Model Transformation Testing: The State of the Art

Gehan M. K. Selim
School of Computing,
Queen's University
Kingston, ON, Canada
gehan@cs.queensu.ca

James R. Cordy
School of Computing,
Queen's University
Kingston, ON, Canada
cordy@cs.queensu.ca

Juergen Dingel
School of Computing,
Queen's University
Kingston, ON, Canada
dingel@cs.queensu.ca

ABSTRACT

Model Driven Development (MDD) is a software engineering approach in which models constitute the basic units of software development. A key part of MDD is the notion of automated model transformation, in which models are stepwise refined into more detailed models, and eventually into code. The correctness of transformations is essential to the success of MDD, and while much research has concentrated on formal verification, testing remains the most efficient method of validation. Transformation testing is however different from testing code, and presents new challenges. In this paper, we survey the model transformation testing phases and the approaches proposed in the literature for each phase.

Keywords

Model Driven Development, Model Transformation Testing, Test Case Generation, Mutation Analysis, Contracts.

1. INTRODUCTION

In MDD, software models or abstractions are first class artifacts, with development beginning with high-level models that are successively refined and transformed into detailed models and finally into code. The correctness of transformations is therefore essential to the success of MDD.

A model transformation is a program that maps input models conforming to a source metamodel to output models conforming to a target metamodel. To reason about the correctness of transformations it is important to ensure that they satisfy their expected properties. While formal methods have been applied to this problem, they are heavyweight.

By contrast, testing is the most popular quality assurance technique for code, since it is lightweight, automatable and can easily uncover bugs, although it does not guarantee their absence. Testing is also a natural candidate for lightweight analysis of transformations' correctness. Transformation testing executes a transformation on input models and validates that the actual output matches the expected output [13].

Several studies have discussed the challenges facing transformation testing [2, 3, 10, 15, 20]. Despite these challenges and despite the fact that testing does not fully verify the correctness of a transformation, it has been gaining increasing interest for several reasons. The major advantage of testing is its usefulness in uncovering bugs while maintaining a low computational complexity [12]. Other advantages include the ease of performing testing activities, the feasibility of analyzing the transformation in its target environment and the ease of automating most of the testing activities [20].

In this paper, we propose four transformation testing phases and survey the state of the art in the proposed phases with the goal of identifying topics that require further research.

We differentiate between a transformation's implementation and specification. A transformation implementation consists of the rules that carry out the mapping between the source and target metamodels. [1] By contrast, a transformation specification includes the source and target metamodels (and their constraints), and the transformation contracts. A contract is composed of three sets of constraints [7]: (1) constraints on input models, (2) constraints on output models, and (3) constraints on relationships that must be maintained between input model elements and output model elements.

Several studies have proposed taxonomies of transformation contracts. Baudry *et al.* [2] define three levels of contracts: transformation contracts, subtransformation contracts and output contracts. (Sub)transformation contracts include pre-/post-conditions of (sub)transformations and their invariants. Output contracts are expected properties of output models. Mottu *et al.* [24] categorized contracts as either syntactic or semantic. Syntactic contracts ensure that the transformation can run without errors. Semantic contracts are context-dependent and can be subdivided into preconditions on input models, postconditions on output models, and postconditions linking input and output models.

The remainder of this paper is organized as follows: Section 2 overviews the proposed model transformation testing phases; Sections 3, 4, and 5 explain the testing phases in more detail and survey the state of the art related to each phase; and finally Section 6 concludes the paper.

[1] We use the notion of a *model transformation* and a *model transformation implementation* interchangeably.

2. PHASES OF MODEL TRANSFORMATION TESTING

We break down the model transformation testing process into four phases, inspired by those defined by Baudry *et al.* [3] with minor changes. The first phase, *test case generation*, involves generating a *test suite* or a set of test models conforming to the source metamodel for testing the transformation of interest. 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 [22] (Eqn. 1).

$$Coverage = \frac{|AdequacyCriteriaSatisfiedByTestSuite|}{|AdequacyCriteria|} * 100\% \quad (1)$$

The second phase is assessing the generated test suite. A test suite that has a positive assessment is more likely to expose faults in an incorrect transformation. A test suite that has a negative assessment can be improved by adding relevant test models to the test suite.

The third phase is building the *oracle function*. The oracle function is the function that compares the actual output of a transformation with the expected output to evaluate the correctness of the transformation [10].

The fourth phase involves running the transformation on the generated test suite and evaluating the actual outputs using the oracle function. For each model in the test suite, if the oracle function detects a discrepancy between its corresponding actual and expected outputs, then the tester can analyze the transformation and fix any faults accordingly.

In the following sections, we explain the first three phases and we survey studies related to them. The fourth phase is a straightforward process given that the test suite and the oracle function are built correctly.

3. PHASE 1: TEST CASE GENERATION

Test case generation involves defining test adequacy criteria and building a test suite that achieves coverage of the adequacy criteria. Defining test adequacy criteria, and hence test case generation, can follow a *black-box*, *grey-box* or *white-box* approach. A black-box approach assumes that the transformation implementation is not available and builds a test suite based on the transformation specification (i.e., source metamodel or contracts). A grey-box approach assumes that the transformation implementation is partially available and builds a test suite based on the accessible parts of the transformation implementation [13]. A white-box approach assumes that the full transformation implementation is available and builds a test suite based on the transformation implementation. We discuss criteria proposed for black- / white-box test case generation in more detail. We do not discuss grey-box test case generation, since it has been rarely investigated in the literature. 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 transformation implementation.

3.1 Black-Box Test Case Generation Based on Metamodel Coverage

Different adequacy criteria have been proposed in the literature to achieve coverage of the different source metamodels of model transformations.[2] For example, if a transformation manipulates class diagrams, then adequacy criteria for class diagrams can be leveraged for black-box testing.

McQuillan and Power [22] surveyed the black-box adequacy criteria proposed in the literature for one structural model (class diagrams) and five behavioural models (sequence diagrams, communication diagrams, state machine diagrams, activity diagrams and use case diagrams). None of the surveyed studies discussed adequacy criteria for deployment diagrams, component diagrams, composite structure diagrams, interaction overview diagrams or timing diagrams. The study also reviewed how the different criteria were evaluated and compared with each other. The paper concluded that little work has been done on evaluating the effectiveness of the criteria in detecting faults and on comparing the criteria in terms of the coverage they provide.

In this section, we summarize adequacy criteria proposed for class diagrams since they are the only structural models with criteria proposed in the literature. Due to space limitations, we summarize the adequacy criteria for only two behavioural models (interaction diagrams and statecharts).

Adequacy Criteria for Class Diagrams: Three criteria have been investigated for class diagrams [1, 10, 11]: 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 the test suite. The GN criterion requires that each subclass gets instantiated. The CA criterion requires that each representative class attribute value gets instantiated.

In the 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* [26] 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 [9].

Some studies [10, 11] propose the notion of a *coverage item*, which is a constraint on the test suite that requires certain combinations of objects, representative CA values and AEM values to be instantiated in the test suite. A test adequacy criterion can then be defined for each coverage item. Fleurey *et al.* [9] also combined classes, representative CA 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 speci-

[2]We survey black-box adequacy criteria for testing models and transformations, since in both cases, criteria are dependent on the input metamodel only.

fying different ways of combining object fragments into a model fragment. A tool was built to implement the proposed criteria and to guide the tester by generating the required model fragments and to point out model fragments that were not covered by the test suite. The tool was found to suggest model fragments that are not feasible, e.g., suggesting a model fragment with zero transitions and one transition in an input state machine.

Adequacy Criteria for Interaction Diagrams: Seven adequacy criteria have been investigated for interaction diagrams [1, 11, 28]: each message on link (EML), all message paths (AMP), collection coverage (Coll), condition coverage (Cond), full predicate coverage (FP), transition coverage, and all content-dependency relationships coverage.

The EML criterion requires that each message on a link connecting two objects gets instantiated in the test suite. The AMP criterion requires that each possible sequence of messages gets instantiated in the test suite. The Coll criterion requires each interaction with collection objects of representative sizes gets instantiated in the test suite. The Cond criterion requires that each condition gets instantiated in the test suite with both *true* and *false*. The FP coverage 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 transition coverage criterion requires that each transition type gets instantiated in the test suite. The all content-dependency relationships coverage criterion is based on extracting data-dependency relationships between system components and requires that each identified relationship gets instantiated in the test suite.

Adequacy Criteria for Statecharts: Six adequacy criteria have been investigated for statecharts [25, 28, 13, 28]: full predicate (FP) coverage, all content-dependency relationships coverage, transition coverage, transition pair coverage, complete sequence coverage, and all-configurations-transitions coverage for statecharts with parallelism.

The FP coverage criterion, the all content-dependency relationships coverage criterion, and the transition coverage criterion are similar to their equivalents for interaction diagrams. The transition pair coverage criterion requires that each pair of adjacent transitions gets instantiated in the test suite. The complete sequence coverage criterion requires that each complete sequence of transitions that makes full use of the system gets instantiated in the test suite. Due to the infinite possible sequences, a domain expert must define a set of sequences that are crucial to be tested. The all-configurations-transitions coverage criterion for statecharts with parallelism requires that all transitions between all state configurations in the reachability tree of a state chart get instantiated. Similarly to class diagrams, coverage items can be created for interaction diagrams and statecharts and test adequacy criteria can be defined accordingly.

3.2 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 model transformations. Fleurey *et al.* [10] proposed constructing an *effective* metamodel composed of only those source metamodel elements that are actually used in the pre-/post-conditions of a transformation. The values of attributes and multiplicities in the effective metamodel can then be partitioned, and the defined partitions can be used to generate coverage items and adequacy criteria. No case study was conducted to evaluate the proposed approach.

Bauer *et al.* [4] propose a *combined specification-based coverage* approach for testing a transformation chain, where contract-based and metamodel-based adequacy criteria were generated from the transformations in the transformation chain. Contract-based criteria were generated that require the execution of each contract by the test suite. Traditional metamodel-based criteria, *e.g.,* the AEM criterion, were used. Using the generated 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 criterion. The quality of the test suite was then measured using the footprints of all the test models to assess the covered criteria, the uncovered 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. A case study was conducted on a commercial 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 transformation chain. Moreover, 19 redundant test models were identified and removed.

Bauer and Küster [5] investigated the relationship between specification-based (black-box) test adequacy criteria used in [4] 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. A positive linear scatter and a correlation coefficient close to one implied that the code implemented its specified behavior. The study used the same transformation chain used in [4] 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.

3.3 White-Box Test Case Generation

Different adequacy criteria have been proposed in the literature to achieve coverage of a model transformation im-

plementation. Fleurey *et al.* [10] proposed using a static type checker to build an *effective* metamodel composed of the source metamodel elements referenced in the transformation implementation. Attributes and multiplicities that constitute the effective metamodel can then be partitioned and the defined partitions can be used to generate coverage items and adequacy criteria. No case study was conducted to evaluate the proposed approach.

Küster and Abd-El-Razik [15] proposed three white-box approaches to test transformations specified as conceptual rules and built using IBM WebSphere Business Modeler. The first 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 parameterized. Thus, several templates were generated from each rule to ensure source metamodel coverage per rule. The second approach was proposed to experiment with output models with constraints. For each constraint on an output model element, a test model that affects the constraint of interest was generated. The third approach generated critical input models that contain overlapping match patterns of rule pairs to test if errors can occur due to the interplay of rules. The study concluded that the third 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 [21] assessed the coverage of ATL rules by profiling their operation during execution. 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 prints out a log file of the executed instructions. To assess rule coverage of ATL transformations, a two phase-approach was proposed. In the first phase, the XML file resulting from compilation of the transformation is processed to extract the available rules and helpers. In the second phase, the transformation was executed using the available test suite. The resulting log file was processed to find out how much of the rules and helpers extracted in the first phase were covered according to three white-box criteria: rule coverage, instruction coverage and decision coverage.

Lämmel [16] proposed a criterion for grammar testing. The criterion can be leveraged for 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 the test suite to trigger each rule in the grammar. The proposed criterion, referred to as context-dependent branch coverage, requires each rule to be triggered 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*. No case study was conducted to evaluate the efficiency of the criterion in detecting faults.

4. PHASE 2: TEST SUITE ASSESSMENT

Many studies use the coverage achieved by a test suite with respect to some adequacy criteria to assess the quality of test suites [1, 10, 11, 9, 28, 25, 13, 22, 4, 5, 15, 21]. Other studies use *mutation analysis* instead [24, 17, 22, 23, 25]. We discussed in Section 2 how to measure the coverage achieved by a test suite with respect to some adequacy criteria. In this section, we discuss mutation analysis in depth.

Mutation analysis [23] is a technique used to evaluate the sensitivity of the test suite to faults in the transformation of interest. Mutation analysis involves applying *mutation operators* to inject faults in the original transformation and generate *mutants*. The injected faults represent *fault models* committed by developers when building transformations. The mutants and the original transformation are then executed using the test suite under assessment. For each mutant, if one test model produces different results for the transformation and the mutant, then the mutant is *killed*. 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. A *mutation score* is computed to evaluate the test suite (Eqn. 2).

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

Mottu *et al.* [23] propose *semantic* mutation operators that 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 mutation operators related to each of the four operations. Using the proposed mutation operators, mutants were generated for a Java transformation and were compared with the mutants generated using a commercial tool, 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- or run-time.

Dinh-Trong *et al.* [8] discussed mutation operators for UML models that can be easily 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. 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 environments. Faults related to behavior result from an incorrect transformation specification that is syntactically correct. Several mutation operators were discussed and classified according to the proposed fault models.

5. PHASE 3: BUILDING THE ORACLE FUNCTION

An oracle function compares the actual output with the expected output to validate the transformation of interest [10]. If the expected output models are available, then the oracle function is a model comparison or a *model differencing* task [19, 14, 20]. However, if the expected output models are not available, then the oracle function validates the transformation's output with respect to predefined output specifications or *contracts* [7, 6, 24, 12, 17].

5.1 Model Comparison as Oracle Functions

Model comparison or differencing has been identified as a major task in transformation testing [14]. Lin *et al.* [20] proposed a framework which was integrated with the transformation engine C-SAW and used model comparison as the oracle function. The transformation language used in C-SAW is ECL, an extension of OCL. In ECL, a transformation can be either a *strategy* or an *aspect*. A strategy specifies the required transformation, while an aspect binds 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 are the paths of the input models and their expected output models, and the strategy specifying the transformation. For each input model, the test constructor generates an aspect. The test engine executes the generated aspects and compares the actual output models with the expected output models. The comparison results are then passed to the test analyzer which visualizes the results using different colors and shapes. A case study was conducted to show how the framework helped detect errors in a transformation example.

However, model comparison or differencing has many dimensions that need to be addressed to be carried out successfully [19]. These dimensions include syntactic/ semantic differencing and visualisation of differences. Thus, in this paper, we do not discuss model differencing any further.

5.2 Contracts as Oracle Functions

Contracts specify expected properties of the transformation's output, and can be used as oracle functions. Many languages for defining transformation contracts have been proposed, e.g., Java Modeling Language (JML) [18] can be used to define contracts for Java transformations. However, OCL [27] has been used in many studies for specifying transformation contracts.

Cariou *et al.* [7] discuss two approaches to specify OCL constraints on relationships between input and output models. In the first approach, OCL expressions that manipulate elements of a single metamodel were specified in the transformation's postcondition. In this approach, 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. Moreover, OCL expressions are simple due to the use of one metamodel for both the source and target. On the other hand, a disadvantage of this approach is that it can only be used when the source and target metamodels are the same since the transformation must be owned by a classifier of one metamodel. Thus, for transformations manipulating different metamodels, a common metamodel needs to be defined and a classifier of the metamodel must own the transformation. 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 easily referenced. The first approach was applied to two transformations [6] to demonstrate the construction of a common metamodel and the choice of the classifier to own the transformation.

In the second approach, OCL expressions that manipulate models as packages were specified in the postcondition. The second approach can be used to define contracts for transformations that manipulate different metamodels. However, disadvantages of the second approach include the need for an OCL extension to define explicit mappings between input and output model elements. Further, OCL navigational expressions can be verbose due to the use of different metamodels. The study applied the second approach to a transformation to demonstrate the definition of an OCL extension and the definition of the contracts.

Gogolla and Vallecillo [12] propose a framework for testing transformations based on a generalized type of contracts called *tracts*. A tract defines a set of OCL 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 and output models; the tract test suite is a test suite built to satisfy the source tract constraints and the source metamodel 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 and was used to verify a transformation. The paper discussed the advantages of using tracts in testing. However, no case study was conducted to evaluate the framework.

Improving Transformation Contracts: Some studies focused on the importance of contracts and the need to improve them. Two approaches were proposed to improve contracts [24, 17]: mutation analysis [24] and mathematical modeling [17]. Both approaches aimed to improve two transformation metrics that reflect the effectiveness of its contracts: *vigilance* and *diagnosability*. Vigilance is the probability that the contracts dynamically detect errors [24, 17]. Diagnosability is the effort needed to locate a fault once it has been detected by a contract [17].

Mottu *et al.* [24] improved the *vigilance* of transformations by improving the consistency between a transformation's test suite, implementation and contracts using mutation analysis in three 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 transformation and fix errors. If the final transformation after fixing errors differs significantly from the original one, mutation analysis was repeated since different mutants can be generated. Finally, the accuracy of the contracts was evaluated using mutation analysis to assess the percentage of mutants detected by 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. The study evaluated their approach on a transformation and were able to improve the contracts' mutation score to detect up to 90% of the mutants detected by the test suite.

Le Traon *et al.* [17] used contracts to improve the *vigilance* and *diagnosability* of a system using mathematical modeling. Although the study focused on systems captured as models with OCL constraints, the approach can be leveraged for transformations with contracts. A system's 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 system failure. 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 and it was proven that a system's vigilance and diagnosability improved significantly with the addition of contracts.

6. CONCLUSION

In this paper, we have reviewed the state of the art in three model transformation testing phases. We discussed different approaches that can be used in each phase, and we surveyed studies related to the different approaches.

Based on our study, we propose requirements that need further research in the three testing phases. For test case generation and test suite assessment, one possible area of future work is relating the two phases by identifying adequacy criteria that are effective in uncovering certain fault models. By relating criteria to fault models, testers can decide on the criteria to use to build their test suites based on the kinds of faults they expect in a transformation. For building the oracle function, if OCL is to be adopted as a standard for defining contracts, then constructs need to be incorporated into OCL to facilitate defining contracts for transformations manipulating different metamodels while avoiding complex navigational expressions and the need for defining explicit mappings between model elements, as done in [7].

7. REFERENCES

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

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

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

[4] E. Bauer, J. Küster, and G. Engels. Test Suite Quality for Model Transformation Chains. *Objects, Models, Components, Patterns*, 2011.

[5] E. Bauer and J. M. Küster. Combining Specification-Based and Code-Based Coverage for Model Transformation Chains. *ICMT*, 2011.

[6] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. Model Transformation Contracts and their Definition in UML and OCL. Technical Report 2004-08, LIFL, 2004.

[7] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. In *OCL and Model Driven Engineering*, volume 12, 2004.

[8] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry, and F. Fleury. A Taxonomy of Faults for UML Designs. In *MoDeVa*, 2005.

[9] F. Fleurey, B. Baudry, P. Muller, and Y. Traon. Qualifying Input Test Data for Model Transformations. *SoSyM*, 8(2), 2009.

[10] F. Fleurey, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *MoDeVa*, 2004.

[11] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test Adequacy Assessment for UML Design Model Testing. In *ISSRE*, 2003.

[12] M. Gogolla and A. Vallecillo. Tractable Model Transformation Testing. In *ECMFA*, 2011.

[13] S. Haschemi. Model Transformations to Satisfy All-Configurations-Transitions on Statecharts. In *MODEVVA*, 2009.

[14] D. Kolovos, R. Paige, and F. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Global Integrated Model Management*, 2006.

[15] J. Küster and M. Abd-El-Razik. Validation of Model Transformations – First Experiences using a White Box Approach. In *MoDeVa*, 2006.

[16] R. Lämmel. Grammar Testing. *FASE*, 2001.

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

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

[19] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *Best Practices for Model-Driven Software Development*, volume 108, 2004.

[20] Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations. *Model-Driven Software Development*, 2005.

[21] J. McQuillan and J. Power. White-Box Coverage Criteria for Model Transformations. In *Model Transformation with ATL*, 2009.

[22] J. A. McQuillan and J. F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, Department of Computer Science, 2005.

[23] J. Mottu, B. Baudry, and Y. Le Traon. Mutation Analysis Testing for Model Transformations. In *ECMDA-FA*, 2006.

[24] J. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *MODELS*, 2006.

[25] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. *UML*, 1999.

[26] T. Ostrand and M. Balcer. The Category-Partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM*, 31(6), 1988.

[27] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting your Models Ready for MDA*. Second edition, 2003.

[28] Y. Wu, M. Chen, and J. Offutt. UML-Based Integration Testing for Component-Based Software. *ICCBSS*, 2003.