Towards Automated Testing of Abstract Syntax Specifications of Domain-Specific Modeling Languages

Daniel A. Sadilek and Stephan Weißleder

Humboldt-Universität zu Berlin
Department of Computer Science
Rudower Chaussee 25
12489 Berlin, Germany

{sadilek|weissled}@informatik.hu-berlin.de

Abstract: The abstract syntax of domain-specific modeling languages (DSMLs) can be defined with metamodels. Metamodels can contain errors. Nevertheless, they are not tested systematically and independently of other artifacts like models or tools depending on the metamodel. Consequently, errors in metamodels are found late—not before the dependent artifacts have been created. Since all dependent artifacts must be adapted when an error is found, this results in additional error correction effort. This effort can be saved if the metamodel of a DSML is tested early. In this paper, we argue for metamodel testing and propose an approach that is based on understanding a metamodel as a specification of a set of models. Example models are given by a user to test if the set of models specified by a metamodel is correct. We present an example from the domain of earthquake detection to clarify our idea.

1 Introduction

Metamodels are a common way to describe the structure of domain-specific modeling languages (DSMLs). Tools for a DSML like editors, interpreters, or debuggers base on this metamodel. Like every other artifact, metamodels contain errors (e.g. wrong specification of classes or associations between them). When errors in a metamodel are found late, dependent models and tools must be adapted. Hence, *detecting errors* in a metamodel early can save time and money.

In software engineering, *testing* is the primary means to detect errors. In this paper, we advocate *testing metamodels* and present an approach for automated testing based on the specification of positive and negative example models.

In Sec. 2, we describe how to *specify* metamodel tests with example models and we describe how to *execute* them in Sec. 3. In Sec. 4, we substantiate our approach with an exemplary development process of a simple DSML. We discuss related work in Sec. 5. We conclude and give an overview of future work in Sec. 6.

2 How to Test Metamodels?

2.1 Example Models and Test Models

How can a metamodel be tested? To answer this question, we have to consider the nature of metamodels. A metamodel is the specification of a set of possible or desired models. What does it mean that a metamodel contains an error? It means that the specified set of models either contains a model that is undesired or that it does not contain a model that is desired.

The set of models specified by a metamodel is a subset of all instances of all possible metamodels expressible as instances of the meta-metamodel used[1]. Figure 1 shows an Euler diagram visualizing this idea. To test a set specification, one could give all elements of the set and check if the set does contain these and only these elements. Since metamodels generally specify an infinite set of models, this is impossible. Instead, representative elements can be given. Each representative element can be either an element or not an element of the set.

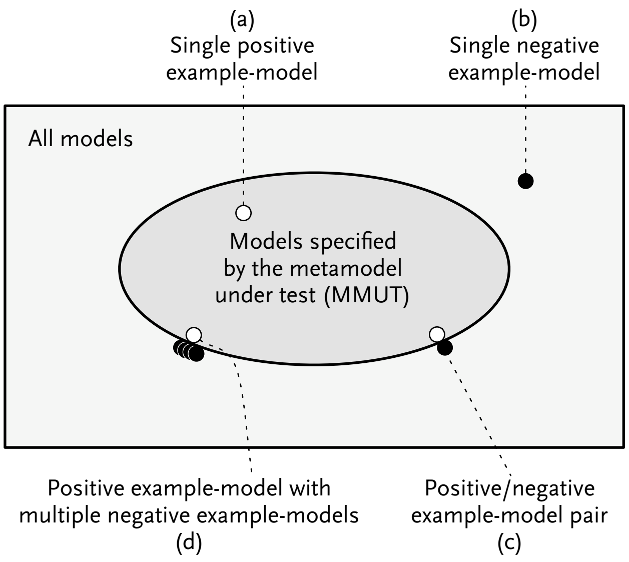

(a)
Single positive example-model
(b)
Single negative example-model
All models
Models specified by the metamodel under test (MMUT)
Positive example-model with multiple negative example-models
(d)
Positive/negative example-model pair
(c)

Figure 1: Metamodel as a set specification; example models as elements.

In the following, we describe the relationship between such representative elements and a *Metamodel Under Test (MMUT)*.

For metamodels, we call representative elements *example models*. Example models that are elements of the set of desired models should be correct instances of the MMUT—hence

[1]We consider metamodels that are instances of MOF.

we call them *positive* example models (Fig. 1a). Example models that are not elements of the set of desired models should not be instances of the MMUT—we call them *negative* example models (Fig. 1b).

Single example models can be anywhere inside or outside the set specified by the MMUT. But, for high discriminatory power of the tests, we propose to give example models as pairs of a positive and a negative example model that differ only in one aspect, for example by an attribute value or by the existence of some object or reference. The positive/negative example model pairs then demarcate the boundary of the set specified by the MMUT (Fig. 1c). The more example model pairs are given by a user, the more precise the boundary is demarcated. This resembles the common testing technique of boundary testing [Bei90].

A positive example model and its negative counterpart differ only slightly. If a user has to specify them separately, this introduces a lot of redundancy. Therefore, we propose to specify them in only one *test model*. A test model is a positive example model extended with *test annotations* that describe which model elements have to be added or removed to make it a negative example model. Thus, a test model allows a user to specify a positive/negative example model pair without redundancy.

We propose to allow the user to annotate more than one model element. Then, one test model can describe one positive and multiple negative example models (Fig. 1d).

2.2 Test Metamodel

2.2.1 Motivation for an Additional Metamodel

Technically, models cannot be created and stored without a corresponding metamodel. Which metamodel should be used for test models? Can we use an existing one, for example the MMUT or the metamodel of UML object diagrams?

Unfortunately, the MMUT cannot be used. The reason is that the MMUT does not allow to express test annotations. Also, a user may want to create test models before the MMUT exists—for example, to sketch how instances may look like or to follow a *test first* approach like in test-driven development [Bec02].

Test models describe instances of the MMUT. UML object diagrams can be used to describe instances of arbitrary classes. Could they be used to describe instances of the MMUT's classes? Unfortunately, the metamodel for UML object diagrams does not contain elements to express test annotations, i.e. there is no way to describe a combination of several example models in one object diagram. Also, UML object diagrams explicitly reference the classes that the modeled objects instantiate. This again forbids to create test models before the MMUT exists.

Therefore, another metamodel for test models is needed. We call it *test metamodel*. Figure 2 shows the test metamodel we propose and other artifacts of our approach: A test model is an instance of the test metamodel and it specifies one positive example model

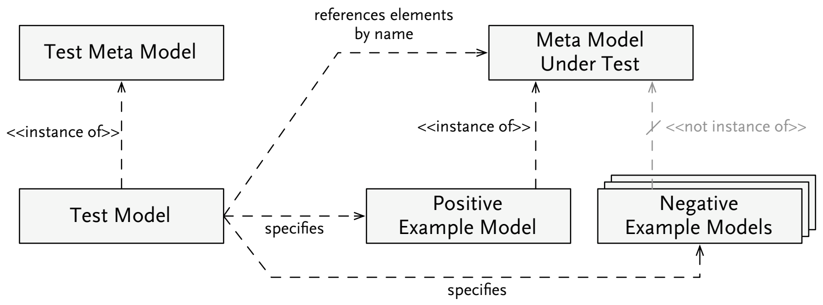


Figure 2: Relations between artifacts of our approach.

and an arbitrary number of negative example models. A test model references elements of the MMUT by name (explained below). For each MMUT, there can be various test models that are all instances of the test metamodel. The test metamodel is fixed, i.e. test models for different MMUTs are all instances of the same test metamodel.

2.2.2 Structure of the Test Metamodel

Figure 3 shows the test metamodel: *Instances* of classes are given with their class name and an optional object name. An instance can have an arbitrary number of *attributes*. Each attribute has a name and a value, which is given generically as a string literal. Instances can be connected by *references*. Reference ends can be named. The name must match an attribute of the instance's class at the opposite end of the reference.

All *model elements* (instances, attributes, and references) have an existence specification that can be *arbitrary* (default value), *enforced*, or *forbidden*. All model elements with existence specification *arbitrary* or *enforced* are part of the specified positive example model. If an element is enforced, removing this element from the model leads to a negative example model. If an element is forbidden, it is not part of the positive example model and adding it leads to a negative example model.

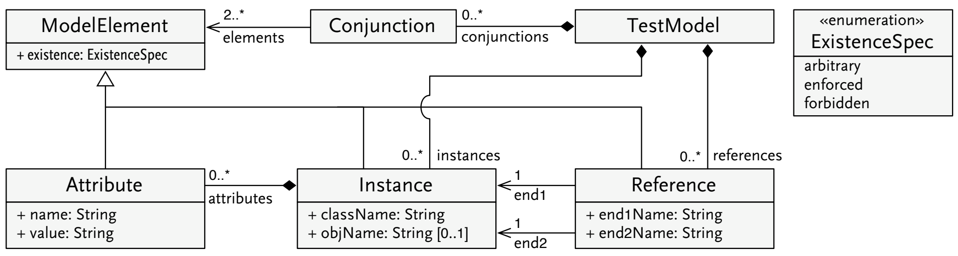


Figure 3: The test metamodel.

If multiple elements in a test model are enforced or forbidden, one test model describes multiple negative example models. If two or more elements should be enforced or forbidden conjunctionally, i.e. they should describe just one negative example model, then they can be connected by a *conjunction*.

3 Test Execution

In this section, we briefly describe how test models are used to test metamodels. The test execution consists of 5 steps:

1. *Resolve references to the MMUT.*
   The test model references the elements of the MMUT by name. In the first step, it is checked whether all references can be resolved. If a reference of a not forbidden element cannot be resolved, the test fails.

2. *Derive example models from the test model.*
   Each test model specifies one positive example model and multiple negative example models. The *positive example model* is derived from the test model by leaving out all forbidden elements. A *negative example model* is derived for each conjunction in the test model and for each enforced or forbidden element that is not connected to a conjunction. Let e be a conjunction of elements or a single element for which a negative example model is to be derived. Then all forbidden elements except e are left out when constructing the model. If e itself is a forbidden element, it is added to the negative example model; if e is enforced, it is left out.

3. *For all example models: Create an instance of the MMUT according to the example model.*

4. *For all example models: Check multiplicities and constraints of the created MMUT instance.*

5. *For all example models: Decide test outcome.*
   If the current example model is a positive one, constraints must *not* be violated in the previous steps; if it is a negative one, *at least one* constraint must be violated.

4 Example: Testing the Metamodel of a Stream-Oriented DSML

In this section, we describe the first step of an exemplary iterative development process of a simple *stream-oriented DSML* for the configuration of an earthquake detection algorithm: A sensor source generates a data stream that can be filtered and that finally streams into a sink. For this example, we use the prototypical implementation of our approach: *MMUnit* (`http://mmunit.sourceforge.net`).

The development of the stream-oriented DSML involves a language engineer and an expert of the domain, a seismologist. As a first step, seismologist and language engineer discuss some example uses of the new language. For the beginning, they concentrate on one specific detection algorithm called *STA/LTA* [Ste77]. They sketch their ideas in an informal ad hoc concrete syntax. Figure 4 shows the resulting model they have drawn on a whiteboard.

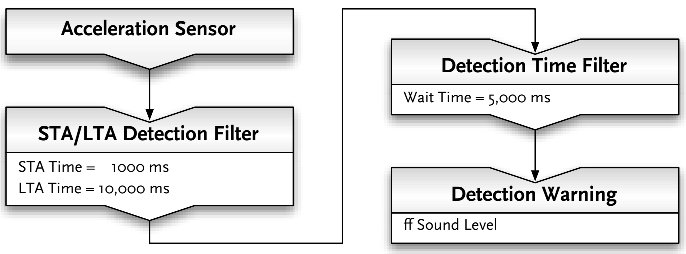
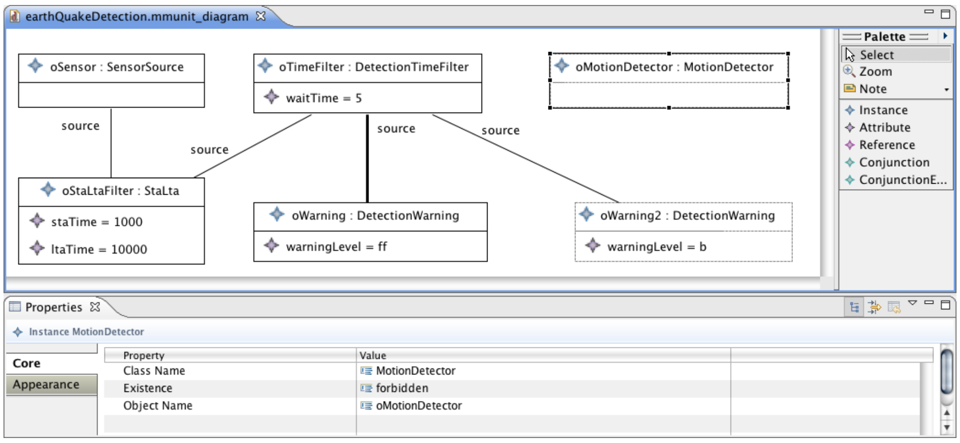


Figure 4: A first whiteboard sketch of a model expressed in a stream-oriented DSML.

The intention behind this model is as follows: Sensor readings from an *acceleration sensor* are piped through a filter that realises the *STA/LTA detection*. The filter forwards sensor readings that are considered to be the beginning of an earthquake and blocks all others. The frequency of sensor readings is limited by another filter, *detection time filter*, before they stream into a stream sink that generates an earthquake *detection warning* whenever a sensor reading streams in, for example by activating a warning horn. The filters and the sink contain attributes influencing their behavior.

The language engineer prefers a test-driven development. Therefore, he creates a metamodel test *before* he creates the metamodel. For this, he derives a test model from the model he and the seismologists sketched on the whiteboard. The result is shown in Fig. 5.[2] The four instances on the left reproduce the model sketch. The *positive* example model specified with the test model consists of only these objects. The test model also specifies three *negative* example models: (1) Each sink must have a reference to a stream source. Therefore, the language engineer sets the existence specification of the reference from *oWarning* to *oTimeFilter* to "enforced". This describes a negative example model in which the reference is missing. (2) Seismologist and language engineer discussed but discarded the idea of a motion detector filter. To ensure that the final metamodel does not support a motion detector, the language engineer adds the forbidden instance *oMotionDetector*. The corresponding negative example model contains this additional instance. (3) Each source must be referenced by exactly one sink. To test this, the language engineer adds the forbidden instance *oWarning2*. Again, the corresponding negative example model contains this additional instance.

[2]The notation we use for test models is similar to that of UML object diagrams. Additionally, enforced elements are marked with a thick border, forbidden elements with a dashed one.



Instance MotionDetector

Property	Value
Class Name	MotionDetector
Existence	forbidden
Object Name	oMotionDetector

Figure 5: Screenshot of the test model for the earthquake detection metamodel. (The test model editor is part of our prototype implementation MMUnit.)

After specifying the test model, the language engineer creates a metamodel for the stream-oriented language (Fig. 6). In order to execute the tests specified by the test model, the language engineer uses MMUnit to generate corresponding JUnit test cases. The generated JUnit tests use a library that implements the test process as described in Sec. 3. Executing the JUnit tests reveals an error: The negative test model that contains the additional instance *oWarning2*, case (3), is not rejected as an instance of the metamodel. The language engineer realizes that he forgot to set the multiplicity of the association between *Source* and *Sink* to 1 on the *Sink* end. He corrects the error and executes the tests again. Now, all tests pass.

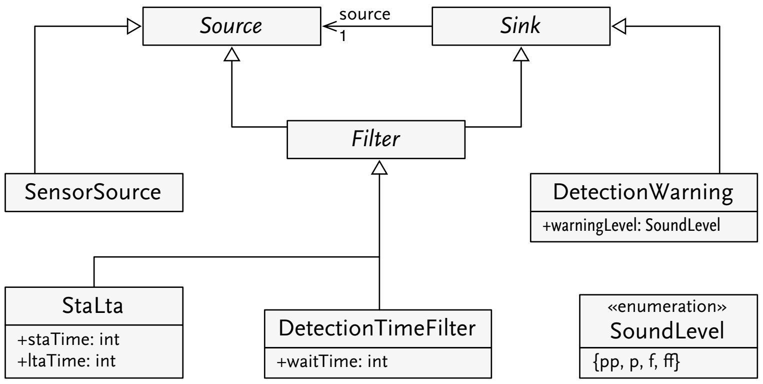


Figure 6: A proposal for a domain-specific metamodel.

5 Related Work

In model-based testing, many approaches use models as specifications to generate test cases for a system under test (SUT) [NF06, PP05, OA99, AO00]. The tests check if the SUT satisfies all constraints of the model. The models themselves are assumed to be correct, whereas we want to test the correctness of (meta-)models.

Tests for model transformations are handled in [Küs06, WKC06, BFS$^+$06]. They all assume that the used metamodels are correct and they focus on testing the transformation process between them. Our approach is complementary to their approaches as it tests the metamodels they assume to be correct.

In grammar testing [Pur72], character sequences are used to test a developed grammar [Läm01]. This generic approach permits to define both words that conform to the grammar and words that do not. While our metamodel also allows to generically describe instances, we target metamodels, not grammars.

6 Conclusion and Future Work

Conclusion. Metamodels play an important role for the definition of the abstract syntax of a DSML. In this paper, we argued that metamodels should be tested systematically. We proposed an approach for testing metamodels and exemplified it with tests of a metamodel for a stream-oriented DSML. Our approach is based on understanding metamodels as set specifications. Our idea is to use example models that may lie either inside or outside of the set specified by the metamodel.

We already did a prototypical implementation of our approach, which we sketched shortly in this paper. It is based on the Eclipse Modeling Framework (EMF). The prototype is called *MMUnit* (`http://mmunit.sourceforge.net`) and provides an editor for test models and can generate JUnit tests from test models. Such a generated JUnit test reads a test model and checks if the described positive example model is an instance of the MMUT and if the described negative example models are not instances of the MMUT. If both checks pass, the test succeeds; otherwise it fails.

Metamodel tests are possible. They can be specified quite easily. By the integration with JUnit, metamodel tests can be executed automatically. Thus, metamodel tests can be integrated into existing software development processes (e.g. metamodel tests can be executed as part of a continuous integration build).

Future work. Currently, our implementation tests classes, attributes, and associations of a metamodel together with their multiplicities. Usually, a constraint language like OCL is used to constrain the set of possible models. We plan to extend our implementation to support the evaluation of OCL constraints during test execution.

Another restriction in our current approach is that a test model always describes exactly one positive example model. We think that one may also want to describe multiple positive example models that differ only slightly or one may also want to describe negative example models only. For this, we could extend the test metamodel with an attribute that states whether the test model describes a positive or a negative example model as the base case. Furthermore, we could add another enumeration value for existence specifications that allows for specifying that a model element can be removed or left in the example model without influencing whether the example model is a positive or a negative one.

We left open whether metamodel tests pay off economically? To answer this question, systematic case studies are necessary.

Acknowledgments. We would like to thank the reviewers for valuable comments. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

References

[AO00] Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In *UML 2000*. University of York, UK, 2000.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[Bei90] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., 1990.

[BFS$^+$06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2006. IEEE Computer Society.

[Küs06] Jochen M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, V5(3):233–259, 2006.

[Läm01] Ralf Lämmel. Grammar Adaptation. In José Nuno Oliveira and P. Zave, editors, *FME'01*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.

[NF06] Clementine Nebut and Franck Fleurey. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, 2006.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *UML'99 — The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999.

[PP05] Wolfgang Prenninger and Alexander Pretschner. Abstractions for Model-Based Testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[Pur72] Paul Purdom. A sentence generator for testing parsers. *bit*, 12(3):366–375, 1972.

[Ste77] S. W. Stewart. Real time detection and location of local seismic events in central California. In *Bull. Seism. Soc. Am.*, volume 67, pages 433–452, 1977.

[WKC06] Junhua Wang, Soon-Kyeong Kim, and David Carrington. Verifying Metamodel Coverage of Model Transformations. In *ASWEC'06*, 2006.