TETRA$_{Box}$ – A Generic White-Box Testing Framework for Model Transformations

J. Schönböck
Upper Austrian University of Applied Sciences, Hagenberg, Austria
Email:johannes.schoenboeck@fh-hagenberg.at

G. Kappel, and M. Wimmer
Vienna University of Technology, Austria
Email: {lastname}@big.tuwien.ac.at

A. Kusel, W. Retschitzegger, and W. Schwinger
Johannes Kepler University Linz, Austria
Email: {firstname.lastname}@cis.jku.at

***Abstract*—Model transformations play a vital role in Model-Driven Engineering. Due to their increasing complexity, proper means for ensuring their quality are needed. Although numerous approaches for testing of model transformations have been proposed, their focus is rather on formal verification than on execution-based testing. Additionally, existing approaches do not consider the actual transformation definition, rarely provide hints for debugging in case of an error and are specific to a certain transformation language. Therefore we propose TETRA$_{Box}$ as a language-independent framework for execution-based white-box testing of transformation languages. For this, we base on symbolic execution of model transformations. Additionally, by employing our Pattern-based Modeling Language for Model Transformations (PaMoMo), we are able to provide dedicated failure traces that are aligned to the actual transformation definition as a hint for debugging.**

I. Introduction

Model-Driven Engineering (MDE) proposes an active use of models to conduct the different phases of software development [1]. For the success of MDE, model transformations are crucial, being comparable in role and importance to compilers for high-level programming languages. Given their prominent role in MDE and their use in increasingly complex scenarios, proper *means for testing model transformations* are inevitable to ensure their *correctness*, i.e., the transformation definition conforms to the specified requirements.

Although, various approaches for testing model transformations have been brought forward, their focus is mainly on formal verification (cf. [2] for an overview) and metrics, which are *not* the focus of this paper. In contrast, *execution-based testing* of model transformations, i.e., testing the transformation by executing it in dedicated test runs, is still in its infancy and suffers from three main shortcomings. Current methods for source model generation, being the first prerequisite for automating the test process, rely on the source metamodel (MM) only, but do not consider the transformation definition itself, i.e., a certain *code coverage* may not be achieved, as envisioned by *white-box testing*. Second, test oracles that decide wether the transformation under test behaves correctly, often do not provide the failing elements, i.e., *failure trace*, in the complex graph structures of the test source model, which would be a prerequisite to provide hints for debugging. Finally, current test methods lack genericity in that they are specific to a certain transformation language, being aggravated by the fact that no transformation language has been accepted as the state-of-the-art transformation language - not even the QVT standard [3].

To tackle the aforementioned limitations of existing approaches, we propose a testing framework called TETRA$_{Box}$ (A Generic White-Box TEsting Framework for Model TRAnsformations) that allows for automatic, *execution-based testing* of model transformations in order to improve the correctness of transformation definitions. To enable *white-box-based testing* of model transformations, *automatic generation of test source models on the basis of the transformation definition*, irrespective of the underlying transformation language is enabled based on the notion of symbolic execution trees [4]. Furthermore, by applying our Pattern-based Modeling Language for Model Transformations (PaMoMo) [5], we are able to provide a dedicated *failure trace*, if a test case fails. By aligning this failure trace with the information of the symbolic execution tree we are able to provide a link to the actual source code in order to provide a starting point for debugging. Finally, since we base on symbolic execution, our approach is not restricted to a certain transformation language.

The rest of the paper is structured as follows. Section II motivates the main challenges of execution based testing by means of an example. PaMoMo is shortly introduced in Section III. After introducing our approach for white-box testing in Section IV, Section V explains how PaMoMo and our approach for white-box testing may be employed for fault localization. Section VI presents our prototypical implementation. A comparison to related work is conducted in Section VII before Section VIII concludes.

II. Model Transformation Testing in a Nutshell

The general pattern for model transformations prescribes that source models conforming to a source metamodel (MM) are translated into target models conforming to a target MM (cf. Fig. 1). For specifying transformations, numerous transformation languages exist (cf. [6] for an overview). The resulting transformation definitions are typically executed by a dedicated transformation engine, realizing their operational semantics. To exemplify this, Fig. 1 provides a small excerpt of the well-known `Class2Relational` transformation, being the pendant of the "Hello World" example for model transformations. As a first step, the requirements of this transformation need to be specified (cf. *specification phase* in Fig. 1), which are that all `Packages` should be transformed into `Schemata`,

This work has been partly funded by the FFG Bridge program under grant 832160, FFG FIT-IT Semantic Systems grant P21374-N13 and 825070 as well as WTZ AR 18/2013 and UA 07/2013.

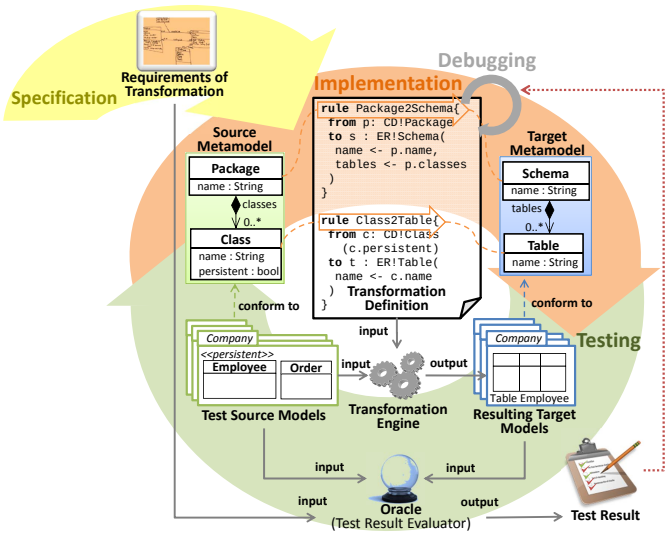

Fig. 1. Model Transformation Testing

whereas persistent `Classes` should result in `Tables`, only. In order to ensure that the requirements are met by the transformation definition (cf. *implementation phase* in Fig. 1), *execution-based testing* is applied (cf. *testing phase* in Fig. 1). For this, in a first step dedicated *test source models*, are required. Due to the complexity of the source models, which are graphs of objects that must conform to the constraints defined in the MM, the manual assembling thereof is tedious and error-prone [7], demanding for an automatic generation. For deciding whether a resulting target model for a provided source model meets the requirements, in a second step, an *oracle* is needed. With this, it is evaluated, if the system under test behaves correctly for a particular execution or not, i.e., it indicates, if a certain test passed or failed [8]. For this, the oracle has to deal again with complex graph structures. In case of an error, the source model elements that caused the test to fail should be presented as a *failure trace*.

After introducing the general steps in testing model transformations, in the following it is explained, how the requirements of a transformation may be formally specified by means of PaMoMo.

III. Specification of Requirements with PaMoMo

Our PaMoMo language (Pattern-based Modeling Language for Model Transformations) [5] provides a visual, declarative, formal specification language to describe, in an implementation-independent way, correctness requirements of transformations as well as their input and output models. PaMoMo specifications express *what* a transformation should do, but not how it should be done, providing an adequate level of abstraction to express transformation requirements.

A PaMoMo contract consists of a set of declarative visual patterns, which can either be *positive* or *negative*. Positive patterns describe necessary conditions to happen (i.e., the pattern is satisfied by a pair of models if these contain certain elements) while negative ones state forbidden situations (i.e., the pattern is satisfied if certain elements are not found). We depict positive patterns in green with its name enclosed in P(...), while negative patterns are shown in red with its name enclosed in N(...). Patterns are made of two compartments containing object graphs, plus a *pattern constraint expression* using the Object Constraint Language (OCL)[1]. The left compartment contains objects typed on the source MM, while the objects to the right are typed on the target MM. Patterns, where both - the source and target compartments - are not empty are called *invariants*, i.e., properties that the transformation specification has to fulfill. Patterns that contain elements in the source compartment only, are called *preconditions*, i.e., properties a source model must fulfill in order to be allowed to participate in a transformation. Finally, patterns that contain elements in the target compartment only, are called *postconditions*, i.e., properties that must be fulfilled by the target model after the transformation is executed. Objects in the source and target compartments may have attributes that can be assigned either a concrete value or a *variable*. A variable can be assigned to several attributes to ensure equality of their values, or can be used in the pattern constraint expression.

In our example in Fig. 1 it is required that every `Package` instance is transformed into an equally named `Schema` element. To ensure this, a positive PaMoMo invariant is employed (cf. Fig. 2). One may see that the left compartment contains elements typed to the `Class` source MM (Package `p`), whereas the right compartment contains elements typed to the `Relational` target MM (Schema *s*). Please note that although in this simple example only one object is contained in every compartment, arbitrary complex graphs are allowed in general. To ensure that packages and schemas are equally named, a variable `X` is introduced, which is assigned to `p.name` as well as `s.name`, stating that both attributes are required to exhibit the same value. For a more detailed discussion of PaMoMo we refer to [5]

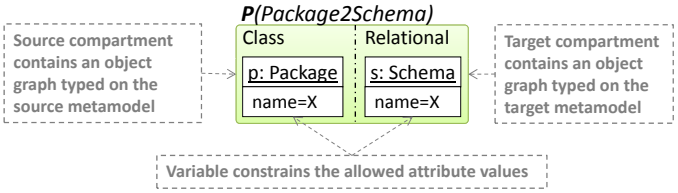

Fig. 2. Requirements Specification using PaMoMo

In order to test an existing transformation against the specified requirements, first a comprehensive set of source models is needed for execution-based testing, as discussed in the following.

IV. Language-Independent White-Box Testing

A main goal of TETRA$_{Box}$ is the provision of adequate test source models with a focus on the language-independent, *white-box-based generation* thereof. For achieving this goal, a *White-box-based Test Source Model Generator* is provided, whose main parts are sketched in Fig. 3 and described in detail in the following.

Metamodel and Coverage Criteria. For the generation of test source models two main sources of information are considered. First, the source MM is incorporated, since a generated source model must conform to the source MM. Second, since we base on a white-box-based testing approach,

[1] http://www.omg.org/spec/OCL/2.3.1

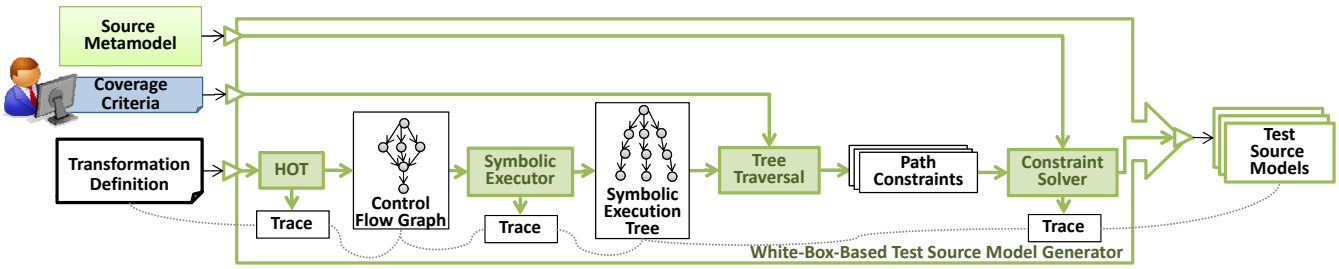

Fig. 3. Language Independent White-Box based Test Source Model Generation

we also include the transformation definition itself in order to generate source models. This is done to ensure that we will cover a certain part of the transformation, i.e., to ensure that each transformation rule is executed once or that each possible path is executed once. In order to allow the configuration of this generation process, so-called coverage criteria are offered, which determine the level of detail, the transformation under test is investigated, e.g., a *rule coverage* ensures that every transformation rule is executed once when the transformation under test is executed with the generated source model.

HOT, Control Flow Graph and Symbolic Execution. To allow the language-independent generation of a set of test source models, the transformation definition needs to be abstracted to a common representation. For this, we base on *control flow graphs* (CFG) [9] since a CFG allows *abstraction by simplification* by omitting parts, which are irrelevant for the source model generation and *abstraction by generalization* by allowing a transformation language-independent representation. For deriving a CFG from a certain transformation definition, a so-called higher-order model transformation (HOT) [10] as shown in Fig. 3 is employed, which takes the transformation definition as source model and produces a corresponding CFG as target model. Such a HOT must be specified once for each transformation language for achieving the goal of *language-independent white-box testing*.

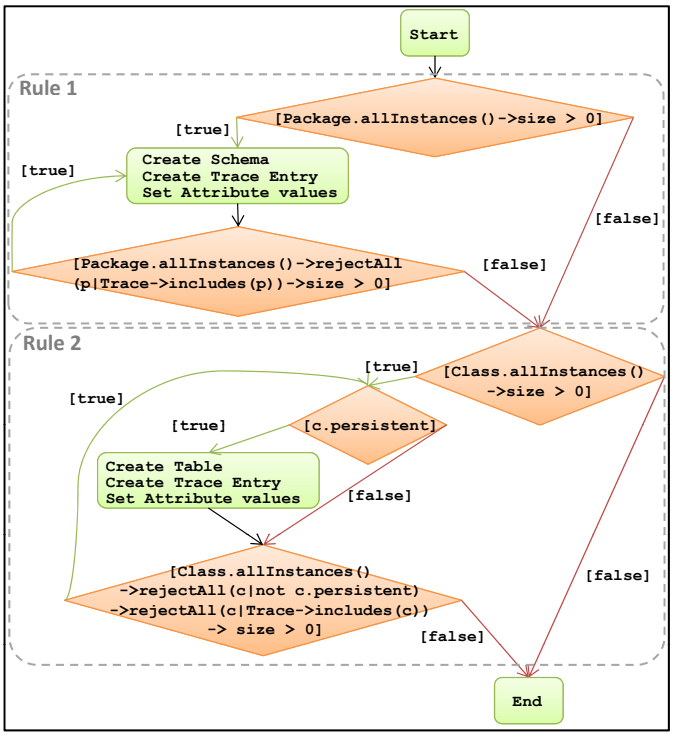

Fig. 4. Control Flow Graph of Example

Example. In order to exemplify the CFG representation, Fig. 4 depicts the CFG for the example introduced in Fig. 1, whereby the setting of links between the objects has been omitted to keep the example simple. The CFG includes a single starting node, representing the entry point of the transformation as well as a single ending node, representing the exit point. In between, each rule is represented as a loop, being responsible for processing the source model elements as well as generating the target model elements. Furthermore, the loop being responsible for transforming classes into tables includes the specified condition as a decision node (cf. `c.persistent`).

In order to explicate all potential execution paths through a CFG, the *Symbolic Executor* component (cf. Fig. 3) converts a CFG into a symbolic execution tree, thereby assigning symbolic values and path constraints to the nodes, following the idea of symbolic execution in software engineering (cf., e.g., [11]).

Tree Traversal and Constraint Solver. In order to cover specific paths of the symbolic execution tree by source models following a certain coverage criterion, various tree traversal algorithms are provided, which collect the path constraints along a certain execution path. The collected path constraints together with the source MM are then used by a dedicated constraint solver, e.g., UMLtoCSP [12] in our case, in order to generate a source model that fulfils all path constraints and that is a valid instance of the source MM. To deal with the well-known *path explosion problem* [13], *hierarchies* are introduced in the CFG - e.g., a whole OCL[2] expression may be either considered as a single node (collapsed form) or a set of nodes (expanded form), resulting in a different number of potential execution paths in the symbolic execution tree.

Example. In order to exemplify this, Fig. 5 depicts the symbolic execution tree for the CFG shown in Fig. 4, whereby at most two iterations have been assumed for existing loops in order to make them finite. Please note that repetitive parts are shown only once in detail for the sake of simplicity. One may see that each node contains certain path constraints on symbolic variables. In order to follow a certain execution path, all path constraints are collected from the start node to a leaf node which must be fulfilled by a certain source model, e.g., in the example "`Package.allInstances()->size > 0`

[2]http://www.omg.org/spec/OCL/2.3.1

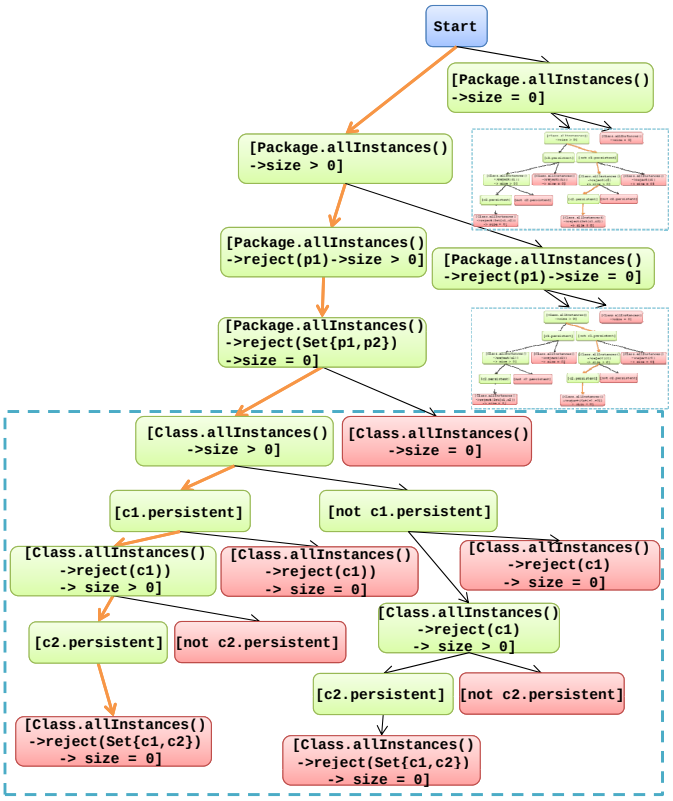

Fig. 5. Symbolic Execution Tree of Example

`and Package.allInstances()-> reject(p1)-> size > 0 and ...`". These constraints are resolved by the constraint solver resulting in a test source model with exactly two `Packages` and two `Classes`, whereby both `Classes` are persistent (cf. test sourced model in Fig. 8).

Adequate Test Source Models. In order to further increase adequacy of the generated test source models, i.e., a test set that is effective in finding errors, the set of generated test source models on basis of the transformation definition is complemented by the specified requirements. This is since the transformation definition may be incomplete with respect to the requirements. Therefore, we remove in on the one hand the MM elements used in the transformation definition form those used in PaMoMo contracts. If the resulting MM is not empty, then the requirements include parts of the MM that have not been covered in the code. Consequently, for those parts dedicated additional test source models need to be generated on a black-box basis. Furthermore,if negative patterns are formulated, e.g., in order to ensure that non-persistent `Class` instances are not translated into according `Table` instances in our example, according test cases should be provided as well. However, according test source models may not be derived form the source code since this information must not be in the transformation definition. On the other hand we also remove the MM elements used in PaMoMo from the MM elements used in the transformation definition. If the set is not empty, the transformation exhibits parts for which no according specification exists. Potentially, this is a hint that certain requirements have not been specified accordingly.

Example. Considering the example depicted in Fig. 6, it

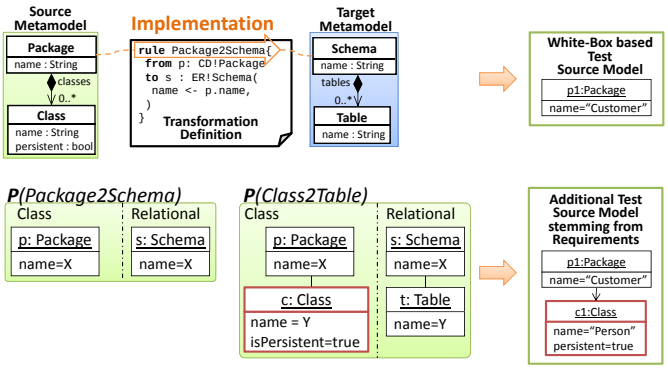

Fig. 6. Adequate Test Source Model Generation

can be seen that we only specified a single rule that transforms `Package` instances into `Table` instances. For this, we could generate a white-box based test source model which contains a Package `p1`. However, with this test source model the PaMoMo invariant `Class2Table` would not raise an error, as it is never executed since there is no `Class` instance available. With the approach described above, we identify that the MM element `Class` has not been used in the transformation definition and we therefore generate an according source model based on the PaMoMo patterns, resulting in an additional test source model. If we now execute the PaMoMo patterns, the invariant `Class2Table` raises an according error.

V. Fault Localization in Model Transformation Testing

After having described our approach for language independent, white-box-based generation of source test models we elaborate in the following on the actual execution of a test case and how we provide means for fault localization in case a test case fails, as depicted in Fig. 7.

Oracle. After executing the transformation under test the resulting target model is obtained. In order to verify, if the specified requirements are fulfilled by the transformation, PaMoMo compares the elements of the test source models with the elements of the resulting target models on the basis of PaMoMo patterns, i.e., PaMoMo acts as an oracle. For this, PaMoMo patterns are compiled and executed by means of QVT-Relations. This is, since QVT-Relations may be executed in a so-called check-only mode, which may be used to identify model elements, which are out of sync. If model elements are out of sync, PaMoMo provides a failure trace.

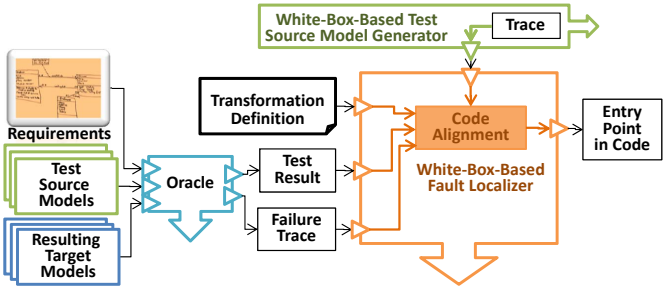

Fig. 7. Overview on Fault Localizer Component

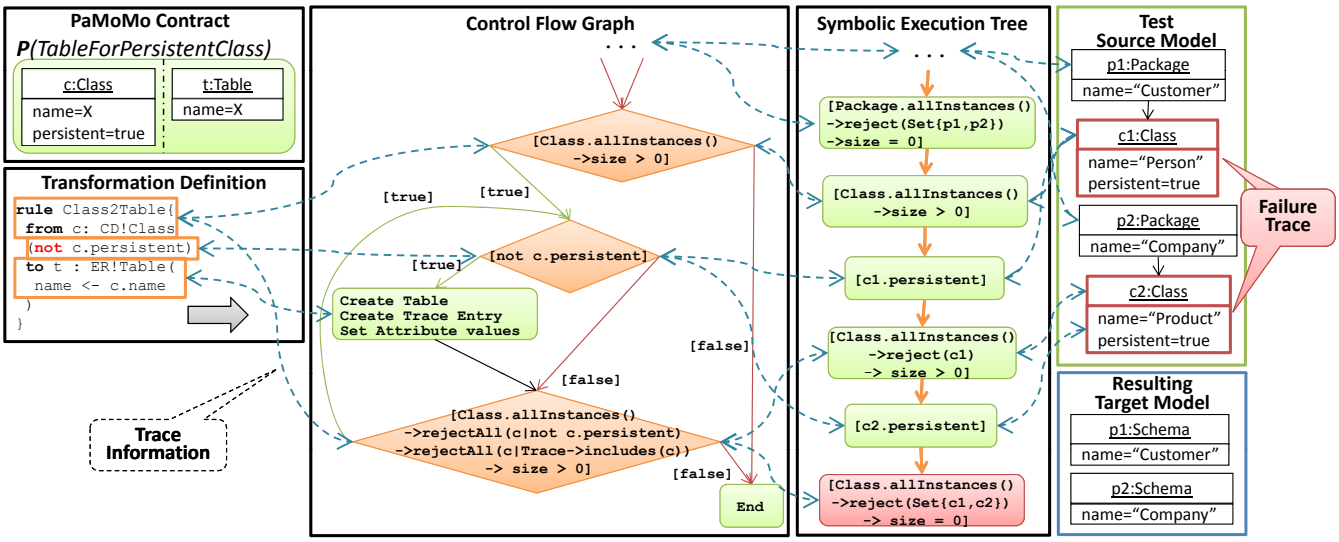

Fig. 8. Overview on Code Alignment

Example. For exemplifying oracles specified by PaMoMo, Fig. 8 shows a simple pattern named `P(TableForPersistentClass)`, which demands that for each persistent `Class` of a source model (cf. left side of pattern) an equally named `Table` must exist in the target model (cf. right side of pattern), being the second requirement of our example in Fig. 1. In the transformation definition shown in Fig. 8, an erroneous condition (cf. `not c.persistent`) has been introduced. Given the PaMoMo pattern `P(TableForPersistentClass)`, the generated test source model would cause the oracle to fail. In our example the oracle would either provide `Class c1` or `Class c2` as dedicated failure trace as highlighted in Fig. 8.

Code Alignment. As an entry point in code for debugging, a precise alignment of the failing source model elements as provided by the oracle to the transformation definition needs to be provided. In order to achieve this goal, we exploit the information provided by the *White-box-based Test Source Model Generator* component, since it produces dedicated trace information between the artifacts, e.g., between the elements of the CFG and the elements of the symbolic execution tree. Since the generated source model objects correspond to path constraints of the symbolic execution tree, the nodes of the symbolic execution tree can be traced back to the nodes of the CFG and finally, the nodes of the CFG have corresponding counterparts in the transformation definition. Consequently, an alignment between failing source model elements provided by the oracle and the responsible transformation definition parts, i.e., *the entry point in code*, is established.

Example. Starting from one of the classes provided by the failure trace, i.e., `Class c1` or `Class c2`, and following the traces backwards, one may end up with the nodes of the transformation definition, which are close to the error, e.g., the rule `Class2Table` and the condition `not c.persistent` itself as may be seen in Fig. 8.

VI. Prototypical Implementation

Fig. 9 provides an overview on the main components of the TETRA$_{Box}$ framework. The TETRA$_{Box}$ framework supports all main phases in testing, being (i) requirements specification, (ii) test source model generation, (iii) test execution, (iv) oracle execution and (v) fault localization. TETRA$_{Box}$ bases on a pluggable architecture on the basis of the Eclipse framework, whereby each component is represented as a separate plugin. Consequently, the TETRA$_{Box}$ framework may be extended, e.g., by existing black-box-based test source model generators. To achieve genericity, i.e., testing of different transformation languages, existing transformation engines might be plugged in. For the specification of requirements we currently provide a graphical, *GMF-based editor* for modeling the PaMoMo patterns, as depicted in Fig. 10. Furthermore, in order to allow for automatic test source model generation, we provide the *White-box-based Test Source Model Generator* component, as described in Section IV. This component may be configured by according coverage criteria. Additionally, we are currently working on including existing black-box-based test source model generators in order to generate further test models. To achieve genericity, i.e., testing of different transformation languages, existing transformation engines might be plugged in and executed within the framework in order to achieve the resulting target models. These are required inputs for the test oracle. Currently, we base on PaMoMo as test oracle. In case of a failing test case, the *White-Box-Based Fault Localizer* component may be activated as described in Section V, which provides an entry point in code and highlights suspicious lines of codes in the transformation definition. Please note that this component requires the *White-box-based Test Source Model Generator* to be used for test source model generation, i.e., dependencies between components may exist.

VII. Related Work

Testing of model transformations has been identified as *the* key challenge in the field already in 2008 at the remarkable workshops on "Model Engineering of Complex Systems"[3] in Dagstuhl and on "Challenges in Model-Driven Software Engineering" in Toulouse at MoDELS [14] as well as in a recent CACM article by Bryent et al. [15] resulting in

[3] http://drops.dagstuhl.de/portals/index.php?semnr=08331

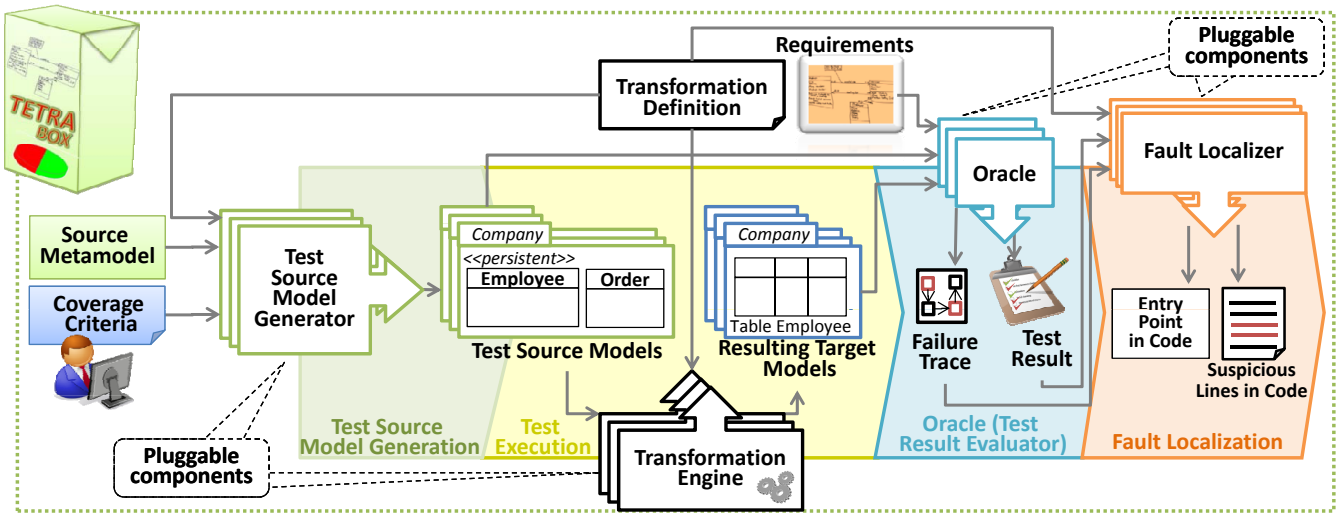

Fig. 9. Overview on TETRA$_{Box}$ Framework

several publications. When surveying the literature focussing on testing of model transformations, one may see that research mainly concentrated on *non-execution-based testing methods*, and thereby mainly on formal verification (cf. [2] for an overview) and metrics. In contrast, less research work has been conducted in the area of *execution-based testing* in general and on white-box testing in particular.

In the following, related work is discussed in detail by targeting relevant research efforts in the area of model transformations as well as useful complementary approaches from the area of testing conventional software with respect to white-box based testing of model transformations and means for fault localization.

A. State-of-the-Art in White-Box Testing of Model Transformations

In the area of white-box based testing of model transformations we first investigate on dedicated approaches for the generation of test source models and second on white-box based coverage criteria that build the basis.

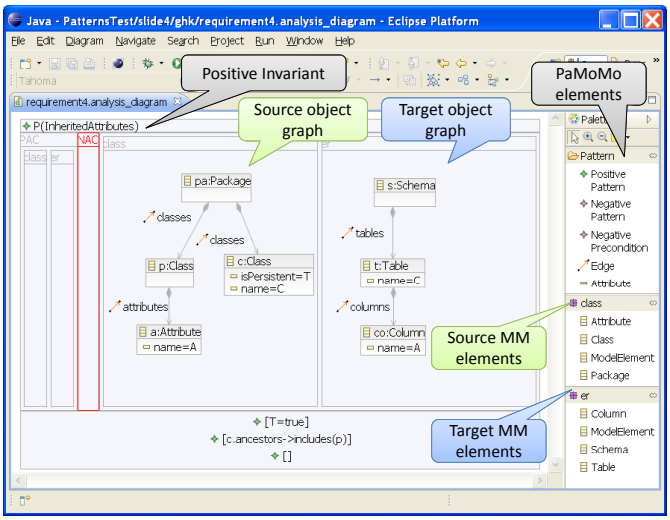

Fig. 10. GMF Editor for Modeling PaMoMo Patterns

White-Box-Based Test Source Model Generation. One of the most important components in a testing framework is an automatic test data generator for a given program [16]. Much research efforts have been investigated in software engineering in this area, including (i) *program-based methods*, i.e., white-box-based methods, and (ii) *syntax-based methods*, i.e., black-box-based methods. Regarding *white-box-based methods* in the area of model transformations, Küster et al. [17] focus on this topic. However, in their approach, the existence of a high-level design of model transformations, consisting of conceptual transformation rules, is assumed. Consequently, to apply this approach to existing model transformations, the manual extraction of these conceptual transformation rules is required, being in contrast to our vision of testing transformations automatically. Furthermore, González et al. [18] propose a first approach for testing ATL transformations. For the automatic generation of test source models, they also envision the application of constraint solvers. However, in contrast to our approach they are specific to ATL since they do not provide a common formalism such as our CFG or the symbolic execution tree. Additionally, no means for configuration, i.e., coverage criteria, or means for fault localization are provided. Additionally, many approaches have been proposed for *black-box testing*, i.e., test source models may be generated either on basis of the source MM (e.g.,[19], [20], [21] or the specified requirements [5], [22]. Finally, [23] provides a semi-automatic approach, since the transformation designer must specify a generation script on basis of the declarative ASSL language [24]. For the actual test source model generation, most of these approaches rely - similar to software engineering - on constraint satisfaction, e.g., by means of SAT solvers [25]. Furthermore, an approach has been proposed, which allows to automatically complete test source models, i.e., the tester has to specify "an intention" by a model fragment, only, and an algorithm complements this fragment to a valid test source model [26].

White-Box-Based Coverage Criteria for Model Transformations. In general, coverage criteria are used to derive certain sets of test input data, i.e., they specify when sufficient testing has been done such that it can be stopped [27]. In software engineering, much research effort has been spent in

the investigation of suitable coverage criteria - including black-box-based as well as white-box based ones (cf., e.g., [27] for an overview). In contrast, in the area of model transformations, research focused mainly on black-box-based coverage criteria (cf. e.g., [5]). Only McQuillan et al. [28] proposed three basic *control-flow-based coverage criteria* (rule coverage, instruction coverage, and decision coverage) specifically for ATL [29] transformations. Although these control-flow-based coverage criteria represent a first step for tackling this challenge, literature in software engineering reveals (cf., e.g., [27] for an overview) that also other categories of white-box-based coverage criteria may be of interest for the domain of model transformations, e.g., *data-flow-based coverage criteria*.

B. State-of-the-Art in Fault Localization

Since fault localization requires information on the elements that caused a test case to fail, we first investigate to which respect existing oracles provide failure traces and second, to which respect means for aligning the failure trace to the transformation definition are provided.

Oracles Providing Failure Traces. Different methods for the specification of oracles have been proposed, including comparison of *pre-computed input/output pairs* [30] or dedicated *specification languages* like Z [31]. In the area of model transformations, similar groups of methods have been proposed, comprising so-called (i) *full oracles* or (ii) *partial oracles*. Approaches employing *full oracles* follow the idea of model comparison, (e.g., [32]), whereby the desired target models are assumed to be provided by the transformation designer. In this context, basic support for a failure trace is provided, since the difference elements (added, updated, and deleted elements) between an actual target model and an expected target model may be calculated, but the tracing to the corresponding source model elements is left open. Approaches using *partial oracles* base on specification languages like OCL (cf. [33] for an overview) or graph patterns [34], [35] and exhibit different capabilities concerning the provision of a dedicated failure trace. The approaches presented in [36] and [37], both basing on OCL do not provide any failure trace. The same is true for the approaches presented in [34], [35], which use graph patterns, and the approach of [22], which employs the Human Usable Textual Notation (HUTN) language[4] for specifying object patterns and the Epsilon Validation Language (EVL)[5] for specifying the actual assertions. Finally, also [38], which uses QVT-Operational [3], is not able to provide any failure trace. In contrast, [23], which bases on the USE tool [39] is capable of providing a dedicated failure trace including those source model elements that caused the contract to fail. However, to benefit from this tool, all artifacts involved in a model transformation have to be represented in the USE-compatible format.

Alignment of Failure to Transformation Definition. Fault localization is one of the most expensive tasks in program testing and debugging [40]. In software engineering, various approaches have been presented, whereby in [41] an overview is given. Thereby, approaches range from *delta debugging* [42], i.e., comparing the input data of succeeding test cases to the input data of failing test cases, over *visualization-based* techniques, e.g., in [43] and *metrics-based approaches* that calculate suspicious ratings for statements [44], to *white-box-based approaches* [45]. Currently, approaches (e.g.,[46], [47]) exist that base on techniques from program slicing [48], in order to extract suspicious lines of codes in a transformation definition. Nevertheless, to the best of our knowledge, up to now, oracles used for testing model transformations provide no means to automatically align a failing test to the suspicious piece of code in a transformation definition.

[4]http://www.omg.org/spec/HUTN

[5]http://www.eclipse.org/epsilon/doc/evl

VIII. Conclusion and Future Work

In this paper we presented TETRA$_{Box}$ as a generic framework for execution-based testing of model transformations in a white-box manner. In a first step we presented our PaMoMo language for specifying requirements on transformations. In a second step, PaMoMo patterns may act as oracle since they may be compiled to check-only QVT-Relations which may be used to identify model elements, which are out of sync. For automatic generation of test source models, we derive a CGF and a symbolic execution tree, being independent of a specific transformation language. Depending on the desired coverage criteria, we collect according path constraints of the symbolic execution tree, which are then used by a constraint solver in order to generate the test source model. Finally, in case a test run fails, we provide means for aligning the failure trace of the oracle to the actual transformation definition in order to provide a starting point for debugging.

Concerning future work, several lines of work remain to be done. First, the set of offered coverage criteria should be complemented. In order to achieve this, coverage criteria will be collected and adapted in a *top-down* approach by investigating coverage criteria proposed in traditional software testing [27]. On the other hand, coverage criteria being specific to the domain of model transformations will be derived in a *bottom-up* approach by investigating existing transformation languages and their peculiarities. Furthermore, several transformation languages should be incorporated in our framework by providing dedicated HOTs, transforming their transformation syntax into our common CFG representation. Finally, an extensive evaluation on basis of real-world examples is envisioned.

References

[1] J. Bézivin, "On the Unification Power of Models," *Software and Systems Modeling*, vol. 4, pp. 171–188, 2005.

[2] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy, "A Tridimensional Approach for Studying the Formal Verification of Model Transformations," in *Proc. of Int. Conf. on Software Testing, Verification, and Validation*. IEEE, 2012, pp. 921–928.

[3] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," http://www.omg.org/spec/QVT/1.1/, 2009.

[4] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 215 – 222, 1976.

[5] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated Verification of Model Transformations Based on Visual Contracts," *Automated Software Engineering*, pp. 1–42, 2012.

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

[7] 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*, vol. 53, pp. 139–143, 2010.

[8] L. Baresi and M. Young, "Test Oracles," Department of Computer and Information Science, University of Oregon, Tech. Rep. CIS-TR01-02, 2001.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, 1991.

[10] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the Use of Higher-Order Model Transformations," in *Proc. of Europ. Conf. on Model Driven Architecture - Foundations and Applications*. Springer, 2009, pp. 18–33.

[11] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu, "Symbolic Execution with Interval Solving and Meta-heuristic Search," in *Proc. of Int. Conf. on Software Testing, Verification and Validation*. IEEE, 2012, pp. 111–120.

[12] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming," in *Proc. of Int. Conf. on Automated Software Engineering*. ACM, 2007, pp. 547–548.

[13] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven Compositional Symbolic Execution," in *Proc. of Int. Conf. of Theory and Practice of Software*. Springer, 2008, pp. 367–381.

[14] R. Straeten, T. Mens, and S. Baelen, "Challenges in Model-Driven Software Engineering," in *Models in Software Engineering*. Springer-Verlag, 2009, pp. 35–47.

[15] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai, "Challenges and Directions in Formalizing the Semantics of Modeling Languages," *Computer Science and Information Systems*, vol. 8, pp. 225–253, 2011.

[16] J. Edvardsson, "A Survey on Automatic Test Data Generation," in *Proc. of Conf. on Computer Science and Engineering*. ACTA Press, 1999, pp. 21–28.

[17] J. M. Küster and M. Abd-El-Razik, "Validation of Model Transformations: First Experiences Using a White Box Approach," in *Proc. of Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 193–204.

[18] C. A. González and J. Cabot, "Atltest: A white-box test generation approach for atl transformations," in *Proc. of 15th Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 449–464.

[19] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool," in *Proc. of Int. Symp. on Software Reliability Engineering*. IEEE, 2006, pp. 85–94.

[20] S. Sen, B. Baudry, and J.-M. Mottu, "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing," in *Proc. of Int. Conf. on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 328–337.

[21] ——, "Automatic Model Generation Strategies for Model Transformation Testing," in *Proc. of Int. Conf. on Theory and Practice of Model Transformations*. Springer, 2009, pp. 148–164.

[22] P. Giner and V. Pelechano, "Test-Driven Development of Model Transformations," in *Model Driven Engineering Languages and Systems*, ser. LNCS. Springer Berlin / Heidelberg, 2009, pp. 748–752.

[23] M. Gogolla and A. Vallecillo, "Tractable Model Transformation Testing," in *Proc. of Europ. Conf. on Modelling Foundations and Applications*. Springer, 2011, pp. 221–235.

[24] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL Models in USE by Automatic Snapshot Generation," *Software and Systems Modeling*, vol. 4, pp. 386–398, 2005.

[25] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional Satisfiability and Constraint Programming: A Comparative Survey," *ACM Computing Surveys*, vol. 38, pp. 1–62, 2006.

[26] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot, "Using Models of Partial Knowledge to Test Model Transformations," in *Proc. of Int. Conf. on Model Transformations*. Springer, 2012, pp. 24–39.

[27] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, pp. 366–427, 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[28] J. McQuillan and J. Power, "White-Box Coverage Criteria for Model Transformations," in *Proc. of Int. Works. on Model Transformation with ATL*. Online Publication, 2009, pp. 63–77.

[29] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Science of Computer Programming*, vol. 72, pp. 31–39, 2008.

[30] D. J. Panzl, "Automatic Software Test Drivers," *Computer*, vol. 11, pp. 44–50, 1978.

[31] J. M. Spivey, "An Introduction to Z and Formal Specifications," *Software Engineering Journal*, vol. 4, pp. 40–50, 1989.

[32] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "EUnit: A Unit Testing Framework for Model Management Tasks," in *Proc. of Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 395–409.

[33] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Model Transformation Testing: Oracle Issue," in *Proc. of Int. Conf. on Software Testing Verification and Validation*. IEEE, 2008, pp. 105–112.

[34] A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró, "Workflow-Driven Tool Integration Using Model Transformations," in *Graph Transformations and Model-Driven Engineering*. Springer, 2010, pp. 224–248.

[35] T. A. Khan, O. Runge, and R. Heckel, "Visual Contracts as Test Oracle in AGG 2.0," *ECEASST*, vol. 47, 2012.

[36] E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "OCL Contracts for the Verification of Model Transformations," *Electronic Communications of the EASST*, vol. 24, 2009.

[37] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Reusable MDA Components: A Testing-for-Trust Approach," in *Proc. of Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 589–603.

[38] A. Ciancone, A. Filieri, and R. Mirandola, "MANTra: Towards Model Transformation Testing," in *Proc. of Int. Conf. on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 97–105.

[39] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based Specification Environment for Validating UML and OCL," *Science of Computer Programming*, vol. 69, pp. 27 – 34, 2007.

[40] T. Ball and S. G. Eick, "Software Visualization in the Large," *Computer*, vol. 29, pp. 33–43, 1996. [Online]. Available: http://dx.doi.org/10.1109/2.488299

[41] J. A. Jones, "Semi-automatic Fault Localization," Ph.D. dissertation, School of Computer Science, Georgia Institute of Technology, Atlanta, USA, 2008.

[42] H. Cleve and A. Zeller, "Locating Causes of Program Failures," in *Proc. of Int. Conf. on Software Engineering*. ACM, 2005, pp. 342–351. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062522

[43] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proc. of Int. Conf. on Software Engineering*. IEEE, 2002, pp. 467–477.

[44] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proc. of Int. Conf. on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[45] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Fault Localization for Dynamic Web Applications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 314 –335, 2012.

[46] J. Schönböck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Catch Me If You Can - Debugging Support for Model Transformations," in *Proc. of Workshops and Symposia of Models in Software Engineering*. Springer, 2009, pp. 5–20.

[47] Z. Ujhelyi, Á. Horváth, and D. Varró, "Dynamic Backward Slicing of Model Transformations," in *Proc. of Int. Conf. on Software Testing, Verification and Validation*. IEEE, 2012, pp. 1–10.

[48] M. Weiser, "Program slicing," in *Proc. of Int. Conf. on Software Engineering*. IEEE, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557