Rule Chains Coverage for Testing QVT-Relations Transformations

Daniel Calegari and Andrea Delgado

Instituto de Computación, Facultad de Ingeniería,
Universidad de la República, 11300 Montevideo, Uruguay
{dcalegar,adelgado}@fing.edu.uy

Abstract. Traditional software testing techniques have been adapted to deal with the verification of model transformations. Black-box techniques have the benefit of simplicity as well as the advantage of being independent of the implementation language, and thus compatible with any model transformation language. Although this is important, the inherent complexity of metamodels may result in a significant amount of non-relevant test models. On the contrary the use of white-box techniques allows generating more effective models at a higher cost. In this paper we propose an approach for the verification of QVT-Relations transformations which considers the dependencies between transformation rules and the standard semantics. Test models generation is based on the construction of what we call a *rule chain*: a set of rule patterns and conditions satisfying a top rule, as well as on an adaptation of other techniques, as grammar testing and partition analysis. We introduce the approach and explain its application using a model transformation devised for the generation of service models from business process models. This approach generates more effective test models than existing approaches, for which we are working on several tests to prove it.

Key words: testing, model transformations, QVT-Relations

1 Introduction

The feasibility of the Model-Driven Engineering paradigm (MDE [1]) is strongly based on the existence of a (semi)automatic construction process driven by model transformations. A transformation basically takes as input a model conforming to certain metamodel and produces as output another model conforming to another metamodel (possibly the same). This very simple transformation schema can be extended to take more than one source model as input and/or produce multiple target models as output, among other extensions. The Query/View/Transformation Relations (QVT-Relations [2]) language follows this schema using a relational approach, which consists on defining transformation rules as relations between source and target elements.

There are several alternatives for assessing the quality of a model transformation [3], from logical inference, which consists of using a mathematical representation of a system and the verification properties, to testing, which relies on the construction of test cases including subsequent execution of the transformation on these models and validate that the output matches the expected one. Although it has some disadvantages, testing is the most popular technique, since it is lightweight, automatable and can easily uncover bugs [4].

The model transformation testing process consists of four phases [5], from the generation of a test suite (set of test models) conforming to the source metamodel for testing the transformation to the execution of the transformation on them and the evaluation of the outputs with respect to the expected ones. The test models generation involves the definition of test adequacy criteria and the building of the test suite that achieves coverage of the adequacy criteria. This generation can follow a black-box approach (only using the source and target metamodels and transformation contracts), grey-box (using partial knowledge of the transformation implementation) or white-box approach (using the full transformation implementation).

In this paper we propose a white-box approach for the generation of test models for QVT-Relations transformations. The approach basically considers the dependencies between transformation rules and the standard semantics of QVT-Relations for the construction on what we call a *rule chain*: a set of rule patterns and conditions satisfying a top rule. The adequacy criterion is defined to be the coverage of every possible rule chain, and thus the whole model transformation. The generation of these rule chains and of the test suite is based on an adaptation of other techniques such as grammar testing [6] and partition analysis [7].

The remainder of the paper is structured as follows. In Section 2 we briefly present some background on QVT-Relations and in Section 3 we introduce a running example. In Section 4 we present our approach. In Section 5 we discuss existing approaches and their relation with our work. Finally, in Section 6 we present a short summary with conclusions and an outline of further work.

2 A Brief Look at QVT-Relations

A QVT-Relations transformation can be viewed as a set of interconnected relations (or rules) which are of two kinds: top-level relations which must hold in any transformation execution, and non-top-level relations which are required to hold only when they are referred from another relation. Every relation has a set `<R_var_set>` of variables occurring in the relation and defines a source and a target domain pattern (`<domain_k_pat>`) which is used to find matching sub-graphs in the source and target models, respectively. Relations can also contain `when` (`<when_cond>`) and `where` (`<where_cond>`) clauses. A `when` clause specifies the conditions under which the relationship needs to hold, whilst the `where` clause specifies the condition that must be satisfied by all model elements participating in the relation. The `when` and `where` clauses may contain arbitrary boolean expressions in addition to the call of other relations. Finally, any relation can define a set of primitive domains (`<R_par_set>`) which are data types used to parameterize the relation. We can view a relation as having the following abstract structure:

```
[top] relation R {
    <R_var_set> <R_par_set>
    Domain {
        <domain_k_pat>
    } //k = 1,2
    [when <when_cond>]
    [where <where_cond>]
}
```

The standard semantics [2] states that a relation holds if for each valid binding of variables of the `when` clause and variables of domains other than the target domain, that satisfy the `when` condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of the target domain that satisfies the target domain pattern and `where` condition. This can be interpreted as a logical formula [2] basically saying that a relation holds if the following formula holds:

$$\texttt{when} \rightarrow (\texttt{<domain\_1\_pat>} \rightarrow (\texttt{<domain\_2\_pat>} \wedge \texttt{where}))$$

In other words, this formula holds if one of the following cases hold: the `when` clause do not holds, (b) the `when` clause holds but the source domain pattern (`<domain_1_pat>`) do not holds, (c) every element in the formula holds (both `<domain_1_pat>` and `<domain_2_pat>` patterns, `when` and `where` clauses).

Finally, dependencies between relations can be represented as a graph where each node represent a relation, and each directed edge is labelled with `when`/`where` representing the invocation of the target relation from the source one within the corresponding clause.

3 Generation of SoaML models from BPMN2 models

As part of our research work we have defined the MINERVA framework [8] which defines several elements to support the continuous improvement of business processes (BPs) implemented by services with a model driven approach. At the heart of our approach is the automation of BPs implementation, based on the generation of service models in Service Oriented Modeling Language (SoaML [9]) from BP models specified in Business Process Model and Notation (BPMN2 [10]). BPMN is a readily understandable standard notation for specifying business processes whilst SoaML provides a metamodel and a UML profile for the specification and design of services within a service-oriented architecture. It defines specific stereotypes to be used when modeling services within a SOA but with no reference to implementation details.

We followed a model-driven approach to define the different type of models we use to provide the implementation of BPs, from the specification of the BP to the model for its execution and the software models to support their implementation. In Figure 1 we illustrate this vision. To generate SoaML models from BPMN2 models [11] we have defined a set of transformations in QVT Relations, based on a mapping between elements from BPMN2 and SoaML metamodels.

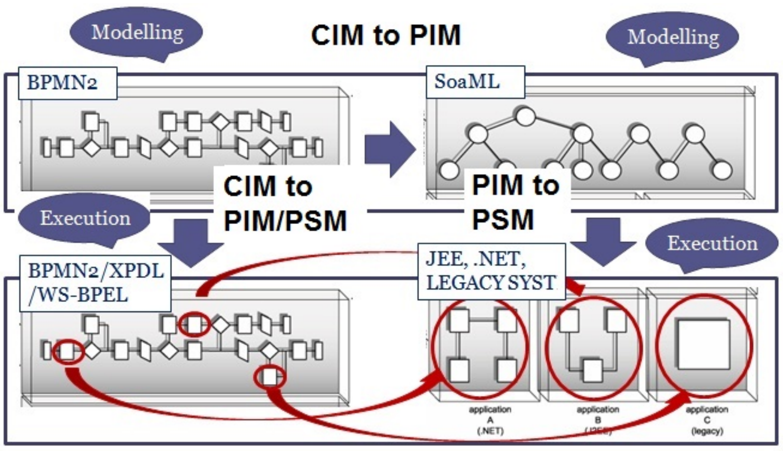

Fig. 1. Model-driven approach for the implementation of BPs

Three main transformations for the generation of services from elements in BPMN2 are provided. Each transformation defines several dependencies between the relations that are stated in the `when` and `where` clauses of each rule, which are the same for all of them. This dependencies graph is shown in Figure 2 where the top relations are shown in grey and the invoked relations are shown in white.

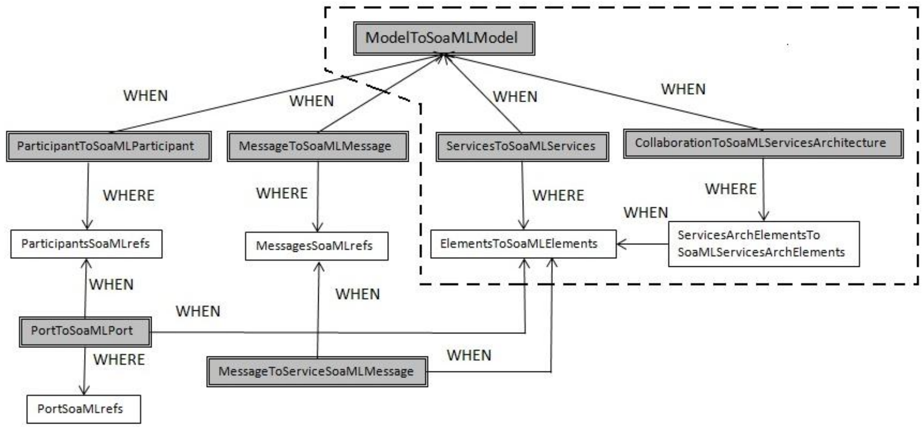

Fig. 2. Dependencies graph of the running example

Due to space reasons we do not give details about the transformations, but we explain the dependencies between rules that constitute a key element of this work. More details can be found in [11]. In the first place, we generate a SOA Model from a process definition, and then we generate Participants, Messages and Services from the messages between process participants. Then, we assign the Messages as the types of the parameters in the generated service operations, and create Ports on the Participants generated, typed with the corresponding Service or Request stereotype, depending on the service being provided or consumed. Finally, the ServicesArchitecture is created referencing every element.

4 Rule Chains Coverage

We define a white-box approach for test models generation, which involves generating a test suite conforming to the source metamodel for testing the transformation of interest, based on the following knowledge: the dependencies graph between rules, the transformation specification, the source metamodel, and the standard semantics. The general idea is based on using the dependencies graph for generating test models not covering the whole transformation (as with rule coverage [12]) but the minimal sets of rules which satisfy every top rule. The information extracted from the dependencies graph is supplemented with the knowledge of the three cases of rule satisfaction defined before, which allows defining a set of *rule chains*: a set of rule patterns and conditions satisfying a top rule. We follow three-steps for the generation of the test suite: grammar generation, model templates generation, and test suite generation, which are explained in the following subsections.

4.1 Grammar Generation

In order to construct a valid source model to test the transformation, we only care about the source pattern of a rule, not the target pattern. In this sense, any source model satisfying a rule must respect one of the cases of rule satisfaction defined in the last section: (a) do not satisfy the source conditions of the `when` clause, (b) satisfy the last but do not satisfy `<domain_1_pat>` (the matching subgraph we are looking for within the model), (c) satisfy both conditions, plus the source conditions of the `where` clause. With this information, we can define a grammar rule, for each transformation rule, as follows.

$$\begin{aligned}\langle \text{Rule} \rangle ::= {} & \neg\ \langle \text{Rule\_When} \rangle \\ \mid {} & \langle \text{Rule\_When} \rangle\ \neg\ \textbf{Rule\_Pat} \\ \mid {} & \langle \text{Rule\_When} \rangle\ \textbf{Rule\_Pat}\ \langle \text{Rule\_Where} \rangle\end{aligned}$$

The source domain pattern is a terminal symbol (represented as **Rule_Pat**) and the `when`/`where` clauses are non-terminal ones (they must represent other transformation rules as well as boolean expressions constraining those rules). It can be noticed that the negation ¬ represents that the pattern must not be satisfied. Moreover, a transformation rule may have empty `when`/`where` clauses. In this case the corresponding production rules will be discarded from the grammar.

The first two cases seem to be unnecessary since the transformation rule will not generate any target model when executed. However, these are considered negative scenarios which are also interesting to test.

The dependencies graph states that the rules are chained through the `when`/`where` clauses. If we combine this information together with the grammar rules of each transformation rule, we can generate a complete grammar. This grammar has a top grammar rule composed by the top transformation rules, as follows:

$$\langle \text{Transformation} \rangle ::= \langle \text{Rule}_1 \rangle \mid ... \mid \langle \text{Rule}_n \rangle$$

As an example, if we focus on the subgraph within the dotted lines in Figure 2, we can extract the following grammar (names are abbreviated), where for example rule ModelToSoaML has empty `when`/`where` clauses.

$$\begin{array}{rcl}
\langle \text{Transformation} \rangle & ::= & \langle \text{ModelToSoaML} \rangle \\
& \mid & \langle \text{ServicesToSoaML} \rangle \\
& \mid & \langle \text{CollToSoaML} \rangle \\
\langle \text{ModelToSoaML} \rangle & ::= & \textbf{ModelToSoaML\_Pat} \\
\langle \text{ServicesToSoaML} \rangle & ::= & \neg\ \langle \text{ModelToSoaML} \rangle \\
& \mid & \langle \text{ModelToSoaML} \rangle\ \neg\ \textbf{ServicesToSoaML\_Pat} \\
& \mid & \langle \text{ModelToSoaML} \rangle\ \textbf{ServicesToSoaML\_Pat} \\
& & \langle \text{ElementsToSoaML} \rangle \\
\langle \text{CollToSoaML} \rangle & ::= & \neg\ \langle \text{ModelToSoaML} \rangle \\
& \mid & \langle \text{ModelToSoaML} \rangle\ \neg\ \textbf{CollToSoaML\_Pat} \\
& \mid & \langle \text{ModelToSoaML} \rangle\ \textbf{CollToSoaML\_Pat} \\
& & \langle \text{ServArchEltToSoaML} \rangle \\
\langle \text{ElementsToSoaML} \rangle & ::= & \textbf{ElementsToSoaML\_Pat} \\
\langle \text{ServArchEltToSoaML} \rangle & ::= & \neg\ \langle \text{ElementsToSoaML} \rangle \\
& \mid & \langle \text{ElementsToSoaML} \rangle\ \neg\ \textbf{ServArchEltToSoaML\_Pat} \\
& \mid & \langle \text{ElementsToSoaML} \rangle\ \textbf{ServArchEltToSoaML\_Pat}
\end{array}$$

It can be noticed that the grammar above was simplified for the matter of presentation, by only considering `when`/`where` clauses composed by other transformation rules. We can generate the grammar by also considering boolean conditions within `when`/`where` clauses. In this case there will be a set of terminal symbols representing these conditions. However, the test case generation will be more complex in the presence of OCL conditions, as we explain later.

4.2 Model Templates Generation

Every possible string generated from the grammar is a rule chain which provides the set of conditions from transformation rules needed for generating valid test models satisfying a top rule. Any combination of rule chains, one for each top rule, completely covers the transformation (not necessarily covers every rule). In our example, we can generate the following rule chains.

With respect to the information provided by a rule chain, for example the seventh rule chain defines one of the possible set of conditions which must satisfy a source model in order to be valid for rule CollToSoaML, and indirectly rule ModelToSoaML. The rule chain defines that the model must satisfy the source pattern of those rules, together with the source pattern of rule ElementsToSoaML, and not to satisfy the pattern of ServArchEltToSoaML.

[1] ModelToSoaML_Pat
[2] ¬ ModelToSoaML_Pat
[3] ModelToSoaML_Pat ¬ ServicesToSoaML_Pat
[4] ModelToSoaML_Pat ServicesToSoaML_Pat ElementsToSoaML_Pat
[5] ModelToSoaML_Pat ¬ CollToSoaML_Pat
[6] ModelToSoaML_Pat CollToSoaML_Pat ¬ ElementsToSoaML_Pat
[7] ModelToSoaML_Pat CollToSoaML_Pat ElementsToSoaML_Pat
¬ ServArchEltToSoaML_Pat
[8] ModelToSoaML_Pat CollToSoaML_Pat ElementsToSoaML_Pat
ServArchEltToSoaML_Pat

The example reflects a limitation with rule chains generation: we are assuming that the grammar has no recursion, and thus all rule chains are finite. We need a deep research on this topic to tackle with this limitation.

4.3 Test Suite Generation

We can use rule chains information to construct valid test models. For this purpose we are in the process of adapting other techniques. Each source rule pattern allows reducing the original metamodel to an effective one for such rule. We can then use partition analysis [7] to find the representative values for each property of those metamodels separately. However this technique must be adapted in order to consider conditions required by domain patterns. For example, if we consider rule ServicesToSoaML, we need to generate partitions ensuring that a Definition has at least two collaborating Process instances. If not, the partition could generate models not satisfying the rule.

The adequacy criterion is defined to be the coverage of every possible rule chain, which is achieved by construction. However, we want to generate more than one input model with just one set of valid bindings to the variables in the relevant rules, for each rule chain. In fact, we use partition analysis to generate different values for each property and thus have a most complete test suite.

We also need an incremental approach to define test models from the information provided by each pattern within the rule chains. We need to systematically construct overlapping partitions, one for each rule chain. This is a non trivial constraint solving problem. For instance we only consider the case where the clause just invokes another relation. However, a complete approach must generate test models considering boolean expressions associated to `when/where` clauses (relation invocations can be arbitrarily combined with other OCL constraints, such that the evaluation of the constraint depends on the relation invocation) as well as rules parametrization. This is subject of future research.

Finally, overlapping partitions can be used as input for the generation of model and object fragments, as in [7]. In order to generate adequate test models we also need to consider those invariants that must hold in any model conforming to the source metamodel. With the use of rule chains we are avoiding the accidentally construction of test models which will never satisfy the source conditions of the transformation. In this sense the construction of test cases is more effective in terms of the transformation execution. As an extremely simple example, by the fourth rule chain above, we know that if a model satisfies the pattern of rule ServicesToSoaML then it must satisfy the pattern of rule ElementsToSoaML. Using the former technique it is possible to generate a model not satisfying this constraint, and thus an invalid source model for the transformation.

No matter how effective we think the resulting test suite is, we also need some coverage analysis for measuring test suite quality, as in [13]. If we cover every rule chain, we have an equivalent coverage as a combination of the rule and (effective) metamodel coverage criteria, since the set of rule chains completely covers the whole model transformation, as well as the conditions expressed by rule chains completely covers the effective metamodel of the transformation. Further work is needed in order to know how our approach performs in relation with other criteria or alternative techniques.

5 Related Work

As stated in [5], the complexity of constraints that define the input domain is the main challenge for automatic test model generation. In this sense, several ideas have been proposed for the generation of an effective and minimal test suite. For example. in [14] the authors propose the construction of an *effective* metamodel composed of the source metamodel elements referenced in the transformation implementation. Ideally, this step reduces the set of test models to be considered. Then, concrete models must be generated according to some adequacy criteria, e.g. achieve metamodel coverage or rule coverage. Although it is important to have generic approaches compatible with any model transformation language, like those referred earlier, the use of techniques based on specific knowledge allows generating more effective models within a smaller test suite, and thus simplifying the testing process. In [15] the authors explored the extraction of partial knowledge from model transformations about its usage of the input metamodel to generate effective test models. They use mutation analysis to experimentally evaluate the fault-detecting effectiveness of the set of test models. We can follow a similar approach to evaluate our test suite generation.

The generation of rule chains is related to the grammar testing approach [6]. In our case we do not get concrete models to be part of the test suite, but model templates. Each rule chain defines the conditions under which concrete test models must be generated. In [16] the authors propose to use pairs of rules in order to define test models, since they argue “that it is useful to construct systematically all possible overlapping models of two rules”. However, they do not take into account the relations between rules as defined with the dependencies graph. We will use the idea of overlapping models to construct our test suite. Finally, we have some sort of refinement of the rule coverage strategy [12], in which we do not only try to generate a test suite exploring all rules, but also a more effective suite by considering how these rules are connected. In [17] dependencies between rules are also used as a coverage criteria. However, that work is based on Triple Graph Grammars and not QVT-Relations, thus the general approach for the test model generation is completely different.

Overlapping partitions are related to [16] in which the authors propose to construct models by taking the left sides of two rules and then calculate all possible overlaps of model elements. In this case, if the overlapping model is syntactically incorrect, it is discarded. However, in our case an inconsistency with the generation of an overlapped partition could results in an incongruent set of transformation rules, e.g. one rule only accepts a positive integer within a property whereas another one requires the same property to be negative, which is identified as an error to be corrected.

6 Conclusions & Future Work

We have presented an approach for the generation of test models for QVT-Relations transformations. The approach is based on using the dependencies graph and the standard semantics for generating test models covering any possible rule chain, i.e. a set of rule patterns and conditions satisfying a top rule. We generate rule chains based on an adaptation of the grammar testing technique. From the information provided by rule chains, we generate an effective source metamodel for each rule, and then we use partition analysis to find the representative values for each property within the metamodel. Finally, we need to systematically construct overlapping partitions, one for each rule chain, and from these partitions generate specific test models. This last step is subject of current work. In fact, we are using the model transformation from BPMN2 models to SoaML models [11] for completing the approach and, at the same time, carry out a complete verification process which is part of our research agenda.

The adequacy criterion is defined to be the coverage of every possible rule chain. This is equivalent to a combination of the rule and (effective) metamodel coverage criteria, since the set of rule chains completely covers the whole model transformation, as well as the conditions expressed by rule chains completely covers the effective metamodel of the transformation. This is a positive coverage, i.e. the rules are supposed to be satisfied but not necessarily to generate a target model. Thus, we are avoiding the accidentally construction of test models which will never satisfy the source conditions of the transformation. Nevertheless, it can be useful to generate negative scenarios, for example satisfying invalid rule chains (e.g. substrings of the rule chains) to have a complete test suite. Although this is an initial work, we expect that this approach generates more effective test models than existent approaches. However, this assertion deserves a deeper comparative analysis, which is intended for future work.

We observed that not every property is valuable when applying partition analysis. Instead, we can focus on those properties that are really *significant* for the transformation, i.e. those whose value affect a condition within the transformation. In this sense, we can identify and apply partition analysis only for those significant model elements and use any default value for the rest of the elements. This is related to [16] in which the authors identify model elements changed by the transformation in order to test that all constraints that may be violated due to the change hold after applying a transformation. This idea may assist in the simplification of the partitions. However, it needs a more complex procedure since we also need to consider the target domain patterns of the transformation.

Finally, we need to evaluate if this approach can be generalized for multiple source metamodels and adapted to other languages. Moreover we need to investigate how the OCL can be considered associated to `when`/`where` clauses.

Acknowledgments. This work has been funded by the Agencia Nacional de Investigación e Innovación (ANII), Uruguay by means of the “Generación Automática de Arquitecturas Orientadas a Servicios desde Procesos de Negocio” project (ANII-FCE-2011-7341).

References

1. Kent, S.: Model driven engineering. In: IFM ’02, Springer (2002) 286–298
2. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation. Final Adopted Specification Version 1.1, Object Management Group (2009)
3. Calegari, D., Szasz, N.: Verification of model transformations: A survey of the state-of-the-art. ENTCS **292** (2013) 5–25
4. Selim, G.M.K., Cordy, J.R., Dingel, J.: Model transformation testing: the state of the art. In: AMT ’12, ACM (2012) 21–26
5. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM **53** (2010) 139–143
6. Lämmel, R.: Grammar testing. LNCS **2029** (2001) 201–216
7. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. SoSyM **8** (2009) 185–203
8. Delgado, A., Ruiz, F., de Guzmán, I.G.R., Piattini, M.: Minerva: Model driven and service oriented framework for the continuous business process improvement and related tools. LNCS **6275** (2010) 456–466
9. OMG: Service Oriented Architecture Modeling Language (SoaML). Technical Report Version 1.0.1, Object Management Group (2012)
10. OMG: Business Process Model and Notation (BPMN). Technical Report Version 2.0, Object Management Group (2011)
11. Delgado, A., Ruiz, F., de Guzmán, I.G.R., Piattini, M.: Model transformations for business-it alignment: from collaborative business process to SoaML service model. In: SAC, ACM (2012) 1720–1722
12. Mcquillan, J.A., Power, J.F.: White-box coverage criteria for model transformations. First Intl. Workshop on Model Transformation with ATL (2009)
13. Bauer, E., Küster, J.M., Engels, G.: Test suite quality for model transformation chains. In: TOOLS’11, Springer (2011) 3–19
14. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: MoDeVVa ’04. (2004) 29–40
15. Mottu, J.M., Sen, S., Tisi, M., Cabot, J.: Static analysis of model transformations for effective test generation. In: ISSRE, IEEE (2012) 291–300
16. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations: first experiences using a white box approach. In: MoDELS’06, Springer (2006) 193–204
17. Hildebrandt, S., Lambers, L., Giese, H.: Complete specification coverage in automatically generated conformance test cases for TGG implementations. LNCS **7909** (2013) 174–188