Combining Unit and Specification-Based Testing for Meta-Model Validation and Verification

Jesús J. López-Fernández, Esther Guerra, Juan de Lara

Computer Science Department, Universidad Autónoma de Madrid (Spain)

Abstract

Meta-models play a cornerstone role in Model-Driven Engineering as they are used to define the abstract syntax of 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 (V&V), which are essential activities for the proper engineering of meta-models.

In order to fill this gap, we propose two complementary meta-model V&V languages. The first one has similar philosophy to the *xUnit* framework, as it enables the definition of meta-model unit test suites comprising model fragments and assertions on their (in-)correctness. The second one is directed to express and verify expected properties of a meta-model, including domain and design properties, quality criteria and platform-specific requirements.

As a proof of concept, we have developed tooling for both languages in the Eclipse platform, and illustrate its use within an example-driven approach for meta-model construction. The expressiveness of our languages is demonstrated by their application to build a library of meta-model quality issues, which has been evaluated over the ATL zoo of meta-models and some OMG specifications. The results show that integrated support for meta-model V&V (as the one we propose here) is urgently needed in meta-modelling environments.

Keywords: Model-Driven Engineering, Meta-Modelling, Domain-Specific Modelling Languages, Validation & Verification, Meta-Model Quality.

1. Introduction

Model-Driven Engineering (MDE) [13] is a Software Engineering paradigm that promotes an active use of models and transformations throughout all phases of software development. Hence, models are used to specify, test, simulate and generate code for the final application. The rationale of MDE is that models have a higher level of abstraction than code, with less accidental details, which promises higher levels of quality and productivity [48]. Sometimes, models are described using general-purpose modelling languages like the UML, but it is also frequent the use of Domain-Specific Modelling Languages (DSMLs) capturing the abstractions within a given domain in a more concise and intuitive way [25, 29].

Email addresses: jesusj.lopez@uam.es (Jesús J. López-Fernández), Esther.Guerra@uam.es (Esther Guerra), Juan.deLara@uam.es (Juan de Lara)

The creation of a new DSML involves the definition of its abstract syntax by means of a meta-model declaring the relevant primitives and relations within a domain. Hence, it is important to validate this meta-model with respect to specifications of the domain, or with the help of domain experts who can provide meaningful examples of correct and incorrect uses of the DSML. Moreover, meta-models are normally defined using an object-oriented approach and are implemented in specific platforms like the Eclipse Modelling Framework (EMF) [46]. Therefore, they should adhere to accepted object-oriented quality criteria and style guidelines in object-oriented conceptual schemas [2, 3], as well as to framework-specific rules and conventions.

However, while meta-models play a central role in MDE, they are often built in an ad-hoc way, without following a sound engineering process [33, 37]. This lack of systematic means for their construction yields non-repeatable processes that may lead to unreliable results, with the aggravating factor that errors in meta-models may be propagated to all artefacts developed for them, like modelling editors, model transformations and code generators. Thus, it is of utmost importance to deliver proven meta-models of high quality. Unfortunately, most efforts on DSML research focus on the implementation aspect, while neglecting domain analysis and DSML validation [31]. Hence, there are scarce methods and tools to validate and verify meta-models against domain requirements, quality guidelines and platform-specific rules.

To fill this gap, we propose two complementary languages and tool support for meta-model Validation and Verification (V&V). Our first language, called *mmUnit*, is similar to the xUnit framework [9]. It enables writing conforming and non-conforming model fragments to check whether the meta-model accepts the former and rejects the latter. Model fragments can be defined either using a dedicated textual syntax, or sketched by domain experts in drawing tools, thus involving domain experts in the meta-model validation process in a more direct way [26]. For non-conforming tests, it is possible to declare assertions that state the expected disconformities and reflect the intention of the test. This is useful for regression testing, when the meta-model evolves.

The second language, called *mmSpec*, allows expressing and checking expected meta-model properties that may arise from the domain, like the existence of a path of associations between two classes, and from the implementation platform, like the existence of a root container class (which is a common practice in frameworks like EMF). The language also permits the specification of quality criteria, like threshold values for the depth of inheritance hierarchies, and style conventions, like naming rules regarding the use of capitalized nouns for class names. The latter is enabled by the use of WordNet [38], a lexical database for English.

To analyse the usefulness of *mmUnit*, we compare it with the use of the JUnit testing framework to implement meta-model test cases. Moreover, we evaluate *mmSpec* from several perspectives. First, we provide a comparison with OCL to evaluate its conciseness and relative performance. To assess its expressiveness and usefulness, we have used *mmSpec* to develop a reusable library of 30 meta-model quality properties. The properties come from quality criteria of conceptual schemas [2], naming guidelines [3], or have been derived by us from experience. The library has been applied to a repository of 295 meta-models from the ATL zoo[1] and to a suite of 30 OMG specifications. The obtained results evince the need of this kind of support for the V&V of meta-models.

This paper extends our previous work with a detailed presentation of all features of *mmUnit* and *mmSpec* (which were only partially described in [35]), and proposing their integration with

[1] `http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore`.

an example-driven meta-model construction approach [33]. In addition, we evaluate several aspects of our languages like their usefulness, conciseness, expressiveness and performance. The analysis in [34] has also been extended to include standardized meta-models from the OMG.

The remaining of this paper is organized as follows. First, Section 2 reviews the state of the art on meta-model validation and verification, identifying existing gaps. Section 3 introduces a running example, and Section 4 overviews our approach. Then, Section 5 presents our language for example-driven testing, while Section 6 describes our language for testing expected meta-model properties. Section 7 shows the tool support and illustrates its use in an example-based process for meta-model construction. Section 8 includes an evaluation of our languages. Finally, Section 9 finishes the paper with the conclusions and future work. An appendix gives the detailed encoding of *mmSpec* primitives in OCL.

2. Approaches to meta-model Validation & Verification

Most efforts towards the V&V of MDE artefacts are directed to test model management operations [22], like model transformations [41], but few works target meta-model V&V. In this paper, we take the classical view of V&V [12]. Thus, while meta-model validation tries to answer the question "are we building the *right* meta-model?", verification addresses the question "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. This branch of works supports the definition of test suites made of models or model fragments, and their validation against a meta-model definition. This is the most usual approach, which follows the philosophy of the xUnit framework [9]. For example, in [43], test models describe instances that the meta-model should accept or reject. In a different style, [18] proposes embedding 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 enables the use of existing xUnit frameworks for meta-model testing, it resorts to a programming language for meta-model construction. The proposal presented in [40] is similar, but using Eiffel as host language. None of these works provides support for asserting the expected test results, though having an assertion language tailored to meta-model testing would enable an intensional description of the test models, documenting and narrowing the purpose of the test.

Other proposals [28, 49] expand general-purpose testing tools (e.g., JUnit) to enable the testing of DSML programs, not necessarily defined by meta-models. In [47], the authors present CSTL, a JUnit-like framework to test executable conceptual schemas written in UML/OCL. Test models in CSTL are described in an imperative way, lacking specialized assertions to check for disconformities. The *de facto* standard meta-modelling technology EMF [46] also provides some support for testing. Given a meta-model, the EMF synthesizes a Java API to instantiate the meta-model, as well as some classes to facilitate the construction of JUnit tests. Such tests must be actually encoded using Java and JUnit assertions by the meta-model developer. Java unit testing is also proposed in [6] as a way to test meta-models. However, we believe that it would be more helpful to have available higher-level assertions to express common failures in the modelling domain (like the lack of a container for an object) instead of lower-level generic assertions like `assertEquals` or `assertFalse`. Similarly, the availability of user-friendlier ways than Java code to specify tests, e.g., by means of graphical sketches, would help engaging domain experts in the meta-model validation process. In [42], the authors use the Human-Usable Textual Notation

(HUTN) to create EMF models that could be used in JUnit tests, instead of programmatically using Java. Still, this notation lacks support for dedicated assertions.

Specification-based approaches. While unit testing proposals work at the model level, specification-based testing approaches allow expressing desired properties of a meta-model. Following this line, [45] presents an approach for checking meta-model integration. It relies on specifying meta-model properties in EVL [30] (a variant of OCL), but as the authors recognise, using EVL/OCL to check meta-model properties is cumbersome, leads to complicated assertions, and demands expert technical knowledge of the used meta-modelling framework. Moreover, OCL does not provide support for visualizing complex validation errors.

Other works define catalogues of quality criteria for meta-models [11] or conceptual schemas. In [20], the authors express meta-model properties using QVT rules which create trace objects to ease problem reporting. However, rules still need to use the abstract syntax of MOF or UML, being cumbersome to specify and comprehend. Moreover, the same property needs to be encoded twice in order to be applicable to both MOF and UML. In [2, 4], quality properties of conceptual schemas are formalised in terms of quality issues, which are conditions that should not happen in schemas. The authors describe such conditions using OCL. In [3], the same authors propose a set of guidelines for naming UML schemas, which can be validated using an Eclipse plugin [1]. The drawback of these approaches is that the languages used to specify the meta-model properties (OCL, QVT) can be difficult to understand by domain experts. This is acceptable if the goal is to define libraries of quality properties for meta-models. However, if the goal is to state properties from the domain, it becomes useful to have a language where these properties can be naturally expressed, so that they can be more easily understood by domain experts.

Reverse testing approaches. These approaches are based on the automatic generation of instance models from a meta-model, likely using constraint solving [14, 21, 23, 50]. A domain expert has to inspect the generated models to detect invalid ones, in which case the meta-model contains errors. This approach is followed by [15] (where the generated snapshots are targeted to test cardinality boundaries) and works like [24, 36, 44]. To guide the model generation process, [24] provides a programming language to define object snapshots, [44] allows defining constraints captured by query patterns or OCL, and [36] provides a dedicated DSL to define seed models and properties that the generated models should or should not meet. In [32], the authors transform meta-models into OWL 2 and use reasoners to validate their consistency; however, their approach only reports the unsatisfiable concepts with no further explanations. In [5], questionnaires with true/false questions are generated from the meta-model, and the domain experts perform the meta-model validation by answering the questionnaires.

Challenges in meta-model V&V. We claim that an integral approach to meta-model V&V needs to combine all mentioned approaches. *Reverse testing* typically tackles validation by domain experts, and it requires a meta-model developed upfront. *Unit testing* integrates better with test-driven and example-driven development approaches (i.e., there is no need for a fully-developed meta-model beforehand), and it can be used to validate requirements and verify design concerns. Finally, *specification-based testing* can deal with the specification of requirements (validation) and meta-model quality concerns (verification).

However, we observe some gaps in the state of the art. First, the few existing *specification-based testing* approaches rely on OCL or QVT, which are not optimal to express meta-model properties and do not provide effective support for error visualization and reporting. Second, *unit*

testing approaches sometimes lack appropriate means to construct faulty models, as frameworks like EMF require correct models and building the meta-model upfront. Additionally, no proposal allows detailing the intension of the expected faults using a dedicated assertion language, or supports user-friendly definitions of model fragments. Thus, this paper proposes a novel integrated framework for the incremental construction and testing of meta-models, which comprises an example-based meta-model construction process, *specification-based testing*, *meta-model unit testing* and reporting facilities. We refer the interested reader to [36] for our contributions on *reverse testing*.

3. Motivation and running example

Assume we are interested in modelling Data-as-a-Service (DaaS) applications [16], for which we are building a DSML. In this kind of applications, the data is the product offered to users, who are charged by their consumption and manipulation.

Figure 1 shows a simplified meta-model for DaaS applications. A DaaSApplication has Users who may access data and functionality according to the Roles they have been assigned (Admin, Member or Guest). 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. Users are charged for the access to any ChargeableElement according to an accessFee. User Accounts accumulate the amount to be charged to the users, as well as payment details.

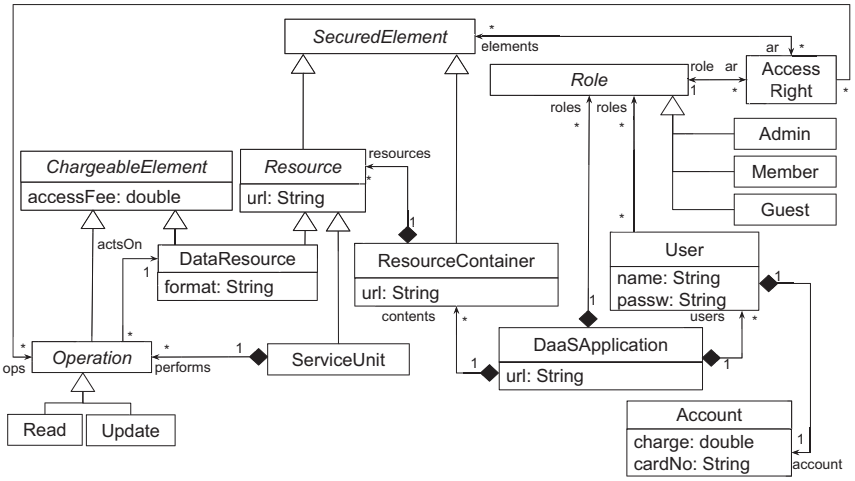


Figure 1: Meta-model excerpt for Data-as-a-Service applications.

Domain experts can provide requirements for the DaaS meta-model in different ways. First, as domain experts may not be meta-modelling experts, they can sketch examples of correct and incorrect configurations of DaaS applications. Figure 2 shows a sketch representing a situation of interest. It expresses that a (member) role with access right to a read operation on some data resource is invalid, unless the role also has access right to that data resource. Thus, this situation is undesirable because an access right to the data resource is missing. The challenge in this case is how to formalize this knowledge as a test that can be used to verify that the meta-model forbids

this incorrect configuration. This way of specifying requirements permits their checking using a unit testing approach.

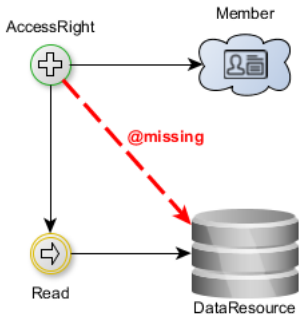


Figure 2: Test sketch.

Other requirements for the meta-model may be captured in natural language. For instance, domain experts in our example expect the DaaS language to ensure the following:

Rq1 *Data resources cannot be accessed directly by users, but through entities modelling access rights.*
Rq2 *Every application should contain at least one element with access charge.*
Rq3 *All elements in an application should be chargeable, or contain chargeable elements.*
Rq4 *Any DSML element with a URL should have a defined access right.*

Meta-modelling experts are also concerned with the quality of meta-models, which should fulfil certain quality criteria and standard style conventions, like:

Rq5 *No class is included in two containers.*
Rq6 *No inheritance hierarchy has a 5-level or greater depth.*
Rq7 *Every class name is a noun, possibly qualified, written in upper camel-case.*

Finally, the meta-model is to be implemented in EMF, which imposes some additional platform-specific requirements, like:

Rq8 *The meta-model needs to define a root class.*

Here, the challenge is to express these properties, to be checked on the meta-model, in a concise, intensional, platform-independent way. Verifying these properties manually by observation is hardly an option, because it is error prone for large, complex meta-models. Moreover, every time the meta-model evolves (which is frequent in test-driven approaches), all properties need to be manually checked again. A better solution involves automation and the generation of reports that document and trace the compliance with requirements. This permits automatic re-evaluation of the properties when the meta-model evolves. This style of expressing requirements enables their checking using specification-based testing approaches.

Using the standard Object Constraint Language (OCL) [39] to define meta-model properties is possible, but sometimes it leads to verbose, complex, poorly understandable, or hardly maintainable expressions. As an example, the next OCL expression can be used to check Rq3:

```
1  // Obtain classes (+ subclasses) reached from DaasApplication through containment relationships
2  EClass.allInstances()->select(c | c.name='DaaSApplication')
3  ->closure(c | EClass.allInstances()->select(c2 |
4        c.eAllReferences->select(r | r.containment=true).eReferenceType
5        ->exists(c3| c3=c2 or c2.eAllSuperTypes->includes(c3))))
6  // All should be chargeable ...
7  ->forAll(c | c.name='ChargeableElement'
8        or c.eAllSuperTypes->exists(sc | sc.name='ChargeableElement')
9  // ... Or contain chargeable elements
10       or Sequence{c}->closure(c2 | EClass.allInstances()->select(c2 |
11             c2.eAllReferences->select(r | r.containment=true).eReferenceType
12             ->exists(c3| c3=c2 or c2.eAllSuperTypes->includes(c3))))
13       ->exists (c2 | c2.name='ChargeableElement' or
14             c.eAllSuperTypes->exists(sc | sc.name='ChargeableElement')))
```

It is interesting to observe that this OCL expression contains recurring patterns for meta-model testing, like traversing inheritance hierarchies (lines 5, 8, 12 and 14) or the recursive navigation of references of a certain type (implemented using the primitive closure in lines 3–5 and 10–12). Moreover, in order to search elements in the meta-model, one has to use their exact name (lines 2, 7, 8, 13 and 14), whereas sometimes, more flexibility is needed to allow searching synonyms as well (e.g., the meta-model may use a synonym of *charge*, like *fee*). These should be ingredients of a Domain-Specific Language (DSL) for meta-model V&V. Finally, the expression assumes an Ecore meta-model, having to define a different expression if other meta-modelling technology (e.g., UML class diagrams) is used. This can hinder the development of reusable libraries to automate the validation of standard quality criteria and style guidelines.

Next, we present our approach targeted at solving these challenges.

4. Our integrated approach to meta-model validation and verification

The left of Figure 3 shows an overview of our proposal, where meta-models can be tested under two perspectives. The first one allows performing unit testing of meta-models with respect to valid and invalid test models, likely provided by domain experts in the form of graphical sketches [33]. A domain expert is a stakeholder with a sufficiently complete understanding of the field for which the DSML is being developed. The success of a DSML heavily depends on an active involvement of domain experts during its construction, and the provision of tests as sketches is a way to promote such an involvement.

The specification of test cases is supported by a DSL called *mmUnit*. Since test cases need to be intensional and focus on specific aspects of the meta-model under test, we support the provision of model fragments (i.e., not necessarily full-fledged models) containing only the information relevant for the test. While this approach promotes succinctness, its realization in standard EMF frameworks may be problematic because EMF models need to conform to their meta-models. In addition, *mmUnit* incorporates an assertion language to describe expected errors in test models and make explicit why a certain test model is incorrect. Section 5 will present all details of this language, while Section 8.1 will evaluate it with respect to using EMF for unit testing.

The second possibility (process "*property checking*" in the figure) concerns the V&V of meta-model properties. These can emerge from domain requirements elicited by domain experts, quality criteria adapted from object-oriented metrics [8], conceptual schema quality rules [2], naming style conventions [3], and platform-specific rules [46]. Although we could employ OCL to specify meta-model properties, we have seen that the resulting expressions may get cumbersome. Thus, we provide a DSL called *mmSpec* to describe meta-model properties in a more compact

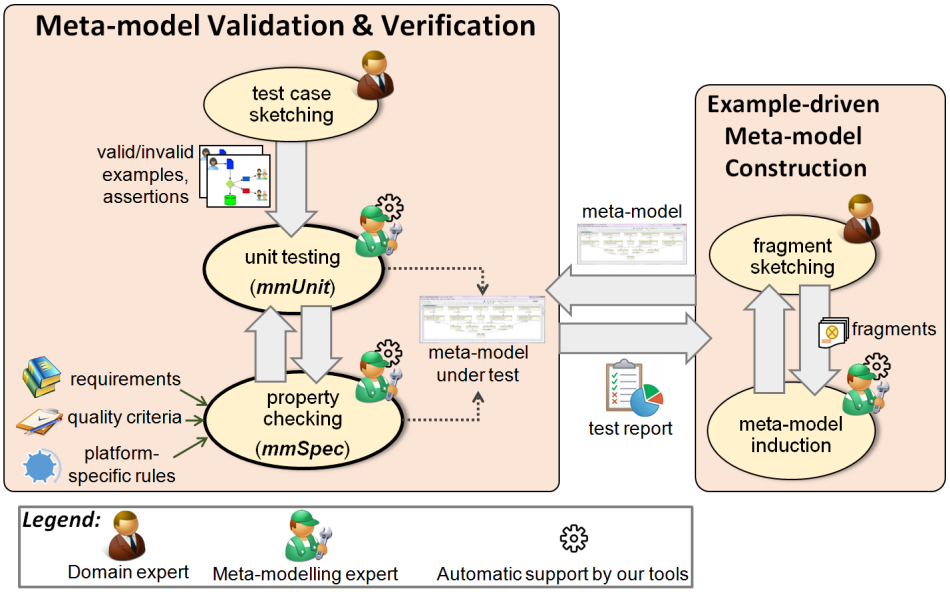


Figure 3: Our approach to meta-model V&V, and its integration with an example-based meta-model construction process.

and meaningful way. For this purpose, it includes high-level primitives for recurrent meta-model checkings, like the existence of a navigation path between two given classes. The features of this language will be explained in Section 6, while its differences with OCL will be discussed in Section 8.

Our languages can be applied in any scenario that requires V&V of meta-models. In particular, the use of examples to drive the incremental development of meta-models [33], schemas [27] and software designs [7] is increasingly favoured nowadays. Hence, in this paper, we show how to integrate our V&V languages in these example-driven approaches. The main idea is to use example models to automatically derive the meta-model needed for them.

The right of Figure 3 shows our example-driven approach to meta-model construction [33]. Any example-driven approach demands an example provider, which in our case is the domain expert. He provides example fragments of the DSML usage as sketches, thus participating actively in the DSML development. Sketches are used to automatically induce an appropriate meta-model for them. Figure 4 details this process. The fragments provided by the domain expert are supervised by a modelling expert, who can attach annotations to their elements to account for technical or more detailed constraints. The annotations may trigger refactorings and enforce certain design decisions when the meta-model is automatically induced from the fragments. Moreover, the derived meta-model can be monitored to detect locations where its design can be improved, in which case, the modelling expert is offered the possibility to apply an appropriate refactoring. This process is iterative, as new sketches can be provided at any time, likely producing an update of the meta-model under construction.

At any moment, it is possible to evaluate the correctness of the meta-model being constructed (tasks "*unit testing*" and "*property checking*" in Figure 3). Unit testing is enabled by our *mmUnit* language, which allows checking example models against the meta-model definition. This

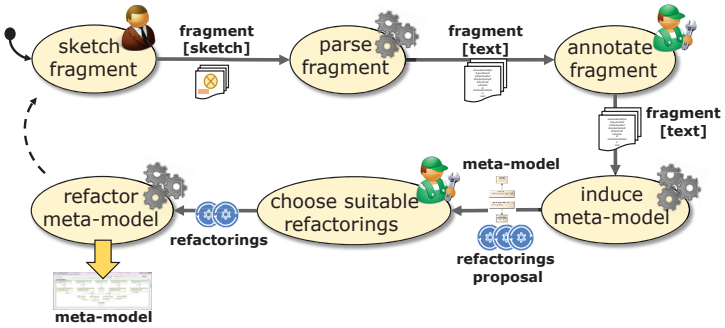


Figure 4: Steps in our example-based meta-model construction process.

can serve to identify meta-model errors as well as gaps in the provided set of examples. In case of gaps, new fragments that consider the missing elements should be provided to make the meta-model evolve. The definition and evaluation of meta-model properties is possible with our *mmSpec* language. While missing domain requirements can be covered by the provision of new fragments, quality issues and platform-specific rules can be tackled through the application of appropriate meta-model refactorings.

It is interesting to note that our V&V approach does not require the existence of a meta-model developed upfront to define tests or properties for the meta-model. This makes it possible to follow test-driven meta-model construction processes (in addition to example-based ones).

The next two sections describe the unit testing and property checking tasks, as well as the languages created for them.

5. Example-based meta-model unit testing

In our approach to unit-test a DSML, the domain expert defines test cases by means of sketches corresponding to either valid or invalid model examples. In case of invalid examples, the sketch may contain annotations that identify incorrect or missing elements. Sketches are imported into the system and used to evaluate the current meta-model version, as shown in Figure 5. Thus, this approach involves domain experts in the validation of the DSML, which is vital to build correct, useful DSMLs [26].

Internally, sketched fragments are parsed and converted into test cases expressed with our textual DSL for unit testing. Annotations indicating errors in the provided examples are translated into assertions in the created test cases. The modelling expert can use this DSL to define new tests, or to enrich parsed sketches with additional assertions. The availability of a language with assertions for the unit-test of meta-models is very useful in iterative processes of meta-model construction, as changes in the meta-model due to the provision of new fragments can be validated with respect to the unit test. This is in-line with modern, agile software engineering processes guided by tests [10].

Our DSL to define test cases is textual and its name is *mmUnit*. Each test case is made of a configuration of objects, and in case the configuration is deemed invalid, a set of assertions stating why it is incorrect. 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

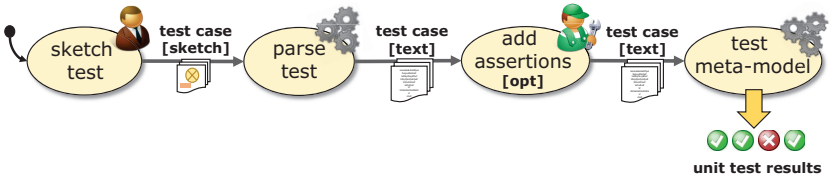


Figure 5: Unit testing of meta-models.

mandatory objects and attributes, and violate the lower bound of cardinalities, because their purpose is concentrating in the nearby context of a particular situation of interest.

Figure 6 shows an excerpt of *mmUnit*'s meta-model. It contains elements to explicitly represent models and meta-models, which provides flexibility to define non-conforming tests. An additional advantage is that meta-models from several technological spaces (e.g., UML class diagrams, CMOF models and EMF Ecore meta-models) can be imported into our neutral format, and so the tests become available for all of them.

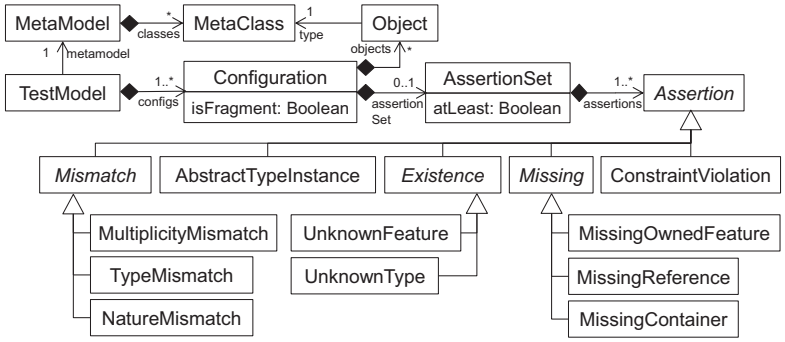


Figure 6: Excerpt of the *mmUnit* meta-model, including the hierarchy of assertions.

The language supports the following types of assertions for checking the correctness of the fragments and examples provided in the test cases:

- *Mismatch*. It states that a certain feature in the test case is in conflict with its definition in the meta-model in one of these aspects: multiplicity, type or nature (i.e., an attribute that should be a reference or vice versa).
- *Abstract type instance*. It signals the presence of an object whose type in the meta-model is abstract and hence cannot be instantiated.
- *Existence*. It states that the type of a certain object, or a particular feature, is not defined in the meta-model.
- *Missing owned feature*. It states that a certain object lacks some mandatory attribute or outgoing reference.
- *Missing reference or container*. It states that a certain object lacks some mandatory incoming reference, or is outside an appropriate container object. In both cases, we support

two possibilities: either indicating a particular source object, or the type that defines the reference. In the latter case, the assertion only fails if there is no instance of the specified type that refers to (or contains) the given object.

- *Constraint violation*. This assertion kind is specific of our example-driven meta-model construction approach, where fragments and meta-models can have attached annotations to constrain the models considered valid. For example, any reference annotated with acyclic should be acyclic, and to enforce this, we generate an appropriate OCL invariant. This assertion kind points to violations of such annotations. The list of supported annotations is detailed in [33].

Domain experts can define their own valid and invalid object configurations by means of graphical sketches drawn in tools like Dia[2] and yED[3], and import them as *mmUnit* test cases (i.e., as instances of the meta-model in Figure 6). The translation of a sketch into an *mmUnit* test case is done with the help of a legend, which is just another sketch that assigns a type name to the figures in the sketches. For instance, Figure 7 shows the legend that is used by all sketches in the running example.

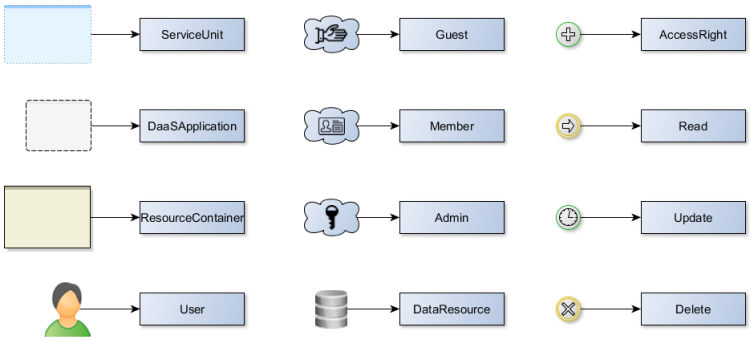


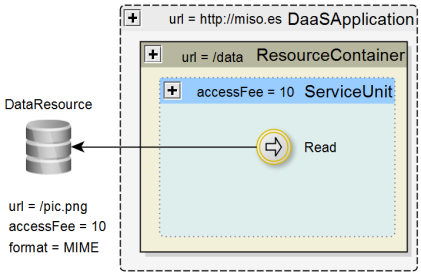
Figure 7: Legend for the sketches.

In addition, sketches can include annotations indicating the fault in the case of incorrect configurations. These annotations become translated into equivalent assertions in the derived test case. For instance, the word "missing" in Figure 2 yields a *missing reference* assertion in the generated test case. The modelling expert can specify additional assertions in the textual test case generated from the sketch, or directly in the sketches. He can also customise the keyword of the annotations (like "missing") that will be converted into assertions.

Figure 8 shows two test cases for the running example. The listing at the top-left has been automatically derived from the sketch in Figure 2. Line 1 indicates that the test includes a model fragment and therefore some mandatory elements may be missing. Lines 2–10 describe the fragment object configuration using a textual notation. Lines 12–13 include an assertion, generated from the reference "missing" in the sketch, describing why the fragment is invalid. Assertions can be inspected in two ways, controlled by the keywords fails and fails at least. The former is

[2] `http://dia-installer.de`
[3] `http://www.yworks.com/yed`

```
1  fragment MissingAccessRight {
2    access : AccessRight {
3      ref role = member
4      ref ops = read
5    }
6    member : Member {}
7    read : Read {
8      ref actsOn = dr
9    }
10   dr : DataResource {}
11
12   fails at least because:
13     missing reference from access to dr
14 }
```




```
1  example DanglingDataResource {
2    misoApp : DaaSApplication {
3      attr url = "http://miso.es"
4      @composition ref contents = resources
5    }
6    resources : ResourceContainer {
7      attr url = "/data"
8      @composition ref resources = ServiceUnit
9    }
10   service : ServiceUnit {
11     attr accessFee = 10
12     @composition ref performs = operation
13   }
14   operation : Read {
15     ref actsOn = dr
16   }
17   dr : DataResource {
18     attr url = "/pic.png"
19     attr accessFee = 10
20     attr format = "MIME"
21   }
22
23   fails because:
24     missing containment from some ResourceContainer to dr,
25     unknown service.accessFee,
26     missing attribute accessFee from operation
27 }
```

Figure 8: *mmUnit* test case for the sketch of Figure 2 (top-left). Another sketch (bottom-left) and generated test case (right).

normally used with examples, and it indicates that the subsequent list of assertions describes all the reasons for non-conformance. The latter is primarily used with fragments, and it indicates just the subset of reasons for non-conformance that are relevant for the intention of the test. In this case, the test fails because the meta-model accepts the fragment even if it is incorrect, i.e., the meta-model does not enforce that a read operation acts only on resources for which the executing role has access rights. This error can be solved by adding an appropriate OCL expression to the meta-model, or using the annotation cycleWith [33].

The right of Figure 8 shows another test, this time an example (i.e., a full-fledged model). The bottom-left of the same figure includes the sketch from which this object configuration was derived. This time, the assertions were encoded by the modelling expert after he imported the sketch. The assertion in line 24 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 resources or any other. The assertion in line 25 states that service should not have an access fee because its type ServiceUnit does not define it, while line 26 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.

6. Specification-based meta-model testing

In specification-based testing, the desired meta-model properties are specified and checked against an existing meta-model definition. The properties may come from domain requirements, design quality standards, style conventions and platform-specific rules. Domain properties describe DSML-specific requirements, like the need to uniquely identify objects of a given type, or

to navigate from some class to another in a limited number of steps. Design quality properties express well-known practices for object-oriented schemas, like avoiding deep inheritance hierarchies, or the fact that a class should be part of at most one container class. Style conventions refer to agreed naming styles, like the use of capitalized nouns for class names. Finally, platform properties refer to specific rules for a given meta-modelling platform. For example, EMF meta-models normally require a root class.

Making explicit the expected meta-model properties is useful in an incremental process for meta-model construction, as the properties can be rechecked every time the meta-model changes. Figure 9 depicts a scheme of the specification-based meta-model testing process. Addressing a failed property typically requires the provision of new fragments when the property captures a domain requirement, and a meta-model refactoring in the rest of cases.

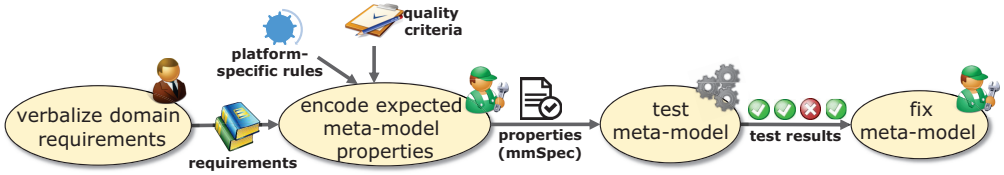


Figure 9: Specification-based testing of meta-models.

To facilitate the specification and testing of expected meta-model properties, we have created a DSL called *mmSpec* that covers all abovementioned kinds of properties: domain requirements, design guidelines, style conventions and platform rules. The language has been designed with simplicity in mind, adhering to a *select-filter-check* execution model for property definition. This style usually leads to structured descriptions of properties, closer to their formulation in natural language than OCL. The language makes available primitives for the main meta-model elements (classes, attributes, references and cardinalities), interesting derived relations (e.g., paths and inheritance hierarchies), and typical query patterns on those elements (e.g., reachability of classes, cyclicity and acyclicity of paths, depth of inheritance, and synonyms for class and feature names).

Figure 10 shows an excerpt of the *mmSpec* meta-model. The definition of each expected meta-model Property includes a Selector to select the set of elements (classes, attributes, references or paths) that meet certain filter criteria. These filtered elements are tested for the satisfaction of some condition (class Qualifier). Then, the number of elements satisfying the condition is compared against a quantifier (every, none, some or an interval) to assess whether the property is satisfied or not.

Overall, property specifications in concrete textual syntax have the following structure:

<quantifier> <selector>{ <filter> } => <condition>.

As an example, the property:

every class {abstract} => super-to some class{!abstract}.

uses the ClassSelector selector and the filter abstract. The condition super-to some class{...} is tested on the elements of the filtered selection, and the resulting set is checked against the quantifier every. In this way, this property checks if every abstract class has some direct or indirect concrete child class.

In order to define filters and conditions on elements, *mmSpec* makes available a hierarchy of specialized Qualifiers (omitted in the meta-model) for each type of element. Qualifiers can be negated using !, can be combined using and/or connectives, and can point to new selectors,

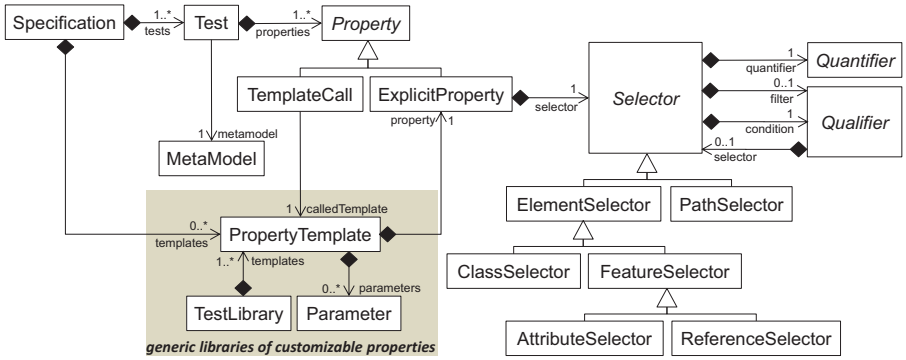


Figure 10: Excerpt of *mmSpec* meta-model, including the structure of properties.

enabling recursive checks. Table 1 lists the most relevant qualifiers, which can be used both as filter in selectors and as conditions. Some qualifiers support a set of prearranged modifiers, expressed between braces and separated by commas.

As the table shows, properties can look for synonyms and assert if a word is a noun, a verb or an adjective. This is possible because the language interpreter integrates WordNet, a database of the English language [38]. Additionally, properties can check the adherence of names to (upper) camel-case, the use of a given prefix or suffix, as well as testing the synonymy or grammatical form of each word in a camel-phrase. This latter feature facilitates a smooth encoding of requirements in natural language.

To deal with inheritance, there are primitives to obtain the super/subclasses of a class, optionally up to a given depth or width, as well as primitives to obtain the root/leaves of a hierarchy. For class reachability, collects calculates the overall cardinality of a composed path of relations, while the jump modifier constrains the path length, and cont considers containment relations only. For containment trees, there are primitives to test whether a class is container/leaf, and to get the absolute root of a tree. For paths, there are primitives to define the starting/intermediate/ending classes of the path, and to check for cycles (among others).

As an example, Figure 11 shows an *mmSpec* test suite with the domain requirements Rq1 to Rq4. The meta-model under test is referenced in line 2. The property for Rq1 (lines 4–7) states that every path from User to DataResource should go through class AccessRight. The property uses the path selector, with modifiers from and to to retain only the paths starting and ending in the given classes. The condition of the property checks that every such path goes through class AccessRight.

The property for Rq2 (lines 10–11) checks whether there is a class named DaaSApplication, from which there is a chain of containment relations with composed lower cardinality at least 1, leading to a class with an attribute ending in a synonym of charge. The property uses the collect primitive to calculate the composed cardinality, with modifier cont for containment.

The property for Rq3 (lines 14–17) first selects all classes directly or indirectly contained in DaaSApplication (line 14), and then, it checks that they are all children of ChargeableElement (line 16), or contain a subclass of ChargeableElement (line 17). Finally, the property for Rq4 (lines 20–23) selects all concrete classes owning an attribute named url, and checks that they define a direct reference to class AccessRight.

If we evaluate this set of properties in the meta-model shown in Figure 1, we discover that only Rq1 holds. Rq2 fails because although there is a navigation path from DaaSApplication to sev-

Qualifiers	Primitives	Modifiers	Description
Element Qualifiers			
Existence	exist	-	The simplest check, for ensuring the presence of elements.
Name	name	noun, verb adjective synonym prefix, suffix camel-phrase	The name of an element. It can be compared for equality with a given string, or to check whether it contains a given string as prefix/suffix/infix. It can be checked whether the name is a noun, a verb, an adjective, a synonym to another word, if it matches a pattern, or if it is in upper/lower camel case.
Class Qualifiers			
Abstractness	abstract	-	It states that a class is abstract (or the contrary with the ! operator).
Features	with	inh	It checks the existence of a reference or an attribute in a class definition. The modifier controls whether inherited features should be considered.
Inheritance	sub-to super-to	depth width or-equal	Set of direct and indirect subclasses or superclasses of a class. It is possible to constrain the depth and width of the inheritance hierarchy to consider (modifiers depth and width), and to include the class itself in the sub/super set (with the or-equal modifier).
Depth of hierarchy	inh-root inh-leaf	-	Depth of a class in the inheritance hierarchy, either from the root or from the leaves.
Depth of containment	cont-root cont-leaf	absolute	Depth of class in a containment hierarchy. It can be checked whether a class is a top container or a leaf. The absolute modifier, in combination with cont-root, checks whether the element is the meta-model root of the containment hierarchy.
Class reachability	reach reached-from collect	jumps cont inh strict	Set of classes that a given one can reach through navigation, or from which it is reachable. The collect primitive is used to check the composed cardinality of the traversed relations. The number and properties (e.g., containment, cardinality) of the traversed relations can be fine-tuned, as well as whether inheritance should be considered.
Feature Qualifiers			
Class	owned-by	inh	The class a feature belongs to (with/without inheritance).
Multiplicity	multiplicity	min max	Minimum and maximum multiplicity of a feature. In both cases, a fixed value or an interval can be given.
Attribute Qualifiers			
Type	type	-	The primitive type of an attribute.
Reference Qualifiers			
Reference ends	from to	inh	The source and target classes of a reference end.
Path Qualifiers			
Path ends	from through to	-	Paths starting, traversing or ending in the given classes. Any combination of primitives is valid.
Path settings	cont strict cycle	-	Characteristics of a given path. It can be restricted to containment relations or references only, it can consider subclasses as path nodes or not, and it can detect cycles.

Table 1: Main qualifiers for selectors and conditions.

eral classes with attributes ending in a synonym of charge (e.g., DataResource defines an inherited attribute accessFee), the lower bound of the composed cardinality is 0 instead of at least 1. This can be solved by increasing the lower bound of relations contents and resources. Rq3 fails because DaaSApplication contains non-chargeable elements like User. Since this is fine for the DSML, we should refine the property, restricting the need to be chargeable to subclasses of Resource. Finally, Rq4 fails for two reasons: first, because DaaSApplication has a url but no mandatory relation to AccessRight; second, because DataResource, ResourceContainer and ServiceUnit have a url and a relation to AccessRight, but is not mandatory. The latter can be fixed by setting the ar cardinality to [1..*]. The first reason for failure is not an error of the meta-model, but it requires refining the property to filter out class DaaSApplication.

In addition to domain-specific properties, which normally need to be defined anew for each DSML, *mmSpec* allows the creation of libraries of reusable parameterised property templates. In our current implementation, we provide a library with typical design quality criteria [20] and style guidelines [3] (see Section 8.4). As an example, lines 1–4 in Figure 12 show an excerpt

```
1  -- "domain"
2  test (metamodel "/dataServices/DataServices.mbup"){
3    -- "Rq1: Data resources cannot be accessed directly by users, but through entities modelling access rights"
4    every path{and{
5        from a class{name=User},
6        to a class{name=DataResource}}}
7    => through a class{name=AccessRight}.
8
9    -- "Rq2: Every application should contain at least one element with access charge"
10   a class {name=DaaSApplication}
11   => collect{cont} [1,*) of a class {with 1 attribute{name{suffix}=synonym{charge}}}.
12
13   -- "Rq3: All elements in an application should be chargeable, or contain chargeable elements"
14   every class{reached-from{cont} a class{name=DaaSApplication}}
15   => or{
16     sub-to a class{name=ChargeableElement},
17     reach{cont} a class {sub-to a class{name=ChargeableElement}}}.
18
19   -- "Rq4: Any DSML element with a URL should have a defined access right"
20   every class {and{!abstract, with 1 attribute{name = url}}}
21   => with a reference {and{to a class{name=AccessRight}, multiplicity{min=[1,*)}}}.
22 }
```

Figure 11: *mmSpec* test suite (domain requirements).

of the library, including the definition of a template called depthOfInheritanceTree. The template has a parameter to configure the threshold depth considered a bad design. Parameters may have a descriptive text (as in this case) to facilitate comprehension. Test suites can import libraries and reuse their templates as in line 13. When a template is called, it is possible to pass a set of elements in place of a parameter. For example, fun(every class{!abstract}) evaluates property fun for all concrete classes in the meta-model.

```
1  -- library quality.mbm
2  define depthOfInheritanceTree :
3    no class
4    => sub-to{depth= [<?:threshold, "Minimum forbidden Inheritance Depth">, *)} some class.
```

```
5  import "/quality.mbm"
6
7  -- "quality" @warning
8  test (metamodel "/dataServices/DataServices.mbup"){
9    -- "Rq5: No class is included in two containers"
10   no class => reached-from{cont, jumps=[1,1]} 2 class.
11
12   -- "Rq6: No inheritance hierarchy has a 5-level or greater depth"
13   depthOfInheritanceTree(threshold=5).
14
15   -- "Rq7: Every class name is a noun, possibly qualified, written in upper camel-case"
16   every class => name = upper-camel-phrase{ends{noun}}.
17 }
18
19 -- "EMF" @warning
20 test (metamodel "/dataServices/DataServices.mbup"){
21   -- "Rq8: The meta-model needs to define a root class"
22   strictly 1 class => cont-root{absolute}.
23 }
```

Figure 12: *mmSpec* test suites (style guidelines, and quality and platform requirements).

Lines 8–17 in Figure 12 show a test suite with properties derived from the quality requirements Rq5 to Rq7. The test suite is marked with warning, so that a warning (instead of an error) will be issued if some of the properties fails. The property in line 10 states that no class can be directly contained (modifiers jumps and cont) in two classes. The property in line 13 calls the depthOfInheritanceTree template with 5 as threshold value. The property in line 16 checks that every class has a name in upper camel-case, the last part being a noun. The meta-model in Figure 1 satisfies all these properties.

The suite in lines 20-23 contains one property specific for EMF which checks that there is strictly one absolute root class. The property fails because AccessRight is not contained in any class. This can be solved by adding a composition relation from DaaSApplication to AccessRight.

As a summary, we have seen that *mmSpec* enables the encoding of domain requirements for DSMLs, as well as the evaluation of quality and platform-specific properties of their meta-model. The possibility of automated checking is useful for regression testing, as the properties can be rechecked every time the meta-model changes.

7. Tool support

We have built an Eclipse plug-in, named *metaBest*[4], that integrates the presented languages and provides rich support to visualize the test results. Figure 13 shows a screenshot of the tool, where a sketched fragment is imported as an *mmUnit* test case in textual format, and then evaluated. Currently, our importer is able to parse Dia and yED drawings. The assertions starting from line 25 have been introduced by the modelling expert in the text editor, but as we explained in Section 5, they can also be derived from annotations in the sketch, like "missing" in Figure 2. The modelling expert can configure the sketch annotations that will generate assertions in the derived test case. The results of the evaluation are displayed in the lower view, where a user-readable sentence explaining why each property holds or not is produced. In this case, the three assertions of the unit test are true in the meta-model under test.

Figure 14 shows the evaluation of the *mmSpec* properties in our running example. The lower view includes a summary of the evaluation results of each test suite (domain, quality and EMF). Expanding each node of the tree displays the results for each property in the suite, including a description. For both passing and failing properties, it is possible to visualize the meta-model elements that fulfil or not the property: the faulty elements are shown in red, the correct ones in green, and those that trigger a warning are shown in amber. The meta-model in the figure shows the result of evaluating Rq4, where the concrete classes with a URL attribute (DaaSApplication, ResourceContainer, ServiceUnit, DataResource) are displayed in red because they don't have a compulsory reference to AccessRight. In addition, a test suite can be evaluated over a set of meta-models in batch mode, in which case, the results for each meta-model are persisted as a CSV file.

metaBest is integrated with our tool *metaBup* [33], which provides support for the example-based approach for meta-model construction presented in Section 4. Both tools use the same syntax and input type files, thus enabling a smooth integration of the testing tasks within the incremental meta-model development process. Figure 15 shows an example of this process, where the meta-model in the upper part is updated upon the import of an example fragment (a sketch). The sketch is parsed to obtain the equivalent textual representation, which has the same

[4]The tool and a screencast demo are available at `http://www.miso.es/tools/metaBest.html`.

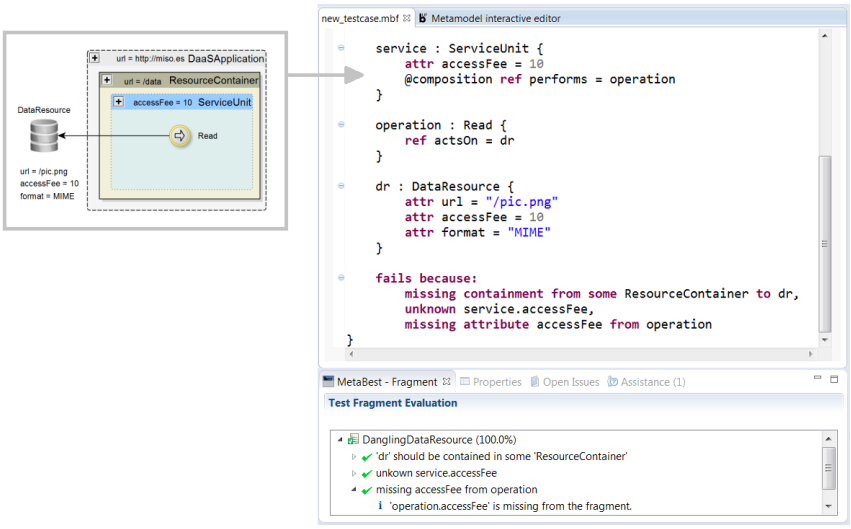


Figure 13: Evaluation and reporting mechanism of *mmUnit* test case generated from a sketch.

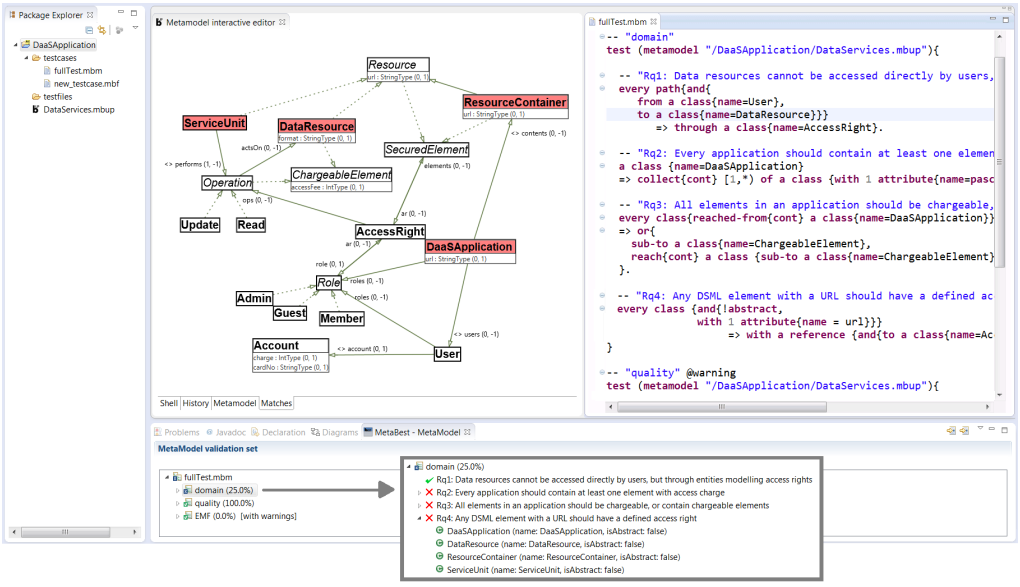


Figure 14: Evaluation and reporting mechanism of *mmSpec* specifications.

syntax used to specify fragments in *mmUnit* test cases. The textual representation can be edited by the modelling expert to include annotations that guide the meta-model induction process. In this example, the modelling expert has attached the annotation general(name = "Operation") to the object delete; in this way, when the meta-model is updated to include the new class Delete, this is created as a subclass of the existing class Operation. The annotation would have enforced the creation of a new class Operation in the meta-model if it didn't exist.

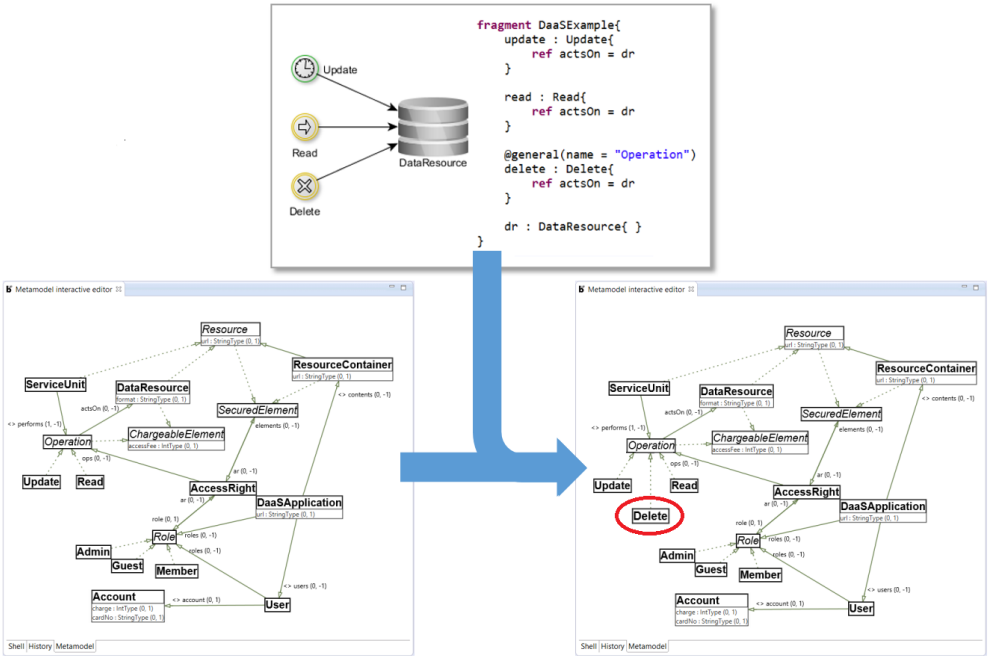


Figure 15: Example-based meta-model development with *metaBup*.

Alternatively, the meta-model under test can be imported from another technical space. We currently support the import and testing of Ecore, EMOF and UML meta-models.

8. Evaluation

In this section, we evaluate five aspects of our framework. First, we compare *mmUnit* with the available means for unit testing using the EMF library and JUnit. Second, we provide an analytical comparison of *mmSpec* and OCL to evaluate the complexity of *mmSpec* expressions with respect to their OCL equivalents. Then, we evaluate the performance of our tool for a representative subset of its primitives, and compare with their equivalent implementation using a standard OCL library for EMF. Next, we evaluate the expressiveness of *mmSpec* by presenting a reusable library that defines and enables the evaluation of 30 quality issues for meta-models, which have been specified as *mmSpec* properties. Finally, to assess the usefulness of our tool in current MDE practice, we present the results of applying our library of quality issues over a wide set of meta-models developed by third-parties. The obtained results show the need for integrated meta-model V&V support.

8.1. Evaluating usefulness: Comparison of mmUnit and EMF utilities

In this section, we compare *mmUnit* with the available means for unit testing in EMF [46], the de-facto standard for meta-modelling. EMF provides a library and infrastructure to create models and meta-models using Java. However, since it does not provide facilities for testing, developers typically resort to the JUnit framework for this task. Using JUnit, test model fragments can be

Assertion		Use case	Model edition	Model loads?	Assertion check (in Java)
Feature mismatch	Multiplicity	fn = 0 and mmMin >0	Tree editor	Yes	Diagnostic
		fn = n <m and mmMin = m	Tree editor	Yes	Diagnostic
		fn = n and mmMax <n	Tree editor	Yes	Diagnostic
	Type	Reference	Text editor	No	Capture 'IllegalValueException'
		Attribute	Text editor	No	Capture 'IllegalValueException'
	Nature		Text editor	Yes*	Cannot be checked with EMF
Abstract type instance			Text editor	No	Capture 'ClassNotFoundException'
Element existence		MetaClass	Text editor	No	Capture 'ClassNotFoundException'
		Feature	Text editor	No	Capture 'FeatureNotFoundException'
Missing feature		Owned	Tree editor	Yes	Diagnostic
		Incoming association	Cannot be inserted	N/A	Cannot be checked
		Incoming containment	Text editor	No	Capture'FeatureNotFoundException'
Constraint violation			Tree editor	Yes	Diagnostic

* The model is loaded, but it omits the faulty elements.

Table 2: Assertion of erroneous properties covered by *mmUnit*, and their resolution using EMF

built using the default EMF tree editor, then persisted in XMI (an XML-based format), and then loaded in the test cases. Alternatively, test models can also be constructed programmatically in Java within the tests. Either way, EMF expects correct models, and creating faulty models (as required by a complete unit test suite) is cumbersome, as we will analyse next.

EMF provides a Diagnostic class to detect errors in models. However, this class only detects the violation of cardinality and OCL constraints. Malformed models (e.g., using incorrect features or assigning incorrect values to features) raise runtime exceptions when they are loaded. This makes it difficult to introduce changes in the meta-model under development, as any defined meta-model instance may become incorrect due to these changes, being not possible to load it again.

To evaluate the effort of using JUnit to define meta-model tests and identify its limitations, we have tried to implement every *mmUnit* primitive using JUnit and EMF. Appendix B includes part of the Java implementation code, while Table 2 shows the equivalence of each *mmUnit* assertion and its encoding as JUnit tests. Column *Assertion* indicates the *mmUnit* assertion primitive being evaluated. Column *Use case* identifies particular cases of these primitives, where fn is the number of objects assigned to the checked feature, and mmMin and mmMax are the minimum and maximum feature cardinality in the meta-model. Column *Model edition* indicates whether a model having the conflictive property can be edited using the default EMF tree editor, which is the preferred option as it is user-friendlier. If it is not possible, then the model should be edited in XMI format with a text editor, which requires deep knowledge of EMF and XMI. Building malformed models (up to cardinality and OCL constraints) programmatically is not possible.

Once the test model has been created, there is the issue of whether it can be successfully loaded into memory (column *Model loads*), as EMF does not load models that define erroneous features. Finally, the last column *Assertion check (in Java)* shows the necessary tasks to accomplish the assertion in Java. If the model loads successfully, the Diagnostic class returns a text description of the error, which must be parsed to check whether the detected error corresponds to the evaluated one. On the contrary, if the load operation cannot be completed, a runtime exception is thrown. In such a case, one needs to study the exception stack to detect the reasons that impeded loading the model. In this way, the type of exception gives a clue on the produced error type, and then there is the need to dig into it to find out the object that triggered the exception.

As an example, Figure 16 shows to the left the *mmUnit* test case already shown in Figure 8, and to the right an attempt towards a similar test in EMF. We had to develop utility classes to load models (class MmUnitModelLoader) and emulate to some extent some of the *mmUnit* assertions (class MmUnitAssertion). Appendix B provides details of the developed library of assertions.

```
1  fragment MissingAccessRight {
2    access : AccessRight {
3      ref role = member
4      ref ops = read
5    }
6    member : Member {}
7    read : Read {
8      ref actsOn = dr
9    }
10   dr : DataResource {}
11
12   fails at least because:
13     missing reference from access to dr
14 }
```

```
1  public class AccessRightTests {
2
3    private String uri = "ResourceContainer.xmi";
4    private MmUnitModelLoader loader;
5    private MmUnitAssertion assertion;
6
7    @Before
8    public void load() {
9      loader = new MmUnitModelLoader();
10     loader.load(uri);
11     assertion = new MmUnitAssertion("DaaS.impl.",
12       DaaSFactory.eINSTANCE,
13       loader.getErrors());
14   }
15
16   @Test
17   public void testAssertion(){
18     assertion.assertMissingFeature( "AccessRight", "elements");
19   }
20 }
```

Figure 16: Example of *mmUnit* test (left) and corresponding EMF test (right).

In this example, we can see that the model fragment and the test are specified together in *mmUnit* (lines 2-10 in the left listing), but the model has to be loaded from an external XMI file in EMF (lines 9-10 in the right listing, where we have omitted the content of the file). This separation of the model under test and the assertions hinders understanding the rationale and objective of the test. Regarding *mmUnit* assertions (line 13 in the left listing), we emulate them in JUnit by using our library of assertions for EMF inside test methods (lines 17-19). The developed assertions use the Diagnostic class and rely on parsing their error messages, which makes testing less robust. Another drawback of the Java-based testing is the difficulty to refer to concrete objects in the model: while *mmUnit* assertions use identifiers access and dr to refer to objects, this is not possible in EMF, where the assertion just checks that there is a missing feature named reference from an object of type AccessRight. Finally, please note that the semantics of the *mmUnit* mechanisms fails at least because and fails because would need to be encoded in Java as well.
From the emulation of *mmUnit* using the JUnit and EMF frameworks, we have learned the following lessons:

- It is difficult to build erroneous (i.e., malformed) models in EMF since their editors are designed to handle only valid models, or models that violate cardinality or OCL constraints at the most. This can be observed in Table 2, by the fact that 7 out of our 13 assertion types cannot be checked on models constructed using the default EMF tree editor. Thus, in the cases when there is the need to introduce faulty features in models, it is necessary to editing XMI code directly.

- When a malformed model cannot be successfully loaded, it is only possible to obtain the first error occurrence. Contrarily, *mmUnit* is able to validate all assertions.

- EMF objects do not necessarily define a "name" attribute. Hence, it is difficult to refer to particular objects individually as we do with *mmUnit*, which simplifies the definition of assertions over them.

- Test models are not easily embedded in tests, so one needs to build them separately in a different environment. Alternatively, models could be implemented using plain Java,

though this is low-level and it is not possible to build malformed models.

- EMF does not provide friendly support for meta-model testing, as encoding assertions over models demands parsing complex textual error descriptions not intended for this purpose. In our experiment, we had to manually encode a library of model-specific assertions which inspect the error messages produced by the Diagnostic class.
- Finally, some *mmUnit* primitives cannot even be reproduced in EMF models, as they cannot be represented in XMI format, or the model loader cannot handle them.

Altogether, we can conclude that *mmUnit* supports a wider range of assertions and makes easy to specify both model tests and assertions in a unified way. The difficulty of accessing elements by name and specifying incorrect models is a strong drawback of directly using EMF for unit testing.

8.2. Evaluating conciseness: Comparison of mmSpec and OCL

mmSpec has been designed to facilitate the definition of meta-model properties by making available high-level primitives like path, inh and collect. To evaluate to which extent our language is concise, we have compared its primitives with their equivalent representation in OCL. We compare with OCL because this is the standard language proposed by the OMG for model queries [39], and a meta-model is just a model.

The appendix in this paper includes the translation of most primitives in *mmSpec* into OCL (36 cases in total). In this section, we comment on the most relevant observations.

First, only in three cases (1 to 3 in the appendix) the translation into OCL yields an expression with the same size (same number of tokens). These cases check the existence of a certain meta-model element kind (class, attribute or reference). While in *mmSpec*, such properties are specified using the expression some <element-kind> => exist, the encoding in OCL is <element-kind>.allInstances()->notEmpty(). Thus, even if they have the same size, the *mmSpec* formulation is more declarative, closer to natural language, and there is no need to manipulate object collections.

Other *mmSpec* primitives related to the nature or synonymy of words cannot be tested using OCL. In detail, it is not possible to check whether the name of a meta-model element is a verb/-noun/adjective, is synonym to a given word, or uses a camel/pascal phrase (see cases 8, 9 and 10 in the appendix).

In the remaining cases (which are the majority) *mmSpec* primitives are more succinct or intensional than the equivalent OCL expressions. This is especially the case for (see appendix): the primitive super-to, which checks the existence of a superclass in a hierarchy of a given length (case 16); the primitive cont-root to check if there is a top container class (i.e., it contains other classes and it is not contained in others, case 19); the primitive cont-leaf to check if there are leaf classes (case 20); the primitive reach with modifier jumps, which checks the reachability of a class in a given number of steps (case 24); the primitive reached-from with modifier jumps, which checks the backwards-reachability from a class in a given number of steps (case 26); and the primitive that checks the existence of a path starting and ending in some given classes (case 34), possibly traversing another third class (case 35).

In practice, expressing a single meta-model requirement often implies combining several *mmSpec* primitives and modifiers, in which case, the equivalent OCL expressions become more verbose as well. For example, Section 3 showed how to express Rq3 in OCL, while in comparison, the equivalent *mmSpec* property (lines 14–17 in Figure 11) is more compact and closer to

natural language. As an additional example, the following OCL expression checks Rq1. In this case, expressing the property in OCL is not direct, as the tester has to concoct a way to encode the requirement based on the reachability of classes. Instead, the *mmSpec* property (lines 4–7 in Figure 11) is more comprehensible and maintainable, due to the use of primitives for path quantification and analysis (e.g., through).

```
1 EClass.allInstances()->select(c | c.name='User')
2 ->closure(c | EClass.allInstances->select(c2 |
3       c.eAllReferences->select(r | r.eReferenceType.name<>'AccessRight').eReferenceType
4         ->exists(c3| c3=c2 or c2.eAllSuperTypes->includes(c3))))
5 ->select(c | c.name='DataResource')->isEmpty()
```

We believe that the reason why OCL gets more complex meta-model properties than *mmSpec* is that OCL was not explicitly designed to evaluate properties over meta-models, as *mmSpec* is. Thus, even if OCL can be used for querying meta-models, it lacks concise primitives to express meta-model facts, which have to be encoded as nested operations on collections, decreasing their understandability. However, we acknowledge that OCL is a richer, more expressive constraint language than *mmSpec*. Some of the features of OCL that we do not support are: explicit types of collections (Set, OrderedSet, Bag, Sequence), definition of variables, relational operators (e.g., union, difference, intersection), as well as arbitrary expressions through the use of collection operators like first, excludes, including and so on. However, our goal is not building a DSL with the same expressive power as OCL, but providing a small, optimized, compact language dedicated to expressing interesting meta-model properties. *mmSpec* provides high-level primitives to make this task easier, like first-order qualifiers for the length of navigation paths and hierarchies, or collectors of the composed cardinality in navigation paths. Finally, *mmSpec* is technology agnostic, meaning that the *same* property can be evaluated on Ecore meta-models, EMOF meta-models, and UML class diagrams. Using OCL, one should define different expressions to evaluate the same property with different meta-modelling technologies.

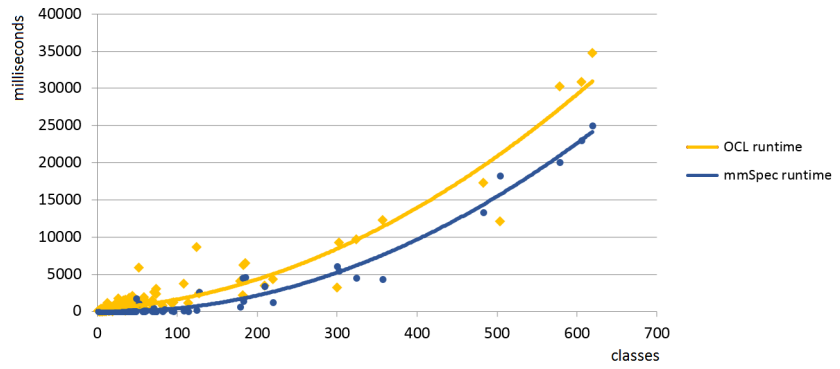
8.3. Evaluating performance: Comparison of mmSpec and OCL

Another interesting facet to look into is whether our language interpreter is capable of delivering runtime figures analogous to the standard OCL. To evaluate this aspect, we have measured the evaluation runtime of a set of properties expressed with *mmSpec* and their equivalent expressions in OCL. We have employed the native org.eclipse.ocl 3.4.2 OCL library, as it is based on Eclipse and therefore it has the same resources at its disposal as our *mmSpec* interpreter. The experiment was performed in a *Windows 7* Eclipse Kepler installation, run on an Intel(r) Core(TM) i7-3770 CPU with a 3.40 GHz processor and 8 GB of RAM.

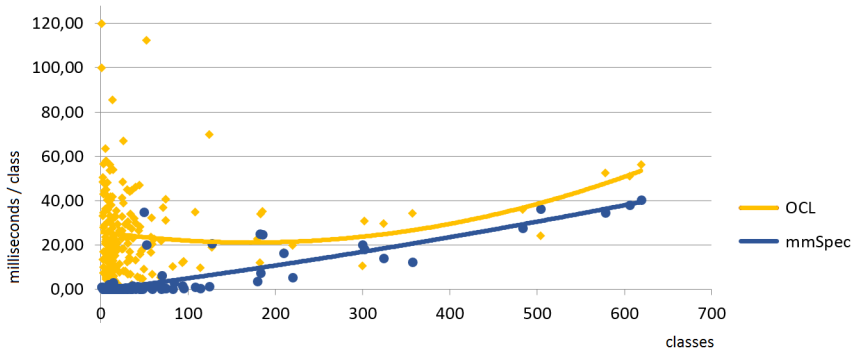
In our experiment, we have evaluated a test set of 33 meta-model properties over 201 Ecore meta-models of varying size and coming from two main sources: the ATL Ecore Zoo and OMG specifications. The meta-models were imported into the input format of *metaBest*. Then, for those meta-models, we evaluated the properties shown in the Appendix, excluding properties 8 to 10 because they cannot be expressed natively in OCL. These properties cover a wide range of *mmSpec* primitives on every kind of meta-model element (classes, references and attributes), and consider different kinds of relations like association, containment and inheritance.

Figure 17 includes some graphics with the obtained results. Overall, *mmSpec* had better performance than OCL for the set of analysed properties: the time to complete the evaluation of the properties in the meta-model test set was 143 seconds in the case of *mmSpec*, and 292 seconds for OCL. Figure 17(a) shows that the runtime increases in absolute terms as the size of the meta-model does. In the case of *mmSpec*, Figure 17(b) shows that the runtime increment is linear on

the number of classes, with an average runtime increment of 15.3 milliseconds per meta-model class.



(a) Performance in absolute time.



(b) Performance in relative time.

Figure 17: Comparison of performance in OCL and *metaBest*: runtime vs meta-model size.

If we look at each particular property in the test set, we find that the properties that calculate paths and class reachability are the most computationally costly (properties 19 to 26, and 33 to 36, in the Appendix). The average runtime per class to evaluate these properties was only slightly smaller in *mmSpec* (27.4 milliseconds) than in OCL (29.3 milliseconds). However, *mmSpec* spent 97% of the overall experiment runtime performing these operations, whereas OCL dedicated 51%. Thus, there is still room for studying how to improve the performance of our tool in these particular cases.

Concerning memory consumption, there was no significant difference between the two interpreters.

To conclude, a threat to the validity of our results is that it depends on the actual OCL encoding of the properties, and we might have not come up with the most efficient expressions. To mitigate this risk, we carefully designed the expressions trying to follow performance optimization patterns for OCL [19], like favouring the use of the exists(...) iterator to evaluate whether a collection contains an element with certain features, instead of using the equivalent but less

efficient expression select(...)->notEmpty(). In any case, results show good average performance for *mmSpec*.

8.4. Evaluating expressiveness: An mmSpec library of quality issues for meta-models

To demonstrate the expressivity of *mmSpec*, we report on a library we have built to discover typical mistakes that designers tend to commit, as well as others that may jeopardize a basic level of meta-model quality. The library contains 30 properties coming from several sources like [3, 4], or have been derived by our own experience. The library has four categories of issues, depending on their nature and relevance:

Design. Properties signalling a faulty design (i.e., an error).

Best practices. Basic design quality guidelines. Their violation is reported as a warning.

Naming conventions. For example, ensuring the use of verbs, nouns or pascal/camel case (warnings).

Metrics. Measurements of meta-model elements and their threshold value, like the maximum number of attributes a class should reasonably define. Most metrics in this category are adapted from the area of object-oriented design [17].

Table 3 lists the properties from these categories. To illustrate *mmSpec*'s expressiveness, next we show the formulation of one property from each category. Listing 12 (Rq8) showed the encoding of BP03 and a similar property to N04 (Rq7). The complete encoding of each property in the table can be found at `http://jesusjlopezf.com/metabup/quality-issue-catalogue/`.

- *D02: There are no isolated classes*. The encoding of this property is:

  ```
  no class => and { sub-to no class, super-to no class,
                    reach no class, reached-from no class}.
  ```

  The aim is to check the absence of classes that are not involved in any association or hierarchy. Thus, we use the no class selector, and check the following conditions: the class is orphan (qualifier sub-to with selector no class), childless (qualifier super-to with selector no class), contains no reference (qualifier reach with selector no class), and is not pointed by any other (qualifier reached-from with selector no class).

- *BP01: There are no redundant generalization paths*. This undesired situation arises when there are two or more inheritance paths from a subclass A to a superclass B. In the literature, this is sometimes called the "diamond problem", and it is problematic when two intermediate subclasses override a method of the superclass B, which then becomes ambiguous in the subclass B. The encoding of the property is:

  ```
  no class => sub-to{width=[2,*)} some class.
  ```

- *N09: No class is named with a synonym to another class name*. Having two different classes with synonym names can make the meta-model difficult to understand, ambiguous or redundant. *mmSpec* can detect such situations due to its integration with WordNet. The encoding of this property is as follows:

Code	Description
Design	
D01	An attribute is not repeated among all specific classes of a hierarchy.
D02	There are no isolated classes (i.e., not involved in any association or hierarchy).
D03	No abstract class is super to only one class (it nullifies the usefulness of the abstract class).
D04	There are no composition cycles.
D05	There are no irrelevant classes (i.e., abstract and subclass of a concrete class).
D06	No binary association is composite in both member ends.
D07	There are no overridden, inherited attributes.
D08	Every feature has a maximum multiplicity greater than 0.
D09	No class can be contained in two classes, when it is compulsorily in one of them.
D10	No class contains one of its superclasses, with cardinality 1 in the composition end (this is not finitely satisfiable).
Best practices	
BP01	There are no redundant generalization paths.
BP02	There are no uninstantiable classes (i.e., abstract without concrete children).
BP03	There is a root class that contains all others (best practice in EMF).
BP04	No class can be contained in two classes (weaker version of property D09).
BP05	A concrete top class with subclasses is not involved in any association (the class should be probably abstract).
BP06	Two classes do not refer to each other with non-opposite references (they are likely opposite).
Naming conventions	
N01	Attributes are not named after their feature class (e.g., an attribute paperID in class Paper).
N02	Attributes are not potential associations. If the name of an attribute is equal to a class, it is likely that what the designer intends to model is an association.
N03	Every binary association is named with a verb phrase.
N04	Every class is named in pascal-case, with a singular-head noun phrase.
N05	Element names are not too complex to process (i.e., too long).
N06	Every feature is named in camel-case.
N07	Every non-boolean attribute has a noun-phrase name.
N08	Every boolean attribute has a verb-phrase (e.g., isUnique).
N09	No class is named with a synonym to another class name.
Metrics	
M01	No class is overloaded with attributes (10-max by default).
M02	No class refers to too many others (5-max by default) – also known as efferent couplings (Ce).
M03	No class is referred from too many others (5-max by default) – also known as afferent couplings (Ca).
M04	No hierarchy is too deep (5-level max by default) – also known as depth of inheritance tree (DIT).
M05	No class has too many direct children (10-max by default) - also known as number of children (NOC).

Table 3: Library of meta-model quality properties.

```
define noSynonymClassNames:
  no class {!name = <?:className>} => name = synonym{<?:className>}.

noSynonymClassNames (className = every class).
```

Thus, the property uses a parameterized template noSynonymClassNames that receives one class as parameter, and checks that no other class (i.e., with a different name) has a synonym name. Then, this template is invoked with every class in the meta-model.

- *M01: No class is overloaded with attributes*. Even in large meta-models, classes with too many attributes often evidence a questionable design. While some entities in certain domains might carry a vast load of information, commonly, this data can be split into smaller entities that are arranged using inheritance or composition. Thus, the following property states that every class should have a maximum of 10 non-inherited (!inh) attributes. Thresholds are adjustable, but they have default values (10 in this case).

```
every class => with {!inh} [0, 10] attribute.
```

To build our library of properties, we considered sources like [3, 4]. We discarded some of those properties, namely, those that were UML-specific properties not shared with MOF, like disjointness of a generalization set, and therefore they did not apply to meta-modelling in general. Other

proposed properties cannot be automated, like detecting whether the name of a class is the most appropriate for the concept that the class represents. Nonetheless, of this exercise we can conclude that *mmSpec* provides sufficient expressivity for its practical use, as we have been able to encode all naming conventions suggested for classes, attributes and binary associations in [3], and 34 out of 44 (77.2%) quality issues for conceptual schemas presented in [4] (we exclude 21 UML-specific or non-automatable checkings from this count). From the 10 properties that we were not able to encode with *mmSpec*, 3 require using a constraint solver as they imply verifying the satisfiability of the meta-model, and the remaining 7 apply to meta-modelling elements currently not supported by our example-based meta-modelling framework *metaBest*, like derived attributes, explicitly defined data types, or arbitrary OCL expressions. We are currently working to add support for these features in our tool.

As a threat to the validity of our conclusions, there is the risk that *mmSpec* lacks further primitives (in addition to those mentioned above) for some relevant meta-model properties that could be evaluated in an automatic fashion. In such a case, this would require extending our language with these primitives. To mitigate the risk, we chose a set of properties developed by a third-party [3, 4], when we built our library of meta-model quality properties.

8.5. Evaluating usefulness: Applying the mmSpec library to meta-model repositories

To evaluate the need for our techniques and have a measure of the quality of current meta-modelling practice, we have applied our library of quality properties to a test set of 338 meta-models of varying sources, size and format. Our purpose is to have an evidence of the appearance of quality defects in existing meta-models, which can shed light on the need for V&V support in meta-model construction tools, like the one *metaBest* provides.

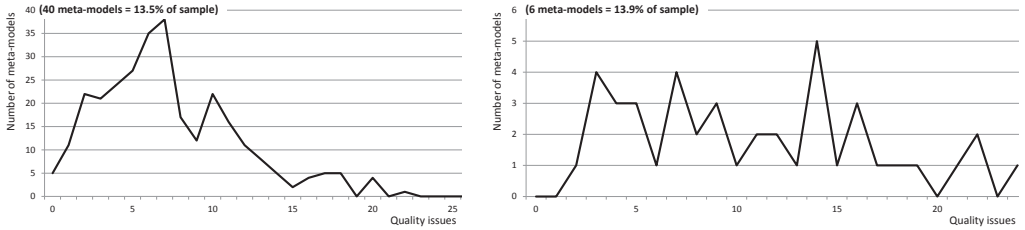
The meta-models used in this analysis come from two different sources: the ATL Ecore zoo (295 meta-models) and specifications of the Object Management Group (OMG[5], 43 meta-models). In particular, our analysis considers all meta-models that the ATL zoo contains, which are defined in Ecore format (i.e., the format used by EMF to store meta-models). Regarding the OMG, we have only included those specifications that make available a meta-model implementation in a format that can be imported by our testing tool (UML, EMOF, CMOF and Ecore). Note that the fact that our analysis has considered a variety of meta-model formats shows the reusability of our approach, as we have applied the same library of properties to all cases regardless their format.

The reason why we have selected meta-models from two different repositories is because, altogether, they cover a wide spectrum of the current meta-modelling practice and practitioners. First, the meta-model contributors are very different in each case: whereas the contributors to the ATL zoo are MDE practitioners, academics and researchers with a heterogeneous background on modelling ranging from novice to proficient, OMG specifications are normally developed by industry experts and professionals. OMG is responsible for widely used modelling standards like the UML or BPMN. Second, OMG specifications regularly state a standard formal definition – which makes them a rigorous sample to analyse – whereas the degree of maturity and completeness of the meta-models in the ATL zoo is generally unknown. Finally, the size of the meta-models in the test set varies from tiny ones with only one class, to meta-models of medium size, the largest one with 699 classes coming from the ATL zoo. This is interesting as one of our goals is to check whether V&V is needed for both large and small meta-models.

Figure 18 shows the number of quality issues detected in both analysed repositories (a more detailed analysis of just the meta-models in the ATL zoo can be found in [34]). The ATL zoo

[5] `http://www.omg.org/spec/`

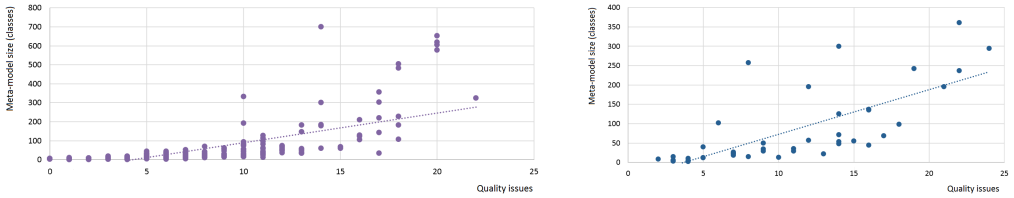
only contains 5 meta-models without issues, no meta-model contains more than 22 issues, and the average number of issues per meta-model is 7.26. The OMG figures are higher, with an average rate of 11 issues per meta-model, a maximum of 24 issues in one meta-model, and zero meta-models having no flaws, which means that every analysed OMG meta-model raised some potential quality error or warning.



(a) ATL zoo (sample of 295 meta-models). (b) OMG (sample of 43 meta-models).

Figure 18: Number of quality issues in meta-models.

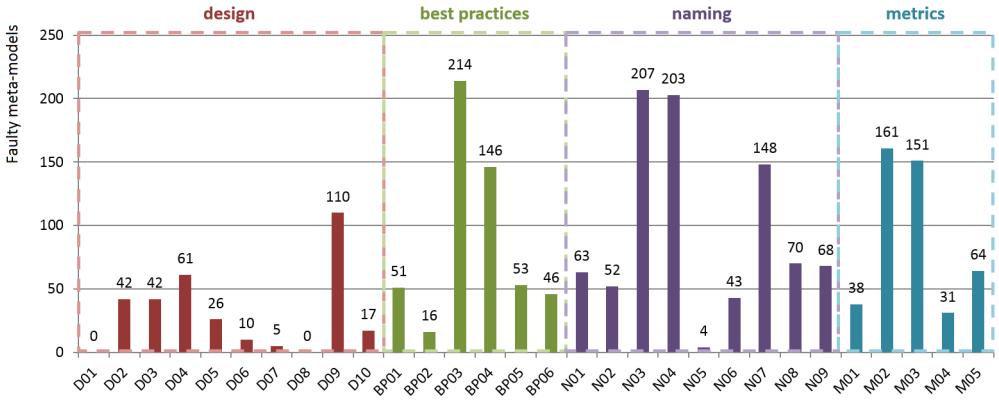
Figure 19 depicts the distribution of detected issues with respect to the meta-model size. In particular, each dot in this graphic corresponds to a meta-model, and the vertical axis indicates the number of classes it contains. In this way, the figure shows that, in both cases, the larger the size of the meta-model, the larger the number of quality issues it tends to have.



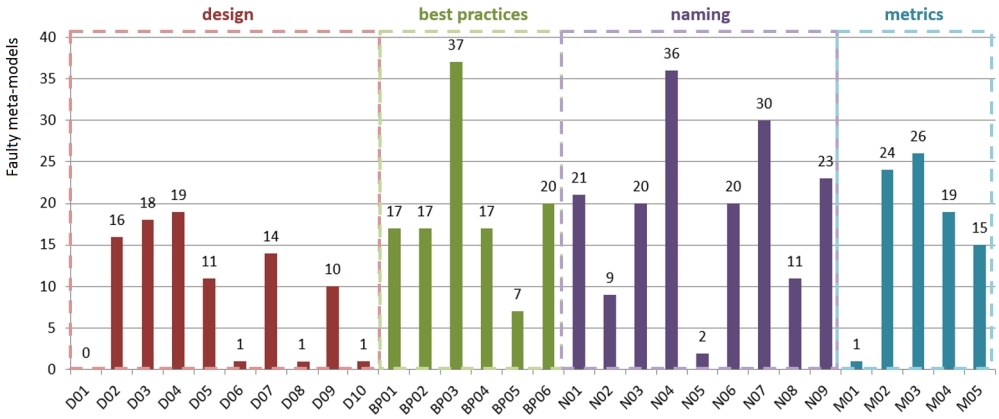
(a) ATL zoo (sample of 295 meta-models). (b) OMG (sample of 43 meta-models).

Figure 19: Number of quality issues in meta-models, with respect to the meta-model size.

Regarding the distribution of issues according to their kind, Figure 20 shows how many meta-models fail each property from Table 3. *Design* is the most relevant category of properties, as it gathers errors that may potentially lead to a faulty design. In this sense, the results for the properties in this category are good in average, as they have low rate of failure. Indeed, there are two *design* properties that every meta-model fulfils in the ATL zoo (Figure 20(a)): D01 and D08. D01 checks the absence of repeated attributes in a whole hierarchy (see *D01* in Figure 21 for a faulty example), while D08 checks that the upper bound of features is not 0. If we take a look at the OMG results (Figure 20(b)), D01 is not present likewise, and there is a mere occurrence of D08, for which we may conclude that these two design guidelines are followed carefully. Other design issues that occur only once in the analysed OMG specifications are D06 (no binary association is composite in both ends) and D10 (no class contains one of its superclasses, with cardinality 1 in the composition end). The occurrence rate of these two issues in the ATL zoo is slightly higher: 3.3% (10 out of 295 meta-models) for property D06, and 5.7% (17 out of 295 meta-models) for D10.



(a) ATL zoo (sample of 295 meta-models).



(b) OMG (sample of 43 meta-models).

Figure 20: Number of meta-models that contain issues of a certain type.

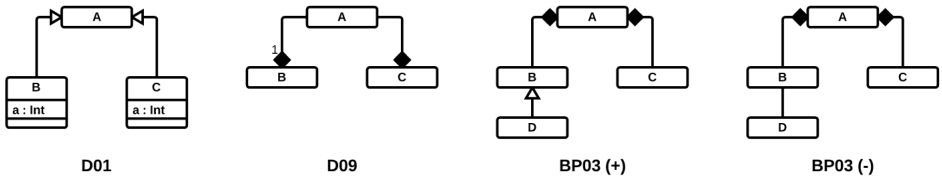


Figure 21: Some quality issues analysed by the library.

However, 110 meta-models from ATL (37.2%) and 10 meta-models from OMG (23.2%) fail property D09. As illustrated in Figure 21, this error consists in making a class to be contained in two other classes, with minimum source multiplicity 1 in one of the containment relationships. This is an error because, at the instance level, an instance of A could never be contained in an instance of C, as it must be mandatorily contained in an instance of B.

Overall, the most failed property in both repositories is BP03: 72.5% meta-models in the ATL zoo, and 86% analysed OMG specifications fail this property. BP03 is an EMF best practice that states the need for a root class whose instances may contain the whole model tree. This is not a problem for most OMG specifications as these are not thought to be implemented with EMF (though they could). However, it is worrying for the meta-models in the ATL zoo as they all are EMF-based. Figure 21 shows an example meta-model that fulfils this property, and an example that does not. In *BP03 (+)*, A contains B and C, and hence D (as it is subclass of B), so A acts as absolute root class. On the contrary, *BP03 (-)* does not meet the property because A does not contain D.

Among the naming conventions, N03, N04 and N07 are scarcely followed. N03 demands the verbalization of binary association names (e.g., reaches), while N04 and N07 check conventions for class and boolean attribute naming (see Table 3).

From the analysis of the meta-model repositories, we can conclude that integrated support for V&V is urgently needed in meta-modelling tools, as evidenced by the low number of meta-models with no issues. This need is real for both professional engineers (as the analysis of the OMG specifications demonstrates) and academics and researchers (as shown by the analysis of the ATL zoo). We believe that a way to improve the quality of meta-models is the inclusion of quality checks in the meta-modelling tools, for example, to discover problems like D09. Such checks should be available while meta-models are being constructed, but also *a posteriori* to enable regression testing. Moreover, for some kinds of problems (like the ones related to metrics), the tool could suggest refactorings that mitigate or even remove the issue. Regarding naming conventions, it would be useful to have integrated "smart" spell checkers able to, e.g., check the correctness of names in camel-case.

Regarding the validity of the conclusions of our study, there are a number of issues that should be mentioned. First, the number of analysed meta-models (338) could be considered insufficient, and indeed, we plan to increase the number of analysed meta-models in future studies. Nonetheless, even if the number of analysed meta-models could be higher, the point is that we detected a high number of issues in most of them, making evident that better support for meta-model V&V is needed. Similarly, the particular meta-models used in the study have a great impact in the analysis results. To mitigate the risk that these results are not general, we have analysed meta-models from two different sources which, altogether, cover meta-models built by industry experts and by developers with different training and background, consider meta-models with different degrees of maturity ranging from toy examples (ATL zoo) to industry standards (OMG), and include meta-models of different size.

Finally, an important remark is that detecting a quality issue does not always imply that a meta-model is "erroneous", as some issues are warnings or bad smells that need to be manually assessed by the meta-model designer. In particular, depending on the purpose or nature of a meta-model, one can obviate certain types of issues. For instance, some meta-models are built to be used as frameworks, and hence, they may contain abstract leaf classes that need to be subclassified by the meta-model users. For this kind of meta-models, our best practice BP02 which checks whether there are abstract leaf classes is not needed. Our study has not removed such cases from the results, therefore, some raised issues may be not actual errors. It is up to future work to

extend our study to exclude such cases from the results.

9. Conclusions and future work

In this paper, we have addressed the V&V of meta-models by proposing two DSLs: *mmUnit* and *mmSpec*. The first one permits defining valid and invalid examples and model fragments. This language enables an intentional description of the reasons why an example is invalid, and the 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. Moreover, its connection with WordNet permits an easier transition from natural language requirements into *mmSpec* properties. Both languages are supported by a tool (*metaBest*), enabling the visualization and reporting of the problematic elements.

We have also presented a library that encodes 30 common quality criteria for meta-models using our language *mmSpec*. The library has been applied to two meta-model repositories, which amount to 338 meta-models altogether. From the analysis of the results, we have concluded that the meta-modelling community would greatly benefit from integrated tool support for checking quality properties during meta-model construction. *metaBest* is one such tool.

In the future, we would like to perform an empirical evaluation of our framework with MDE developers in order to analyse effectively to which extent our languages are helpful in the construction of proved, high-quality meta-models. We would also like to improve our framework with *quick fixes* and recommendations, suggested upon test failures. Finally, we will improve our framework with the automatic generation of graphical modelling environments, where the visual syntax for models will be inferred from the example fragments.

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Flexor" (TIN2014-52129-R), the region of Madrid with project "SICOMORO-CM" (S2013/ICE-3006), and the EU commission with project "MONDO" (FP7-ICT-2013-10, #611125).

References

[1] D. Aguilera, R. García-Ranea, C. Gómez, and A. Olivé. An eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops*, volume 6999 of *LNCS*, pages 323–327. Springer, 2011.

[2] D. Aguilera, C. Gómez, and A. Olivé. A method for the definition and treatment of conceptual schema quality issues. In *ER*, volume 7532 of *LNCS*, pages 501–514. Springer, 2012.

[3] D. Aguilera, C. Gómez, and A. Olivé. A complete set of guidelines for naming UML conceptual schema elements. *Data Knowl. Eng.*, 88:60–74, 2013.

[4] D. Aguilera, C. Gómez, and A. Olivé. Enforcement of conceptual schema quality issues in current integrated development environments. In *CAiSE*, volume 7908 of *LNCS*, pages 626–640. Springer, 2013.

[5] M. Autili, A. Bertolino, G. D. Angelis, D. D. Ruscio, and A. D. Sandro. A tool-supported methodology for validation and refinement of early-stage domain models. *IEEE Trans. Software Eng.*, 42(1):2–25, 2016.

[6] O. B. Badreddin, A. Forward, and T. C. Lethbridge. A test-driven approach for developing software languages. In *MODELSWARD*, pages 225–234. SciTePress, 2014.

[7] K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside. Example-driven modeling: model = abstractions + examples. In *Proc. ICSE '13*, pages 1273–1276. IEEE / ACM, 2013.

[8] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.

[9] K. Beck. Simple smalltalk testing: with patterns. Technical Report 4 (2), The Smalltalk Reports, 1994.

[10] K. Beck. *Test Driven Development: by Example*. Addison-Wesley Professional, 2003.

[11] M. F. Bertoa and A. Vallecillo. Quality attributes for software metamodels. In *QAOOSE'10*, 2010.

[12] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.
[13] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
[14] J. Cabot, R. Clarisó, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014.
[15] J. J. Cadavid, B. Baudry, and H. A. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *ICST*, pages 131–140. IEEE, 2012.
[16] M. J. Carey, N. Onose, and M. Petropoulos. Data services. *Commun. ACM*, 55(6):86–97, 2012.
[17] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
[18] A. Cicchetti, D. D. Ruscio, A. Pierantonio, and D. Kolovos. A test-driven approach for metamodel development. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 319–342. IGI Global, 2012.
[19] J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin. Deriving OCL optimization patterns from benchmarks. *ECEASST*, 15, 2008.
[20] M. Elaasar, L. C. Briand, and Y. Labiche. Domain-specific model verification with QVT. In *ECMFA*, volume 6698 of *LNCS*, pages 282–298. Springer, 2011.
[21] L. Gammaitoni, P. Kelsen, and F. Mathey. Verifying modelling languages using lightning: a case study. In *MoDeVVA@MODELS'14*, volume 1235 of *CEUR Workshop Proceedings*, pages 19–28. CEUR-WS.org, 2014.
[22] 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 *MoDELS*, volume 6981 of *LNCS*, pages 395–409. Springer, 2011.
[23] A. Garis and A. Sánchez. Verification and validation of domain specific languages using Alloy. In *CACIC*, 2015.
[24] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4(4):386–398, 2005.
[25] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.*, 89:144–161, 2014.
[26] J. L. C. Izquierdo, J. Cabot, J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara. Engaging end-users in the collaborative development of domain-specific modelling languages. In *CDVE'13*, volume 8091 of *LNCS*, pages 101–110. Springer, 2013.
[27] P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, and D. Mayorova. A requirement-driven approach to the design and evolution of data warehouses. *Inf. Syst.*, 44:94–119, 2014.
[28] L. C. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: Enabling test-driven language development. In *OOPSLA '11*, pages 139–154, New York, NY, USA, 2011. ACM.
[29] S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.
[30] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *LNCS*, pages 204–218. Springer, 2009.
[31] T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. *Information & Software Technology*, 71:77–91, 2016.
[32] Y. Liu, S. Höglund, A. H. Khan, and I. Porres. A feasibility study on the validation of domain specific languages using OWL 2 reasoners. In *TWOMDE*, volume 604 of *Ceur Workshop Proceedings*, pages 1–13. CEUR, 2010.
[33] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara. Example-driven meta-model development. *Software and Systems Modeling*, 14(4):1323–1347, 2015.
[34] J. J. López-Fernández, E. Guerra, and J. de Lara. Assessing the quality of meta-models. In *MoDeVVA@MODELS'14*, volume 1235 of *CEUR Workshop Proceedings*, pages 3–12. CEUR-WS.org, 2014.
[35] J. J. López-Fernández, E. Guerra, and J. de Lara. Meta-model validation and verification with metabest. In *ASE'14*, pages 831–834. ACM, 2014.
[36] J. J. López-Fernández, E. Guerra, and J. de Lara. Example-based validation of domain-specific visual languages. In *SLE*, pages 101–112. ACM, 2015.
[37] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, 2005.
[38] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
[39] OMG. OCL 2.4. `http://www.omg.org/spec/OCL/`, 2014.
[40] R. F. Paige, P. J. Brooke, and J. S. Ostroff. Specification-driven development of an executable metamodel in Eiffel. In *WISME'04*, 2004.
[41] L. A. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and Systems Modeling*, 14(2):1003–1028, 2015.
[42] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. Constructing models with the human-usable textual notation. In *MODELS*, volume 5301 of *LNCS*, pages 249–263. Springer, 2008.

[43] D. A. Sadilek and S. Weißleder. Testing metamodels. In *ECMDA-FA*, volume 5095 of *LNCS*, pages 294–309. Springer, 2008.
[44] O. Semeráth, A. Barta, A. Horváth, Z. Szatmári, and D. Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and Systems Modeling*, In press, 2015.
[45] S. Sobernig, B. Hoisl, and M. Strembeck. Requirements-driven testing of domain-specific core language models using scenarios. In *QSIC*, pages 163–172. IEEE, 2013.
[46] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.
[47] A. Tort and A. Olivé. An approach to testing conceptual schemas. *Data Knowl. Eng.*, 69(6):598–618, 2010.
[48] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.
[49] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw., Pract. Exper.*, 38(10):1073–1103, 2008.
[50] H. Wu, R. Monahan, and J. F. Power. Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *TASE'13*, pages 175–182. IEEE, 2013.

Appendix A. Encoding of *mmSpec* primitives in OCL

This appendix provides the encodings in OCL of most primitives offered by *mmSpec*. The OCL expressions are specific to EMF meta-models; other meta-modelling architectures (like UML or MOF) would require a different encoding. While some *mmSpec* primitives related to word nature cannot be directly encoded in OCL, other primitives are much concise in *mmSpec* than in OCL.

	mmSpec	**OCL**
Element existence	(1) Some class exists:	
	`some class => exist.`	`EClass.allInstances()->notEmpty()`
	(2) Some attribute exists:	
	`some attribute => exist.`	`EAttribute.allInstances()->notEmpty()`
	(3) Some reference exists:	
	`some reference => exist.`	`EReference.allInstances()->notEmpty()`

	mmSpec	**OCL**
Element name	(4) There is an element named *x*:	
	`some element => name = x.`	`ENamedElement.allInstances()` `  ->exists(e | e.name = 'x')`
	(5) The name of some element starts with *a*:	

	`some element => name{prefix} = a.`	`ENamedElement.allInstances()`
		`  ->exists(e |`
		`  if e.name.oclIsUndefined() then false`
		`  else e.name.indexOf('a') = 1 endif)`

	(6) The name of some element contains *a*:	

	`some element => name{infix} = a.`	`ENamedElement.allInstances()`
		`  ->exists(e |`
		`  if e.name.oclIsUndefined() then false`
		`  else e.name.indexOf('a') <> 0 endif)`

	(7) The name of some element ends with *a*:	

Continue on next page

Continue from previous page (element name)

	mmSpec	OCL

	`some element => name{suffix} = a.`	`ENamedElement.allInstances()`
		`  ->exists(e |`
		`  if e.name.oclIsUndefined() then false`
		`  else (e.name.indexOf('a') <> 0 and`
		`  e.name.indexOf('a') =`
		`  (e.name.size() - 'a'.size() +1)) endif)`

	(8) The name of some element is a verb / noun / adjective:	

	`some element => name = verb.`
	`some element => name = noun.`
	`some element => name = adjective.`
		not supported

	(9) The name of some element is a synonym of *x*:	
	`some element => name = synonym{x}.`	*not supported*
	(10) The name of an elem. is a camel/pascal phrase ("likeThis" or "LikeThis"):	
	`some element => name = camel-phrase.` `some element => name = pascal-phrase.`	*not supported*

	mmSpec	OCL
Class abstractness	(11) Some class is abstract:	
	`some class => abstract.`	`EClass.allInstances()` `  ->exists(c | c.abstract)`

	mmSpec	OCL
Class features	(12) Some class contains a feature (attribute or reference) named *x*:	

	`some class => with a feature{name = x}.`	`EClass.allInstances()`
		`  ->exists(c |`
		`  c.eStructuralFeatures->exists(f |`
		`  f.name = 'x'))`

	mmSpec	OCL
Class inheritance	(13) Some class is subclass of another:	
	`some class => sub-to some class.`	`EClass.allInstances()` `  ->exists(c | c.eSuperTypes->notEmpty())`
	(14) Some class is subclass of another in at most *n* steps in the hierarchy:	

	`some class`
	`  => sub-to{depth=[1,n]} some class.`
		`EClass.allInstances()`
		`  ->exists(c | Sequence{1..n}`
		`  ->iterate(i:Integer; super:Set(EClass)=Set{} |`
		`  super->union(super->including(c)`
		`  ->select(c2 |`
		`  c2.eSuperTypes->notEmpty())`
		`  ->collect(c2 |`
		`  c2.eSuperTypes)->asSet()))`
		`  ->notEmpty())`

	(15) Some class is superclass of another:	

Continue on next page

Continue from previous page (class inheritance)

	mmSpec	OCL
	`some class => super-to some class.`	`EClass.allInstances()->exists(c | EClass.allInstances()->exists(subclass | subclass.eSuperTypes->includes(c)))`
	(16) Some class is superclass of another in at most *n* steps in the hierarchy:	
	`some class => super-to{depth=[1,n]} some class.`	`EClass.allInstances()->exists(c | Sequence{1..n}->iterate(i:Integer; sub:Set(EClass)=Set{} | sub->union(sub->including(c)->select(c2 | EClass.allInstances()->exists(super | super.eSuperTypes->includes(c2)))->collect(c2 | EClass.allInstances()->select(super | super.eSuperTypes->includes(c2)))->asSet()))->notEmpty())`

	mmSpec	OCL
Depth of hierarchy	(17) Some class is at the top of an inheritance hierarchy:	
	`some class => inh-root.`	`EClass.allInstances()->exists(c | c.eSuperTypes->isEmpty() and EClass.allInstances()->exists(subclass | subclass.eSuperTypes->includes(c)))`
	(18) Some class is at the bottom of an inheritance hierarchy:	
	`some class => inh-leaf.`	`EClass.allInstances()->exists(c | EClass.allInstances()->forAll(subclass | subclass = c or subclass.eSuperTypes->excludes(c)))`

	mmSpec	OCL
Depth of containment	(19) Some class is a top container:	
	`some class => cont-root.`	`EClass.allInstances()->exists(c | c.eAllReferences->exists(ref | ref.containment = true) and EClass.allInstances()->forAll(contclass | if container = c then true else not contclass.eAllReferences->reject(ref | ref.contclass = false)->includes(c) endif)))`
	(20) Some class is contained in another class, but it does not contain others:	

Continue on next page

Continue from previous page (depth of containment)

	mmSpec	**OCL**

	`some class => cont-leaf.`	`EClass.allInstances()`
		`->exists(c |`
		`not c.eReferences->exists(ref |`
		`ref.containment = true) and`
		`EClass.allInstances()->exists(contclass |`
		`contclass.eReferences->exists(ref |`
		`ref.containment = true and`
		`ref.eType = c)))`

	mmSpec	**OCL**
Class reachability	(21) Some class contains another:	

	`some class`
	`=> reach{cont} some class.`
		`EClass.allInstances()`
		`->exists(c |`
		`c.eAllReferences->reject(ref |`
		`ref.containment = false)`
		`->notEmpty())`

	(22) Some class is contained in another:	

	`some class`
	`=> reached-from{cont} some class.`
		`EClass.allInstances()`
		`->exists(c |`
		`EReference.allInstances()->exists(ref |`
		`ref.eType = c and`
		`ref.containment = true))`

	(23) Some class reaches another:	
	`some class` `=> reach some class.`	`EClass.allInstances()` `->exists(c | c.eAllReferences->notEmpty())`
	(24) Some class reaches another in up to *n* jumps:	

	`some class`
	`=> reach{jumps=[1,n]} some class.`
		`EClass.allInstances()`
		`->exists(c | Sequence{1..n}`
		`->iterate(i:Integer; reaches:Set(EClass)=Set{} |`
		`reaches->union(reaches->including(c)`
		`->select(c2 |`
		`c2.eAllReferences->notEmpty())`
		`->collect(c2 |`
		`EClass.allInstances()`
		`->select(c3 |`
		`c2.eAllReferences->exists(ref |`
		`ref.eType = c3)))-> asSet()))`
		`->notEmpty())`

	(25) Some class is reached from another:	

	`some class`
	`=> reached-from some class.`
		`EClass.allInstances()`
		`->exists(c | EReference.allInstances()`
		`->select(ref | ref.eType = c)->notEmpty())`

	(26) Some class is reached from another in up to *n* jumps:	

Continue on next page

Continue from previous page (class reachability)

	mmSpec	OCL

	`some class`
	`=> reached-from{jumps=[1,n]}`
	`some class.`
		`EClass.allInstances()`
		`->exists(c | Sequence{1..n}`
		`->iterate(i:Integer; targets:Set(EClass)=Set{} |`
		`targets->union(targets->including(c)`
		`->select(c2 |`
		`EClass.allInstances()->exists(tar |`
		`tar.eAllReferences->exists(ref |`
		`ref.eType = c2)))`
		`->collect(c2 |`
		`EClass.allInstances()->select(tar |`
		`tar.eAllReferences->exists(ref |`
		`ref.eType = c2)))->asSet()))`
		`->notEmpty())`

	mmSpec	OCL
Feature class	(27) Some class has a feature:	
	`some class => with some feature.`	`EClass.allInstances()` `->exists(c | c.eAllStructuralFeatures->notEmpty()`
	(28) Some class has a feature, not inherited from its superclasses:	
	`some class => with{!inh} some feature.`	`EClass.allInstances()` `->exists(c | c.eStructuralFeatures->notEmpty())`

	mmSpec	OCL
Feature multiplicity	(29) Some feature has a minimum/maximum multiplicity between *n* and *m*:	

	`some feature => multiplicity{min=[n,m]}.`
	`some feature => multiplicity{max=[n,m]}.`
		`EStructuralFeature.allInstances()`
		`->exists(f | f.lowerBound >= n and`
		`f.lowerBound <= m)`
		
		`EStructuralFeature.allInstances()`
		`->exists(f | f.upperBound >= n and`
		`f.upperBound <= m)`

	mmSpec	OCL
Attribute type	(30) Some attribute has a primitive type *t*:	
	`some attribute => type = t.`	`EAttribute.allInstances()` `->exists(attribute | attribute.eType.name = 't')`

	mmSpec	OCL
Reference ends	(31) Some reference starts in a class:	

	`some reference => from a class.`	`EReference.allInstances()`
		`->exists(ref | EClass.allInstances()`
		`-> exists(c | c.eAllReferences->includes(ref)))`

	(32) Some reference ends in a class, or in one of its superclasses:	

Continue on next page

Continue from previous page (reference ends)

	mmSpec	OCL
	1 **a reference** => **to**{!**inh**} **a class**.	1 EReference.**allInstances**() 2 −>**exists**(ref | EClass.**allInstances**() 3 −> **exists**(c | 4 ref.eType = c **or** 5 c.eAllSuperTypes−>**exists**(super | 6 ref.eType = super)))

	mmSpec	OCL
Paths	(33) Some path starts in a class *x* and ends in a class *y*:	
	1 **some path** => **and**{ 2 **from a class** {**name** = x}, 3 **to a class** {**name** = y}}.	1 EClass.**allInstances**() 2 −>**exists** (class | 3 class.name=x **and** 4 **Sequence**{class}−>**closure**(class | 5 class.eAllReferences−>**collect**(eType)) 6 −>**exists**(classy | classy.name = y))
	(34) Some containment path starts in a class *x* and ends in a class *y*:	
	1 **some path** => **and**{ 2 **cont**, 3 **from a class** {**name** = x}, 4 **to a class** {**name** = y}}.	1 EClass.**allInstances**() 2 −>**exists** (class | 3 class.name = x **and** 4 **Sequence**{class}−>**closure**(class| 5 class.eAllReferences−>**reject**(ref | 6 ref.containment.**oclIsUndefined**() **or** 7 ref.containment = false)−>**collect**(eType)) 8 −>**exists**(classy | classy.name=y))
	(35) Some path starts in a class *x*, goes through a class *y*, and ends in a class *z*:	
	1 **some path** => **and**{ 2 **from a class** {**name** = x}, 3 **through a class** {**name** = y}, 4 **to a class** {**name** = z}}.	1 EClass.**allInstances**()−>**exists**(class | 2 class.name = x **and** 3 **Sequence**{class}−>**closure**(class | 4 class.eAllReferences−>**collect**(eType)) 5 −>**exists**(classy | 6 classy.name = y **and** 7 **Sequence**{classy}−>**closure**(classy | 8 classy.eAllReferences−>**collect**(eType)) 9 −> **exists**(classz | classz.name = z)))
	(36) Some path is cyclic:	
	1 **some path** => cycle.	1 EClass.**allInstances**() 2 −>**exists**(c | **Sequence**{c} 3 −>**closure**(c | c.eAllReferences 4 −>**collect**(eType))−>**includes**(c))

Appendix B. Encoding of *mmUnit* primitives in Java

The following library aims to reproduce the behaviour of *mmUnit* primitives using JUnit test cases. We show relevant excerpts of classes MmUnitModelLoader (for model loading) and MmUnitAssertion (a library of assertions for testing EMF models). Each assertion method in MmUnitAssertion corresponds to a *mmUnit* primitive, with the exception of those that cannot be checked with the only aid of EMF utilities.

```
1 public class MmUnitModelLoader {
2
3   private Resource resource = null;
```

```
4   List<String> errors = new ArrayList<String>();
5   private boolean loaded;
6
7   public Resource load(String fileLocation){
8     try{
9       File modelFile = new File(fileLocation);
10      ResourceSet resourceSet = new ResourceSetImpl();
11      resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put
12        (Resource.Factory.Registry.DEFAULT_EXTENSION,
13         new XMIResourceFactoryImpl());
14
15      resourceSet.getPackageRegistry().put(DaaSPackage.eNS_URI, DaaSPackage.eINSTANCE);
16      URI uri = modelFile.isFile() ? URI.createFileURI(modelFile.getAbsolutePath()):
17            URI.createURI(modelFile.getAbsolutePath().toString());
18
19      this.resource = resourceSet.getResource(uri, true);
20
21      for (Iterator<?> j = resource.getContents().iterator(); j.hasNext(); ) {
22        EObject eObject = (EObject)j.next();
23
24        Map<Object, Object> context = new HashMap<Object, Object>();
25        Diagnostic diagnostic = Diagnostician.INSTANCE.validate(eObject, context);
26
27        if (diagnostic.getSeverity() != Diagnostic.OK)
28          errors.add(diagnosticToString(diagnostic, ""));
29      }
30    }catch(RuntimeException exception){
31      if(exception.getCause() instanceof IllegalValueException){
32        IllegalValueException cause = (IllegalValueException) exception.getCause();
33        errors.add(exception.getCause().getMessage()
34            + "\nEObject:␣" + cause.getObject()
35            + "\nEStructuralFeature:␣" + cause.getFeature());
36      }
37
38      if(exception.getCause() instanceof ClassNotFoundException)
39        errors.add(exception.getCause().getMessage());
40
41      if(exception.getCause() instanceof FeatureNotFoundException)
42        errors.add(exception.getCause().getMessage());
43    }
44
45    if(resource != null) loaded = true;
46    else loaded = false;
47
48    return resource;
49  }
50
51  protected static String diagnosticToString(Diagnostic diagnostic, String indent) {
52    String diagMssg = "";
53    diagMssg += "\n" + indent;
54    diagMssg += "\n" + diagnostic.getMessage();
55
56    for (Iterator<?> i = diagnostic.getChildren().iterator(); i.hasNext(); )
57      diagMssg += diagnosticToString((Diagnostic)i.next(), indent + "␣␣");
58
59    return diagMssg;
60  }
61
62  //...
63 }
```

Listing 1: Excerpt of MmUnitLoader class to help in loading EMF models

```
1  public class MmUnitAssertion extends Assert {
2    private String packagePrefix;
3    private EFactory factory;
4    private List<String> errors;
5    //...
6
```

```
7   public void assertMismatchOnFeatureMultiplicity(String objectType, String feature){
8     for(String e : errors){
9       List<String> matchingSegments
10        = Arrays.asList("Diagnosis␣of␣" + packagePrefix + objectType + "Impl@",
11                "The␣feature␣\'" + feature + "\'␣of␣\'" + packagePrefix + objectType + "Impl@",
12                "with",
13                "values␣must␣have␣at␣least");
14
15      if(stringMatches(e, matchingSegments)) assertTrue(true);
16    }
17
18    fail("There␣is␣not␣a␣multiplicity␣mismatch␣on␣" + objectType + "." + feature);
19  }
20
21  public void assertMismatchOnFeatureType(String objectType, String feature){
22    for(String e : errors){
23      List<String> matchingSegments
24        = Arrays.asList("Value␣",
25                "is␣not␣legal.␣",
26                "EObject:␣" + packagePrefix + objectType + "Impl@",
27                "EStructuralFeature:␣org.eclipse.emf.ecore.impl.",
28                "(name:␣" + feature + ")");
29
30      if(stringMatches(e, matchingSegments)) assertTrue(true);
31    }
32
33    fail(objectType + "." + feature + "␣has␣the␣proper␣nature␣in␣all␣its␣occurences.");
34  }
35
36  public void assertAbstractTypeInstance(String objectType){
37    for(String e : errors){
38      List<String> matchingSegments
39        = Arrays.asList("Class␣\'",
40                objectType + "\'␣is␣not␣found␣or␣is␣abstract.");
41
42        if(stringMatches(e, matchingSegments)){
43          EClassifier classifier = factory.getEPackage().getEClassifier(objectType);
44          if(classifier == null) continue;
45
46          if(classifier instanceof EClass)
47            if(((EClass) classifier).isAbstract())
48              assertTrue(true);
49        }
50    }
51
52    fail(objectType + "␣is␣not␣abstract␣or␣it␣doesn't␣exist.");
53  }
54
55  public void assertInexistentMetaClass(String objectType){
56    for(String e : errors){
57      List<String> matchingSegments
58        = Arrays.asList("Class␣\'",
59                objectType + "\'␣is␣not␣found␣or␣is␣abstract.");
60
61        if(stringMatches(e, matchingSegments)){
62          EClassifier classifier = factory.getEPackage().getEClassifier(objectType);
63          if(classifier == null) assertTrue(true);
64          if(!(classifier instanceof EClass)) assertTrue(true);
65        }
66    }
67
68    fail(objectType + "␣exists.");
69  }
70
71  public void assertInexistentFeature(String feature){
72    for(String e : errors){
73      List<String> matchingSegments
74        = Arrays.asList("Feature␣\'",
75                feature + "\'␣not␣found.");
76
```

```
77          if(stringMatches(e, matchingSegments)) assertTrue(true);
78        }
79
80        fail(feature + "␣exists.");
81      }
82
83      public void assertMissingFeature(String objectType, String feature){
84        for(String e : errors){
85          List<String> matchingSegments
86            = Arrays.asList("Diagnosis␣of␣",
87                    "The␣feature␣\'" + feature + "\'␣of␣\'" + packagePrefix + objectType + "Impl@",
88                    "with␣0␣values␣must␣have␣at␣least␣1␣values");
89
90          if(stringMatches(e, matchingSegments)) assertTrue(true);
91        }
92
93        fail(objectType + "." + feature + "is␣not␣missing.");
94      }
95
96      public void assertUncontainedObject(String objectType){
97        for(String e : errors){
98          List<String> matchingSegments
99            = Arrays.asList("Feature␣\'",
100                   objectType + "\'␣not␣found.");
101
102         if(stringMatches(e, matchingSegments)) assertTrue(true);
103       }
104
105       fail("All␣the␣" + objectType + "␣are␣contained.");
106     }
107
108     public void assertConstraintViolation(String objectType){
109       for(String e : errors){
110         List<String> matchingSegments
111           = Arrays.asList("Diagnosis␣of␣",
112                   "The␣\'",
113                   "\'␣constraint␣is␣violated␣on␣\'" + packagePrefix + objectType + "Impl@");
114
115         if(stringMatches(e, matchingSegments)) assertTrue(true);
116       }
117
118       fail(objectType + "␣doesn't␣violate␣any␣constraint.");
119     }
120
121     //...
122 }
```

Listing 2: Excerpt of Library of EMF-based assertions