Meta-Model Validation and Verification with metaBest

Jesús J. López-Fernández, Esther Guerra, Juan de Lara
Universidad Autónoma de Madrid (Spain)
{Jesusj.Lopez, Esther.Guerra, Juan.deLara}@uam.es

ABSTRACT
Meta-models play a cornerstone role in Model-Driven Engineering as they are used to define the abstract syntax of Domain-Specific Modelling Languages, and so models and all sorts of model transformations depend on them. However, there are scarce tools and methods supporting their validation and verification, which are essential activities for the proper engineering of meta-models.

In this paper we present an Eclipse-based tool that aims to fill this gap by providing two complementary meta-model testing languages. The first one has similar philosophy to the xUnit framework, enabling the definition of meta-model unit test suites comprising model fragments and assertions on their (in-)correctness. The second one is directed to verify expected properties of the meta-model, including domain and design properties, quality criteria and platform-specific requirements. Both tools are integrated within a framework for example-based, incremental meta-model development.

Categories and Subject Descriptors
D.2.4 [Software]: Model checking; I.6.4 [Computing Methodologies]: Model Validation and Analysis

Keywords
Meta-modelling; Validation and verification (V&V); Meta-model testing; Example-based meta-modelling

1. INTRODUCTION
Model-Driven Engineering (MDE) promotes the use of models and transformations throughout all phases of software development. Models are frequently defined using Domain-Specific Languages (DSLs), which previously need to be constructed in collaboration with domain experts. The abstract syntax of DSLs is described by a meta-model that includes the relevant abstractions, primitives and relations within the domain. Hence, it is important to validate the DSLs w.r.t. specifications of the domain, or with the help of domain experts who can provide meaningful examples of correct and incorrect uses of the DSL. Moreover, meta-models are normally defined using an object-oriented approach, and implemented in specific platforms like the EMF [13]. Hence, they should adhere to accepted object-oriented quality criteria and style guidelines in conceptual schemas [1], as well as to framework-specific rules and conventions.

Unfortunately, while meta-models play a central role in MDE, they are often built in an ad-hoc way, without following a sound engineering process [8]. This lack of systematic means for their construction may lead to unreliable results, with the aggravating factor that errors in meta-models may be propagated to all artefacts developed for them, like model transformations and code generators. Part of this situation is due to the fact that there are scarce methods and tools to validate and verify meta-models against domain requirements, quality guidelines and platform-specific rules.

In order to fill this gap, we deliver metaBest, an Eclipse-based tool that facilitates the integral testing of meta-models by making available two dedicated testing languages. The first one, called mmUnit, is inspired by the xUnit framework [2], as it enables writing conforming and non-conforming model fragments to check whether the meta-model accepts the former and rejects the latter. The language, which was initially proposed in [7], has been recently extended with an assertion language tailored for meta-models. The second language, mmSpec, is conceived to test expected meta-model properties that may arise from the domain, the implementation platform, quality criteria and style conventions.

metaBest is integrated with an example-based meta-model construction framework [7], which facilitates the involvement of domain experts in the DSL construction process. Although our tool is directed primarily to DSL designers, the tool is kept independent from the meta-modelling platform, so that it can also be used within the wider scope of software design, e.g., to validate and test UML conceptual schemas for information systems.

Paper organization. Sec. 2 analyses the state of the art, motivating the need for metaBest. Sec. 3 shows the tool with an example and Sec. 4 poses conclusions and future work.

2. METABEST CONTRIBUTION TO V&V
The classical view of V&V [3] poses validation as the answer to the question “are we building the right meta-model?”, while verification is set to address “are we building the meta-model right?”. The literature reports on three main approaches to meta-model V&V, which we classify as unit testing, specification-based testing and reverse testing.
Unit testing approaches define test suites of models or model fragments, which get validated against a meta-model. For example, in [11], test models describe instances that the meta-model should accept or reject. In a different style, [5] embeds meta-modelling languages into a host programming language like PHP, and then inject the meta-model back into a meta-modelling technological space. While this allows the use of existing xUnit frameworks, it resorts to a programming language to build meta-models. Moreover, these proposals lack an assertion language tailored to meta-modelling language. While this allows the use of existing xUnit frameworks, it resorts to a programming language to build meta-models. Moreover, these proposals lack an assertion language tailored to meta-model testing which enables an intensional description of the test models, documenting and narrowing the purpose of the test.

Specification-based testing allows expressing desired properties of a meta-model. In this line, [12] specifies meta-model properties in EVL [6] (a variant of OCL), but as the authors recognise, using EVL/OCL to check meta-model properties is cumbersome, leading to complicated assertions and demanding expert technical knowledge of the used meta-modelling framework. Moreover, OCL does not provide support for visualizing complex validation errors.

Reverse testing automatically generates instance models from a meta-model, e.g., using constraint solving [4]. A domain expert has to evaluate the generated models to detect errors, in which case the meta-model is deemed incorrect.

An integral approach to meta-model V&V needs to encompass the benefits of all mentioned approaches, and fill the following gaps in the state of the art. First, although there are a few specification-based testing approaches that support the specification of requirements (validation) and meta-model quality concerns (verification), they rely on OCL, which is not optimal for expressing meta-level properties and does not provide appropriate support for error visualization (sets of problematic elements) and reporting. Second, no meta-model unit testing proposal allows describing the intension of the expected faults using an appropriate assertion language, or supports user-friendly definitions of model fragments. Additionally, even if approaches for unit testing are suitable for test-driven development, they sometimes lack means to construct faulty models, as frameworks like EMF require correct models and building the meta-model upfront.

Hence, metaBest promotes an iterative, incremental meta-model construction, and enables domain experts to provide sketches that cover interesting features of the system. Additionally, our tool also allows importing existing models in Ecore format. Independently of the meta-model construction process – either from example models or not – the meta-model can be tested using our two testing languages. We illustrate both languages in the next two subsections.

3. METABEST BY EXAMPLE

Assume we are interested in modelling Data-as-a-Service (DaaS) applications, and need to define a DSL for this. In DaaS applications, the data is the product offered to users, who are charged by their consumption and manipulation.

Using our metaBup tool, the domain expert can introduce fragments of example models of the DSL as sketches, like the one shown at the bottom-left in Fig. 1. This sketch corresponds to a situation of interest, where a role having access rights to a read operation on some data resource, also provides an access right to that data resource. Sketches can be drawn using traditional drawing tools like yEd (in the figure) or Dia. The sketch is parsed into an internal textual format (upper-left in Fig. 1), and the tool induces and updates the underlying meta-model, suggesting the user patterns and notions of quality. The domain expert also needs to provide a legend assigning a name to each different kind of object appearing in the sketch, which will be used as the name of the classes in the derived meta-model.

Fig. 1 shows a simplified meta-model for DaaS applications. A DaaSApplication contains Users who may access data and functionality according to the Roles they have been assigned. In particular, AccessRights grant access to users with a certain role to some Resources (either data or services) and Operations on data (like Read and Update). Applications organize data and functionality in ResourceContainers, while ServiceUnits perform operations on a DataResource.

Thus, metaBup promotes an iterative, incremental meta-model construction, and enables domain experts to provide sketches that cover interesting features of the system. Additionally, our tool also allows importing existing models in Ecore format. Independently of the meta-model construction process – either from example models or not – the meta-model can be tested using our two testing languages. We illustrate both languages in the next two subsections.

3.1 Example-based meta-model unit testing

Our first testing language, called mmUnit, allows the definition of test cases. Each test case includes a configuration of objects, which can be defined either using a dedicated textual syntax, or a sketching tool like yEd. In the latter case, sketches are imported and automatically translated into this textual format to facilitate their subsequent processing. In order to allow building more intensional tests, for the structural part, we support both examples of full-fledged models and model fragments. Fragments may miss certain mandatory objects and attributes, and violate the lower bound of cardinalities, as their purpose is concentrating in the nearby context of a particular situation of interest.

As an example, the upper window in Fig. 2 shows a sketch of a model fragment that has been drawn using yEd, and part of the equivalent textual format once the sketch is imported in metaBest (lines 12–23). The test includes assertions (lines 25–28) stating why the situation is incorrect. Once the test is ready, the tool checks it against the meta-model in Fig. 1, and provides a report view with the results.
For instance, in our running example:

among them, the requirements elicited from domain experts.

meta-model properties. These may come from several sources,

mmSpec language is targeted to express and evaluate expected

terms of meta-modelling concepts, thus bridging the gap

between requirements in natural language and the meta-

modelling space. For example, for Rq3, we need to check

that the classes reachable from DaasApplication inherit from

ChargeableElement, or contain classes inheriting from it.

Moreover, the meta-modelling expert may also like to en-
sure certain quality attributes and platform-specific require-
ments in the meta-model, as well as to adhere to standard
guidelines and style conventions. For instance:

Rq2 Applications contain at least one chargeable element.

Rq3 All elements in an application are chargeable, or con-
tain chargeable elements.

Rq4 The access to any element with a URL needs to be
controlled.

These high-level requirements need to be interpreted by
the meta-model designer, who needs to formulate them in
terms of meta-modelling concepts, thus bridging the gap
between requirements in natural language and the meta-
modelling space. For example, for Rq3, we need to check
that the classes reachable from DaasApplication inherit from
ChargeableElement, or contain classes inheriting from it.

Figure 2: Definition and evaluation of test case.

A distinguishing feature of mmUnit is that it allows declar-
ing the reasons why a test case is deemed incorrect by asserting
errors in meta-models. It supports the following types of assertions:

- **Multiplicity, type or nature mismatch** of some feature.
- **The test includes instances of abstract types**.
- **Missing meta-model elements** (i.e., the meta-model lacks
  the type of an object or a particular feature).
- **Missing mandatory feature** on an object.
- **Missing incoming reference or container** for an object.
- **Constraint violation**. metaBup fragments and meta-models can be annotated to constrain the models considered
  valid. For instance, any reference annotated with acyclic
  in the meta-model is forced to be acyclic in every model.
  This assertion kind points to violations of such annotations.
  The complete list of supported assertions is det-
tailed in [7].

The first assertion in Fig. 2 states that the test should fail
because object dr should be contained in some object of type
ResourceContainer. The assertion does not explicitly say the
object container, which could be any in the model fragment.
The next one states that service should not have an access
fee, since its type ServiceUnit does not define it. The last
assertion identifies that the operation object lacks an access
fee. In this case, the test passes because the meta-model
rejects the model for exactly those reasons.

A video featuring this example is available at: http://
youtu.be/fC8J5YkPCHE.

3.2 Specification-based meta-model testing

While the previous language allows testing the conformance
of models to meta-models, our second testing language
mmSpec is targeted to express and evaluate expected
meta-model properties. These may come from several sources,
among them, the requirements elicited from domain experts.
For instance, in our running example:

Rq1 Data resources must be accessible by users with access
rights.
(selector) contained in a DaaSApplication (filter), and then checks whether they are subclasses of ChargeableElement or they reach a class that is subclass of it (condition).

Properties can be assigned a name, organized in groups, and annotated with a severity (e.g., warning). Named properties may define parameters and then be called considering quantifiers in the parameters.

The lower window of Fig. 3 shows a summary of the evaluation of all properties in the meta-model, another feature of metaBest. For each property, it is possible to obtain a user-friendly visualization of its results over the meta-model: the faulty elements are shown in red, the correct ones are green-coloured, and elements that triggered a warning are amber-coloured. In our example, the classes Role and User do not meet Rq3, and hence they are coloured in red when the user double-clicks that property. Notice that Resource is not highlighted, since it actually meets the property, containing one of its children (ServiceUnit) a chargeable element (Operation).

An alternative to mmSpec would be the use of the more general-purpose OCL [10] to express meta-model properties. However, although OCL is richer and more expressive than mmSpec, it was not designed for meta-model testing, and thus lacks many of the high-level primitives we provide for this task. As a result, OCL properties tend to be more complex, while our language yields more compact expressions, closer to natural language, and more understandable by non-meta-modelling experts. In http://goo.gl/6eVBgy, we provide a thorough comparative study of OCL and mmSpec. Moreover, the integration with WordNet and the rich reporting facilities are unique in our proposal to specification-based testing. A video featuring this example is available at: http://youtu.be/V8t0HT-rw7k.

4. CONCLUSIONS AND FUTURE WORK

We have presented metaBest, a tool for V&V of meta-models. The tool integrates two DSLs: mmUnit and mmSpec. The first one permits defining valid and invalid examples and fragments, and enables an intentional description of the reasons why an example is invalid. Its importer of graphical sketches encourages the engagement of domain experts in the V&V process. The second language enables a succinct expression of expected (domain, quality, style and platform) meta-model properties and automates their checking. The tool is integrated with metaBup, which permits a test-driven development approach to meta-model construction.

We are working on more advanced ways to annotate graphical sketches with errors, and on quick fixes suggested upon test failures. We also plan to support reverse testing using model generation by constraint solving.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitiveness with project Go-Lite (TIN2011-24139) and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).

5. REFERENCES