D2.3 – Integrated tools supporting the development processes

Version 1.0
15 November 2015
Final

Public Distribution

Autonomous University of Madrid

Project Partners: ARMINES, Autonomous University of Madrid, BME, IKERLAN, Soft-Maint, SOFTEAM, The Open Group, UNINÓVA, University of York

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Project Partners accept no liability for any error or omission in the same.

© 2015 Copyright in this document remains vested in the MONDO Project Partners.
## PROJECT PARTNER CONTACT INFORMATION

<table>
<thead>
<tr>
<th><strong>ARMINES</strong></th>
<th><strong>Autonomous University of Madrid</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Massimo Tisi</td>
<td>Juan de Lara</td>
</tr>
<tr>
<td>Rue Alfred Kastler 4</td>
<td>Calle Einstein 3</td>
</tr>
<tr>
<td>44070 / 3 Nantes Cedex, France</td>
<td>28049 Madrid, Spain</td>
</tr>
<tr>
<td>Tel: +33 2 51 85 82 09</td>
<td>Tel: +34 91 497 22 77</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:massimo.tisi@mines-nantes.fr">massimo.tisi@mines-nantes.fr</a></td>
<td>E-mail: <a href="mailto:juan.delara@uam.es">juan.delara@uam.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>BME</strong></th>
<th><strong>IKERLAN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Daniel Varro</td>
<td>Salvador Trujillo</td>
</tr>
<tr>
<td>Magyar Tudosok korutja 2</td>
<td>Paseo J.M. Arizmendiarrrieta 2</td>
</tr>
<tr>
<td>1117 Budapest, Hungary</td>
<td>20500 Mondragon, Spain</td>
</tr>
<tr>
<td>Tel: +36 146 33598</td>
<td>Tel: +34 943 712 400</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:varro@mit.bme.hu">varro@mit.bme.hu</a></td>
<td>E-mail: <a href="mailto:strujillo@ikerlan.es">strujillo@ikerlan.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Soft-Maint</strong></th>
<th><strong>SOFTTEAM</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vincent Hanniet</td>
<td>Alessandra Bagnato</td>
</tr>
<tr>
<td>Rue du Chateau de L’Eraudiere 4</td>
<td>Avenue Victor Hugo 21</td>
</tr>
<tr>
<td>44300 Nantes, France</td>
<td>75016 Paris, France</td>
</tr>
<tr>
<td>Tel: +33 149 931 345</td>
<td>Tel: +33 1 30 12 16 60</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:vhanhniets@sodifrance.fr">vhanhniets@sodifrance.fr</a></td>
<td>E-mail: <a href="mailto:alessandra.bagnato@softeam.fr">alessandra.bagnato@softeam.fr</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>The Open Group</strong></th>
<th><strong>UNINOVA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott Hansen</td>
<td>Pedro Maló</td>
</tr>
<tr>
<td>Avenue du Parc de Woluwe 56</td>
<td>Campus da FCT/UNL, Monte de Caparica</td>
</tr>
<tr>
<td>1160 Brussels, Belgium</td>
<td>2829-516 Caparica, Portugal</td>
</tr>
<tr>
<td>Tel: +32 2 675 1136</td>
<td>Tel: +351 212 947883</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
<td>E-mail: <a href="mailto:pmm@uninova.pt">pmm@uninova.pt</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>University of York</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimitris Kolovos</td>
</tr>
<tr>
<td>Deramore Lane</td>
</tr>
<tr>
<td>York YO10 5GH, United Kingdom</td>
</tr>
<tr>
<td>Tel: +44 1904 325167</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:dimitris.kolovos@york.ac.uk">dimitris.kolovos@york.ac.uk</a></td>
</tr>
</tbody>
</table>
### DOCUMENT CONTROL

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial draft</td>
<td>23/09/15</td>
</tr>
<tr>
<td>0.2</td>
<td>Committed version to the project repository</td>
<td>30/10/15</td>
</tr>
<tr>
<td>0.9</td>
<td>Incorporated suggestions from BME</td>
<td>10/11/15</td>
</tr>
<tr>
<td>1.0</td>
<td>Incorporated suggestions from UNINOVA</td>
<td>15/11/15</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Requirements and Transition to Design ................................................................. 6  
   1.1 Gathering Requirements with DSL-maps .......................................................... 6  
      1.1.1 Tool support ......................................................................................... 7  
   1.2 From DSL-maps to an initial meta-model design .................................................. 9  
   1.3 Gathering Requirements with sketches ............................................................... 13  
   1.4 Transition to Design: From Sketches to Meta-Models ........................................ 15  
      1.4.1 Parsing model examples .......................................................................... 15  
      1.4.2 Meta-model induction ............................................................................. 16  
      1.4.3 Meta-model design refactorings and recommendations ............................. 17  
      1.4.4 Tool support ......................................................................................... 18  
2. Testing .................................................................................................................... 20  
   2.1 Specification-based testing with mmSpec ......................................................... 21  
   2.2 Unit testing with mmUnit ................................................................................ 25  
   2.3 Reverse testing with mmXtens ......................................................................... 28  
   2.4 Tool Integration .............................................................................................. 32  
3. Related Work .......................................................................................................... 33  
   3.1 Extracting and representing requirements for DSMLs ....................................... 33  
   3.2 Testing Meta-Models ...................................................................................... 34  
REFERENCES ............................................................................................................. 36
EXECUTIVE SUMMARY

Model-Driven Engineering (MDE) is a software engineering methodology aiming at reducing the accidental complexity of software systems by promoting the use of models that focus on the essential complexity of systems, hence being first-class artefacts of the software development process. Such models are often described using Domain-Specific Modelling Languages (DSMLs) rather than with general purpose modelling languages like UML [WHR14].

However, as MDE is increasingly applied to larger and more complex systems, the models and the modelling languages may become large and unmanageable. This poses several challenges, like the lack of systematic approaches for the engineering of DSMLs, which becomes more acute when such DSMLs become large.

This document describes tool support for different activities needed for constructing DSMLs following sound engineering principles. In particular, we describe tool support for:

- Describing and organizing DSML requirements using a flexible notation, called DSL-maps (similar to the widely used mind-maps), and the transition into an initial meta-model design draft using customizable transformations. This notation is integrated into the DSL-tao tool.

- Support for DSML requirements in the form of drawings/sketches, and the automated derivation of an initial meta-model design. The tool supporting this approach is called metaBUP.

- Testing the meta-model design using three different languages: mmUnit (for unit-tests), mmSpec (for formalizing and testing requirements at the meta-model level), mmXtens (for expressing interesting features to be found in models, and the automatic generation of such models). The tool supporting these languages is called metaBEST.

The tool to design meta-models and to produce modelling environments for them (implementation), DSL-tao, was described in deliverable D2.2. The methodological guidelines on how to use the supported techniques are described in deliverable D2.4. This document describes DSL-maps, metaBUP and metaBEST, and how they have been integrated to yield a coherent family of tools covering the complete DSML construction life-cycle.

The developed tools are available at:

- DSL-tao and DSL-maps: http://miso.es/tools/DSLtao.html
- metaBUP: http://miso.es/tools/metaBUP.html
- metaBEST: http://miso.es/tools/metaBest.html

The web sites include source code, executables, user guides, screenshots and videos.
1. **Requirements and Transition to Design**

A DSML should contain useful, appropriate primitives and abstractions for a particular application domain. Hence, the input from domain experts and their active involvement in the meta-model development process are essential to obtain effective, useful DSMLs [KKP09, KP09, LCGdL15, MHS05, V13].

While we analyse different alternatives to requirements representation and analysis for DSMLs in Section 4, here we present two approaches that have been tackled in the MONDO project.

1.1 **Gathering Requirements with DSL-maps**

DSL-maps is a notation to visually gather and organize requirements for DSMLs, inspired by mind-maps [WM06]. A mind-map is a diagram used to organize information in a visual way. The objective is to foster brainstorming in the initial phases of DSML construction, and to organize hierarchically the different requirements along axes. Figure 1 shows a DSL-map as supported in DSL-tao.

![Figure 1: DSL-map (excerpt) for the IKERLAN case study](image)

The nodes in the diagram are called “ideas”. There is a root idea named “Wind Turbines”, and further ideas (“Structure”, “Behaviour”) stem from the root. Different branches represent the different aspects or concerns to be considered in the DSML. Branches are coloured differently to ease distinction between different DSML concerns.

Ideas are indexed taking into account their distance to the root (Structure and Behaviour are indexed as 1.1 and 1.2, Component is labelled 1.1.2). Ideas can have attached notes, in order to explain or emphasize some aspect of the idea. In Figure 1, the root idea has an attached label, shown as a yellow rectangle.

There are two kinds of edges in a DSL-map: *hierarchy* edges and *references*. Hierarchical edges connect an idea to its parent, and can optionally be labelled. For example, the connection between “Wind Turbines” and “Structure” is a hierarchical edge. Hierarchical edges span a tree structure. Reference edges may connect ideas in different branches of the DSL-map, are directed and can also be labelled. Figure 1
shows one reference edge, labelled “stored into”, connecting ideas “Component” and “Subsystems”.

Ideas may have a description and a list of features, represented as key-value pairs. These are used to provide details or features of the container idea.

Figure 2 shows the meta-model for DSL-maps with the mentioned concepts. References are modelled by the Link class, while hierarchical edges are modelled by the reference contains. Please note that more than one root idea can be present in a DSL-map.

Figure 2: Meta-model for DSL-maps

1.1.1 Tool support
DSL-tao includes an editor for DSL-maps, and provides a suitable perspective to work with DSL-map diagrams. The perspective consists of four areas, as seen in Figure 3:

1. The project explorer.
2. A canvas where the DSL-map is built and displayed.
3. The palette, with the elements that can be added to the diagram.
4. The view area, which collects the properties of the diagram elements.

Figure 3: Analysis perspective showing the editor structure.
Next, we describe the main functionalities of the editor

**Creating DSL-tao projects and diagrams.** The tool manages DSL-tao projects and diagrams. Figure 4 shows how to create a new DSL-tao project or diagram by invoking a wizard from the “File” menu of Eclipse.

![Figure 4: Creating a new DSL-tao project and diagram.](image)

A DSL-tao project provides an environment with all the resources needed for the construction process of DSMLs aided by patterns. The new project includes a repository of patterns (within the folder called "patterns"), so the user has the possibility of adding new ones, and deleting and updating the predefined ones. The project structure and the pattern definition model are showed in Figure 5.

![Figure 5: DSL-tao project structure.](image)

**Interactive creation of DSL-maps.** The editor permits creating the hierarchy of concepts, links, and features as well as adding notes to the concepts. The addition of elements to the diagram is performed by “drag and drop” from a palette into the correct diagram container.
A root idea is created dropping the idea in the canvas, while a sub-idea is created dropping the idea on top of the parent idea. Figure 6 shows the creation of a root idea (left) and a sub-idea (right). The colour and the classification indexes are automatically handled by the tool.

![Figure 6: Creating root ideas (left) and sub-ideas (right)](image)

Ideas have a name and description, which can be edited using the “Properties” view and corresponding tab as shown to the left of Figure 7. The name of the idea can also be changed directly on the associated pictogram in the canvas. Ideas can have associated features as lists of key/value pairs, which can be edited from the property view, in the “Features” tab.

![Figure 7: Properties of ideas (left) and defining features for ideas (right).](image)

1.2 FROM DSL-MAPS TO AN INITIAL META-MODEL DESIGN

DSL-maps can help in the transition to a meta-model design in two ways: by a pattern recommender and by an automated transformation from the DSL-map into an initial meta-model draft.

**Pattern recommendation.** The DSL-maps editor is equipped with a pattern assistant, which can recommend suitable patterns to be used in the design. These patterns may be those predefined in DSL-tao, or the designer can define their own. The recommended patterns get attached to the corresponding idea. Figure 8 shows how the assistant is invoked. The assistant analyses the name of the currently selected idea, and its description, and compares the different words with the descriptive tags and roles of the patterns in the DSL-tao repository. Tags for patterns are hierarchically organized, from more general (e.g., “behaviour”) to more specific tags (e.g., “workflow”, “state machines”).

![Figure 8: Pattern recommendation assistant](image)
Suggested patterns are ranked according to the number of matching tags. Then, the user is presented with a dialog (see the left of Figure 9), where he can choose different variants of the suggested pattern(s), or to discard the suggestions. In Figure 9, the assistant is recommending to use some variant of the State machine pattern. Once a pattern is selected, it becomes attached to the idea, as shown to the right of the Figure.

Figure 8: Invoking the pattern assistant

Figure 9: Choosing a variant of the suggested pattern (left). Pattern attached to an idea (right).

**Automated transformation.** In addition, the tool contains a wizard that produces an initial draft meta-model from the DSL-map. This transformation is customizable, and works as follows:

- A package is created for the DSL-map.
- A class is created for each idea. There is the option to put the class name in singular form, as depicted in Figure 10.
By default, the transformation produces composition references between an idea and its sub-ideas. However, this rule may change depending on the label of the hierarchical edge. If the edge is tagged as “may-be”, “can-be”, or “is-a”, then an inheritance relation is produced instead in the meta-model, as shown in Figure 11.

If a hierarchical edge in the DSL-map is translated into a composition reference in the meta-model, some heuristics can be applied. First, there is the option to generate bidirectional references between the created parent and children classes, as shown in Figure 12.

Regarding the cardinality of the created compositions, by default it is set to 1..1. However, if the DSL-map edge is labelled as “can”, “may”, “may have” or “can have”, then the lower bound is set to 0. If the target idea is a plural name, the upper bound is set to *. Plural names are detected using simple rules checking terminations and a catalogue of irregular forms, similar to [LCGdL15]. These two options are schematically shown in Figure 13.
Features in the mind map are translated into attributes for the corresponding class. The type of the attribute is induced from the “value” of the feature. Hence, if the value is an integer, then the type is “int”; if a floating point number, then the type is “float”; if “true” or “false”, then the type is “boolean”, and otherwise the type is “String”. Alternatively, there is the option to create a class for each feature. These options are shown in Figure 14.

Notes in the DSL map are transformed into annotations in the meta-model, as shown in Figure 15.

If some idea has an attached recommended pattern, then the selected variant is produced in the generated meta-model, as shown in Figure 16.
1.3 GATHERING REQUIREMENTS WITH SKETCHES

Another way to capture requirements about a DSML is by asking the domain experts to produce example models. This approach is useful, as in some domains, experts might find easier to draft example models first, and then abstract those into classes and relations in a meta-model. As Oscar Nierstrasz put it, “... in the real world, there are only objects. Classes exist only in our minds” [N10]. In this way, domain experts and final users of MDE tools are used to working with models reflecting concrete situations of their domain of expertise, but not with meta-models. Asking them to discuss at the abstract level of language primitives or meta-models is often too demanding if they are not MDE experts. In general, an early exploratory phase of model construction, to understand the main concepts of the language and document the language requirements, is recommended for DSML engineering [CSG11, KKP09, LCGdL15].

Figure 16: Instantiating the recommended pattern.

Figure 17 shows the wizard that permits customizing the different aspects of the transformation. Hovering over each option makes the image change, to illustrate the different option.

Figure 17: Customizing the generation of the initial meta-model through a wizard
While MDE experts are used to work with specialized meta-modelling tools – like those provided by Eclipse EMF [SBP08] – this is seldom the case for domain experts. These may find easier to use sketching and drawing tools in the style of PowerPoint or Visio to build models and examples, than using, e.g., the EMF’s tree-based editor.

This is why, we do not provide tool support for sketching, but let the users use their favourite tool. However, as we will see later, we can automatically process sketches generated using the yED (http://www.yworks.com/en/products_yed_about.html) and DIA (http://projects.gnome.org/dia/) editors. Hence, the examples drafted with these tools are not mere documentation, but are parsed into an internal representation, and then used to automatically induce a meta-model.

Following with the IKERLAN case study, as an example, Figure 18 shows a model fragment as would be drawn by a domain expert (an electrical engineer) using a drawing tool (yEd in this case). The fragment contains a component with an input and an output port.

![Figure 18: An example fragment for the IKERLAN case study](image)

In order to link the symbols used in the sketches with their meaning, we use another diagram serving as a kind of legend, or basic reference model for them. Figure 19 shows the legend for the example. This is a natural, technology-agnostic way for non-experts to specify the meta-model types, resembling the legend of a map, as suggested by Bézivin in [B05].
1.4 Transition to Design: From Sketches to Meta-Models

Example models are not passive documentation describing DSML requirements, but can be processed and used to automatically derive an initial meta-model. Figure 20 shows an overview of this process (simplified from [LCGdL15]). In a first stage, the domain experts draft example models using drawing tools. These examples are parsed into an internal representation (step 2). Then, they are used to induce an initial meta-model draft (step 3). This can be refactored (step 4), following suggestions from an automated assistant [LCGdL15], which detects opportunities for improvement and smells of bad designs (e.g., if several classes are found sharing a number of attributes, the assistant would recommend creating a parent class with the shared features). These steps are supported by our metaBUP tool (http://miso.es/tools/metaBUP.html) and explained in the following subsections.

![Figure 20: Inducing a meta-model from sketches](image)

1.4.1 Parsing model examples

Our parser is able to recognise DIA and yED drawings. It assigns type names to objects using the names in a legend, which is also represented as a drawing, as Figure 19 showed. Listing 1 shows the parsed textual representation of the example in Figure 18.

```
fragment example1 {
    component : Component {
        @overlapping ref ports - out, in
    }
    out : OutPort {
        @overlapping ref component = component
    }
    in : InPort {
        @overlapping ref component = component
    }
}
```

Listing 1: Example fragment of Figure 18, once parsed

Our parsing detects spatial relationships, like containment, adjacency and overlapping. These relations can be configured to be relevant for the abstract syntax. As Figure 21 shows, the detection of @overlapping triggers the creation of links (ports, component) between the corresponding objects. Of course, arrows between graphical objects are
interpreted as links as well. As we will see later, these annotations can be used by the mmXtens testing language, for the automatic rendering of models. Please note that other annotations, not coming from spatial relations, but to express constraints are also supported. The full list of supported annotations is described in [LCGdL15].

1.4.2 Meta-model induction

Whenever the user enters a new fragment, the meta-model is updated accordingly to consider the new information. The annotations in the fragment are transferred to the meta-model, and this may trigger meta-model refactorings. Any conflicting information within and across fragments, like the assignment of non-compatible types for the same field, is reported to the user and automatically fixed whenever possible. Moreover, a virtual assistant provides suggestions on possible meta-model refactorings, applicable on demand.

Given a fragment, our algorithm proceeds by creating a new class in the meta-model for each object with distinct type. If a class already exists in the meta-model due to the processing of previous fragments or other objects within the same fragment, then the class is not newly added. Then, for each slot in any object, a new attribute is created in the object’s class, if it does not exist yet. Similarly, for each reference stemming from an object, a reference type is created in its class, if it does not exist. The lower bound of references is set to the minimum number of target objects connected to each object of source type, while the upper bound is set to the maximum number of target objects in the fragment. Actually, the user can configure the defaults for the lower (0 or the minimum in the fragment) and upper (unbounded or the maximum in the fragment) bounds of references. In case of selecting an unbounded maximum by default, the algorithm checks if the reference name is singular, in which case it keeps the maximum of the fragment (and the user gets the recommendation of changing the name to plural if such maximum is greater than one).

Once the meta-model has been produced, the user is allowed to decrease the lower bound and augment the upper bound of any reference. Moreover, as a consequence of processing a new fragment, the cardinality of a reference in the induced meta-model might also be relaxed: its new lower bound is set to the minimum between its current value in the meta-model and the minimum in the fragment, while its new upper bound is set to the maximum between its current value in the meta-model and the maximum in the fragment. Figure 21 shows a scheme of this situation.

![Figure 21: Processing a reference with different cardinalities in the meta-model and a fragment](image-url)
If two references with the same name and stemming from objects with the same or compatible type, point to objects of different type, our algorithm creates an abstract superclass as target of the reference type, with a subclass for the type of each target object. This situation is illustrated in Figure 22, where the new abstract class BC is created as parent of both B and C. Should the B class be abstract and the C object define features that are compatible with those in B, then BC would not be generated, but the new class C would be created as a child of B. The lower bound of the reference type r is set to \( \min(a, 1) \) because it should accept at least one element (the one provided in the fragment), but the previous lower bound (value a) may be zero. As the fragment has just one reference of type r, the upper bound b of the reference is kept in the meta-model. Any automatic design decision made by the induction algorithm is reported to the user, who can change the design. More details on the algorithm can be found at [LCGdL15].

Figure 22: Processing a reference with different target type in the meta-model and a fragment

As an example, Figure 23 contains the meta-model resulting from processing the fragment of Listing 1. The abstract class Port is created, because both an InPort and an OutPort object have been included in reference ports. Please note that the algorithm initially sets the name of the new class as “InPortOutPort”, but the user has the chance to use a more appropriate name (in this case, it was set to “Port”).

Figure 23: Resulting meta-model from processing the fragment in Listing 1

1.4.3 Meta-model design refactorings and recommendations

The annotations in fragments are transferred from the fragments to the meta-model and may trigger refactorings in it. For example, Figure 24 shows a scheme of the refactoring triggered by the @general annotation applied to a reference, which is similar to the pull-up refactoring [F99]: It pulls up the annotated attribute or reference as general as possible in the inheritance hierarchy. If the annotated attribute or reference is shared by two classes that are not related through inheritance, then an abstract, parent class is created for them so that the attribute or reference can be pulled up (i.e., Fowler’s extract superclass refactoring [F99] is applied). The target end of the pulled reference receives
as lower bound the minimum of the original lower bounds, and as upper bound the maximum of the original upper bounds.

![Diagram of processing a reference with different target type in the meta-model and a fragment](image)

**Figure 24: Processing a reference with different target type in the meta-model and a fragment**

Additionally, we have integrated a virtual assistant which continuously monitors the meta-model to detect places where the meta-model design can be improved and recommend solutions, based on well-known design patterns, refactorings and style guidelines. Guidelines are divided into *structural* and *style* suggestions.

Structural recommendations include: inline class (merging two classes within one), pullup features (detects maximal sets of common features among classes, and proposes either pulling the features up to a common superclass, or creating a common abstract superclass), generalize references (creates a common abstract superclass for a set of classes that receive a set of references from another class), replace class by integer (replaces a featureless class with an integer attribute in the unique referencing class), remove abstract class (removes an intermediate featureless abstract class in an inheritance hierarchy).

Regarding naming recommendations, if a reference is multivalued but its name is singular, the assistant suggests changing the name to plural. If a reference is monovalued but its name is plural, the assistant suggests either changing the name to singular, or increasing the upper multiplicity to *. If an attribute name contains the name of the owning class as prefix, the assistant suggests the removal of the prefix (as recommended in [B90]). Further suggestions take care of the capitalization of feature and class names, reflecting widely used modelling style guidelines [AGO13b].

The full list of supported refactorings and recommendations can be checked at [LCGdL15].

### 1.4.4 Tool support

These processes are supported by our metaBUP tool. Some screenshots of its working scheme are shown in Figure 25. In the figure, a yED drawing, labelled as 1, is imported into the tool, and shown using the internal concrete textual syntax (label 2). Then, the fragment is further annotated (see the @general annotations) and used to update the current version of the meta-model. Such change may trigger some meta-model refactoring (e.g., including an extra abstract superclass), for which the input of the user might be required (step 3, which requires adding a name to such class). The resulting meta-model is shown in the window with label 4.
Figure 25: Induction process, as supported by metaBUP
2. **Testing**

While meta-models play a central role in MDE, they are often built in an ad-hoc way, without following a sound engineering process [LCGdL15, MHS05]. 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, 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 have created three complementary languages and tool support for meta-model Validation and Verification (V&V).

The first 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 [M95], a lexical database for English. mmSpec will be explained in Section 2.1.

The second language, called **mmUnit**, is similar to the xUnit framework [B94]. 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 [ICL13]. 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. mmUnit will be explained in Section 2.2.

The third language, called **mmXtens**, is used for the automated generation of example models conforming to the meta-model. The idea is producing example instantiations of the meta-model being developed, which both domain experts and engineers can inspect more easily to detect possible flaws in the meta-model and reason on the properties that instances should have. The example generation process can be fine-tuned using mmXtens, which allows constraining the number and type of objects in the example, stating their connectivity, and providing a seed model either in graphical or textual format. The semantics of mmXtens is given in terms of OCL, but it provides a more concise syntax. mmXtens will be explained in Section 2.3.
2.1 **SPECIFICATION-BASED TESTING WITH mmSpec**

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 in a mandatory way. 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 metamodel changes.

To facilitate the specification and testing of expected meta-model properties, we have created a DSL called mmSpec [LGdL14, LGdL14b] 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 26: Excerpt of mmSpec meta-model](image)

Figure 26 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:

\[
\langle\text{quantifier}\rangle \langle\text{selector}\rangle\{\langle\text{filter}\rangle\} \Rightarrow \langle\text{condition}\rangle.
\]

As an example, the property:

\[
\text{every class}\ \{\text{abstract}\} \Rightarrow \text{super}-\text{to}\ \text{some class}\{!\text{abstract}\}.
\]

uses the ClassSelector selector and the filter abstract. The condition \text{super}-\text{to}\ \text{some class}\{\ldots\} 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, 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.

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 [M95]. 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 27 shows some the evaluation of some quality criteria using metaBest. The tool contains a library of common quality criteria [AGO13b], and the Figure shows the mmSpec encoding of one of the properties (name N4, which demands each class to be named in pascal-case, with a singular-head noun phrase). The Figure also shows the result of evaluating the quality criteria on the example meta-model. In particular two properties fail, one (N3) demanding a certain style convention for association names, and the other one (N4), fails on classes “WTComponents” (ends with a plural name) and “DocumentElt” (“Elt” is not a valid noun). Please note that mmSpec properties are evaluated by checking the structure of the meta-model, with no need to look at meta-model instances.
<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Primitives</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence</td>
<td>exist</td>
<td>-</td>
<td>The simplest check, for ensuring the presence of elements.</td>
</tr>
<tr>
<td>Name</td>
<td>name</td>
<td>noun, verb adjective synonym prefix, suffix camel-phrase</td>
<td>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.</td>
</tr>
<tr>
<td>Abstractness</td>
<td>abstract</td>
<td>-</td>
<td>It states that a class is abstract (or the contrary with the ( \theta ) operator).</td>
</tr>
<tr>
<td>Features</td>
<td>with</td>
<td>inh</td>
<td>It checks the existence of a reference or an attribute in a class definition. The modifier controls whether inherited features should be considered.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>sub-to super-to depth width or-equal</td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td>Depth of hierarchy</td>
<td>inh-root inh-leaf</td>
<td>Depth of a class in the inheritance hierarchy, either from the root or from the leaves.</td>
<td></td>
</tr>
<tr>
<td>Depth of containment</td>
<td>cont-root cont-leaf absolute</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Class reachability</td>
<td>reach reached-from collect jumps cont inh strict</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Class owned-by</td>
<td>inh</td>
<td>The class a feature belongs to (with/without inheritance).</td>
<td></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>multiplicity</td>
<td>min max</td>
<td>Minimum and maximum multiplicity of a feature. In both cases, a fixed value or an interval can be given.</td>
</tr>
<tr>
<td>Attribute Qualifiers</td>
<td>type</td>
<td>-</td>
<td>The primitive type of an attribute.</td>
</tr>
<tr>
<td>Reference Qualifiers</td>
<td>Reference ends from to</td>
<td>inh</td>
<td>The source and target classes of a reference end.</td>
</tr>
<tr>
<td>Path Qualifiers</td>
<td>Path ends from through to</td>
<td>-</td>
<td>Paths starting, traversing or ending in the given classes. Any combination of primitives is valid.</td>
</tr>
<tr>
<td>Path settings</td>
<td>cont strict cycle</td>
<td>-</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 1: Main qualifiers for selectors and conditions

Hence, mmSpec is useful to express quality criteria or domain requirements at the meta-model level. It is useful for regression testing, where the defined properties can be re-checked upon changes in the meta-model.
Figure 27: Evaluating meta-model quality criteria in metaBEST
2.2 UNIT TESTING WITH mmUNIT

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. Thus, this approach involves domain experts in the validation of the DSML, which is vital to build correct, useful DSMLs [ICL13].

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 inline with modern, agile software engineering processes guided by tests [B03].

Our mmUnit language [LGdL14] has a concrete textual syntax. 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 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 28 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 28: Excerpt of mmUnit meta-model](image)

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 [LCGdL15].

As an example, Figure 29 shows a mmUnit test (in textual format) for the example IKERLAN case study. The test defines a context made of a Subsystem object (named subsystem), which contains a Component. Then, it specifies that this configuration is incorrect because subsystem.ensembles has type Architecture, and hence cannot hold objects of type Component. Evaluating the test on the meta-model of Figure 27 succeeds, because the context model is rejected by the meta-model exactly for the given reasons. The lower part of the image shows the actual result obtained in the metaBEST tool.

```
metamodel /*WT_test/WT.mUp*/ fragment metaup_fragment {
  subsystem: Subsystem {
    constraint refs ensembles = component
  }
  component: Component()
}
```

**Figure 29: Screenshot of metaBEST with a mmUnit fragment, and the result of its evaluation on the meta-model of Figure 27**

Figure 30 shows another test case, which sets a context made of an InPort, an OutPort and a Component object. The latter contains the OutPort. The assertion states that the test should fail because the InPort object should be contained in the Component object as well. Again, evaluating the test on the meta-model of Figure 27 succeeds, because the context model is rejected by the meta-model exactly for the given reasons.
Hence, altogether, with mmUnit, it is possible to document expectations on correct and incorrect models, and in the latter case the reasons for considering them incorrect. This is especially useful if the meta-model is to contain complex OCL invariants. The negative tests would document scenarios for non-conformance. Generally, mmUnit tests are useful both for regression testing and test-driven development.
2.3 REVERSE TESTING WITH mmXTENS

In “reverse testing”, the system produces example models (conformant to the current version of the meta-model), which then are checked by the domain expert to assert whether they meet his expectations. If they don’t, then the meta-model contains some error.

In order to ensure the generation of interesting example models, we have created the mmXtens language [LGdL15]. An mmXtens specification is made of a seed model and a set of extension constraints, specifying the conditions that extensions of the seed models should satisfy. Our engine translates the mmXtens specification into OCL and uses an OCL-based model finder [KG12] to produce a model satisfying the specification and conformant to the meta-model under test. The produced model (if any exists that satisfies the extension constraints) can be inspected to validate whether it complies with the requirements, the expectations about the meta-model, or the intuitions about the domain.

Listing 2 shows an excerpt of mmXtens syntax. An mmXtens specification (lines 1-4) can be tagged as positive (by default) or negative, depending on whether we expect the specification to be satisfiable or not. In the former case, we expect the system to produce a model that contains the given seed model fragment and satisfies the extension constraints, while in the latter, we expect that no such model exists. Providing a seed fragment is optional; if given, it does not need to be a full model, but it may contain just a set of initial objects together with their features of interest (i.e., the fragment may break some lower cardinality and OCL constraints, and objects may not specify values for uninteresting attributes). It is possible to define a list of extension constraints with conditions that the model to be generated should satisfy (line 3). These constraints are expressed using a simple syntax, and then internally translated into more complex OCL expressions that, transparently to the user, are used for generating the sought example model.

Listing 2: Excerpt of the definition of the mmXtens textual syntax
Extension constraints may refer either to specific objects in the seed fragment (line 9), or to arbitrary objects which can exist in the seed fragment or may need to be created new (line 10). In both cases, we can use a CONDITIONER stating required properties for the object.

A CONDITIONER (line 13) may require an object to be connected with some other using the keyword reference (line 14). If we use the keyword contain instead, then, the reference should be a composition. Optionally, we may specify a reference name using the keyword via. Finally, we can use a SELECTOR (line 18) to choose the target object of the reference. In the simplest case, it will be an object present in the seed fragment though it can also be a new object created in the extended model, or a set of objects of a given type (we give examples below).

Alternatively, a CONDITIONER may define requirements on the attribute values of the selected object (line 15 in the listing). In the current version of mmXtens, these requirements must be concrete values. Finally, conditioners can be combined using the logical primitives and, or a nd not (omitted in the listing for simplicity).

Extension constraints can also require properties on sets of objects of a certain type, without referring explicitly to an existing object (line 10). In such a case, we use a QUANTIFIER to select the objects. Valid quantifiers include intervals, strictly a given number, at least a given number, every object of a given type, no object of a given type, or some (i.e., at least one) object of a given type. If the type name is preceded by the keyword new, then, the selected objects must not belong to the fragment but they must be new in the extended model. If new is omitted, the selection is among both existing and new objects.

In this latter constraint type, it is also possible to define a FILTER to indicate required properties of the selected objects. The definition of filters is similar to the one for conditioners, and may include conditions over attributes and references.

An mmXtens specification may have zero or more extension constraints, and the extended model to produce must fulfil all of them. Moreover, we can customize the minimum and maximum number of objects in the model extension.

The abstract syntax of mmXtens has been defined through a meta-model, which is partly shown in Figure 31. As mmXtens is integrated with our two other languages for meta-model V&V, it shares the notion of seed fragment with mmUnit, but while mmUnit assertions are contained in class TestAssertionSet, mmXtens assertions are contained in ExtensionAssertionSet. Both mmXtens and mmSpec specifications follow a similar selector/filter/condition style, because our goal is to supply users with a homogeneous family of DSLs.

As an example, Listing 3 shows an mmXtens specification setting as seed fragment a Component with an input and an output port. Then, the extension constraints demand 3 new components, some input ports and some output ports. The system returns the model shown in Listing 4 in textual syntax. Large models in textual syntax become cumbersome to understand, and hence mmXtens can depict the model in graphical syntax, using yED. The graphical version of the model in Listing 4 is shown in Figure 32.
Figure 31: Excerpt of the mmXtens meta-model

Listing 3: mmXtens specification

Listing 4: Model satisfying the specification of Figure 33
Please note that each object is represented with the icon specified in the legend (see Figure 19), and spatial relationships are generated from the annotations (@overlapping between components and ports). Please note also that the generated models can be used as mmUnit tests, as both languages share the same syntax for specifying model fragments.

Hence, altogether mmXtens is useful to make the system produce models, which are then verified by the domain expert for conformity. If some produced model is deemed incorrect, then the meta-model needs to be revised. mmXtens is also useful to generate model to test other MDE artefacts, in addition to a meta-model. For example, it can be used to generate test cases for model transformations, or model storage tools.
2.4 **Tool Integration**

While DSL-tao, metaBUP and metaBEST can work as standalone tools, we have integrated them so that they can work together in a smooth way. This way, metaBEST tests can be applied on meta-models built with either metaBUP or DSL-tao.

From DSL-tao, we can create a metaBEST project, by exporting a meta-model within DSL-tao, as shown to the left of Figure 33. This triggers the metaBEST wizard shown to the right of the same Figure.

![Figure 33: Creating a metaBEST project from within DSL-tao](image1)

Figure 34 shows the created metaBEST project, with the imported meta-model. Once in metaBest, it is possible to define test cases, as illustrated in Sections 2.1—2.3.

![Figure 34: Resulting metaBEST project](image2)
3. **RELATED WORK**

Next, we review some works dealing with the extraction and representation of requirements for DSMLs (Section 4.1) and testing of meta-models (4.2).

3.1 **EXTRACTING AND REPRESENTING REQUIREMENTS FOR DSMLs**

Several approaches to represent and analyse DSML requirements have been proposed in the literature. In [MHS05] the authors suggest three different types of analysis methods for DSLs: informal, formal, or extract from code.

In the first case, the domain is analysed in an informal way. In [MHS05], it is reported as the most common scenario.

In the second case, some methodology, probably adapted from software analysis is used. In particular, the author suggests using Feature Oriented Domain Analysis (FODA) [KCH90]. FODA requires the construction of a feature model capturing commonalities and variabilities of the domain at hand, so that commonalities are built-in into the DSL engine whereas variabilities are supported as parameters to be set by the DSL user [MHS05]. For example, in [DP12] the authors develop a DSL to configure wikis, for which they first create a feature model characterizing the domain. The resulting language is based on mind-maps.

In the third scenario, the DSML is created by generalizing an existing legacy system by inspection or by using software tools, or a combination of both. As mentioned in [V13], this scenario requires factoring the knowledge into a language, where reasonable defaults are given for some of the framework features, increasing the level of abstraction and making framework use easier. An example of this scenario is [BGdL13], and automated frameworks to automate the transition from an API into a DSL is provided in [CTC15].

Our proposal fit both in the informal and formal approaches, while the third approach is not currently supported. In the informal approach, our sketches are able to collect the DSML requirements by example, in a graphical way. However, we can process the sketches and use them to generate a first version of the meta-model. In the more formal side, DSL-maps serve as a notation to reason about requirements of the language. We purposely refrained from a notation like feature models in order to promote flexibility and enable discussion between stakeholders. While DSL-maps lack a formal semantics as feature models have, the customizable transformation into an initial meta-model permits providing different semantics to the DSL-map elements.

In [WSA13], the authors use mind-maps in order to gather requirements for software systems. Then, such requirements are automatically transformed into a class diagram. However, the transformation is fixed, and is only able to produce a class for every idea, with compositions between them. In contrast, our DSL-maps can make use of patterns, and the customizable transformation is able to produce e.g., inheritance hierarchies or references. In [WSA14] the technique was evaluated, showing good results (in terms of time taken and quality of the result) with respect to producing a conceptual schema without creating a mind-map first. In [WSAL12], the same authors present a method to derive a feature model from a mind-map.
3.2 Testing Meta-Models

Most efforts towards the V&V of MDE artefacts are directed to test model management operations [GKR11], like model transformations [RW15], but few works target meta-model V&V. We take the classical view of V&V [B84]. 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 [B94]. For example, in [SW08], test models describe instances that the meta-model should accept or reject. In a different style, [CRPK12] 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 [PBO04] is similar, but using Eiffel as host language. None of these works provide support for asserting the expected test results, though having an assertion language tailored to meta-model testing would enable an intentional description of the test models, documenting and narrowing the purpose of the test.

Other proposals [KVE11, WGM08] expand general-purpose testing tools (e.g., JUnit) to enable the testing of DSML programs, not necessarily defined by meta-models. In [TO10], 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 [SBP08] 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 [BFL14] as a way 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 [RPKP08], 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, [RPKP08] presents an approach for checking meta-model integration. It relies on specifying meta-model properties in EVL [KPP09] (a variant of OCL), but as the authors recognise, using EVL/OCL to check meta-model properties is cumbersome, leading to complicated assertions and
demanding expert technical knowledge of the used meta-modelling framework. Moreover, OCL does not provide support for visualizing complex validation errors.

Other works define catalogues of quality criteria for meta-models [BV10] or conceptual schemas. In [EBL11], 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 [AGO12, AGO13], 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 [AGO13b], the same authors propose a set of guidelines for naming UML schemas, which can be validated using an Eclipse plugin [AGCO11]. 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 instance models from a meta-model, likely using constraint solving [CCR07, GKM14, WMP13]. 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 [CBS12] (where the generated snapshots are targeted to test cardinality boundaries) and [GBR05] (where a language to define object snapshots is proposed). In [BASS11], questionnaires with true/false questions are generated from the meta-model, and the domain experts perform the meta-model validation by answering the questionnaires.

Altogether, we are not aware of a tool supporting all phases in the development of DSMLs, as we do. Deliverable 2.4 will discuss on the different processes and guidelines to be used when designing DSMLs with our tools.
REFERENCES


D2.3. – Integrated tools supporting the development processes


D2.3 – Integrated tools supporting the development processes


