the Alloy analyzer. The Alloy analyzer can also be used to produce several instances of the model transformation to explore different possible mappings between the source and target metamodels. Finally, the Alloy analyzer can be used to perform assertion checking. The approach was demonstrated on a model transformation for business process models. Analysis using the Alloy Analyzer produced a counter example and revealed an error in the transformation. However, the approach was found to be non scalable, inapplicable to non-declarative transformations, not capable of analyzing non-integer properties of the Alloy model and not capable of reasoning about dynamic properties of the transformation, i.e. Alloy does not have any representation for a system's behaviour such as state machines.

A model transformation can also be analyzed for preservation of type consistency

and multi-view consistency by analyzing its output models only. Paige *et al.* [65] used PVS [64] and Eiffel [58] to formalize BON [82] metamodels and perform model conformance checking and multi-view consistency checking (MVCC) between different diagrams of a model. BON is an object-oriented modeling language, that does not have formally defined semantics. PVS is a general purpose theorem proving environment that is not object-oriented. Eiffel is a declarative object-oriented programming language that supports contracts. To represent the BON metamodel and its constraints in PVS, entities of the BON metamodel were represented using *theory* constructs and the BON metamodel constraints were represented as axioms that manipulate the defined theory constructs. To prove model conformance in PVS, a BON model was encoded as a set of PVS expressions and the PVS theorem prover was used to prove that the encoded BON model satisfies the axioms encoding the BON metamodel constraints. To perform MVCC between static and dynamic diagrams in PVS, PVS was extended to represent different views of a system and to define constraints on these views. To perform MVCC between contracts of diagrams, preconditions of successive routines were composed into one axiom. Then, a BON model was encoded as PVS conjecture that satisfied the axiom. On the other hand, all constructs in BON have an equivalent in Eiffel. Thus, representing the BON metamodel in Eiffel was mostly straight forward, besides two issues that needed additional handling. First, code was added to initialize structures and allocate memory to them. Second, class invariants were translated into boolean functions whereas preconditions and postconditions were translated into predicates that use quantifiers. To check model conformance in Eiffel, a BON model was encoded as an object, and the metamodel rules were executed on the object to determine if the model was a valid instance of the metamodel. MVCC between static and dynamic diagrams was achieved in a similar manner to model conformance checking. MVCC between contracts of different diagrams was achieved by generating unit tests from the encoding of dynamic diagrams, and generating Eiffel code from the encoding of class diagrams and running the unit tests against the Eiffel code. The study compared using PVS and Eiffel for model conformance checking and MVCC with respect to several qualitative measures. However, no quantitative case study was carried out to evaluate the two approaches.

5.2 Model Checking

Many studies analyzed model transformations specified using some formalization (Section 2) by *model checking* the state space of the transformation. In many cases, studies also built tools that perform model checking for a specific formalization.

Model checking Maude specifications: Boronat *et al.* [18] used Maude (Section 2.5) to formalize and model check endogenous (Section 1), non-

confluent model transformations. The metamodel manipulated by a transformation was formalized as a membership equational theory, and models conforming to the metamodel were formalized as terms of the membership equational theory corresponding to the source metamodel. Accordingly, a model transformation with NACs was formalized as a rewrite theory that operates on terms of a membership equational theory. Maude was then used to perform three types of analysis: simulation to execute model transformations, reachability analysis to prove satisfaction of invariants and analysis of linear temporal logic (LTL) [25] properties. The proposed approach was implemented as an Eclipse plugin, MOMENT2. MOMENT2 can create QVT-like transformations that manipulate MOF metamodels and models defined in the Eclipse Modelling Framework (EMF). MOMENT2 can also be used to specify invariants as OCL expressions over QVT model patterns and LTL formulae that use predicates as propositions. Using Maude as the underlying engine, MOMENT2 then compiles model transformations, models, metamodels, invariants and LTL properties to perform model checking. The tool and the proposed approach were demonstrated on a transformation example. Analysis using Maude showed that an LTL property was violated and the model transformation was corrected accordingly.

Rivera *et al.* [74] mapped graph rewriting systems to Maude and used Maude analysis techniques to analyze the graph rewriting system. The proposed approach automatically mapped graph rewriting systems to Maude due to the support for analysis in Maude. The study integrated the Maude code generator with AToM3 as a visual front-end to specify graph rewriting systems. A graph rewriting system and its manipulated graphs were automatically transformed into Maude for analysis. Reachability analysis results were transformed back to the visual language of AToM3. Visual support for specifying LTL properties was left as future work. The study demonstrated how the approach helped in revealing properties that were not satisfied by an example transformation. However, no rigorous case study was carried out to assess the proposed approach.

Model checking graph rewriting systems: Rensink [70] used model checking to analyze temporal properties of graph transition systems. Temporal properties of graph transition systems include logic expressions on edge labels, node set expressions and formulae manipulating any of the two former expressions. The study discussed the semantics of the temporal expressions and their evaluation for a graph. Using the proposed semantics, temporal properties can be easily analyzed for the states in a graph transition system. Later, Rensink [72] extended his work in [70] to formally define graph-based, linear temporal logic and the evaluation of temporal logic expressions for graphs with NACs. The proposed approach was implemented as a tool called GROOVE (GRaph-based Object-Oriented VErification) [71]. GROOVE was implemented in Java and supports SPO-

based, graph rewriting systems with NACs. GROOVE supports both stepwise, manual execution of rules on a graph and automatic generation of a graph transition system. When generating the graph transition system, new states are compared to previously-generated ones and matching states are merged. The current version of GROOVE only simulates a graph rewriting system by generating the resultant graph transition system (i.e., the state space). Model checking the graph transition system for temporal properties was left for future work. The tool was demonstrated on a sample model transformation and the results were discussed in terms of the size of the generated state space and the number of outputs produced.

Rensink *et al.* [73] compared two tools, CheckVML and GROOVE, for model checking graph rewriting systems. CheckVML transforms a graph rewriting system system and an initial graph to a Promela model. In a Promela model, graphs are encoded as fixed state vectors and graph rewriting rules are encoded as guarded commands that modify the state vectors. The Promela model is then verified using the SPIN model checker. On the other hand, GROOVE was used to execute a graph rewriting system and build its state space for model checking. The two approaches were evaluated on three model transformations with respect to the size of the generated state space, the memory usage and the execution time. The study concluded that GROOVE is better for problems with dynamic allocation or symmetric nature (i.e., processes and resources are not distinguished from one another; no concurrency).

Narayanan and Karsai [61] used the GReAT model transformation framework [2] to analyze the *bisimilarity* between the input and output models of a graph rewriting system. Two models are said to be bisimilar if one model simulates the other and vice versa. The approach was demonstrated on a graph rewriting system that transforms state charts to Extended Hybrid Automata (EHA) models. EHA models provide formal operational semantics for state charts and hence are more appropriate to use for analysis. An EHA model is *bisimilar* to a state chart model with respect to reachability if a reachable state configuration in a state chart has an equivalent reachable state configuration in an EHA model and vice versa. GReAT maintains cross-links between input and output model elements that can be used to prove bisimilarity. For every transition in a state chart model and its equivalent transition in its corresponding EHA model, the minimal source state configuration is computed for the transition in both models. Equivalence between the start and end state configurations of each pair of equivalent transitions implies that the state chart and the EHA models are bisimilar with respect to reachability. If the models have been proven to be bisimilar, then the EHA model can be transformed to a Promela model and analyzed for reachability using the SPIN model checker. The trace generated by SPIN to prove reachability of a state configuration corresponds to a transition sequence in an EHA model. Using the cross links in GReAT, the transition sequence in the EHA model can be traced back to the transition sequence

in the state chart. The proposed approach uses both Type III formal methods (Section 5.1) to analyze bisimilarity and model checking to analyze reachability.

Model checking Petri Nets: König and Kozioura [44] proposed a tool, Augur2, that approximates graph rewriting systems with Petri Nets for which several analysis techniques have already been developed. The tool then analyzes structural properties of the resultant Petri Nets for all reachable markings. The graph rewriting system and the initial graph are used to generate the state space of graphs which is transformed to a state space of Petri Net markings. The user can then specify a property to be verified as a graph pattern which is transformed by Augur2 to an equivalent Petri Net marking. Accordingly, Augur2 either verifies that the property is satisfied or produces a counter example which is an execution of the Petri Net producing a marking that represents a graph violating the property being analyzed.

Model checking programs generated by model-to-code generators: Ab Rahim and Whittle [1] analyzed the semantic conformance of UML State Machine-to-Java code generators with respect to the source language semantics, UML state machines, using model checking. The approach used a code generator to transform a UML state machine to its corresponding Java code. In parallel, assertions that capture the semantics of state machines were defined and a transformation was implemented which compiled the assertions into a single *verification component.* The transformation also appended annotations within the generated code that referred to the verification component. Using the appended annotations, Java Path Finder (JPF) was used to model check the semantic-conformance of the generated code to the source language semantics by checking if any of the assertions were violated. The approach was implemented as a tool and was demonstrated on a sample state machine-to-Java code generator. Two case studies were carried out on the state machine-to-Java code generators of two commercial tools: IBM Rhapsody [69] and Visual Paradigm [40]. The results revealed that the commercial tools did not fully conform to the semantics of UML state machines. This was mainly because some assertions were violated or because assertions could not be verified since the commercial tools did not support the relevant UML notations required in the assertions.

5.3 Design Space Exploration (DSE)

Design Space Exploration *(DSE)* involves analyzing several model transformation outputs that meet some design constraints or that achieve acceptable values for non-functional metrics [39]. DSE and model checking operate in the same way;

both use the transformation to generate the state space of intermediate models and both check each state for some property. However, DSE and model checking differ in the intent of the analysis. DSE has been used to analyze different outputs of non-confluent transformations, all of which are valid solutions with respect to the model transformation specification (e.g., invariants or safety requirements) but represent different design options or different values of non-functional metrics. On the other hand, model checking analyzes the state space with respect to the specification of the model transformation of interest (e.g., states that violate invariants are ruled out).

Hegedus *et al.* [39] proposed a framework that performs guided DSE of graph rewriting systems. The inputs to the framework are an initial graph, a graph rewriting system, global constraints on all states, and goals on solution states. Global constraints and goals are numerical or structural constraints on states. The framework uses *selection criteria* and *cutoff criteria* to *guide* the DSE. Selection criteria prioritize promising paths and are defined based on the dependencies between the graph rewriting rules. Cut off criteria identify and prune unpromising paths and are defined based on an algebraic analysis of the Petri Net abstraction of the graph transition system. Using the defined criteria, DSE is executed in a series of steps. For the current state, the framework evaluates all cutoff and selection criteria and identifies the rules applicable to the current state. If one of the cutoff criteria holds or if there are no applicable rules then the state is marked as a dead end. Otherwise, the framework selects the next applicable rule based on the selection criteria and applies it to the current state to generate a new state. If the new state is a solution as specified by the goals, then the solution trajectory (i.e., applied rules and final state) is saved and the next applicable rule is applied to a new state or to the previous state. However, if the new state does not satisfy the global constraints then search continues from the previous state. If the new state is not a solution but satisfies the global constraints, search continues from the same state. DSE stops if a predefined number of solutions are found or if the state space was searched exhaustively. The framework was evaluated on two case studies. The study concluded that using the guided DSE finds an optimal solution (i.e., a trajectory that uses a minimal number of rule applications) much earlier than the traditional depth first DSE and the additional load of evaluating the criteria is negligible.

Drago *et al.* [29] proposed a framework, QVT-Rational, that performs DSE of solutions that satisfy non-functional requirements or desired values for quality metrics. QVT-Rational is executed in three phases. In the first phase, a domain expert specifies the manipulated metamodels, the quality metrics of interest, a quality prediction tool chain and a quality-driven model transformation.[6] To build

[6]Quality-driven model transformations are implemented in model transformation languages with constructs that support representing non-functional attributes of transformations.

quality-driven model transformations, the QVT-Operational language was extended with constructs to support the definition of more than one mapping (i.e., the definition of a non-confluent transformation) and allow quality metrics to be bound to them. In the second phase, a designer specifies the input model and the (hard or soft) requirements of the quality metrics. In the third phase, the designer runs the framework to get viable outputs and their corresponding quality predictions. The framework provides an interactive mode and an automatic exploration mode. In the interactive mode, every application of a predefined mapping requires the designer's intervention until a final output model is obtained. In the automatic exploration mode, the transformation generates all possible outputs based on the defined mappings, checks the generated outputs with respect to the specified requirements and proposes the viable outputs. The framework was demonstrated on two model transformation problems and was found to scale well for input models of varying sizes. However, the framework was found to have a few disadvantages. To transform large models, exploring the entire state space might be impossible, and thus the generated solution might not be an optimal one. Moreover, the run time of the quality prediction tool affects the efficiency of the framework. Also, the quality of the viable outputs generated by the framework is dependent on the domain expert who designed the model transformation and bound different mappings to different quality metrics according to his/her experience.

Schätz *et al.* [76] formalized the solution space of non-confluent model transformations using a relational, rule-based characterization of the transformation's constraints. Models were represented as Prolog terms and transformation rules were represented as Prolog predicates that relate models before and after a rule application. The state space of a model transformation was formalized using a relational, rule-based description of design constraints in terms of the predicates representing the transformation rules. The characterization was formalized as a set of (pre-model,post-model) relations. This formalization was then interpreted by Prologas a non-confluent transformation. The approach was implemented as an Eclipse plugin and can be used to specify and execute transformations for EMF Ecore models. The plugin transforms the resultant Prolog solutions back into their equivalent EMF Ecore model. The approach was demonstrated for the deployment of logical software components and buses to their corresponding physical units and links. The possible solutions had to satisfy two non-functional constraints: completeness and resource consistency. Completeness implies that all input model elements are mapped to output model elements. Resource-consistency implies that the load required by the input components and buses do not exceed the load provided by the output units and links. Several viable outputs were generated and the study proposed and implemented optimizations to the approach that helped cutting down on the execution time and memory usage.

5.4 Instrumentation

Instrumentation is a dynamic analysis approach originally developed for analyzing source code. Instrumentation involves monitoring the inner workings of a model transformation by adding *instrumentation code* to the transformation to debug or analyze the transformation. Model transformation instrumentation may induce the *probe effect* which can affect factors such as the execution time. A possible solution to take the probe effect into account is to control the system clock as seen by the transformation of interest [30]. Although instrumentation of source code has been extensively researched, not many studies explored instrumentation of model transformations.

Dhoolia *et al.* [27] used *dynamic tainting* or dynamic tagging to instrument model-to-text transformations and debug faulty input models. In dynamic tainting, tags or *taints* are added to input model entities (i.e., elements and attributes) and the model transformation propagates the taints of input model entities to the corresponding output substrings. The study differentiated between two kinds of taints: data taints and control taints. Data taints are taints propagated by the transformation to output substrings generated as a result of assignment statements or statements that generate an output string. Control taints are taints propagated by the transformation to output substrings generated as a result of a conditional evaluating to a specific value. Loop taints are control taints which mark the scope of a loop. The proposed approach is composed of three phases. First, the user specifies *markers* in the output string to mark a faulty or missing output substring. Second, the transformation is executed in the debug mode where the transformation associates a unique taint with each input model entity and propagates the taints to the corresponding output substrings. Execution of the transformation in the debug mode generates a log file with taints, where each taint is identified by its type and corresponding input model entity. Finally, the user analyzes the log file with the preset fault marker to locate the faults in the input file. Backward traversal of the faulty output is carried out so that if the initial taints do not identify the fault, the next set of enclosing control taints are investigated iteratively until the fault is identified. The approach was implemented as a framework and a case study was conducted on six model-to-text transformations. For all faulty input models, the fault search space was either significantly decreased or precisely identified. However, the run-time and the size of the instrumented transformation significantly increased in some cases. Although the approach was used to debug faulty input models, it can be easily extended to debug and analyze faulty transformations. The generated taints can save the statement in the transformation that was used to generate it, besides the taint's type and the corresponding input element. Thus, assuming that the input model is correct and that a fault in the transformation resulted in a faulty output, the taints can be used to identify which statements of the transformation produced the fault.

5.5 Testing

Testing executes a model transformation on input test models and validates that the generated, actual output model or code matches the expected output model or code [38]. Although testing does not fully verify the correctness of a model transformation, it has been gaining increasing interest due to several factors. The major advantage of testing is its usefulness in uncovering bugs while maintaining a low computational complexity [37]. Other advantages of testing include the ease of performing testing activities, the feasibility of analyzing the model transformation in its target environment and the feasibility of automating most of the testing activities [52].

In this section, we differentiate between a model transformation implementation and a model transformation specification. A model transformation implementation is the source code that carries out the expected mapping between the source and target metamodels.[7] Whereas, a model transformation specification includes the source and target metamodels manipulated by the model transformation, the constraints of the source and target metamodels and the contracts of the model transformation. A contract is usually composed of three sets of constraints [24]: (1) constraints on input models to classify them as valid inputs, (2) constraints on output models to classify them as valid outputs and (3) constraints on relationships that must be maintained between input model elements and output model elements. Several studies proposed taxonomies of model transformation contracts. Baudry *et al.* [11] defined three levels of contracts: contracts of transformations, contracts of subtransformations and contracts of output models. Contracts of transformations include preconditions and postconditions of transformations and transformations' invariants. Contracts of subtransformations include preconditions and postconditions of subtransformations constituting the transformation of interest and their invariants. Contracts of output models are expected properties of output models such as the target metamodel and its constraints. Mottu *et al.* [60] proposed another taxonomy where contracts were categorized as either syntactic or semantic contracts. Syntactic contracts ensure that the transformation can successfully run without errors. Examples of syntactic contracts are the source and target metamodels of a model transformation. Semantic contracts are context-dependent and can be subdivided in to three types of contracts: a transformation's preconditions on input models; a transformation's postconditions on output models; and constraints linking input and output model elements (usually specified in a transformation's postconditions).

We break down the process of model transformation testing into three phases, inspired by those defined in [12] with minor changes. The first phase in

[7] In this section, we use the notion of a *model transformation* and a *model transformation implementation* interchangeably.

model transformation testing is *test case generation* which involves generating a set of input, test models conforming to the source metamodel for testing the transformation of interest. The generated models are referred to as the *test suite*. Efficient criteria are necessary to generate an *adequate* test suite to test the transformation. Such criteria are referred to as *adequacy criteria*. The percentage of adequacy criteria satisfied by a test suite is referred to as the *coverage* achieved by the test suite [55], shown in Equation (1).

$$Coverage = \frac{|AdequacyCriteriaCoveredByATestSuite|}{|AdequacyCriteria|} * 100\% \tag{1}$$

Thus, test case generation approaches aim to generate test suites that achieve full *coverage* of predefined test adequacy criteria.

The second phase in model transformation testing is assessing the test suite generated in the first phase. Although this step is not necessary, it can be used to evaluate the test suite and the reliability of the testing process; test suites that produce positive assessment results are more likely to result in a non-faulty transformation that can be reused with a higher confidence. To this end, many studies used *mutation analysis* to assess the fault detection capability of a test suite.

The third phase in model transformation testing is building the oracle function. The oracle function is the function that compares the actual output of a model transformation with the expected output to evaluate the correctness of the transformation [33]. If the expected output models are available, then the oracle function is a model comparison or a *model differencing* task ([51, 43, 52]) between the actual and the expected outputs. However, in most cases, the expected output is in the form of output specifications or output *contracts* ([24, 23, 60, 37, 48]). In that case, the oracle function ensures that the actual output of the transformation conforms to the specified contracts.

In the following subsections, we first discuss the challenges in model transformation testing. Then, we explain the three phases of model transformation testing in more details.

5.5.1 Challenges of Model Transformation Testing

Three characteristics of model transformations present challenges in model transformation testing [12, 11]: complexity of the manipulated data, limitations of development environments supporting MDD, and the diversity of model transformation languages. We argue that all the other model transformation analysis techniques face these challenges too.

Complexity of the manipulated data Model transformations manipulate input and output models that can vary in complexity, size and constraints [12]. Moreover, the input and output models can have multiple views which need to be kept consistent throughout a transformation.

Complexity of the input models poses challenges in test case generation [12, 11, 46]. Several adequacy criteria were defined, and no study has been conducted to evaluate the usefulness of the criteria. Moreover, the generated test suite must conform to the source metamodel while satisfying the source metamodel constraints, the transformation's preconditions and the predefined adequacy criteria. Accordingly, three approaches can be followed to generate a valid test suite. The first approach involves manually building the test suite then checking that the test suite conforms to the source metamodel constraints, the transformation's preconditions and the predefined adequacy criteria. Using the first approach, many of the generated test models may turn out to be unfit for testing. The second approach involves automatically generating a test suite using constraint solving techniques to solve source metamodel constraints, the transformation's preconditions and the predefined adequacy criteria. The limitation of the second approach is the cost of constraint solvers and that models generated automatically may not be easily understood by testers. The third approach is an interactive one and involves manually building an initial test suite that can be automatically assessed with respect to some adequacy criteria. If the test suite is inadequate, a tool can provide suggestions of the additional test models required.

Complexity of the output models or code poses challenges in building the oracle function [12, 46, 11, 52]. Three approaches for building the oracle function were identified. The first approach is applicable if the expected output models are available. In that case, building the oracle function is a model comparison or a model differencing task. In model differencing, models that are syntactically different but semantically equivalent should be identified as equivalent models. Visualizing techniques are also needed to capture model differences in a comprehensive manner. The second approach is applicable if the expected output models are not available. In that case, a partial oracle function can be built where the expected properties of the output models, i.e., contracts, are described in a suitable language. The oracle function checks the conformance of the output models to such contracts. The third approach is applicable if the model transformation produces an executable output. In that case, the oracle function should generate a test suite to test the executable output produced by the model transformation of interest.

Limitations of development environments supporting MDD Development environments supporting MDD have not yet fully evolved to support model manipulation, hence facilitating model transformation testing. Model manipulation

includes model building, editing, visualization and analysis [12, 46]. Building and editing models in such environments is usually error prone since there is little support for automatic model checking and valid initialization of model properties. Model visualization requires support for automatic layout of models. To support model analysis, development environments should integrate model comparison and versioning capabilities to ensure that the tester can analyze the transformation and its input and output models in one environment. Moreover, to support model transformation testing, debuggers for model transformations are needed [52].

Diversity of model transformation languages Due to the diversity of model transformation languages currently in use, testing techniques that take this language diversity into account are needed [12, 33]. If white-box testing (Section 5.5.2) will be used, then the choice of the transformation language affects the implementation of the white-box testing approach. Black-box testing (Section 5.5.2) can be used instead since it is language-independent. However, black-box testing does not take advantage of the purpose of the transformation and hence is not as effective as white-box testing. One possible solution is to define different white-box test adequacy criteria that account for different model transformation languages.

5.5.2 Phase I: Test Case Generation

Test case generation involves defining test adequacy criteria and building a test suite that achieves coverage of the adequacy criteria. The test suite can be generated automatically, manually or interactively as explained in Section 5.5.1.

Defining test adequacy criteria, and hence test case generation, can follow a *black-box*, *grey-box* or *white-box* approach. A black-box test case generation approach assumes that the implementation of the transformation of interest is not available and builds a test suite based on the specification of the model transformation (i.e., source metamodel or contracts). A grey-box test case generation approach assumes that the implementation of the transformation of interest is partially available and builds a test suite based on the accessible parts of the implementation [38]. A white-box test case generation approach assumes that the full implementation of the transformation of interest is available and builds a test suite based on the implementation of the transformation. We discuss black-box and white-box test case generation in more details. We do not discuss grey-box test case generation any further, since it has been rarely investigated for testing model transformations. Moreover, grey-box test case generation can use the same approaches as those proposed for white-box test case generation but only on the accessible parts of the implementation.

Black-Box Test Case Generation Based on Metamodel Coverage Different adequacy criteria have been proposed in the literature to achieve coverage of the source metamodel of the model transformation of interest.

Adequacy criteria for class diagrams were heavily investigated. Andrews *et al.* [5] proposed three adequacy criteria for class diagrams: the association-end multiplicity (AEM) criterion, the generalization (GN) criterion and the class attribute (CA) criterion. The AEM criterion requires that each representative multiplicity-pair of two association ends gets instantiated in at least one test model. The GN criterion requires that each subclass gets instantiated in at least one test model. The CA criterion requires that each representative attribute value gets instantiated in at least one test model. In AEM and CA criteria, representative values are used since the possible values of multiplicities and attributes can be infinite. Representative values are created using *partition analysis* where multiplicity and attribute values are *partitioned* into mutually exclusive ranges of values. A representative value from each range must be covered in the test suite. For building partitions, either default partitions can be automatically generated or knowledge-based partitions can be generated by the tester. Other studies [33, 35] also used the AEM and CA criteria and proposed the notion of a *coverage item* which is a constraint on the test suite that requires certain combinations of representative attribute values, representative AEM values and objects of classes to be instantiated in the test suite. A test adequacy criterion can then be defined for each coverage item. Fleurey *et al.* [32] also combined classes, representative attribute values and representative AEM values into coverage items. A coverage item for an object was referred to as an *object fragment*. A coverage item for a model was referred to as a *model fragment* and is composed of several object fragments. The study then proposed different adequacy criteria, each criterion specifying a different way of combining object fragments into a model fragment. The Meta Model Coverage Checker (MMCC) tool was built to implement the proposed criteria. MMCC takes a coverage criterion and a source metamodel as inputs and generates the required model fragments to guide the tester in test case generation. MMCC can then be used to assess if a test suite achieves coverage of the generated model fragments and points out model fragments that were not covered to aid the tester in improving the test suite. A case study was conducted on a model transformation and the tool was found to be useful in test case generation with the drawback of suggesting model fragments that are not feasible, e.g., MMCC suggested a model fragment with zero transitions and one transition in an input state machine.

Adequacy criteria for interaction diagrams were also discussed in the literature. Five adequacy criteria were proposed in [5]: condition coverage (Cond), full predicate coverage (FP), each message on link (EML), all message paths (AMP) and collection coverage (Coll). The Cond criterion requires that each condition

gets instantiated in the test suite with with both *true* and *false*. The FP criterion requires that each clause in every condition gets instantiated in the test suite with both *true* and *false* such that the value of the condition will always be the same as the value of the clause being tested. The EML criterion requires that each message on a link connecting two objects gets instantiated in at least one test model. The AMP criterion requires that each possible sequence of messages gets instantiated in at least one test model. The Coll criterion requires each interaction with collection objects of various representative sizes gets instantiated in at least one test model. Similar to class diagrams, coverage items can be created for interaction diagrams and test adequacy criteria can be defined accordingly. Ghosh *et al.* [35] also used the Cond Criterion, the FP criterion, the EML criterion and the AMP criterion. Wu *et al.* [84] used the AMP criterion, the transition coverage criterion and proposed the all content-dependency relationships criterion for collaboration diagrams. Transition coverage criterion requires that each transition type gets instantiated in at least one test model. The all content-dependence relationships criterion is based on extracting data-dependency relationships between system components and requires that each identified relationship gets instantiated in at least one test model.

Offutt and Abdurazik [63] proposed four adequacy criteria for UML statecharts: the transition coverage criterion, the full predicate coverage criterion, the transition pair coverage criterion, and the complete sequence coverage criterion. The transition coverage criterion requires that every transition type in a state chart gets instantiated in at least one test model and was reused in [84]. The full predicate coverage criterion is similar to the FP criterion of collaboration diagrams and works on the predicates of each available transition. The transition pair coverage criterion requires that each pair of adjacent transitions gets instantiated in at least one test model. The complete sequence coverage criterion requires that each complete sequence of transitions that makes full use of the system of interest gets instantiated in at least one test model. Due to the infinite possible complete sequences, a domain expert must define a set of expected sequences that are crucial to be tested. The study further developed a tool, UMLTest, that generates test suites for UML state charts built using IBM Rational Rose. An experiment was conducted and test suites were created for a system of moderate size. Mutation analysis (Section 5.5.3) was used to assess and compare the proposed criteria. It was found that the full-predicate coverage criterion had the best fault detection capability followed by the transition pair coverage criterion. Haschemi [38] proposed the all-configurations-transitions criterion for statecharts with parallelism. The criterion requires that all transitions between all state configurations in the reachability tree of a state chart get instantiated by the test suite. Wu *et al.* [84] proposed the all content-dependency relationships criterion for statecharts. The criterion is based on extracting data-dependency relationships between system components and requires that each identified relationship gets instantiated in at least one test model.

Mc Quillan and Power [55] surveyed black-box, adequacy criteria proposed in the literature for class diagrams, sequence diagrams, communication diagrams, state machine diagrams, activity diagrams and use case diagrams. No studies discussed adequacy criteria for deployment diagrams, component diagrams, composite structure diagrams, interaction overview diagrams and timing diagrams. The study also reviewed how the different criteria were evaluated for effectiveness and how were they compared with each other. The paper concluded that little work was done on evaluating the effectiveness of the criteria in detecting faults and on comparing the criteria in terms of the coverage they provide.

Black-Box Test Case Generation Based on Contract Coverage Different adequacy criteria have been proposed in the literature to achieve coverage of the input contracts of the model transformation of interest. Fleurey *et al.* [33] proposed constructing an *effective* metamodel composed of the source metamodel elements referenced in the preconditions and postconditions of a model transformation. The effective metamodel can then be used to define test adequacy criteria by partitioning the values of attributes and multiplicities in the effective metamodel, generating coverage items from the resultant partitions and deriving a test adequacy criterion for each coverage item. No case study was conducted to verify the usefulness of the proposed approach.

Bauer *et al.* [13] proposed a *combined specification-based coverage* approach for testing a model transformation chain, where contract-based and metamodel-based adequacy criteria were generated from the transformations in the transformation chain. Traditional metamodel-based adequacy criteria, such as the AEM criterion, were used. Contract-based adequacy criteria were generated that require the execution of each contract by at least one test model in the test suite. Using the generated coverage criteria and an initial test suite, a *footprint* was generated for each test model. A footprint is a vector of the number of times a test model covers each adequacy criterion. The quality of the test suite was then measured using the footprints of all the available test models to assess the covered adequacy criteria, the uncovered adequacy criteria and the redundant test models. The generated information was used to guide the tester to add or remove test cases to improve the quality of the test suite. The study proposed a tool that implements the proposed approach called the Test Suite Analyzer for Model Transformation Chains. A case study was conducted on a commercial model transformation chain with a test suite of 188 test models. Several test adequacy criteria were found to be unsatisfied and adding test models to cover these criteria revealed faults in the model transformation chain. Moreover, 27 redundant test models were identified, 19 of which were removed after manual examination. Bauer and Küster [14] investigated the relation between specification-based (black-box) test adequacy criteria proposed in [13] and code-based (white-box) test adequacy criteria derived

from the control-flow graph of a transformation chain. Such a relation can be useful in many ways. First, the relation can be used to determine parts of the specification that are relevant to a code block and vice versa. Second, the relation can be used to identify code and specification relevant to a test model to facilitate debugging the transformation for failing test models. Third, the relation can be used to determine how closely related the two types of criteria are and hence how closely the implemented code reflects the specification. The relation between specification-based and code-based test adequacy criteria was generated using the test suite in the following manner: if a test model satisfies a code-based test adequacy criterion *c1* and a specification-based test adequacy criterion *s1*, then *c1* and *s1* are related. The coverage of the two types of criteria were computed for each test model and was used to generate a scatter plot and a correlation coefficient for the test suite. A positive linear scatter and a correlation coefficient close to one implied that the code implemented its specified behavior. The study used the same tool and model transformation chain used in [13] to investigate the relation between the two types of criteria using the proposed approach. Several conclusions were reached. For example, the coverage achieved for the code-based and specification-based criteria were found to be linearly correlated. Thus, properties of code blocks were deduced from their related specifications without having to manually analyze the code.

White-Box Test Case Generation Different adequacy criteria have been proposed in the literature to achieve coverage of the implementation of the model transformation of interest. Since the criteria are transformation language-dependent, different criteria have to be defined for model transformations implemented in different languages.

Fleurey *et al.* [33] proposed statically analyzing a model transformation to build an *effective* metamodel from the model transformation implementation that can be used to define test adequacy criteria. The effective metamodel is composed of the source metamodel elements referenced in the model transformation implementation. The effective metamodel can be generated automatically using a static type checker of the transformation language used. During type checking, all metamodel elements referenced in the transformation are collected to form the effective metamodel. Attributes and multiplicities that constitute the effective metamodel can then be partitioned to choose representative values of the input domain and the defined partitions can be used to generate coverage items. Accordingly, a test adequacy criterion can be generated for each coverage item to ensure coverage of the effective metamodel. No case study was conducted to verify the usefulness of the proposed approach.

Küster and Abd-El-Razik [46] investigated white-box test case generation for business process development. The study proposed three white-box test case generation approaches to test five business process model transformations built

using IBM WebSphere Business Modeler. The five model transformations were specified as a set of conceptual transformation rules that were later used to implement the transformation in Java. The first white-box test case generation approach was based on transforming a conceptual rule into a source metamodel template, from which model instances can be created automatically. To create a source metamodel template from a transformation rule, abstract elements in conceptual rules must be made concrete, i.e., parameterized. Thus, several source metamodel templates were generated from each rule to ensure source metamodel coverage per rule. The second white-box test case generation approach was proposed to experiment with output models with constraints. Output model elements affected by the transformation of interest were identified, then constraints dependent on these output model elements were identified. For each identified constraint, a test case that affects the constraint of interest was generated. The third white-box test case generation approach used rule pairs for generating test cases. The approach was based on the idea that errors can occur due to the interplay of rules if the transformation of interest is not confluent. Critical input models were constructed that contain overlapping match patterns of rule pairs. The output of the transformation was then analyzed to test confluence of the model transformation. After evaluating the three approaches, the study concluded that the approach based on rule pairs revealed fewer errors than the first two approaches. However, no detailed results were demonstrated in the study.

McQuillan and Power [56] assessed the coverage of ATL rules by profiling the operation of each rule during the execution of the transformation. ATL has two features which allow it to support such profiling. First, compiled ATL rules are stored in XML files and are executed on top of a special purpose virtual machine. Second, ATL can run in debug mode and can print out a log file of the executed instructions. To assess rule coverage of ATL transformations, the study proposed a two phase-approach. In the first phase, the XML file resulting from compilation of the model transformation is processed to extract statistics about ATL code structures in the transformation. From the first phase, compiled ATL code was found to have three main classes of code structures: irrelevant *scaffolding code* (e.g., routines for resolving references); code corresponding to rules (i.e., each rule had two functions used to match input model elements and to generate output model elements) and code corresponding to helpers (each functional or attribute helper had a corresponding function). The study extracted the available rules and helpers from the XML file. Scaffolding code was not considered in the study. In the second phase, the transformation was executed using the available test suite. For each test model, the resulting log file was processed to find out how much of the code structures extracted from the first phase were covered according to three white-box coverage criteria: rule coverage, instruction coverage and decision coverage. Accordingly, the cumulative coverage achieved by the entire test suite was calculated by accumulating the coverage results achieved for each test model.

The approach was applied to two transformations from the ATL transformation zoo [34]. The results revealed that improving the decision coverage can improve coverage of the transformation as a whole.

Lämmel [47] used white-box test case generation for grammar testing. Although the study proposed an adequacy criterion for grammar testing, the criterion can be leveraged for model transformation testing since using a grammar or a parser to transform a language is similar to using a transformation to transform models. The study proposed a modified version of the rule coverage criterion that requires at least one test model in the test suite to trigger each rule in the grammar. The modified version of the criteria, referred to as context-dependent branch coverage, requires at least one test model in the test suite to trigger each rule in every possible context. For example, if the outcome of rule *r1* can trigger either rule *r2* or rule *r3*, then there must be one test model that triggers *r1* then *r2*, and another test model that triggers *r1* then *r3*. However, no case study was conducted to evaluate the efficiency of the proposed criterion in detecting model transformation faults.

5.5.3 Phase II: Test Suite Assessment

Many studies used the coverage achieved by a test suite with respect to some adequacy criteria as an assessment of the quality of test suites ([5, 33, 35, 32, 84, 63, 38, 55, 13, 14, 46, 56]). Other studies used *mutation analysis* instead ([60, 48, 55, 59, 63]). We discussed in Section 5.5.2 how to measure the coverage achieved by a test suite with respect to black-box or white-box adequacy criteria. In this section, we discuss mutation analysis in more depth.

Mutation analysis [59] is a technique used to assess the quality of a test suite by evaluating the sensitivity of the test suite to faults or *mutations* in the model transformation of interest. Mutation analysis involves injecting faults in a transformation to generate *mutants* or faulty versions of the transformation. *Mutants* are generated by applying *mutation operators* or operators that inject faults in the original transformation. The injected faults represent *fault models* or common errors committed by developers when building transformations. The mutants and the original transformation are then executed using each test model in the test suite under assessment. For each mutant, the output of the mutant is compared to the output of the original transformation. If one test model produces different results for the transformation and the mutant, then the mutant is *killed*; i.e., a test model from the test suite detected the injected fault. The mutant stays *alive* if no test model detects the injected fault. A mutant that can not be killed by any test model is an *equivalent mutant* and has to be discarded from the set of mutants. A *mutation score* is computed to evaluate the test suite as shown in

Equation (2).

$$MutationScore = \frac{|KilledMutants|}{|Mutants| - |EquivalentMutants|} \quad (2)$$

A high mutation score indicates that the test suite is sensitive to faults in the transformation and that the test suite is adequate. A low mutation score indicates that the test suite needs to be improved. Unlike measuring the coverage of adequacy criteria, mutation analysis evaluates a test suite in terms of its fault revealing power.

Mottu *et al.* [59] proposed *semantic* mutation operators to assess test suites of model transformations. *Semantic* mutation operators model semantic faults which are normally not detected when programming, compiling or executing a transformation. Four basic operations were identified in any transformation: input model navigation, filtering of the navigation result, output model creation or input model modification. The study then proposed fault models and hence, mutation operators related to each of the four operations. An example of a mutation operator related to the navigation operation is navigating the wrong association. An example of a mutation operator related to the filtering operation is removing the filtering applied to the result of a navigation operation. An example of a mutation operator related to output model creation or input model modification is creating or modifying a model element compatible with the desired model element. Using all the proposed mutation operators, mutants were generated for a Java transformation and were compared with the mutants generated using a commercial tool called MuJava. MuJava uses classical mutation operators that exist in any programming language and are not dedicated to MDD. MuJava generated almost double the number of mutants generated from the proposed operators, with more mutants being not viable, i.e., detected at compile-time or run-time. The study concluded that the easiness of implementing a mutation operator is dependent on the mutation operator and the language used. For example, it was found more difficult to implement mutation operators in Java than in any other language.

Dinh-Trong *et al.* [28] discussed fault models and mutation operators for UML models. Although the study focused on mutation analysis of models and not model transformations, we discuss how the ideas can be leveraged for transformations. Three main fault models were identified: design-metric related faults, faults detectable without execution and faults related to behavior. Design metric related faults result in undesirable values for design metrics, such as cohesion. Undesirable values for such metrics do not necessarily imply a fault, but can imply problems in non-functional properties of the transformation, such as understandability. Faults detectable without execution result from syntactic errors and are easily killed by MDD development environments. Faults related to behavior are usually more difficult to detect. They arise due to incorrect specification of a transformation that is syntactically correct, hence can easily go undetected. Several mutation operators

were discussed and classified according to the proposed taxonomy. Proposed mutation operators included mutating assignments of class attribute values and swapping compatible parameters in an operation definition. The study claimed that equivalent mutants usually occur if the mutation operator results in a weaker constraint, e.g., a mutation operator that removes the filtering applied on navigation results can result in an equivalent mutant. Only static analysis techniques can detect equivalent mutants.

5.5.4 Phase III: Building the Oracle Function

In model transformation testing, an oracle function compares the actual output with the expected output to validate the transformation of interest. If the expected output is known, then the oracle function is a model differencing or a model comparison task. However, if the expected output is not known, then the oracle function validates the transformation's output with respect to some predefined output contracts.

Model differencing or model comparison has been identified as a major task in model transformation testing [43]. Lin *et al.* [52] proposed a complete model transformation testing framework which was integrated with the model transformation engine C-SAW and used model comparison as the oracle function. The model transformation language used in C-SAW is ECL, an extension of OCL. In ECL, a transformation specification can be either a strategy or an aspect. A strategy specifies the required transformation, while an aspect specifies the binding of a strategy to an input. The proposed framework has three components: a test case constructor, a test engine and a test analyzer. The input of the test case constructor is a set of *test cases* composed of the paths of the input models and their expected output models, and the strategy specifying the transformation. For each test case, the test constructor generates an executable test case with defined aspects. The study assumed that the input models and their expected output models were manually built. Automation was in the execution of the tests and in integrating all the testing steps into one framework. The test engine executes the generated test cases and compares the generated output models with the expected output models. The output of the test engine (i.e., the result of the comparison) is then passed to the test analyzer. The test analyzer visualizes the comparison's result using different colors and shapes to facilitate analysis. The study demonstrated a case study to show how the framework helped detect errors in a model transformation example.

However, model comparison is a complete field and has many dimensions that need to be addressed to be carried out successfully [51]. These dimensions include syntactic differencing, semantic differencing and visualisation of model differences.

Thus, in this paper, we do not discuss model comparison or differencing any further. In this section, we only discuss using contracts as oracle functions.

Contracts as Oracle Functions Contracts can be used as partial oracle functions, where contracts specify expected properties of the transformation and its output, and are used to verify that the specified properties hold.

Many specification languages for defining model transformation contracts were proposed. For example, Java Modeling Language (JML) [49] is a language that can be used to define contracts for model transformations in Java. However, OCL [83] has been used in many studies for specifying model transformation contracts. Cariou *et al.* [24] discussed how to use OCL to specify model transformation contracts. Constraints on input and output models were specified using OCL invariants on profiles of the UML metamodel. Constraints on relationships between input and output models were specified using two approaches. In the first approach, OCL expressions that manipulate model elements were specified in the postcondition of the transformation. In the second approach, OCL expressions that manipulate models as packages were specified in the postcondition of the transformation. The first approach has several advantages. The mapping between input and output model elements is implicit, i.e., input model elements that are not manipulated in the postcondition will automatically be maintained in the output model. Moreover, OCL navigational expressions are simple due to the common metamodel. On the other hand, a major disadvantage of the first approach is that it can only be used in transformations where the source and target metamodels are the same since the transformation operation must be owned by a classifier (e.g., a class or a data type) of one metamodel. For transformations with different source and target metamodels, a common metamodel needs to be defined and a classifier of the common metamodel must be chosen to own the transformation operation. Finding a common metamodel is not always easy; the metamodels may have contradicting constraints. Further, the classifier must be carefully chosen to enable all elements in the OCL expressions to be referenced and to make OCL navigational expressions as simple as possible. The first approach was demonstrated on two transformations [23]. The first transformation is a simple model refactoring where the target metamodel is a constrained profile of the UML metamodel. Both the source and target metamodels conform to the UML metamodel which was used as the common metamodel and a contract was defined. The second transformation was more complex where the source and target metamodels had conflicting constraints. The constraints on the target profile were relaxed so that a common metamodel can be found and a classifier of the common metamodel was chosen to define the contract. In the second approach, contracts were specified as OCL expressions in the postcondition of a model transformation that manipulate models as packages. The main advantage of the second approach is that it can

be used to define contracts for model transformations that manipulate different source and target metamodels. However, disadvantages of the second approach include the need for an OCL extension to define explicit mappings between input and output model elements and their relationships. Since explicit mappings are required, the transformation specification becomes more verbose, hence difficult to read and maintain. Further, OCL navigational expressions can also be verbose due to the use of different metamodels. The study demonstrated the second approach to define contracts for a model transformation example. An extension for OCL was defined and accordingly, contracts were specified for the packages containing the source and target metamodels.

Mottu *et al.* [60] proposed an integrated design and test approach for building *vigilant* model transformations using contracts. Vigilant model transformations have contracts that are precise enough to detect errors at run time. Vigilant transformations were built by checking the consistency between a transformation's test suite, implementation and contracts using mutation analysis (Section 5.5.3) in three iterative steps. First, an initial test suite was analyzed repeatedly using mutation analysis until an acceptable mutation score was achieved. Second, the optimized test suite was used to test the implementation of the transformation and fix errors. If the final model transformation implementation after fixing errors differs significantly from the original implementation, mutation analysis was repeated since different mutants can be generated. Finally, the accuracy of the transformation's contracts as an embedded oracle for the test suite was evaluated using mutation analysis to assess the percentage of mutants detected by running the contracts. If a mutant was killed by a test model but was not killed by any contract, then a contract had to be added to reflect an accurate oracle function of the test suite. The study demonstrated their approach on a simple model transformation example. It was found that successive iterations of the approach increased the mutation score of the contracts to detect up to 90% of the mutants detected by the test suite.

Gogolla and Vallecillo [37] proposed a framework for testing model transformations based on a generalized type of contracts called *tracts*. A tract defines a set of constraints (source tract constraints, target tract constraints, source-target tract constraints) and a tract test suite. Source tract constraints are constraints on input models; target tract constraints are constraints on output models that must be satisfied together with the target metamodel constraints; source-target tract constraints are constraints on relationships between input models and their corresponding output models; the tract test suite is a test suite built to satisfy the source tract constraints and the source metamodel constraints. OCL was used to specify the tract constraints. The context of the tract constraints was a *tract class* that contained functions and attributes used to specify the tract constraints. A framework was implemented as a proof of concept and was used to verify an ATL transformation. The framework integrates different tools into UML-based

Specification Environment (USE). USE was used to define the tract class and tract constraints. A Snapshot Sequence Language (ASSL) was used to implement a program that automates the generation of a test suite that covers the source metamodel constraints and the source tract constraints. The ASSL program was specified and executed within USE. The generated test suite was transformed by the transformation of interest, and the output models were verified with respect to the tract constraints and the target metamodel constraints. The paper discussed the advantages of using tracts in model transformation testing. However, no case study was carried out to evaluate the framework and report on the results.

Le Traon *et al.* [48] used contracts to improve the *vigilance* and *diagnosability* of a system. Vigilance is the probability that the contracts dynamically detect errors caused by faulty statements in a model transformation. Diagnosability is the effort needed to locate a fault once it has been detected by a contract. Although the study focused on improving the vigilance and diagnosability of systems captured as models with OCL constraints, the approach can be leveraged for model transformations with contracts. A systems vigilance was expressed as a function of the *isolated* and *local* vigilance of its constituent components and the probability that this specific component causes a failure in the system. Similarly, a system's diagnosability was expressed as a function of two attributes: the probability that a faulty statement in a set of statements bounded by two consecutive contracts is detected by any contract that comes after the fault and the diagnosis scope. Three case studies were conducted in Eiffel, an object-oriented language that supports contracts. The case studies proved that a systems vigilance and diagnosability improved significantly with the addition of contracts.

6 Discussion

In this section we discuss the surveyed studies from different perspectives. First, we collectively discuss the surveyed studies with respect to the three dimensions, mentioned in Section 1. Then, we independently discuss each of the three dimensions for the surveyed studies and any lessons learnt from them. We also discuss relations between the three dimensions, and how can such relations be useful to developers and researchers.

6.1 Overview of the Surveyed Studies with Respect to the Three Dimensions

For each surveyed study, we show what analysis technique was used from the proposed taxonomy (first dimension), the property analyzed (second dimension) and the formalization used to specify the model transformation of interest (third dimension). Table 2 summarizes this information for the surveyed static analysis techniques. Table 3 summarizes this information for the surveyed dynamic analysis techniques.

Analysis Technique	**Property**	**Formalization**	**Surveyed Studies**
Type I Formal Methods	Type Consistency and Semantical Properties	Algebraic Formalization	[79]
	Semantics Preservation	Alloy	[54]
	Confluence	Graph Rewriting Systems	[67], [31], [26], [21], [7]
	Termination	Petri Nets	[81]
	Termination	Graph Rewriting Systems	[68], [50], [7], [19], [20], [31], [26], [10]
Type II Formal Methods	Assertions	First-Order Logic	[8]
	Property Preservation	TGGs	[36]
	Property Preservation	Graph Rewriting Systems	[15]
	Preservation of Syntactic Relations	Graph Rewriting Systems	[53]

Table 2: Projection of the three dimensions for static analysis techniques

Analysis Technique	Property	Formalization	Surveyed Studies
Type III Formal Methods	Structural Correspondence	Graph Rewriting Systems	[62]
	Semantics Preservation	Graph Rewriting Systems	[9]
	Type Consistency and Multi-View Consistency	Graph Rewriting Systems	[16]
	Type Consistency and Multi-View Consistency	Alloy	[4]
	Type Consistency and Multi-View Consistency	PVS and Eiffel	[65]
Model Checking	Invariants	Petri Nets	[44]
	Invariants, Safety and Liveness Properties	Maude	[18], [74]
	Invariants, Safety and Liveness Properties	Graph Rewriting Systems	[70], [72], [71], [73]
	Bisimilarity	Graph Rewriting Systems	[61]
	Semantics Preservation	Java	[1]
DSE	Design Options	Graph Rewriting Systems and Petri Nets	[39]
	Non-functional Requirements	QVT Operational Language	[29]
	Design Options	Prolog	[76]
Instrumentation	Faults	XSL Language	[27]
Testing	Faults	Declarative/ Imperative Languages	[38], [5], [33], [35], [32], [84], [63], [55], [13], [14], [46], [56], [47], [59], [28], [60], [48], [43], [52], [51], [49], [24], [23], [37], [11], [12]

Table 3: Projection of the three dimensions for dynamic analysis techniques

6.2 First Dimension: The Analysis Technique Used

Based on Table 2 and Table 3 several conclusions can be drawn with respect to the analysis techniques used.

Testing has been heavily investigated, followed by Type I formal methods. The interest in testing can be mainly attributed to the fact that many of the well-established ideas of source code testing were adapted to model transformation testing (e.g., white-box and black-box testing, and mutation analysis). However, we noticed that testing model transformations has been mainly researched for imperative and declarative languages and not for other formalizations such as graph rewriting systems. Type I formal methods have also been heavily investigated due to the many studies that proposed sufficient termination criteria.

On the other hand, we found that instrumentation was the least analysis technique investigated, despite its ease of implementation (as was demonstrated in [27]) and despite the existence of many studies for instrumenting source code. Thus, instrumentation of model transformations presents potential for further research. Its minimal need for a rigorous mathematical background and its usefulness for debugging model transformations are bound to attract more studies to investigate instrumentation.

We also found that for certain classes in the proposed taxonomy, all techniques in the same class suffer from common disadvantages. For example, all Type I formal methods require a rigorous mathematical background which not all researchers have, and are costly to implement. Thus, despite their usefulness in guaranteeing properties for any model transformation when run on any input model, they cannot be easily used. Model checking can not guarantee a property for transformations with a large state space, which are common for real world model transformations. Similarly, DSE can not guarantee optimal design options or optimal values of quality metrics for transformations with a large state space. For testing, many studies proposed different white-box and black-box adequacy criteria. However, to the best of our knowledge, no study evaluated the usefulness of different adequacy criteria for uncovering different fault models. Such an evaluation is crucial for testers to help them choose an adequacy criteria based on the expected faults.

6.2.1 A Comparison Between the Three Types of Formal Methods

We can not compare different classes in the proposed taxonomy with each other since each class is best suited for analyzing different properties and for use with different formalizations. However, we provide a comparison between the three types of formal methods due to their common use of formal methods for model transformation analysis. Type I formal methods (Section 4.1.1), Type II formal methods (Section 4.1.2) and Type III formal methods (Section 5.1) represent a

continuous spectrum in the ease of implementation, generalization and required automation.

Type I formal methods are the most challenging to implement since they require a rigorous mathematical background. However, Type I formal methods are the most generalizable since they can be used to analyze certain properties for any model transformation when run on any input model based on some mathematical proof. Since Type I formal methods are implemented once and reused for many model transformations, they require little automation. For example, studies that propose sufficient termination criteria ([54, 31, 26, 19, 81, 67, 79, 7, 20, 50]) or studies that propose languages that preserve termination for any model transformation ([10]) usually demonstrate a hand-written mathematical proof for their criteria and languages. Nevertheless, some studies still automate their approaches using existing tools such as theorem provers (e.g., [79]).

Type II formal methods and Type III formal methods are easier to implement than Type I formal methods since they attempt to analyze properties for a certain transformation (Type II formal methods) or a certain transformation execution (Type III formal methods) and are not intended for any transformation. More specifically, Type III formal methods are considered *light-weight* since they do not analyze a transformation per se; they analyze a transformation execution. However, since they are not intended for any transformation, they are less generalizable than Type I formal methods. Such methods can be used repetitively to analyze different transformations (Type II formal methods) or different executions of the same transformation (Type III formal methods) and thus, they require automation to be used effectively. For example, we surveyed studies that automate Type II formal methods using model transformation checkers [53]. We also surveyed studies that automate Type III formal methods using constraint solvers such as Alloy [4].

6.3 Second Dimension: Property Analyzed

Model transformation faults have been heavily investigated followed by termination. All the other properties listed in the second column of Table 2 and Table 3 have been almost equally investigated. This highlights the need for further research in analyzing such properties.

One major issue we noticed is that different studies have different definitions for some properties. For example, studies analyzing semantics-preservation had different definitions for the property. Some used conformance to the target metamodel constraints as a definition for semantics-preservation, while other studies statically analyzed input and output models of a model transformation to prove semantics-preservation. A standardized definition of properties with broad definitions is needed to allow comparing different approaches analyzing the same property.

6.4 Third Dimension: The Formalization Used to Specify the Model Transformation of Interest

Although imperative and declarative languages have been heavily used for testing model transformations to analyze faults, graph rewriting systems have been used to analyze a broader range of properties using different analysis techniques. The high interest in graph rewriting systems lends itself to the fact that they are a graphical formalization and an intuitive way for capturing transformations.

However, the former conclusion highlights the need for more studies investigating the analysis of different properties for model transformations implemented in imperative and declarative languages due to the widespread use of such languages in the real world. This will help developers analyze existent model transformations implemented in imperative and declarative languages without having to transform them to an intermediate formalization.

6.5 Relations Between the Three Dimensions

Table 2 and Table 3 can be used to deduce binary relations between the three dimensions. Between the used analysis technique and the analyzed property, Type I formal methods have been heavily used to analyze termination, model checking has been heavily used to analyze invariants and LTL properties, and testing has been heavily used to analyze faults.

Between the used formalization and the analyzed property, graph rewriting systems have had a widespread use to analyze many properties, with termination being the most well-researched. Further, imperative and declarative languages have had a widespread use to analyze faults.

Between the used analysis technique and the used formalization, instrumentation and testing have been researched only for transformations implemented in imperative and declarative languages. While the rest of the analysis techniques have been researched for model transformations implemented in different formalizations, with graph rewriting systems being the most widely used formalization.

Such binary relations can be used by researchers to determine what other combinations require further investigations. The binary relations can also be used by developers as a reference to determine, for example, what analysis technique to use if they would like to analyze a certain property.

Table 2 and Table 3 can be used as a reference such that given two dimensions (e.g., a transformation implemented in some formalization and an available analysis technique), what property (i.e., third dimension) can be analyzed. Further, a researcher interested in one of the three dimensions (e.g., analyzing a certain

property) can use the two tables to determine the combination of the other two dimensions to use (e.g., the formalization and analysis techniques to use to analyze the property of interest).

7 Conclusion

In this paper we proposed a taxonomy for model transformation analysis techniques (Section 3), and we grouped the techniques proposed in the literature according to the proposed taxonomy (Section 4 and Section 5). The techniques surveyed ranged from formal methods (Section 4.1) to light-weight testing (Section 5.5). We also investigated our taxonomy from the perspective of three dimensions (Section 6): the analysis technique used, the property analyzed and the formalization used. The relation between the three dimensions and how such a relation can be useful was also discussed.

The proposed taxonomy extends a former taxonomy we proposed in [3], with additions and minor changes. In this paper, we included testing, instrumentation and design space exploration (DSE) to our taxonomy, which were not surveyed in [3]. Further, in this paper, we consider model checking as a separate class of analysis techniques, and not a subclass of formal methods as we did in [3]. This is mainly due to the existence of a large base of studies on model checking for transformations specified in different formalizations. We also introduced in this paper another level of classification in our taxonomy, static analysis techniques and dynamic analysis techniques, which were not used in [3]. Accordingly, all the classes of analysis techniques were grouped as static or dynamic analysis techniques.

Due to the numerous studies discussing model transformation analysis techniques, this work surveyed only a sample of the studies in the literature that fall under each class of techniques in the proposed taxonomy. More work is needed to make the survey a complete one.

References

[1] L. Ab Rahim and J. Whittle. Verifying Semantic Conformance of State Machine-to-Java Code Generators. *Proceedings of the 13th International conference on Model driven engineering languages and systems (MODELS)*, pages 166–180, 2010.

[2] A. Agrawal, G. Karsai, and Á. Lédeczi. An End-to-End Domain-Driven Software Development Framework. In *Companion of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 8–15. ACM, 2003.

[3] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J.R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. *Verification and validation Of model Transformations (VOLT) Workshop, colocated with ICST*, pages 915–922, 2012.

[4] K. Anastasakis, B. Bordbar, and J.M. Küster. Analysis of Model Transformations via Alloy. *Proceedings of the 4th Model-Driven Engineering, Verification and Validation (MoDeVVa) workshop*, pages 47–56, 2007.

[5] A. Andrews, R. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[6] B.K. Appukuttan, T. Clark, L. Tratt, S. Reddy, R. Venkatesh, P. Sammut, A. Evans, and J. Willans. Revised Submission for MOF 2.0 Query/Views/-Transformations RFP. 2003.

[7] U. Assmann. Graph Rewrite Systems for Program Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):583–637, 2000.

[8] M. Asztalos, L. Lengyel, and T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 15–24, Paris, France, 2010. IEEE.

[9] T. Baar and S. Marković. A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. In *Proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of Systems Informatics (PSI)*, pages 70–83, Akademgorodok, Novosibirsk, Russia, 2007.

[10] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Proceedings of the Third international conference on Software language engineering (SLE)*, pages 296–305, Eindhoven, The Netherlands, 2011. Springer.

[11] B. Baudry, T. Dinh-Trong, J.M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, Bilbao, Spain, 2006.

[12] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.M. Mottu. Barriers to Systematic Model Transformation Testing. *Communications of the ACM*, 53(6):139–143, 2010.

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

[14] E. Bauer and J. M. Küster. Combining Specification-Based and Code-Based Coverage for Model Transformation Chains. *Proceedings of the 4th international conference on Theory and practice of model transformations (ICMT)*, pages 78–92, 2011.

[15] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proceedings of the 28th international conference on Software engineering (ICSE)*, pages 72–81. ACM, 2006.

[16] B. Becker, L. Lambers, J. Dyck, S. Birth, and H. Giese. Iterative Development of Consistency-Preserving Rule-Based Refactorings. *Proceedings of the 4th international conference on Theory and practice of model transformations (ICMT)*, pages 123–137, 2011.

[17] D. Blostein and A. Schürr. Computing with graphs and graph transformations. *Software: Practice and Experience*, 29(3):197–217, 1999.

[18] A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In *12th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 18–33, York, UK, 2009. Springer.

[19] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Termination of High-Level Replacement Units with Application to Model Transformation. *Electronic Notes in Theoretical Computer Science*, 127(4):71–86, 2005.

[20] P. Bottoni and F. Parisi-Presicce. A Termination Criterion for Graph Transformations with Negative Application Conditions. *International Colloquium on Graph and Model Transformation (GraMoT10), Electronic Communications of the EASST*, 30(0), 2010.

[21] P. Bottoni, G. Taentzer, and A. Schurr. Efficient Parsing of Visual Languages Based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proceedings of IEEE International Symposium on Visual Languages*, pages 59–60. IEEE, 2000.

[22] A. Bouhoula, J.P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science - Trees in algebra and programming*, 236(1-2):35–132, 2000.

[23] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. Model Transformation Contracts and their Definition in UML and OCL. Technical Report 2004-08, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2004.

[24] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. In *Workshop of OCL and Model Driven Engineering, in UML 2004*, volume 12, pages 69–83, 2004.

[25] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and C.L. Talcott. All About Maude–A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of LNCS. *Springer*, 4:50–88, 2007.

[26] J. de Lara and G. Taentzer. Automated Model Transformation and its Validation Using AToM3 and AGG. *Diagrammatic Representation and Inference (Diagrams)*, pages 182–198, 2004.

[27] P. Dhoolia, S. Mani, V.S. Sinha, and S. Sinha. Debugging Model-Transformation Failures Using Dynamic Tainting. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, pages 26–51, Maribor, Slovenia, 2010. Springer.

[28] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry, and F. Fleury. A Taxonomy of Faults for UML Designs. In *In Proceedings of Model Design and Validation (MoDeVa) workshop associated to MoDELS'05*, Montego Bay, Jamaica, 2005.

[29] M.L. Drago, C. Ghezzi, and R. Mirandola. Towards Quality Driven Exploration of Model Transformation Spaces. *Proceedings of the 14th international conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 2–16, 2011.

[30] G.T. Eakman. Verification of Platform Independent Models. In *UML Workshop: Model Driven Architecture in the Specification, Implementation and validation of Object-oriented Embedded Systems, San Francisco, California (SIVOES-MDA'03)*, San Francisco, California, USA, 2003.

[31] H. Ehrig, K. Ehrig, J. De Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination Criteria for Model Transformation. In *In Proc. Of 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 49–63, Edinburgh, UK, 2005. Springer.

[32] F. Fleurey, B. Baudry, P.A. Muller, and Y.L. Traon. Qualifying Input Test Data for Model Transformations. *Software and Systems Modeling (SoSyM)*, 8(2):185–203, 2009.

[33] F. Fleurey, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *First International Workshop on Model, Design and Validation (MoDeVa)*, pages 29–40. IEEE, 2004.

[34] The Eclipse Foundation. ATL Transformations Website. *http://www.eclipse.org/m2m/atl/atlTransformations/*, 2012.

[35] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test Adequacy Assessment for UML Design Model Testing. In *In Proc. Of the 14th Intl. Symposium on Software Reliability Engineering (ISSRE)*, pages 332–343. IEEE, 2003.

[36] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In *Proc. of the 3rd International Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA), Genova, Italy*, pages 78–93. Citeseer, 2006.

[37] M. Gogolla and A. Vallecillo. Tractable Model Transformation Testing. In *In Proceedings of the Seventh European Conference on Models Foundations and Applications (ECMFA)*, pages 221–236, Birmingham, UK, 2011. Springer.

[38] S. Haschemi. Model Transformations to Satisfy All-Configurations-Transitions on Statecharts. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation (MODEVVA)*, 2009.

[39] A. Hegedus, A. Horváth, I. Ráth, and D. Varró. A Model-Driven Framework for Guided Design Space Exploration. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 173–182, Lawrence, Kansas, USA, 2011. IEEE.

[40] Visual Paradigm International. Visual Paradigm for UML, http://www.visual-paradigm.com/, 2010.

[41] D. Jackson. Alloy Analyzer Website. *URL http://alloy.mit.edu*, 2002.

[42] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2011.

[43] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20. ACM, 2006.

[44] B. König and V. Kozioura. Augur 2A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 211:201–210, 2008.

[45] J.M. Küster. Systematic Validation of Model Transformations. *WiSME'04 (associated to UML'04)*, 2004.

[46] J.M. Küster and M. Abd-El-Razik. Validation of Model Transformations – First Experiences using a White Box Approach. In *In Proceedings of Model Design and Validation (MODEVA'06) Workshop, Associated to MODELS'06*, pages 62–77. Springer, 2006.

[47] R. Lämmel. Grammar Testing. *Proceedings of the 4th Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 201–216, 2001.

[48] Y. Le Traon, B. Baudry, and J.M. Jézéquel. Design by Contract to Improve Software Vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006.

[49] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[50] T. Levendovszky, U. Prange, and H. Ehrig. Termination Criteria for DPO Transformations with Injective Matches. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 175(4):87–100, 2007.

[51] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, volume 108, pages 219–236, Vancouver, Canada, 2004.

[52] Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations. *Model-driven software development*, pages 219–236, 2005.

[53] L. Lúcio, B. Barroca, and V. Amaral. A Technique for Automatic Validation of Model Transformations. *Model Driven Engineering Languages and Systems (MODELS)*, pages 136–150, 2010.

[54] T. Massoni, R. Gheyi, and P. Borba. Formal Refactoring for UML Class Diagrams. In *19th Brazilian Symposium on Software Engineering (SBES), Uberlândia, Brazil*, pages 152–167, 2005.

[55] J. A. Mc Quillan and J. F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, Department of Computer Science, Maynooth, Co. Kildare, Ireland, 2005.

[56] J.A. McQuillan and J.F. Power. White-Box Coverage Criteria for Model Transformations. In *1st International Workshop on Model Transformation with ATL*, Nantes, France, 2009.

[57] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[58] B. Meyer. EIFFEL: The Language and Environment. *PRENTICE HALL PRESS, 200 OLD TAPPAN ROAD, OLD TAPPAN, NJ 07675(USA), 300*, 1991.

[59] J.M. Mottu, B. Baudry, and Y. Le Traon. Mutation Analysis Testing for Model Transformations. In *Second European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, pages 376–390, Bilbao, Spain, 2006. Springer.

[60] J.M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *Model Driven Engineering Languages and Systems (MODELS)*, pages 589–603, Genova, Italy, 2006. Springer.

[61] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. *Electronic Notes in Theoretical Computer Science*, 211:191–200, 2008.

[62] A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. *Electronic Communications of the European Association of Software Science and Technology (EASST)*, 10(0), 2008.

[63] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. *In Proceedings of the 2nd international conference on The unified modeling language (UML)*, pages 416–429, 1999.

[64] S. Owre, N. Shankar, JM Rushby, and DWJ Stringer-Calvert. PVS Language Reference (Version 2.3), 1999.

[65] R.F. Paige, P.J. Brooke, and J.S. Ostroff. Metamodel-Based Model Conformance and Multiview Consistency Checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):1–48, 2007.

[66] J.L. Peterson. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[67] D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. *Term graph rewriting: theory and practice*, 15:201–214, 1993.

[68] D. Plump. Termination of Graph Rewriting is Undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[69] IBM Rational. Rhapsody System Designer, http://www-01.ibm.com/software/awdtools/rhapsody/.

[70] A. Rensink. Towards Model Checking Graph Grammars. In *3rd Workshop on Automated Verification of Critical Systems (AVoCS), Technical Report DSSE–TR–2003–2*, pages 150–160, University of Southampton, 2003. Citeseer.

[71] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 479–485, 2004.

[72] A. Rensink. Explicit State Model Checking for Graph Grammars. *Concurrency, Graphs and Models*, pages 114–132, 2008.

[73] A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. *International Conference on Graph Transformations (ICGT)*, pages 219–222, 2004.

[74] J.E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. *Software Language Engineering*, pages 54–73, 2009.

[75] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 2003.

[76] B. Schätz, F. Hölzl, and T. Lundkvist. Design-Space Exploration Through Constraint-Based Model-Transformation. In *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 173–182. IEEE, 2010.

[77] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[78] R. Soley and the OMG Staff. Model Driven Architecture. *OMG Document*, 2000.

[79] K. Stenzel, N. Moebius, and W. Reif. Formal Verification of QVT Transformations for Code Generation. *Model Driven Engineering Languages and Systems (MODELS)*, pages 533–547, 2011.

[80] G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. *Applications of Graph Transformations with Industrial Relevance*, pages 333–341, 2000.

[81] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. *International Conference on Graph Transformations (ICGT)*, pages 260–274, 2006.

[82] K. Walden and J.M. Nerson. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, 1995.

[83] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., second edition edition, 2003.

[84] Y. Wu, M.H. Chen, and J. Offutt. UML-Based Integration Testing for Component-Based Software. *In Proceedings of the Second International Conference on COTS-Based Software Systems (ICCBSS)*, pages 251–260, 2003.