Assessing the Quality of Meta-models

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

Universidad Autónoma de Madrid (Spain)
{Jesusj.Lopez, Esther.Guerra, Juan.deLara}@uam.es

Abstract. Meta-models play a pivotal role in Model-Driven Engineering (MDE), as they define the abstract syntax of domain-specific languages, and hence, the structure of models. However, while they play a crucial role for the success of MDE projects, the community still lacks tools to check meta-model quality criteria, like design errors or adherence to naming conventions and best practices.
In this paper, we present a language (*mmSpec*) and a tool (*metaBest*) to specify and check properties on meta-models and visualise the problematic elements. Then, we use them to evaluate over 295 meta-models of the ATL zoo by provisioning a library of 30 meta-model quality issues. Finally, from this evaluation, we draw recommendations for both MDE practitioners and meta-model tool builders.

1 Introduction

Model-Driven Engineering (MDE) considers models as the main assets in software development. Frequently, such models are not built using general-purpose modelling languages, like UML, but with Domain-Specific Languages (DSLs). Recent surveys on the use of MDE in industry [8] observe that nearly 40% of respondents use in-house DSLs, likely developed using meta-models. Hence, a critical factor in the success of MDE projects is the quality of the meta-models. However, the MDE community still lacks flexible tools permitting the specification, evaluation and user-friendly report of desired properties of meta-models.

A meta-model is considered of quality if it serves its purpose (contains all needed abstractions of the domain), and is technically built using sound principles (e.g., there are no repeated attributes among all sibling classes) [5]. The first concern is related to meta-model validation (*"are we building the right meta-model?"*) while the second refers to verification (*"are we building the meta-model right?"*).

In this paper, we present the language *mmSpec* and its supporting tool (*metaBest*), directed to the specification of desired meta-model properties, their evaluation, and the visualization of non-conforming parts of meta-models. We have used this tool to define a reusable library of 30 meta-model quality properties coming from quality criteria of conceptual schemas [2], naming guidelines [3], or from our experience. The properties have been classified in four categories depending on their relevance and nature. Moreover, we have evaluated the properties over 295 meta-models from the ATL zoo[1]. The ATL zoo is an open source

[1] http://www.emn.fr/z-info/atlanmod/index.php/Ecore.

repository to which any author can contribute, being the authorship of its meta-models attributed to a wide range of MDE community members. From the results of this study, we provide suggestions for MDE practitioners and meta-modelling tool builders.

This paper extends [9] (a tool demo paper) by presenting a library of quality properties that is evaluated over a repository of meta-models.

The remaining of this paper is organized as follows. First, Section 2 introduces the *mmSpec* language and tool, and overviews the library of quality properties. Section 3 presents the evaluation of the library over the meta-models of the ATL zoo. Section 4 compares with related research, and Section 5 ends with the conclusions and future work.

2 Specification of meta-model properties with *mmSpec*

We have developed a domain-specific language, named *mmSpec*, to specify meta-model properties and automate their evaluation on meta-models. In the following two subsections, we first introduce the main constructs of our language, and then we describe its use to construct a generic library of quality properties which can be applied on meta-models to assess their quality in an automatic way.

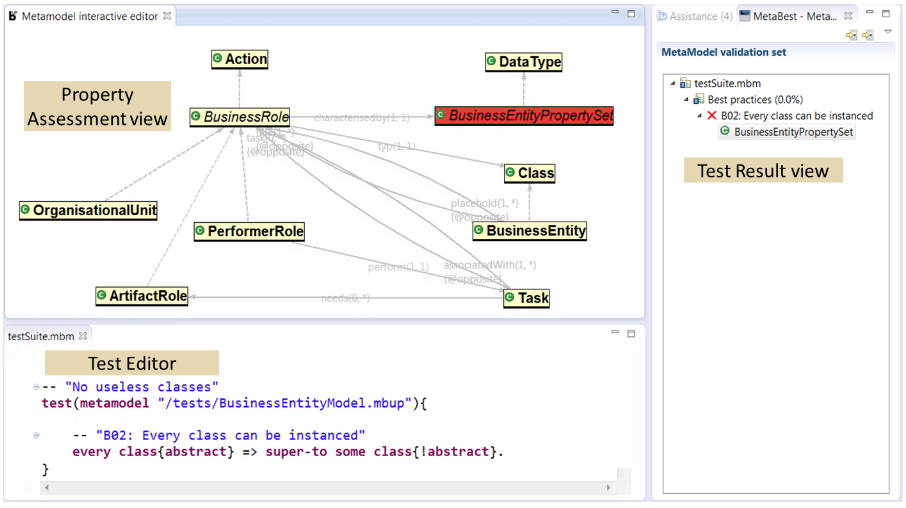
2.1 Definition and evaluation of meta-model properties

mmSpec allows expressing meta-model properties in a concise, intensional, declarative, platform-independent way. It provides high-level primitives that simplify the definition of meta-model properties, like first-order qualifiers for the length of navigation paths or collectors of the composed cardinality in navigation paths. Moreover, it is integrated with WordNet [11], which allows testing the nature of words (i.e., nouns and verbs) and synonymy.

The aim of this language and its companion tool (*metaBest* [9]) is providing a sound framework for expressing and evaluating quality issues and structural properties of meta-models. To favour simplicity, *mmSpec* properties follow a *select-filter-check* style that includes:

- A *selector* of the type (class, attribute, reference or path) and amount (a quantifier like every, some, none or an interval) of elements that should satisfy a given condition.
- An optional *filter* over the elements in the selector.
- A *condition* that is checked over the filtered elements.

Filters and conditions consist of *qualifiers* which can be negated, combined through and/or connectives, and point to new selectors, enabling recursive checks. The main qualifiers allow expressing conditions on the existence of elements, their name (nature, synonymy, prefix, suffix, camel-phrase), abstractness, multiplicity, type, length of navigation paths, inheritance relationships, depth and width of hierarchies and trees of containment relationships, collectors of the composed



```
-- "No useless classes"
test(metamodel "/tests/BusinessEntityModel.mbup"){

    -- "B02: Every class can be instanced"
    every class{abstract} => super-to some class{!abstract}.
}
```

Fig. 1. Defining and evaluating a meta-model property.

cardinality in navigation paths, reachability from/to classes, and (a)cyclicity. Altogether, *mmSpec* promotes first-class primitives for elements (like paths or inheritance hierarchies) that need to be checked in meta-models frequently.

To evaluate a property on a meta-model, our Java-coded evaluator gathers the meta-model elements that match the property filter (or all of them, if there is no filter), verifies which ones meet the condition, and checks whether the size of the resulting subset is consistent with the selector's quantifier.

The bottom left of Fig. 1 shows a property using *mmSpec*. It states that every class (selector) that is abstract (filter) should have some concrete child class (condition), or as it is expressed, be *super to some class that is not abstract*. If an abstract class does not fulfil this condition, it is useless because it cannot be instantiated. The meta-model in Fig. 1 (taken from the zoo) does not satisfy the property, as class BusinessEntityPropertySet is abstract with no children.

Each property can be assigned a description of its intention, which gets shown in the *Test Result* view (see right of Fig. 1). While this view summarizes the results of all evaluated properties, the *Property Assessment* view highlights the relevant elements that did (green-coloured) or did not (red-coloured) fulfil a particular property. In this case, the class BusinessEntityPropertySet is displayed in red when we double-click on the property in the *Test Result* view.

The language offers abstraction mechanisms to package properties into functions with parameters. When the functions are called, it is possible to include 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.

Finally, *metaBest* supports batch evaluation of a set of properties over a set of meta-models, generating a CSV file with the results. In this way, we have managed to deliver the results of evaluating a library of meta-model quality issues for a large number of meta-models, as we explain in the following sections.

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 attribute name 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) – a.k.a. efferent couplings (Ce).
M03	No class is referred from too many others (5-max by default) – a.k.a. afferent couplings (Ca).
M04	No hierarchy is too deep (5-level max by default) – a.k.a. depth of inheritance tree (DIT).
M05	No class has too many direct children (10-max by default) - a.k.a. number of children (NOC).

Table 1. Library of meta-model quality properties.

2.2 A library of quality properties for meta-models

In order to test the quality of meta-models, we have built an *mmSpec* library covering typical mistakes (some of them from [2]) that designers tend to commit, as well as others that may jeopardize a basic level of meta-model quality. The library has four categories of issues, depending on their nature and relevance:

Design. Properties signalling a faulty design (an error).
Best practices. Basic design quality guidelines (a warning).
Naming conventions. For example, use of verbs, nouns or pascal/camel case.
Metrics. Measurements of meta-model elements and their threshold value, like the maximum number of attributes a class should reasonably define. Most metrics are adapted from the area of object-oriented design [6].

Table 1 lists the properties from these categories. To illustrate *mmSpec*'s expressiveness, next we show the formulation of a property from each category:

– *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).

– *BP03: There is a root class that contains all others.* This is a common best-practice in EMF, where meta-models define a class from which all other classes can be reached through composition relations. Its encoding is:

```
strictly 1 class {cont−root {absolute}} => exists.
```

A class satisfying cont-root is the root of a containment tree; if the root is absolute, then it contains all classes. This illustrates how *mmSpec* provides primitives that simplify the definition and checking of meta-model properties.

– *N04: Every class is named in pascal-case, with a singular-head noun phrase.* To obtain intuitive class names, these should be composed by a sequence of words starting with capital letters, and with a singular noun as the last word [3]. For instance, WashingMachine is a good class name, but Washing_Machine and MachineWashing are not. The connection of *mmSpec* with WordNet enables checking whether a word is a singular noun.

```
every class => name = pascal−phrase{end{noun{singular}}}.
```

– *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.

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

3 Assessing the quality of existing meta-models

To evaluate our tool and have a measure of the quality of current meta-modelling practice, we have applied our library of quality properties to the Atlan Ecore zoo of 295 meta-models. We have chosen this repository as it is representative of the meta-models that MDE practitioners build in practice. The size of the meta-models varies from tiny ones with just one class, to meta-models of medium size, the largest one having 699 classes. This is interesting as one of our goals is to detect whether the kind of quality issues depends on the meta-model size, and whether big meta-models are faultier (even if in average) than smaller ones.

Fig. 2 shows the number of quality issues detected in the analysed meta-models. Interestingly, only 5 meta-models have no issue, while no meta-model contains more than 22. The average number of issues per meta-model is 7.26.

Regarding the distribution of issues according to their kind, Fig. 3 shows how many meta-models fail each property from Table 1. *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: D01 and D02. D01 checks the absence of repeated attributes in a hierarchy (see *D01* in Fig. 4 for a faulty example), while D02 checks that the upper bound of features is not 0. However, 110 meta-models fail property D09 (37% of analysed meta-models). 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, as shown in Fig. 4. 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.

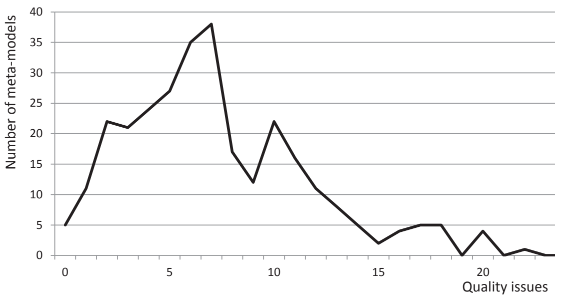


Fig. 2. Number of quality issues in meta-models.

Surprisingly for a set of EMF meta-models, the top unmet property is BP03, an EMF best practice that states the need for a root class whose instances may contain the whole model tree. Fig. 4 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.

The next two properties not satisfied by more meta-models are N03 and N04, which are naming conventions. N03 demands the verbalization of binary association names (e.g., reaches). N04 checks the conventions for class names, as explained in Section 2.2.

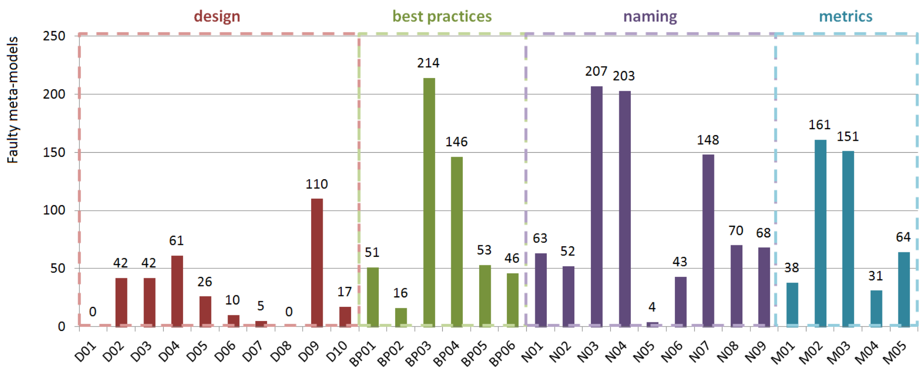


Fig. 3. Number of meta-models that contain issues of a certain type.

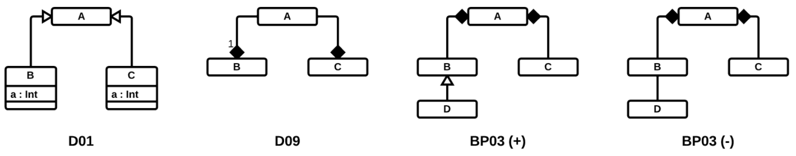


Fig. 4. Some quality issues of the library.

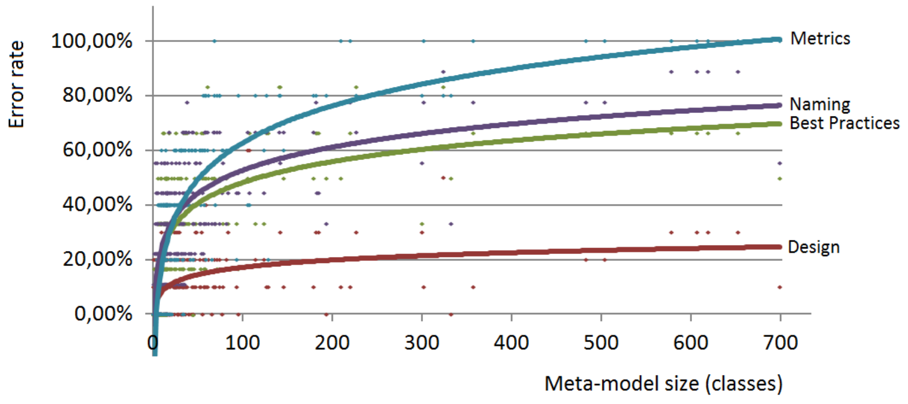
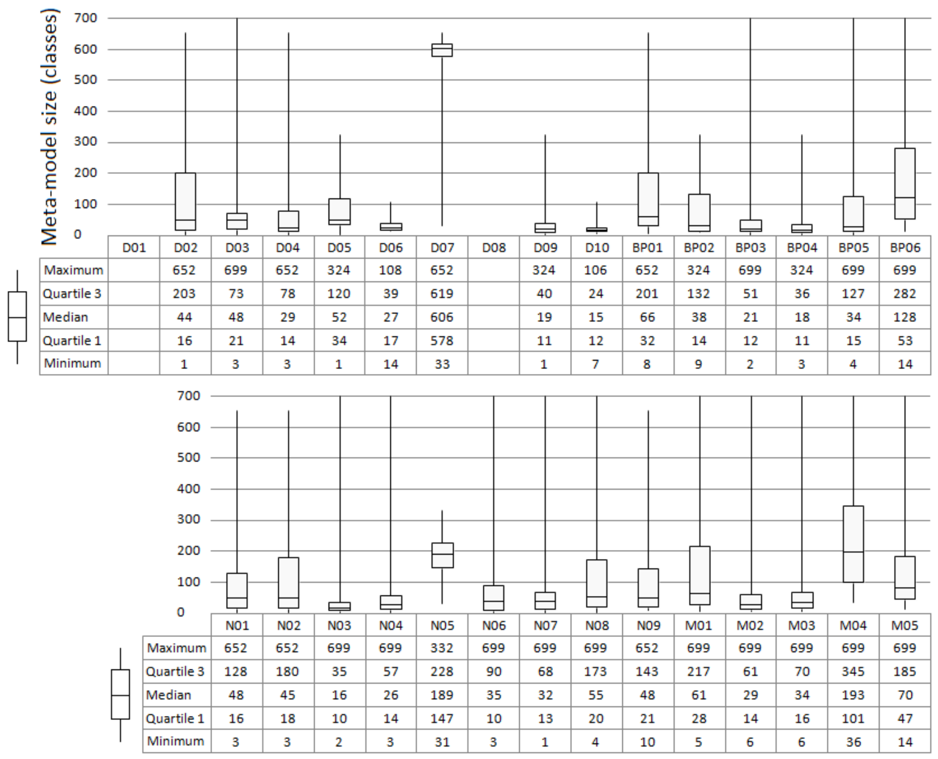


Fig. 5. Percentage of non-fulfilled issues in each category, w.r.t. meta-model size.

Regarding the error rate evolution of each category of issues with respect to the meta-model size (measured in number of classes), Fig. 5 shows that all categories present an upward error trend as meta-models enlarge. The vertical axis in this diagram corresponds to the percentage of issues in the category that were not met by some meta-model. The growth ratio is higher in small/medium size meta-models (up to 100 classes). Then, the error growth is steadier, particularly on the *design* category, which remains around 20% even at the largest meta-model size. The issues of type *metrics* grow as the meta-model size increases, peaking 100% (i.e., all issues of the category fail in large meta-models). This might be comprehensible, since a greater number of classes usually demands a greater number of features and relationships. However, properties such as M01, M02 or M03 might be considered independent from the meta-model size, as they take care of the class feature overpopulation, which is a bad practice despite the meta-model size. More worrisome is the evolution of the *best practice* category. This category reflects less severe concerns than *design*, but they still are bad modelling practices. It is worth noting that most meta-models present a 40 to 65% error rate, which together with the *design*'s - almost permanent - 20%, denote an average design quality that may be improved.

Fig. 6 shows the distribution of meta-model sizes where each issue type tends to occur. For most properties, faulty meta-models have between 1 and 200 classes. However, properties M04, BP06, D07 tend to fail in large meta-models. This is natural in some cases, e.g., D07 checks the presence of overridden at-



	D01	D02	D03	D04	D05	D06	D07	D08	D09	D10	BP01	BP02	BP03	BP04	BP05	BP06
Maximum		652	699	652	324	108	652		324	106	652	324	699	324	699	699
Quartile 3		203	73	78	120	39	619		40	24	201	132	51	36	127	282
Median		44	48	29	52	27	606		19	15	66	38	21	18	34	128
Quartile 1		16	21	14	34	17	578		11	12	32	14	12	11	15	53
Minimum		1	3	3	1	14	33		1	7	8	9	2	3	4	14

	N01	N02	N03	N04	N05	N06	N07	N08	N09	M01	M02	M03	M04	M05
Maximum	652	652	699	699	332	699	699	699	652	699	699	699	699	699
Quartile 3	128	180	35	57	228	90	68	173	143	217	61	70	345	185
Median	48	45	16	26	189	35	32	55	48	61	29	34	193	70
Quartile 1	16	18	10	14	147	10	13	20	21	28	14	16	101	47
Minimum	3	3	2	3	31	3	1	4	10	5	6	6	36	14

Fig. 6. Meta-model size dispersion by property.

tributes in inheritance hierarchies. Instead, properties D06, D09, D10, BP04, N03 tend to occur in small meta-models. The fact that 3 important design properties fail in small meta-models might mean that such meta-models were built by more unexperienced designers, compared to large ones. As a matter of fact, properties with a greater number of failure occurrences (as seen in Fig. 3: D09, BP03, BP04, N03, N04, N07, M02, M03) mainly appear in really small meta-models.

Finally, if we look at the average number of quality issues per class, we find that the most frequent categories of issues are *best practices* and *naming conventions* (0.18 issue occurrences per class in both cases). *Design* (0.07 issues per class) and *metrics* (0.05 issues per class) are less frequent; nonetheless, if they are considered together, the error rate seems worrisome at least.

3.1 Discussion

From the analysis of the meta-model repository, we realize that a way to improve the quality of meta-models is the inclusion of these quality checks in the meta-modelling tools, for example, to discover problems like D09. Actually, for some of these problems (like the ones related to metrics) the tool could trigger some refactoring suggestions. Regarding naming conventions, we noticed the usefulness of having integrated "smart" spell checkers (i.e., to check correctness of names in camel-case).

It is worth mentioning that *mmSpec* is integrated with *metaBup* [10], a tool for the example-based construction of meta-models. This means that the meta-model builder can check these quality issues during meta-model construction.

4 Related work

We could use OCL instead of *mmSpec*. However, OCL expressions tend to be more complex, as OCL lacks primitives (for gathering paths or inheritance hierarchies) which are part of *mmSpec* and have been designed to express properties on meta-models. Moreover, *mmSpec* supports the visualization of problematic elements (i.e., properties do not report just true/false). A comparison of OCL and *mmSpec* is available at: `http://www.miso.es/tools/metaBest.html`.

In [7], quality properties are defined as QVT-Relations transformations which produce a model with the problems in a meta-model. They define a catalog of problems for MOF-based meta-models, categorized into: syntactic (i.e., well-formedness constraints), semantic (i.e., poor design choices), and convention. Interestingly, *mmSpec* fulfils the features that [7] demands from any automated model verification approach: it is declarative, generic, flexible (though not standard), direct, and it has easy-to-inspect reporting facilities. Some of its primitives are specific for meta-model verification and would be difficult to specify with QVT-Relation patterns.

In [2, 4], quality properties of conceptual schemas are formalised in terms of quality issues, which are conditions that should not happen. Our approach also aims at detecting errors or bad smells; however, we focus on meta-models (not schemas). Thus, our library considers all issues in [2] that are meaningful in meta-modelling, as well as others specific to meta-models (like the existence of a root for EMF meta-models). While in [2], the method is evaluated on schemas developed by students, our library is applied to a public repository of meta-models built by developers. The same authors propose guidelines for naming UML schemas in [3], for which they provide a tool [1]. Interestingly, [3] presents a study on the effectiveness of current UML modelling environments for building schemas (not meta-models), and concludes that by including more quality issues in the IDEs, the quality of the developed schemas increases.

Some works aim at characterizing meta-model quality. For example, in [5], the authors adapt the ISO/IEC 9126 for meta-models, proposing concepts like completeness, conciseness, detailedness or complexity. However, there is no concrete proposal on how to measure such properties.

Few works analyse the quality of real meta-models. In [13], the authors take some basic size metrics (e.g., number of classes) over meta-models from different repositories (including the ATL zoo). In the same line, [12] correlates meta-model metrics, like the usage of inheritance w.r.t. meta-model size. Instead, we focus on detecting patterns that may indicate flaws in meta-models.

5 Conclusions and future work

In this paper, we have introduced *mmSpec*, a language directed to the specification of properties to be checked on meta-models, and *metaBest*, a tool to visualize and report the problematic elements. We have used both to build a catalog with 30 meta-model quality properties, which has been evaluated over 295 meta-models. The obtained results show that most meta-models contain some issue; hence, the community would benefit from integrated tool support (like *metaBest*) for checking quality properties during meta-model construction.

In the future, we plan to analyse correlations between meta-model flaws, provide a catalog of quick fixes and recommendations, and support the creation of user-defined categories for properties. We also plan to develop a further language for meta-model testing based on constraint solving.

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

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'12*, volume 7532 of *LNCS*, pages 501–514. Springer, 2012. See also `http://helios.lsi.upc.edu/phd/catalog/issues.php`.
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. F. Bertoa and A. Vallecillo. Quality attributes for software metamodels. In *QAOOSE'10*, 2010.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
7. 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.
8. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *ICSE*, pages 471–480. ACM, 2011.
9. J. J. López-Fernández, E. Guerra, and J. de Lara. Meta-model validation and verification with MetaBest. In *ASE*, pages 1–4 (to appear). ACM, 2014.
10. J. J. López-Fernández, J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Example-driven meta-model development. *SoSyM*, in press, 2014, see also `http://www.miso.es/tools/metaBUP.html`.
11. G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
12. J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Mining metrics for understanding metamodel characteristics. In *MiSE*, pages 55–60. ACM, 2014.
13. J. R. Williams, A. Zolotas, N. D. Matragkas, L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. What do metamodels really look like? In *EESS-MOD@MoDELS*, volume 1078 of *CEUR*, pages 55–60, 2013.