Ph.D dissertation

An agile process for the example-driven development of modelling languages and environments

Author: Jesús J. López-Fernández
Supervisors: Esther Guerra, Ph.D / Juan de Lara, Ph.D

Universidad Autónoma de Madrid
Madrid, Spain

Doctorado en Ingeniería Informática y Telecomunicaciones

May 2017

Abstract

Domain-Specific Modelling Languages (DSMLs) are heavily used in model-driven and end-user development approaches. Compared to general-purpose languages, DSMLs present numerous benefits like powerful domain-specific primitives, an intuitive syntax for domain experts, and the possibility of advanced code generation for narrow domains. While a graphical syntax is sometimes desired for a DSML, constructing graphical modelling environments is a costly and highly technical task. This relegates domain experts to a rather passive role in their development and hinders a wider adoption of graphical DSMLs.

The aim of this dissertation is achieving a simpler DSML construction process where domain experts can contribute actively. For this purpose, an example-based process for the automatic generation of modelling environments for graphical DSMLs is proposed. This way, starting from examples of the DSML likely provided by domain experts using drawing tools, the proposed system synthesizes a graphical modelling environment that mimics the syntax of the provided examples. This includes a meta-model for the abstract syntax of the DSML, and a graphical concrete syntax supporting spatial relationships.

Moreover, despite being essential activities in this field, there are scarce tools and methods supporting the Validation and Verification (V&V) of DSMLs. In order to fill this gap, three complementary meta-model V&V languages are presented. These languages provide means for testing a DSML through three V&V approches: unit testing, specification-based testing and reverse testing.

These two contributions receive tool support in a prototype application comprising *metaBup*, a system for building DSML editors from graphical examples, and *metaBest*, its corresponding testing suite. The process and the prototype have been validated through several experiments, including a user study.

Keywords: Model-Driven Engineering, Domain-Specific Modelling Languages, Meta-Modelling, Graphical Modelling Environments, Example-Based Meta-Modelling, Validation & Verification, Meta-Model Quality

Resumen

Los Lenguajes de Modelado de Dominio Específico (Domain-Specific Modelling Languages (DSMLs)) se utilizan con frecuencia en la Ingeniería del Software Dirigida por Modelos y en métodos de desarrollo por el usuario final. En comparación con los lenguajes de propósito general, los DSMLs presentan numerosas ventajas como la posibilidad de emplear primitivas de dominio específico, una sintaxis intuitiva para expertos en un dominio, y la posibilidad de generar código fuente complejo para dominios concretos. Algunos DSMLs tienen una sintaxis gráfica; sin embargo, construir entornos de modelado gráfico es una tarea altamente costosa a nivel técnico. Esto relega a los expertos del dominio a un rol pasivo en el desarrollo de dichos entornos, e impide una adopción más extensa de los DSML gráficos.

El propósito de este trabajo de tesis es lograr un proceso sencillo para la construcción de DSMLs, en el que los expertos del dominio puedan contribuir de forma activa. Para ello, se propone un proceso basado en ejemplos para la generación automática de entornos de modelado de DSMLs gráficos. El sistema propuesto sintetiza, a partir de ejemplos del DSML esbozados por el experto del dominio en herramientas de dibujo, un entorno gráfico de modelado que emula la sintaxis de dichos ejemplos. Esto abarca un meta-modelo para la sintaxis abstracta del DSML, y una sintaxis concreta que además soporta la inclusión de relaciones espaciales.

Adicionalmente, pese a tratarse de actividades esenciales en el campo del Desarrollo Dirigido por Modelos, existen muy pocas herramientas o métodos que den soporte a la Validación y Verificación (V&V) de DSMLs. Para cubrir esta necesidad se presentan tres lenguajes complementarios para la V&V de meta-modelos a través de tres métodos distintos: las pruebas unitarias, la comprobación de propiedades sobre el meta-modelo, y las pruebas inversas.

Las dos contribuciones se han implementado en dos prototipos: metaBup, un sistema para construir editores de DSMLs a partir de ejemplos gráficos, y metaBest, su correspondiente entorno de pruebas. Tanto el proceso como el prototipo han sido validados mediante diversos experimentos, entre los que se incluye una validación con usuarios.

Contents

List of Figures

List of Tables

Glossary

ATL Atlas Transformation Language. iii, 130, 133, 134

BPMN Business Process Management Notation. iii, 3, 130

CMOF Complete MOF. iii, 75, 130

DE Domain Expert. iii, 4–6

DFD Data Flow Diagrams. iii, 3

DSL Domain-Specific Language. iii, ix, 25, 31, 32, 40, 46, 52, 64, 66, 69, 71–73, 78, 85, 105–116

DSML Domain-Specific Modelling Language. i, iii, 3–7, 10, 12, 14–18, 20, 24–26, 28, 30, 39, 72, 78, 97, 102, 139–143

DSVL Domain-Specific Visual (Modelling) Language. iii, 12, 16, 52, 53, 97, 135

EMF Eclipse Modeling Framework. iii, 4, 14, 21, 23, 26, 27, 32, 40, 71, 75, 78, 85, 93, 97, 111, 119, 121, 122, 130, 133, 135

EMOF Essential MOF. iii, 124, 130

GPML General-Purpose Modelling Languages. iii, 12

HUTN Human Usable Textual Notation. iii, 38

MDE Model-Driven Engineering. iii, 3–5, 10, 11, 15–17, 24–26, 30, 93, 111, 130, 140

ME Modelling Expert. iii, 4–6

MOF Meta-Object Facility. iii, 22, 40, 129

OCL Object Constraint Language. iii, 14, 21, 22, 44, 46, 71, 79, 87, 119, 122–126, 129, 135, 136

OMG Object Management Group. iii, 123, 125, 130, 133, 134

TDD Test-Driven Development. iii, 15

UML Unified Modeling Language. iii, 22, 75, 124, 129, 130

XMI XML Metadata Interchange. iii, 119–122

XML Extensible Markup Language. iii, 119

XSD XML Schema Definition. iii, 96

1

Introduction

The aim of this section is introducing the motivation for conducting this work, its conceptual and technical contributions, and the core outcome resulting from it. Additionally, it includes a short description of each of the next chapters and the list of published works during the realization of the PhD.

1.1 Motivation

Model-Driven Engineering (MDE) relies on the use of models to raise the level of abstraction and automation in the development of software. One of the ways to achieve this goal is by the use of Domain-Specific Modelling Languages (DSMLs) [60].

DSMLs are languages tailored to a specific task or domain, capturing its main primitives and abstractions [60, 100]. Examples of DSMLs include dedicated languages for web engineering, requirements specification, business modelling, or data querying. BPMN [6] or Data Flow Diagrams (DFD) [96] would be concrete examples in these domains. DSMLs are useful in many diverse areas and disciplines, such as biology, physics, management or education, where the domain experts are not necessarily computer scientists, or have knowledge of MDE platforms and tools.

A DSML is defined by its abstract syntax, concrete syntax and semantics. The abstract syntax describes the concepts of the domain, their features and relations. In MDE, the abstract syntax is built through a meta-model, normally a class diagram increased with further constraints. The concrete syntax describes how models are represented, either graphically (e.g., an electrical circuit), textually (e.g., an SQL query), or a combination of both. The semantics describes the meaning of models, by providing e.g. a description of its execution, or a mapping into a semantic domain [20, 37].

There are many workbenches for constructing DSML editors [28, 31, 41, 46, 60, 99, 100], all of them requiring at least a Domain Expert (abbreviated DE; the person knowledgeable about the problem to solve) and a Modelling Expert (abbreviated ME; the one experienced in the development of DSMLs); however, all these frameworks suffer from several drawbacks, being the first of them that they are directed to computer scientists with a background in MDE. This is so as the usual process of DSML construction is *top-down*, that is, it requires building (a part of) the meta-model upfront, and only then it can be used for building instance models. In this way, even though MEs are used to this process, it may be counter-intuitive, difficult and most often too demanding for DEs, who most likely will not be familiar with such abstractions, and may prefer drafting example models first, share and discuss them with their peers, and then abstracting those into classes and relations in a meta-model.

Moreover, most approaches do not foster the active participation of the DEs in the DSML design process. Their role is limited to providing background knowledge of the domain, and evaluating the DSML proposals created by the MDE experts. This often leads to misunderstandings of the domain concepts, omissions or non-optimal solutions.

Finally, the available tools for building environments for DSMLs are technical, complex and time-consuming. Usually, they require manual programming [46] or building models for describing different aspects of the expected editor. These "auxiliary" models can become very detailed, with different syntax, large and hard to build and maintain for non-experts, and frequently they must be constructed using unhandy tree-based editors [41, 99].

On another note, there are scarce methods and tools to Validate and Verify (V&V) the quality, platform-specific rules, and accuracy with respect to the domain specifications, of the constructed DSMLs. In fact, most efforts on DSML research focus on the implementation aspect, rather than on DSML validation [66].

The lack of systematic means for DSML 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 artifacts developed for them, like modelling editors, model transformations and code generators. Moreover, meta-models are normally defined using an object-oriented approach and are implemented in specific platforms like the Eclipse Modeling Framework (EMF) [95]. Therefore, they should adhere to accepted object-oriented quality criteria and style guidelines in object-oriented conceptual schemes [3, 5], as well as to framework-specific rules and conventions.

In addition, the gap in information exchange between DEs and MEs, demands for ways

to validate meta-models with respect to specifications of the domain actively involving the DE, who could provide meaningful examples of correct and incorrect uses of the DSML.

In summary, the traditional approach for meta-model and DSML development, and the available environments that support it, present a series of limitations, which can be summarized as:

- Excessive focus on MDE practitioners.
- Centralization of the development process over the meta-model, disregarding the role of examples [105].
- Lack of interaction promotion between DEs and MEs.
- Technology for building graphical DSMLs has a steep learning curve.
- Lack of V&V mechanisms.

The DSML development process introduced in this work aims at solving the aforementioned drawbacks by proposing a *bottom-up* methodology, in opposition to the traditional *top-down* strategy, integrated with three V&V languages.

1.2 Contribution

This dissertation presents an iterative process for the development of DSMLs in which, instead of building a meta-model first and describing its concrete syntax at the meta-model level, model fragments are given either sketched by DEs using drawing tools, or using a compact textual notation suitable for engineers (not necessarily meta-modelling experts). The framework processes the provided examples to induce a meta-model, and is also capable of extracting a description of the graphical concrete syntax for the later generation of a modelling environment for the DSML, in the event that it is visual. The mechanism aims at the generation of a DSML editor mimicking the graphical syntax used in the examples, but in addition, it enforces the well-formedness rules of the DSML and enables the creation of models (in contrast to drawings) that can be manipulated using MDE technology (e.g., transformations and code generators). In this way, both experts (the ME and the DE) can work using their own syntax, and the communication gap is greatly decreased. The framework supporting the completion of all these tasks is given the name of **metaBup**.

The main benefits this new approach presents are:

- The workload is more balanced, as it does not delegate most of the effort to the ME.
- Using examples empowers the role of models in the development process, decentralizing the focus on the meta-model, which is especially good for DEs, as it is closer to their syntax than meta-models.
- Active collaboration between DEs and MEs is fostered.
- DEs do not have to learn complex technologies, as they are allowed to draw their examples with common tools in a WYDIWYG (*'What You Draw Is What You Get'*) style.

Within the scope of *metaBup*, a complementary approach for the Validation and Verification of meta-models, named **metaBest**, is proposed, supported by three different languages:

1. **mmUnit**. It enables writing conforming and non-conforming model fragments to check whether a meta-model accepts the former and rejects the latter. For non-conforming tests, it is possible to declare assertions that state the expected disconformities and reflect the intention of the test, using a *xUnit* style [15].
2. **mmSpec**. This language allows expressing and checking expected meta-model properties that may arise from the domain, from the implementation platform or from quality criteria.
3. **mmXtens**. This DSML provides a mechanism for automatically generating model examples from a meta-model, satisfying a set of properties. These are created using the same concrete syntax used to provide the examples during the meta-model induction process.

These two main conceptual contributions give shape to the design of a new *bottom-up* process for the development of modelling languages and their editors, aiming for the mitigation of the drawbacks implied in the traditional *top-down* approach.

1.2.1 Technical Contribution

This dissertation provides the following technical contributions:

1. Design and implementation of *metaBup*, which features:
   - A mechanism for bottom-up meta-model development.
   - Automatic generation of visual DSML editors.
2. Design and implementation of the three *metaBest* Validation & Verification languages:
   - *mmUnit*, for meta-model unit testing based on model examples.
   - *mmSpec*, for meta-model property check.
   - *mmXtens*, for automatic example model generation.
3. Experimental evaluation of the aforementioned contributions, including:
   - User validation of the meta-model development process and visual DSML editor generation facility.
   - Validation of the usefulness of the three V&V languages.
   - Evaluation of the conciseness, performance and expressiveness of *mmSpec*.

1.2.2 Publications

This presented dissertation leads to the following publications:

Journals (3):

3. *Example-driven meta-model development*. Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara. Software and Systems Modeling (SoSyM). Volume 14 Issue 4, October 2015. Pages 1323-1347. DOI: 10.1007/s10270-013-0392-y. Springer Berlin Heidelberg. JCR 2015: 0.99 (Q3).

2. *Combining unit and specification-based testing for meta-model validation and verification*. Jesús J. López-Fernández, Esther Guerra, Juan de Lara. Information Systems. Volume 62, December 2016. Pages 104-135. Elsevier Science Ltd. DOI: 10.1016/j.is.2016.06.008. JCR 2015: 1.832 (Q1).

1. *An example is worth a thousand words: creating graphical modelling environments by example*. Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra, Juan de Lara. Software and Systems Modeling (SoSyM), Springer. Submitted for second round of revision (major changes). November 2016 as of invitation to participate in special issue with best papers of ECMFA'16. JCR 2015: 0.99 (Q3).

International conferences and workshops (5):

5. *Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages*. Javier Luis Cánovas Izquierdo, Jordi Cabot, Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara. 10th International Conference on Cooperative Design, Visualization and Engineering, CDVE 2013, Alcudia, Mallorca, Spain, September 22-25, 2013. Proceedings. pp 101-110. 2013. DOI: 10.1007/978-3-642-40840-3_16. Lecture Notes in Computer Science, Volume: 8091. Springer Berlin Heidelberg. Core 2013: C.

4. *Meta-Model validation and verification with MetaBest*. Jesús J. López-Fernández, Esther Guerra, Juan de Lara. ASE 2014 Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. Vasteras, Sweden — September 15 - 19, 2014. DOI: 10.1145/2642937.2648617. Core 2014: A.

3. *Assessing the Quality of Meta-models*. Jesús J. López-Fernández, Esther Guerra, Juan de Lara. 11th Workshop on Model Driven Engineering, Verification and Validation, MoDeVVa 2014. Proceedings published as Vol. 1235 of CEUR. pp 3-12.

2. *Example-based validation of domain-specific visual languages*. Jesús J. López-Fernández, Esther Guerra, Juan de Lara. SLE 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering. Pittsburgh, PA, USA — October 26 - 27, 2015. Pages 101-112. DOI: 10.1145/2814251.2814256. Core 2015: B.

1. *Example-Based Generation of Graphical Modelling Environments*. Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra, Juan de Lara. Modelling Foundations and Applications. 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings. pp 101-117. 2016. DOI: 10.1007/978-3-319-42061-5_7. Lecture Notes in Computer Science, Volume: 9764. Springer International Publishing.

During the PhD period, the following publications were also contributed:

Journals (1):

- *PTL: A model transformation language based on logic programming*. Jesús M. Almendros-Jimenez, Luis Iribarne, Jesús J. López-Fernández, Ángel Mora-Segura. Journal of Logic Programming, Volume 85(2): 332-366 (2016). Elsevier. DOI: http://dx.doi.org/10.1016/j.jlamp.2015.06.006. JCR: 0.636 (Q1).

International conferences and workshops (1):

- *An XQuery-Based Model Transformation Language*. Jesús M. Almendros-Jimenez, Luis Iribarne, Jesús J. López-Fernández, Ángel Mora-Segura. Model and Data Engineering. 4th International Conference, MEDI 2014, Larnaca, Cyprus, September 24-26, 2014. Proceedings. pp 330-338. 2014. DOI: 10.1007/978-3-319-11587-0_31. Lecture Notes in Computer Science, Volume: 8748. Springer International Publishing.

1.2.3 Tool Support

The process introduced in this dissertation is supported by a prototype application that can be downloaded at: www.jesusjlopezf.com/metabup. The website includes a gallery of examples, demonstration videos, tutorials and the installable version of *metaBup*, in the form of an Eclipse plug-in.

1.2.4 Research Visits

The realization of this dissertation included a stay at the Business Informatics group of the Technische Universität Wien (Austria), supervised by Dr. Manuel Wimmer. During this visit, the batch example generation feature presented in Section 5.2.3 was designed and implemented.

1.3 Support

This dissertation has been funded by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) through the grant number BES-2012-060153.

1.4 Organization

The rest of this document is organized as follows:

Chapter 2 introduces the context and related work on the field of MDE, example-driven meta-modelling and testing.

Chapter 3 details the proposed process for bottom-up meta-modelling (*as supported by metaBup*), including the automatic transformation of drawn sketches into fragments, how the engine uses those for creating and evolving an abstract syntax (meta-model), and the mechanisms involved in the generation of a fully operational DSML editor for the domain.

Chapter 4 tackles the design of the main components of *metaBest*, the test suite complementing *metaBup*. The usage of the three proposed languages (*mmUnit*, *mmSpec* and *mmXtens*) is given a place in the central process.

Chapter 5 is a description of the technical contributions achieved in this work including an overall description of the architecture of both *metaBup* and *metaBest*.

Chapter 6 features a series of experimental evaluations on the proposed framework. It is divided in two sections, the first one being a user study of the core process, and the second one going through 6 different experiments on several aspects of the testing suite, like its conciseness, its expressiveness or its performance.

Chapter 7 finalizes this document providing some conclusions on the accomplished research. It also provides a set of tasks that could be performed in future works.

2

Background and Related Work

This chapter tackles the basics of the field in which this work is encompassed, MDE. The aim is guiding the reader in the comprehension of the PhD.

The text is divided into two separate sections: **background**, in which the essential concepts and tools for MDE are introduced, and **related work**, in which the state of the art is discussed, going through interesting publications on the matter, identifying gaps, and discussing their relationship with this work.

2.1 Background

2.1.1 Model-Driven Engineering

MDE [20] is a Software Engineering paradigm that promotes an active use of models and transformations throughout all phases of software development. In MDE, 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 [101].

The main concepts involved in MDE processes are shown in Figure 2.1. **Models** represent an abstraction of real-world elements [18]. These are expressed using a **modelling language**. A modelling language is defined by means of a meta-model, which represents its abstract syntax and is likewise constructed using a **meta-modelling language**. **Transformations** are used to modify models, or to generate new models from existing ones. A particular case of these are **code generators**. More particularly, we talk about *Model-to-Model* (*M2M*) transformations when the output is a model, and *Model-to-Text* (*M2T*) transformations when

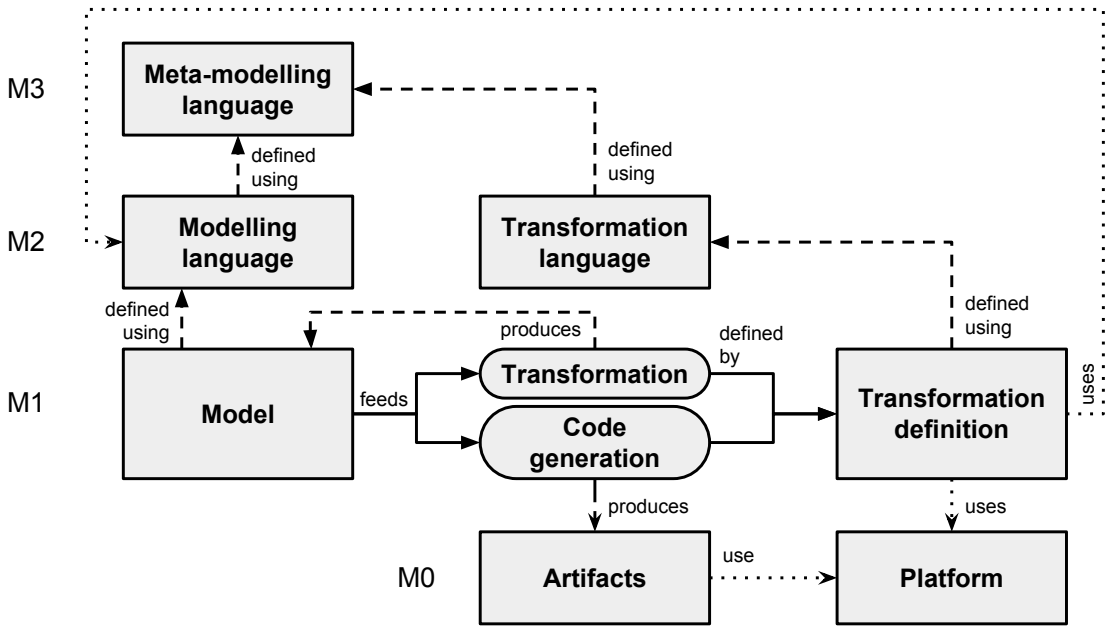


Figure 2.1: Model-Driven Engineering architecture, inspired from [20].

the output comes in the form of text, and many times, source code. Transformations are defined by rules expressed in a **transformation language**. Code generators can produce computable **artifacts** from models, able to be processed by an execution **platform**.

Modelling languages can be classified by the degree of coverage of the domains they represent, into two different types: General-Purpose Modelling Languages (GPMLs) and Domain-Specific Modelling Languages (DSMLs). The former are designed to be employed in arbitrary domains, whereas the latter are designed to resolve problems in a specific field.

In addition, modelling languages can be classified by the format employed in their concrete syntax: whereas Textual Domain-Specific Modelling Languages aim at creating instances of a meta-model using text, Domain-Specific Visual Languages (DSVLs) enable modelling with a graphic syntax.

Figure 2.2 shows in the upper part an example of a meta-model for representing house blueprints (a). To the middle (b) left, we find an instance model of the proposed meta-model, and to the middle right, its representation in a graphical modelling language.

Being known the structure of the meta-model, we could design transformations and apply them to the models produced. For instance, we might have rules to refactor or optimize blueprints. This way, we might be interested in simplifying blueprints to obtain versions of

them limited to a single bathroom. The figure shows a transformation definition (a *'only one bathroom'*, labelled *c*) that aims at obtaining such a house configuration from any input blueprint model. The result is shown to the bottom of the figure (d).

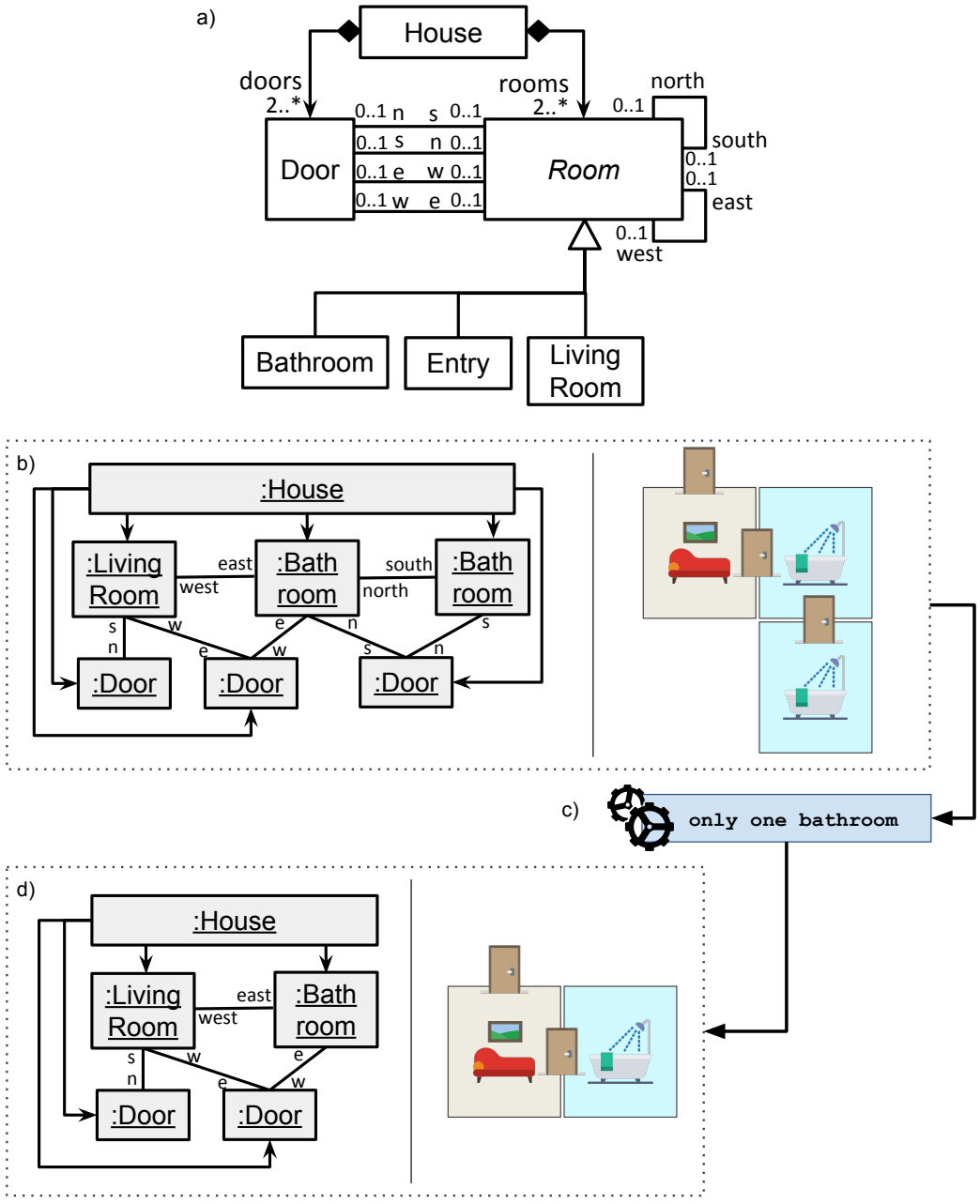


Figure 2.2: Example of a meta-model, model, graphical concrete syntax and model transformation

The aforementioned elements in this software development paradigm are currently supported by different technologies. The most widespread environment for MDE is the dedicated

distribution of Eclipse, called EMF [95]. Because of its extensible architecture, open-source code, and extended use, the majority of industry-ready solutions for MDE are released as complementary tools for EMF.

Xtext [31] and GMF [41] are the main frameworks for developing textual and visual DSMLs on EMF, respectively. In recent years, Graphiti [46], EuGENia [63] and Sirius [99] exploited the features of the latter for providing more advanced tools in the creation of visual editors of DSMLs. CEViNEdit [45] enables generating GMF cognitively efficient editors.

Regarding model transformation languages, the most common tools are ATL [53], QVT [86] or ETL [64]. Object Constraint Language (OCL) [81] is another relevant language in the EMF framework, as it serves for describing constraint rules for models and meta-models.

Outside the Eclipse environment, we find tools like MetaDepth [27], which introduces the concept of *deep* meta-modelling, or JetBrains MPS [52], a framework for developing advanced text and visual DSML editors.

2.1.2 Approaches for creating DSMLs

When it comes to the methodology involved in the development of DSMLs, the variety of available processes are grouped into two main categories: *top-down* and *bottom-up* approaches.

Top-down DSML development methods promote creating modelling languages following an approach in which the meta-model is built first, and then it is used to create instance models. This strategy carries with it the realization of a meta-model in a syntax which shall normally be alien for a Domain Expert, and hence the biggest part of the work relies in the Modelling Expert. This approach is used by the vast majority of the state-of-the-art technology.

On the other hand, a *bottom-up* meta-modelling development process starts with the provision of models, i.e., examples, from which a meta-model is induced, and then fine-tuned by the Modelling Expert. This turnaround poses a big advantage with respect to *top-down* development, as it lets the Domain Expert provide the information in a format that is known to him, and hence the gap between the concrete and the abstract syntax is reduced.

Note that *bottom-up* development is much closer to other software development approaches, like interactive development [84], which promotes rapid feedback from the programming en-

vironment to the developer. Typically, a programming language provides a *shell* to write pieces of code, and the running system is updated accordingly. This permits observing the effects of the code as it is developed, and exploring different design options easily. This approach has also been regarded as a way to allow non-experts to perform simple programming tasks or to be introduced to programming, since a program is created by defining and testing small pieces of functionality that will be composed bottom-up instead of devising a complete design from the beginning.

In a similar vein, example centric programming [29] promotes examples as first-class citizens in the programming process, as programs (abstractions) are iteratively and interactively developed from *concrete* examples. Comparably, Test-Driven Development (TDD) [16] fosters the inference of software from test cases, similar to the derivation of meta-models from models in a *bottom-up* approach. However, in contrast to the widespread status of *top-down* approaches in the MDE community, there are scarce tools and available literature on *bottom-up* strategies for building DSMLs.

2.1.3 Approaches to meta-model V&V

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 [17]. Two main questions arise when assessing the quality of a meta-model: validation (*"are we building the right meta-model?"*), and verification (*"are we building the meta-model right?"*).

The literature reports three main approaches to meta-model V&V: *unit testing*, *specification-based testing* and *reverse testing*.

Unit testing approaches support 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 [15].

Specification-based approaches allow expressing desired properties of a meta-model, and then checking the compliance of models with respect to these assertions.

Reverse testing approaches are based on the automatic generation of instance models from a meta-model, likely using constraint solving [21, 35, 38, 102]. A Domain Expert has to inspect the generated models to detect invalid ones, in which case the meta-model contains errors and needs to be corrected accordingly.

The use of any of these three approaches is not limited to *top-down* or *bottom-up* methodologies, but they can be accommodated into the workflow of any meta-model or DSML development process.

2.2 Related Work

Once we have introduced the basic concepts of MDE, in this section we will review related works divided in two blocks: those approaches aimed for DSVL development (Section 2.2.1), and those directed to DSML testing (Section 2.2.2).

2.2.1 DSVL development

Next, we revise related approaches to the definition of DSML requirements and the generation of modelling tools, and perform a feature-based comparison of the main tools for the flexible creation of graphical DSMLs.

2.2.1.1 DSML requirements

According to [66, 76], domain analysis techniques are still missing for DSMLs, as this phase is most frequently done in an informal way in the develpoment of both graphical and textual modelling languages. This work aims at proposing a concrete approach by introducing a format which is familiar to general users, i.e., drawings which are built with diagramming tools to represent DSML requirements. Other ways to represent requirements include feature diagrams [55] or notations inspired by mind-maps [85]. While these approaches focus on representing and classifying desired DSML features, the example-based approach here presented relies on concrete examples of language use, and promotes a more direct involvement of Domain Experts. In some sense, the approach presented in this PhD is conceptually similar to Test-Driven Development approaches [16], as these promote the creation of software programs using tests which are provided iteratively. This dynamic somehow resemble our process, in which a meta-model (from which the editor shall be generated) is built from *fragments* (which can be considered tests as they represent tentative portions of models).

In [7], domain analysis is application-driven and DSMLs are produced as output artefacts

when developing a software framework, while in [93], the Question-Options-Criteria is used to guide the design decisions involved when creating UML-based DSLs. However, none of these two works propose a concrete notation to represent DSML requirements.

2.2.1.2 Flexible modelling

While MDE is founded on the ability to process models with a precisely defined syntax, some authors have recognised the need for more flexible and informal ways of modelling. This is useful in the early phases of system design [75, 90, 104], or as a means to promote an active role of Domain Experts in DSL development [23, 43, 106], as we advocate in this work.

There are two orthogonal design choices enabling flexible modelling in DSML development: (i) the use of examples to drive the construction process, and (ii) the explicit generation of a meta-model and a modelling tool different from the drawing tool used to build the initial examples.

Regarding the first design choice, "by-demonstration" techniques have been applied to several MDE artefacts, like model transformations [10, 12, 56, 61, 97] or model refactorings [40]. However, their use is not so common to describe graphical modelling environments. The closest work to what this PhD proposes is the MLCBD framework [23], which describes a system atop Microsoft Visio to derive DSMLs by demonstration. Given a single example, the system derives the concrete syntax from the icons in the palette, and some abstract syntax constraints, e.g., concerning the connectivity of elements. This information is recorded and used within Microsoft Visio. Instead, the mechanism here proposed derives an explicit meta-model, infers spatial relationships like containment and overlapping, and generates a modelling tool. Moreover, the meta-model the proposed system is able to induce supports modelling concepts like abstract classes, inheritance, compositions and attributes, which are not found in [23].

The approach in [106], called Muddles, uses yED to draw examples of the DSML. Types are assigned to elements on the basis of labels, and some predefined functions check for shape overlapping, colour or proximity. All modelling is performed within yED, and no meta-model or dedicated modelling environment are generated. Instead, the authors use the Epsilon model management languages [83] to query and manipulate the yED models directly.

The Free Modeling Editor (FME) presented in [43] permits developing DSMLs starting either from example models or from meta-model concepts. The proposal is based on the

Openflexo tool, which supports the concurrent development of both models and meta-models, and has importers for models stored in Powerpoint and Word formats. The approach has been successfully applied to an industrial project [43].

Some tools for DSML development generate an external modelling tool different from the one used to define the DSML. For instance, EuGENia Live [87] is a web tool for designing graphical DSLs. It supports on-the-fly meta-model editing while the user is building a sample model and its concrete syntax. From this definition, the tool exports a meta-model in Ecore format, enriched with concrete syntax annotations for EuGENia, which can be used to generate an Eclipse GMF-based environment.

Some modelling tools promote flexibility in the early phases of system design by offering sketching capabilities similar to pen-and-paper drawing. For instance, SKETCH [90] provides an API to enable sketch-based editing on Eclipse; Calico [75] is a sketching tool for electronic whiteboards, where the sketched elements can be scrapped and reused in other parts of the diagrams; and FlexiSketch [104] derives simple meta-models from sketches, although the extracted meta-models do not support meta-modelling elements like class inheritance, abstract classes or different association types (e.g., compositions).

Finally, although not specific for DSML creation, there is a trend in recent modelling tools to promote flexibility by relaxing the conformance relationship in early phases of modelling, while enforcing strictness in later phases [48, 94]. These tools profit from the flexibility of JavaScript as the underlying implementation language.

2.2.1.3 Comparison of flexible tools to build graphical DSMLs

Table 2.1 compares *metaBup* with the most prominent tools for the flexible creation of graphical DSMLs, which are: EuGENia Live [87], FlexiSketch [104], FME [43], MLCBD [23] and Muddles [106]. This section goes through the differences between them.

Regarding the overall approach the tools implement, *metaBup* and EuGENia Live rely in informal drawing tools to specify examples, and then generate an external modelling tool that mimics the exemplified graphical syntax. In particular, *metaBup* generates a Sirius-based modelling editor, and EuGENia Live generates a GMF-based one. In contrast, MLCBD and Muddles also start from informal drawings, but then modelling is performed in the same drawing tools (i.e., no external modelling tool is generated).

	Approach	**Process**	**DSL examples**	**DSL meta-model**	**Modelling environment**	**Advanced recognition**
metaBup	informal drawings + editor generation	examples + meta-model induction	yED, DIA	automatic (complex)	generated (Eclipse)	spatial relations
EuGENia Live [87]	informal drawings + editor generation	examples + meta-model	inside tool	manual	generated (Eclipse)	-
FlexiSketch [104]	flexible tool (sketches+models)	examples + meta-model induction	inside tool	automatic (simple)	inside tool	sketching
FME [43]	flexible tool (examples+models)	examples + meta-model	inside tool, Office	manual	inside tool	-
MLCBD [23]	informal drawings + informal editor	examples	Ms Visio	-	Ms Visio	-
Muddles [106]	informal drawings + informal editor	examples	yED	-	yED	spatial relations

Table 2.1: Flexible approaches for DSML creation.

The remaining cases (FlexiSketch and FME) do not rely on external drawing tools or graphical modelling frameworks, but they are themselves flexible self-contained modelling environments. FME, in addition, can also import drawings from some external tools, although it does not induce or extract a conceptual meta-model from them.

Concerning the process followed to define the DSML, *metaBup* implements a bottom-up approach, where the provided examples are used to automatically induce a meta-model making explicit the syntactic rules of the DSML. FlexiSketch also induces a meta-model. Instead, MLCBD and Muddles do not infer or need an explicit meta-model, while EuGENia Live and FME require creating a meta-model manually.

Table 2.1 also compares how the examples are created in the different tools. Some of them use existing popular drawing tools like yED, Dia or the Microsoft Office suite. The advantage is that Domain Experts might find some of them familiar. Instead, EuGENia Live and FlexiSketch require using the modelling tool itself to draft the examples.

As abovementioned, only *metaBup* and FlexiSketch induce meta-models from the examples. Nonetheless, the meta-models generated by the second tool are simpler, lacking regular conceptual modelling concepts like inheritance, abstract classes, compositions or constraints. Moreover, *metaBup* incorporates a catalogue of refactorings that can be used to improve the meta-model quality.

Regarding the modelling environment obtained from the DSML definition, *metaBup* and EuGENia Live generate external modelling environments for Eclipse, MLCBD and Muddles enable the tools they use to draw examples as modelling environments, and FlexiSketch and

FME support modelling inside themselves.

Creating a meta-model and a modelling environment on top of a meta-modelling framework has several benefits: first, it guides the user in filling slot values and provides type checking for them, which otherwise should be done via tags in a diagramming tool like Visio; second, the created models can be manipulated using standard model management languages for model transformation or code generation.

Related to this dimension, the degree in which complex graphical aspects are extracted from the examples is likewise relevant. Among the approaches that generate a dedicated environment, *metaBup* is the only one able to identify spatial relationships between elements, and enforce them in the generated modelling environments. Muddles identifies spatial relationships as well, though they are not enforced when modelling. Finally, FlexiSketch is the only analysed tool that supports sketching, but it does not provide recognition of spatial relationships.

2.2.1.4 Summary

Altogether, the approach presented in this work is novel as it enables the creation of graphical DSML editors based on drawings produced by domain experts, generating a meta-model and a dedicated modelling environment. This approach helps in transitioning from informal modelling in a diagrammatic tool, to formal modelling in a modelling tool, where models are amenable to automated manipulation.

2.2.2 DSML testing

Most efforts towards the V&V of MDE artifacts are directed to test model management operations [36], like model transformations [1], but few works target meta-model V&V. Looking at the most remarkable publications and tools on each of the approaches for meta-model testing, the next sections analyze the main related works.

2.2.2.1 Unit testing

Unit testing is a widely used approach in programming, and some approaches have taken inspiration from it, and adapted it to test meta-models.

In [89], test models describe instances that the meta-model should accept or reject. In a different style, [24] 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 [82] 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 [58, 103] expand general-purpose testing tools (e.g., JUnit) to enable the testing of DSML programs, not necessarily defined by meta-models. In [98], 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 [95] also provides some support for testing. Given a meta-model, 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 [9] as a way to test meta-models. However, it does not permit stating higher-level assertions to express common failures in the modelling domain (like the lack of a container for an object); instead, it only allows 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 [88], 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.

2.2.2.2 Specification-based testing

In specification-based testing, the meta-model is tested to comply with some properties, given in a specification.

An approach for checking meta-model integration is presented in [92]. It relies on specifying meta-model properties in EVL [65] (a variant of OCL), but as the authors recognize, 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.

In [51], the authors define a formal meta-modelling framework based on algebraic data types and constraint logic programming, so that proofs and test-case generation can be encoded as open world query operations and automatically solved. Domain constraints must be encoded as CPL rules, which may be difficult to express by domain experts or engineers without a strong mathematical background.

Other works define catalogs of quality criteria for meta-models [17] or conceptual schemas. In [30], 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 [3], quality properties of conceptual schemas are formalized in terms of quality issues, which are conditions that should not happen in schemas. The authors describe such conditions using OCL. In [5], the same authors propose a set of guidelines for naming UML schemas, which can be validated using an Eclipse plugin [2]. 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.

2.2.2.3 Reverse testing

In reverse testing, the system generates some artefact (e.g., a model), so that the designer can check whether it can be considered correct or not.

This approach is followed by [44], where the generated snapshots are targeted to test cardinality boundaries, and works like [42, 91]. To guide the model generation process, [42] provides a programming language to define object snapshots, while [91] allows defining constraints captured by query patterns or OCL.

In [68], the authors transform meta-models into OWL2 and use reasoners to validate their consistency; however, their approach only reports the unsatisfiable concepts with no further explanations. In [8], questionnaires with true/false questions are generated from the meta-model, and the Domain Expert performs the meta-model validation by answering the questionnaires.

This study concludes that there is a lack of high level mechanisms to specify interesting models to be generated, combined with a graphical concrete syntax, so that the Domain Experts can easily evaluate their correctness.

2.2.2.4 Summary

The main gap in the state of the art is the absence of an integral approach for meta-model V&V in all its three approaches, but it is also important to mention that the scarce available solutions tackling testing approaches individually present some kind of drawback.

First, the few existing *specification-based testing* approaches rely on OCL or QVT, which are not optimal to express properties at the meta-model level and do not provide effective support for error visualization and reporting. Second, *unit testing* approaches sometimes lack means to construct faulty models, as frameworks like EMF hardly accept incorrect models and require 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.

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).

This work proposes a novel integrated framework (detailed in Section 4) 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.

This section has presented the main concepts involved in the MDE methodology, particularly focusing on DSMLs, the key role that meta-models play in their construction, and the different kinds of approaches followed for building them. In this sense, this work proposes a *bottom-up* methodology, an alternative to the most widespread solutions in the market and in the available literature, which heavily rely in *top-down* approaches.

On the other hand, the three main styles of meta-model and DSML V&V have been introduced, going through the most salient contributions on each one of them, and identifying challenges and drawbacks in these works. This work contributes three V&V languages that overcome the identified gaps.

3

Example-driven Meta-model Development

In this chapter, the process for DSML development by example is presented. For that, traditional strategies for building these languages are discussed, identifying the most significant drawbacks to them and how *bottom-up* methodologies help overcoming them. Then, the concrete *bottom-up* process introduced in this work is explained in detail, presenting all its involved roles (a Domain and a Modelling Expert) and artifacts (sketches, fragments, annotations and meta-model), and how the latter are created and transformed to produce a resulting DSML visual editor. Descriptions are illustrated with a running example.

Development strategies for Domain-Specific Modelling Languages – DSMLs, or simply DSLs - typically involve two roles: the Domain Expert (DE) and the Modelling Expert (ME). While Domain Experts have the knowledge about the concepts of the domain, they usually lack the skills to build meta-models, which normally need to be tailored according to their future usage and specific implementation platform. This kind of knowledge is available only to engineers with great expertise in specific MDE platforms, who would play the ME role. In this way, engineers are forced to achieve a nearly complete understanding of the domain, which may be the cause of misinterpretations which hamper the development process and negatively affect the quality of the implemented solution. Typically, it is this gap between the domain and the target platform that leaves the Domain Expert in a highly passive role for language development.

In order to alleviate this situation, this section introduces an interactive, iterative approach to meta-model construction enabling the specification of example model fragments by domain experts, with the possibility of using informal drawing tools. These fragments are presented to the ME in a text representation for a review and refactoring phase, and from them, a meta-model which is automatically induced, can be refactored interactively,

and eventually compiled into an *implementation* meta-model from which different platform-specific implementations can be derived.

This chapter is partially based on [69] and [70], although certain details have been extended or modified for reflecting later additions and improvements performed over the originally published work.

3.1 Motivation

Before going into detail with the specific solution, it seems convenient to provide a general explanation of what an *example-based* meta-modelling methodology is, how the meta-model development process can benefit from it, and the challenges it currently presents.

MDE makes heavy use of models during the software development process. Models are usually built using DSMLs which are themselves specified through a meta-model. A DSML should contain useful, appropriate primitives and abstractions for a particular application domain. Hence, the input from DEs and their active involvement in the meta-model development process are essential to obtain effective, useful DSMLs [49, 57, 59, 76, 100].

The usual process of meta-model construction requires first building (a part of) the meta-model which only then can be used to build instance models. Development strategies approaching the development in this way are known as *top-down* methodologies. However, even though software engineers are used to following this kind of process, it may be counter-intuitive and difficult for non-meta-modelling experts, who may prefer drafting example models first, and then abstract those into classes and relations in a meta-model. This second approach to meta-modelling is called *bottom-up* [69].

A change in the development process like this is in many ways positive, if we consider that DEs 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, and consequently asking them to build a meta-model *before* drafting example models is often too demanding if they are not MDE experts.

Nevertheless, a *bottom-up* strategy raises two main challenges. First of all, while MDE experts are used to work with specialized meta-modelling tools – like those provided by Eclipse EMF [95] – this is seldom the case for DEs. DEs may find difficult the use of modelling

tools which force them to build models in a very constrained and systematic way, while on the contrary, they will be familiar and expecting to work with more intuitive and flexible sketching and drawing tools (in the style of *PowerPoint* or *Visio*) to build models. Moreover, once an initial version of a meta-model is built, it needs to be validated in collaboration with the DEs, and this will certainly require the adaptation of modelling tools to a more intuitive format.

Basically, while MDE experts are used to inspect meta-models, for DEs a validation based on examples (again, built using drawing tools) would be more adequate, as DEs may lack the required expertise in conceptual modelling to fully understand a meta-model.

Another issue that makes meta-model construction cumbersome is the fact that meta-models frequently need to be fine-tuned depending on their intended use: designing a textual modelling language (e.g., with Xtext [31]), a graphical language (e.g., with GMF [47], Eugenia [63] or Sirius [99]), or the source or target of a transformation.

As illustrated in Figure 3.1, the particular meta-model usage may impact on its design, for instance to decide whether a connection should be implemented as a reference (e.g., for simple graphical visualization), as an intermediate class (e.g., for a more complex visualization, or to enable iterating on all connection instances), as a bidirectional association (e.g., to allow back navigation if it is used in a transformation), or as an intermediate class with composition (e.g., to enable scoping). The use of a specific technological platform, like EMF [95], has also an impact on how meta-models are actually implemented, e.g., regarding the use of composition, the need to have available a root class, and the use of references. As a consequence, the *implementation* meta-model for a particular platform may differ from the *conceptual* one as elicited by DEs. Specialized technical knowledge is required for this implementation task, hardly ever found in DEs, which additionally has a steep learning curve. Therefore, we may conclude that there is a need to align the capabilities of the DE when expressing meta-model requirements with the actual possibilities of the available meta-modelling tools.

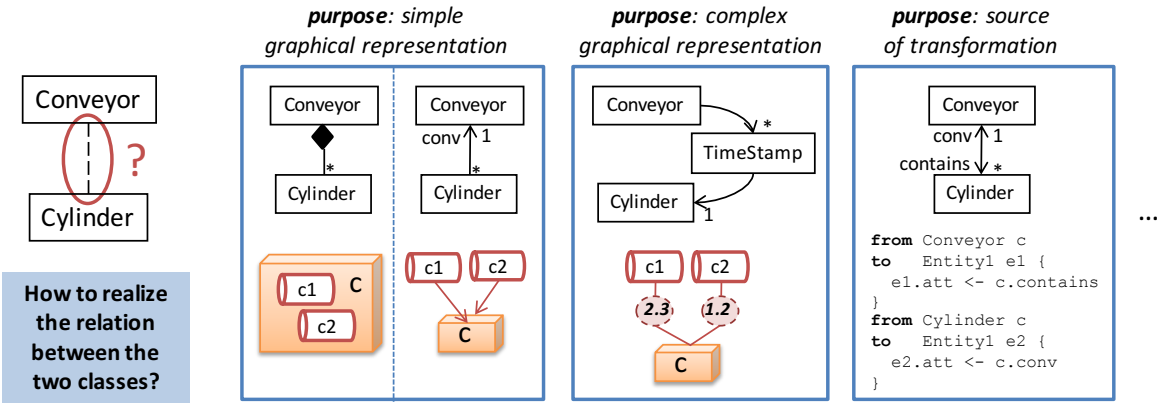


Figure 3.1: Different meta-model realizations depending on its future usage.

3.2 Overview and Running Example

3.2.1 Overview

Once we have explained the benefits that a *bottom-up* strategy can bring to a meta-model development process, we present our own solution for it, a novel way to define meta-models and modelling environments. It is particularly important mentioning that this framework was designed with the observance of the following requirements:

- *Bottom-up*. Whereas meta-modelling requires abstraction capabilities, the design of DSMLs demands, in addition, expert knowledge about the domain in two dimensions: horizontal and vertical [11]. The former refers to technical knowledge applicable to a range of applications (e.g., the domain of Android mobile development) and experts are developers proficient in specific implementation technologies. The vertical dimension corresponds to a particular application domain or industry (e.g., insurances) where experts are usually non-technical people. Our proposal is to let these two kinds of experts build the meta-models of DSMLs incrementally and automatically starting from example models.

  Using example models is appropriate in this context, as these two kinds of users may not be meta-modelling experts. Example models document *requirements* of the DSML to be built, provide concrete evidence on the specific use of the primitives to be supported by the DSML, and can be used for the automated derivation of its meta-model.

Afterwards, the induced meta-model can be reviewed by a meta-modelling expert who can refactor some parts if needed.

Finally, DEs also play a crucial role in meta-model validation. Thus, we encourage their collaboration in this task by proposing an example-based validation process where end-users can feed the system with concrete examples models, and the system reports whether they are correct according to the current version of the meta-model, and the reason why they are not. Moreover, we also enable the automatic generation of example models fulfilling certain criteria, which can be populated either from a seed model fragment or from scratch. In this way, DEs can easily perform validation by just inspecting the generated models.

- *Interactive*. A meta-model can become large, and it may address different separate concerns. In practice, its construction is an iterative process in which an initial meta-model is created, then it is tested by trying to instantiate it to create some models of interest, and whenever this is not possible, the meta-model is changed to accommodate these models [57]. The performed changes may require the detection of broken models and their manual update.

  Our proposal aims at supporting this interactive meta-model construction process. Hence, we do not advocate building a complete meta-model in one step, but the meta-model is "grown" (using the terminology of Test-Driven Development [34]) as new fragments gathering more requirements are inserted. If a new version of the meta-model breaks the conformance with existing models, the problem is reported together with possible fixes.

- *Exploratory*. The design of a meta-model is refined during its construction, and several choices are typically available for each refinement. To support the exploration of design options, we should let the developer annotate the example models with hints about the intention of the different model elements, which are then translated into some meta-model structural design decision or into additional integrity constraints.

- *Guided by best-practices*. Since the users of this approach may not be meta-modelling experts, we provide a virtual assistant which suggests the application of meta-modelling design guidelines, best practices and refactorings that help to improve the quality of the current version of the meta-model.

- *Implementation-agnostic*. The platform used to implement a meta-model may enforce certain meta-modelling decisions (e.g., the use of compositions vs. references, or the

inclusion of a root node). This knowledge is sometimes not even available to meta-modelling experts, but only to experts of the particular platform. For this reason, we postpone any decision about the target platform to a last stage. The meta-models built interactively are neutral or implementation-agnostic, and only when the meta-model design is complete, it is compiled for a specific platform.

In this context, this work aims giving support to four main functionalities in the meta-model development process:

1. Provide a way for DEs to be able to sketch their own model fragments in a free, visual format. This platform needs to be extensible, that is, permitting new formats to be easily added, as this will make the meta-modelling tool independent from a specific drawing tool.
2. Automate a process in which input fragments are used to grow a meta-model which can also be changed by the ME.
3. Facilitate validation and verification activities over the fragments and the meta-model under development, including an automatic instance generation facility.
4. Enable a range of generators that provide the abstract syntax meta-model with a concrete syntax as similar as possible in its format to the input fragments provided by the DE.

The ultimate goal of this work is to facilitate the creation of DSMLs by DEs without proficiency in meta-modelling and MDE platforms and technologies, and to aid the ME in the process. For that, the *bottom-up* process shown in Figure 3.2 has been designed. This chapter tackles the core part of that process, gray-shaded in the Figure. If we take a look at it, we can see that it sketches an iterative cycle, started by the DE providing input examples in the form of drawings, that is, portraying how models should look like (label 1). Then, the examples are automatically parsed into models, more amenable to manipulation (label 2). The parsed models are represented textually, making explicit the existing objects, attributes and relations in the examples. The ME can edit this textual representation (label 3) to set more appropriate names to the derived relations, or to trigger refactorings in the meta-model induction process which takes place next (label 4). Thus, an iteration step finishes when the meta-model under construction is evolved to accept the revised fragment.

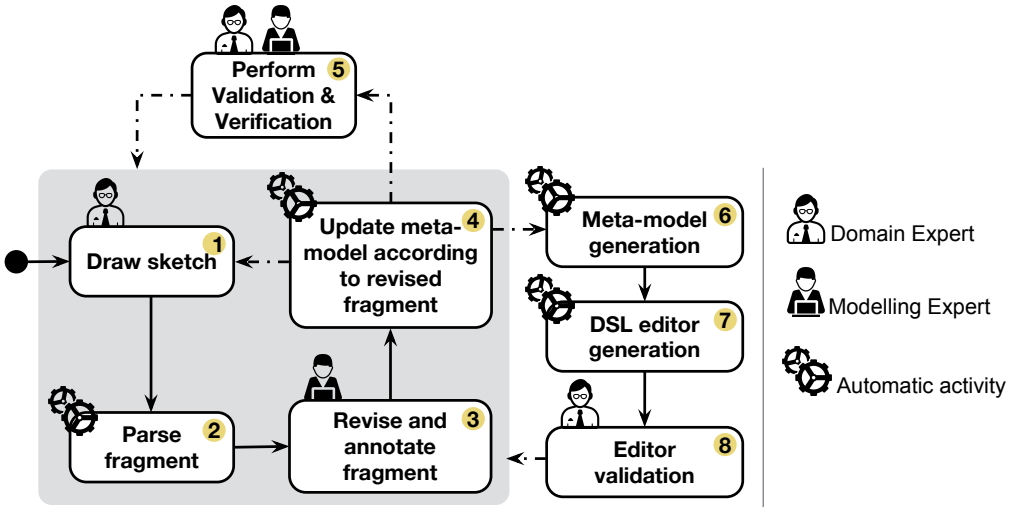


Figure 3.2: Scheme of the overall process.

Furthermore, our framework enables test case definition and property check over the ever growing meta-model (label 5), as well as the automatic generation of examples for their validation. This task, explained in detail in Chapter 4, normally demands the collaboration of both modelling and DEs, and, as it encompasses a broad variety of tools and languages, it is presented as an independent test framework itself.

After processing all provided examples, the ME can export the induced meta-model to a suitable format (label 6), and invoke our DSL editor generator to obtain a fully operating editor mimicking the concrete syntax of the examples (label 7). The DE can validate the editor (label 8), and if necessary, he can refine the DSL by providing further examples and re-generating the editor. We fully detail the DSL editor generation process in Chapter 3.5.

3.2.2 Running example

As a running example, we will develop a DSL in the home networking domain. The DSL is inspired by one of the case studies in the Sirius gallery[1]. In this DSL, we would like to represent the ºcustomer data held by internet service providers (ISPs), the possible configurations of home networks, and their connection with the ISP infrastructure. Customer homes are connected via cable modems to the ISP network. Typically, each home has a (normally WiFi-enabled) router to which the landline phone is connected, and with a number of Ethernet cable ports. WiFi networks are password protected and work in a frequency

[1] https://eclipse.org/sirius/gallery.html

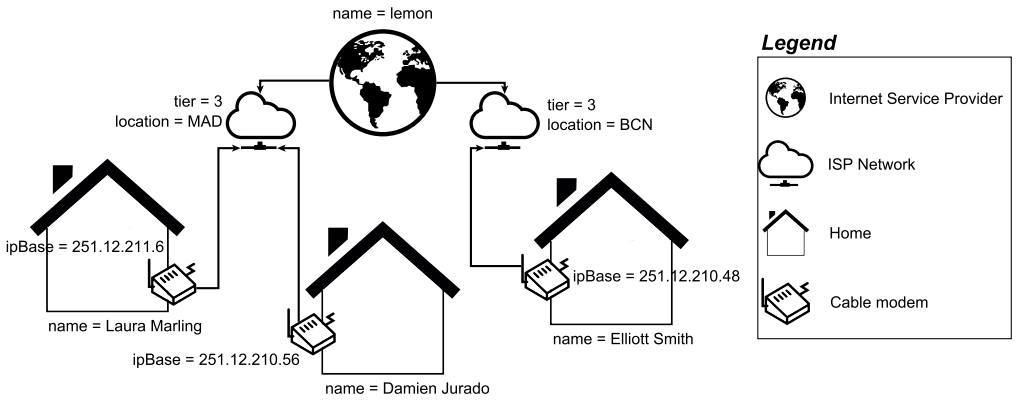


Figure 3.3: Fragment showing a connection between customer homes and an ISP.

range. Moreover, each home may have both cabled devices (e.g., PCs, printers or laptops) and wireless devices (e.g., smartphones, tablets or laptops).

Using the presented approach, DEs provide example fragments that illustrate interesting network configurations and depict the desired graphical representation for them. As an example, Figure 3.3 shows an example of a fragment representing the connection between some customer homes and the ISP through cable modems. The elements in the drawing define some properties, like the ipBase of cable modems, the name of the home owner, the tier and location of the ISP network, and the name of the ISP. The legend to the right assigns a name to every picture used in the drawing.

Figure 3.4 shows a real snapshot of the process, in which a sketch has been drawn by a DE, and automatically turned into a text fragment from which a meta-model has been induced. The meta-model is then converted into a visual DSML editor.

3.3 From Sketches to Text Fragments

The first challenge for implementing our proposed solution is being able to offer an environment in which DEs are enabled to sketch example models without having to become familiar with the insights of the modelling environment, and, at the same time, in a format that can be transformed into machine-processable data. In this way, from these fragments we will eventually generate valid EMF models and make them the fragments to our framework.

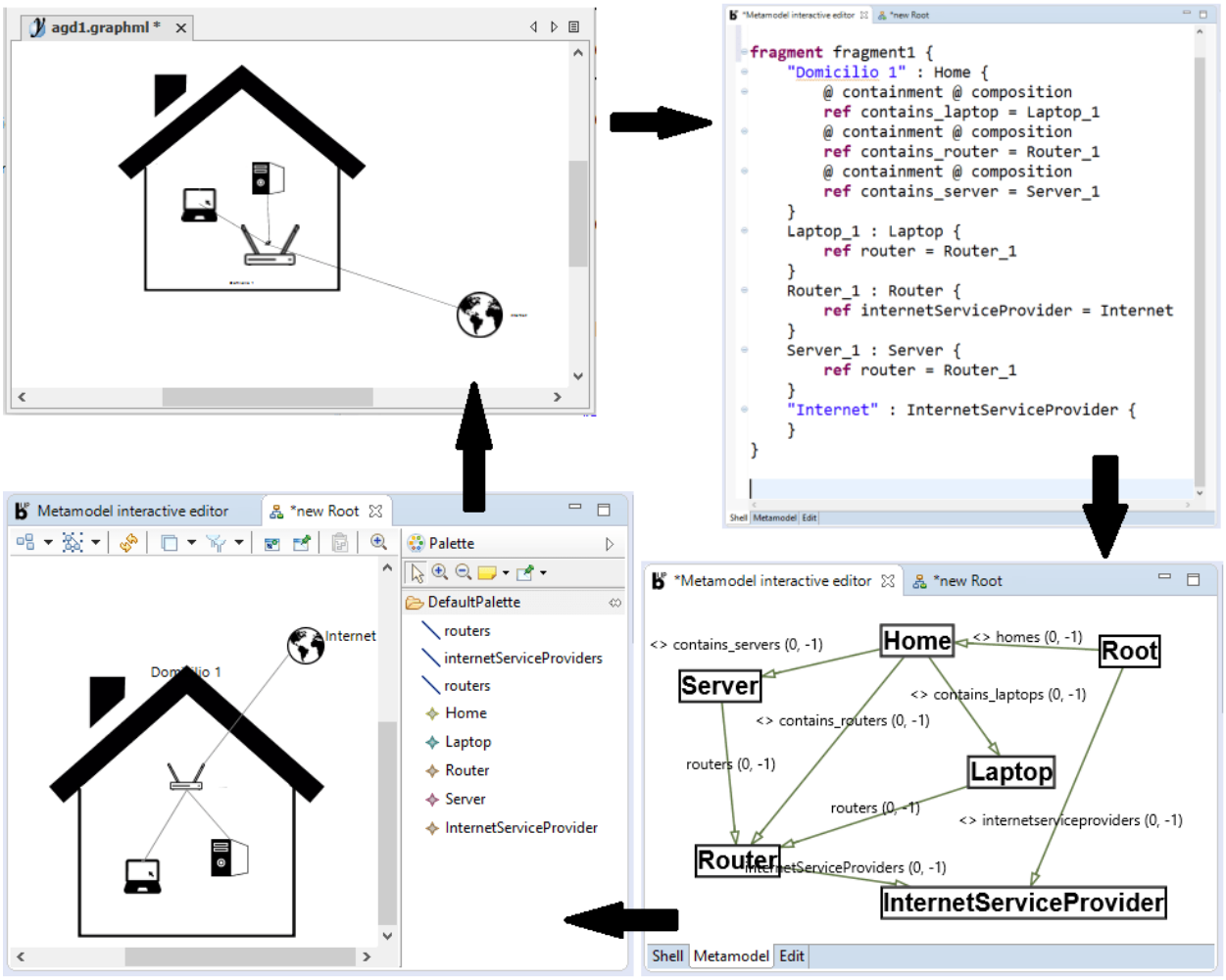


Figure 3.4: From a sketch to an editor.

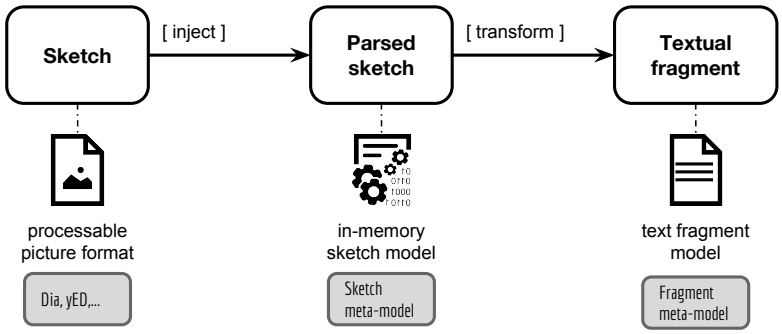


Figure 3.5: Derivation of a textual fragment from a sketch.

Although in this document we mostly refer to *fragments*, in the proposed framework, these can be declared either *fragments* or *examples*, being the latter a specialization of the former. Hence, examples are full-fledged models, necessarily complying with the whole structure of the meta-model. In contrast, fragments may miss certain mandatory objects and attributes, and violate the lower bound of cardinalities, as their purpose is concentrating in the nearby context of a particular situation of interest. In a regular context, the DE shall start providing fragments, which are not as restrictive as examples, and, as the process iterates, the use of examples could be more intensive. Moreover, the use of fragments becomes necessary when testing the DSL under construction (see Chapter 4).

The designed process splits the derivation of a fragment in two main steps: *injection* and *transformation*. The former takes an input sketch, made with a drawing tool, and analyzes it in order to create a new model that stores its graphical features in a tool-independent format. This model will be an instance of our own Sketch meta-model. The resulting model will be the input of the *transformation*, which shall eventually generate a second and final model conforming to our own Fragment meta-model. This meta-model aims to decouple the details of the example model from its graphical properties, applying a group of heuristics on how graphical properties ought to be interpreted and transformed into features and relationships at the abstract syntax. Figure 3.5 shows a schema of the sketch-to-text-fragment flow.

The first step for increasing the versatility of our framework, is defining a common meta-model to store the drawings that the DE is expected to deliver as model sketches. It should be extensible, permitting the easy addition of new drawing formats to our system. It needs to capture most the properties that can be found in drawings, including:

- Nodes.

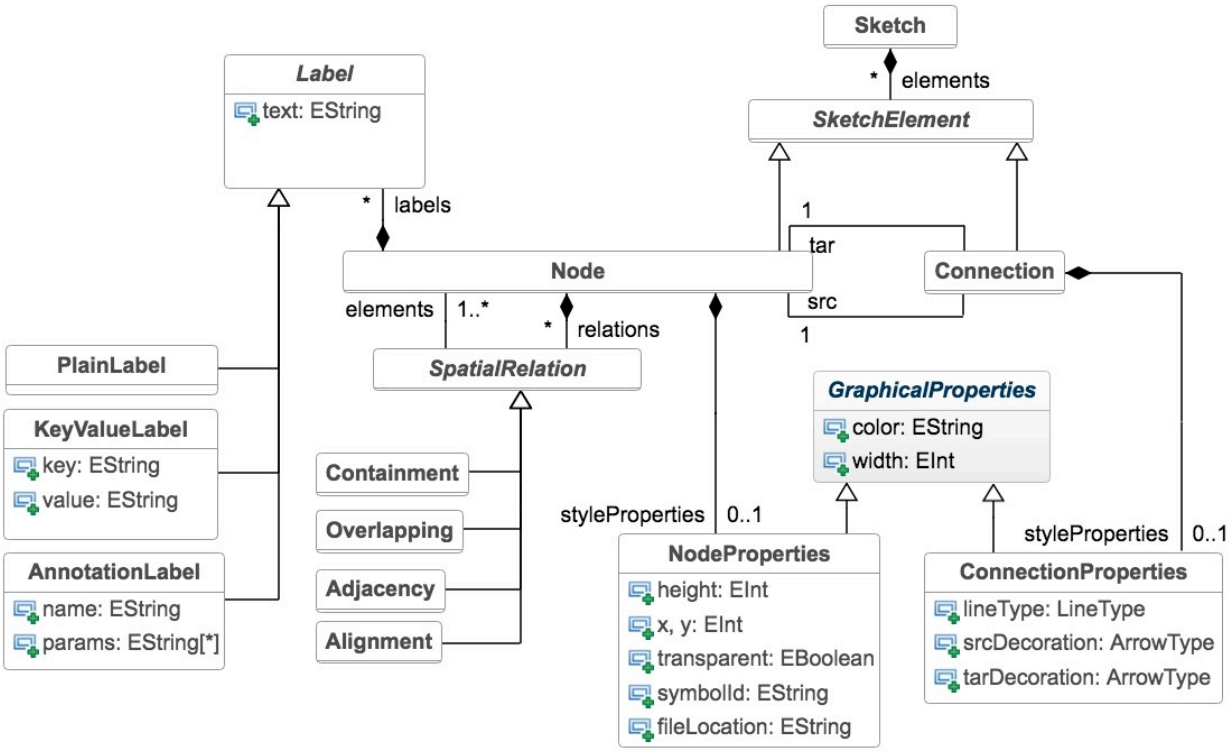


Figure 3.6: Meta-model for representing sketches.

- Lines, arcs and arrows.
- Node properties (color, size, placement).
- Line properties (color, width, style, connecting objects, edge decorators).
- Node and line labels.
- Spatial relationships among objects (containment, adjacency, overlapping, alignment).

Figure 3.6 shows the meta-model that captures the graphical representation elements in drawings. In our meta-model, files with drawings are called Sketches. Every sketch contains a series of elements. SketchElements can be either Nodes or Connections, as these are the most basic units that can be found in drawings. Once identified, sometimes it might be necessary to store certain style facets of each element that may give rise to domain constraints. These are assigned in a styleProperties component, attached to each SketchElement. The sort of style information each element is enabled to use depends on whether it is a Node or a Connection. NodeProperties include color, size (width and height), placement in the diagram (x and y) and transparency.

Elements may contain Labels, which may store either plain text (PlainLabel), key-value pairs (KeyValueLabel) or annotations (AnnotationLabel). Normally, the instantiation of one or another will depend on the content of the label string. For instance, a KeyValueLabel is identified with a string in the form <key> = <value>, being <key> and <value> arbitrary strings; an AnnotationLabel is commonly @ followed by a string value, as its parameters, if any, usually appear in parenthesis and separated by commas (occasionally in the form of key-value pairs). Finally, if none of the aforementioned patterns are met by the label string, we assume a PlainLabel occurrence. However, although these are the heuristics we have elaborated for our implementation system, new label-type sort policies can be implemented by the ME by providing a new Sketch importer.

The objects from the domain are connected to their concrete syntax using a *Legend*. Legend elements can be tracked in two different ways when generating new Sketch models. The first one is the attribute symbolId in NodeProperties. This field serves as a way to trace each node with its concrete syntax in case they need to be identified in an external source (like an index or an XML file). The second one is fileLocation, which explicitly links an image file containing the sole representation of a node. This is necessary for both detecting object type names and visual properties as fragment parsing take place, and specially when it comes to the concrete syntax generation phase. Hence, while symbolId assumes the node needs some kind of identifier for recovering it from auxiliary (legend) files, fileLocation is aimed to set a specific location where its graphical representation is to be found. As we will see in Chapter 5, the parsing of different formats as well as legend handling are given as an extensible feature in the framework.

When it comes to representing the spatial relationship between nodes, we consider cases in which geometric relations between them are considered significant for the domain. The supported relations are: containment, overlapping and adjacency, all children of SpatialRelation in Figure 3.6.

Figure 3.7 shows an excerpt from our running example representing a home router connected to a local network with access to the Internet. The upper part shows the drawing delivered by a DE, while the bottom contains the sketch model representation created from it.

The model representation of the sketch contains a root Sketch object holding every Node (isp, ispNetwork1, home1, cableModem1) and Connection (one from isp to ispNetwork1 and one from ispNetwork1 to cableModem1). Labels are likewise represented, having detected our system that

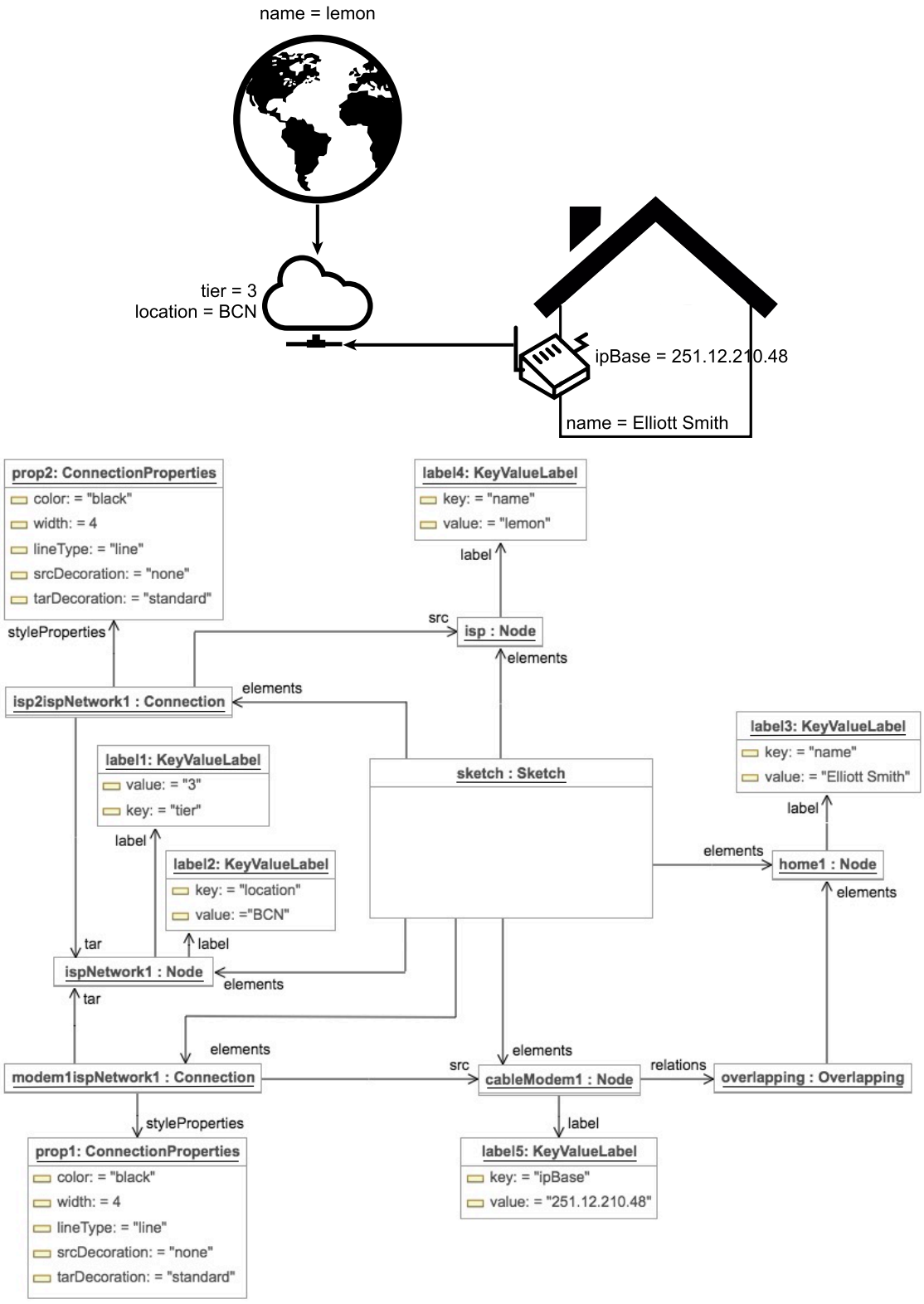


Figure 3.7: A sketch from the running example and its representation as a Sketch model.

they are key-value pairs, and the Node each one of them is attached to.

Finally, there is a spatial relationship between cableModem1 and home1 that is represented with an Overlapping object, and each Connection has its ConnectionProperties object associated, indicating the color, width, lineType, srcDecoration and tarDecoration of each edge. The details on how these graphical properties are detected and represented are explained in Section 3.5.

Once we have the user sketch stored in an in-memory model, we transform these sketches into model fragments. While the meta-model in Figure 3.6 captures the graphical elements in drawings (the concrete syntax of the model), our system infers the underlying abstract syntax, producing what we call model fragments.

Before going into detail with the specifics of the sketch-to-fragment transformation, we need to describe the meta-model for model fragments, shown in Figure 3.8. In our schema, a Fragment is made of Objects with Attributes and connected by References, all of which can be annotated. Annotations can have an arbitrary number of parameters (AnnotationParam). Each parameter must have a name. Annotations may refer either to elements in the fragment, or to primitive values (Integer, String, Boolean, etc.). It must be stressed that annotations are a powerful element that plays a pivotal role in our meta-model development process.

The relatively straightforward transformation uses some heuristics to improve the quality of the generated fragment. The mapping is as follows:

- Each `Node` is transformed into an `Object`. If the node contains a `NodeProperties` instance, then the corresponding `Object` is added the annotation *@style*, with one parameter for each attribute of `NodeProperties`.
- Each `Containment` spatial relation within a `Node` is transformed into a `Reference` annotated with *@composition* in the corresponding `Object`. The contained objects are the target of the reference.
- A `Connection` without key-value labels is transformed into a `Reference` owned by the object that corresponds to the source node of the connection. To avoid generating many monovalued references, connections with the same name pointing to objects of the same type (that is, with the same `symbolId` or `fileLocation`) are grouped into a multivalued reference. If the connection contains a `ConnectionProperties` instance, then the corresponding `Reference` is annotated with *@style*, with one parameter for each attribute of `ConnectionProperties`.

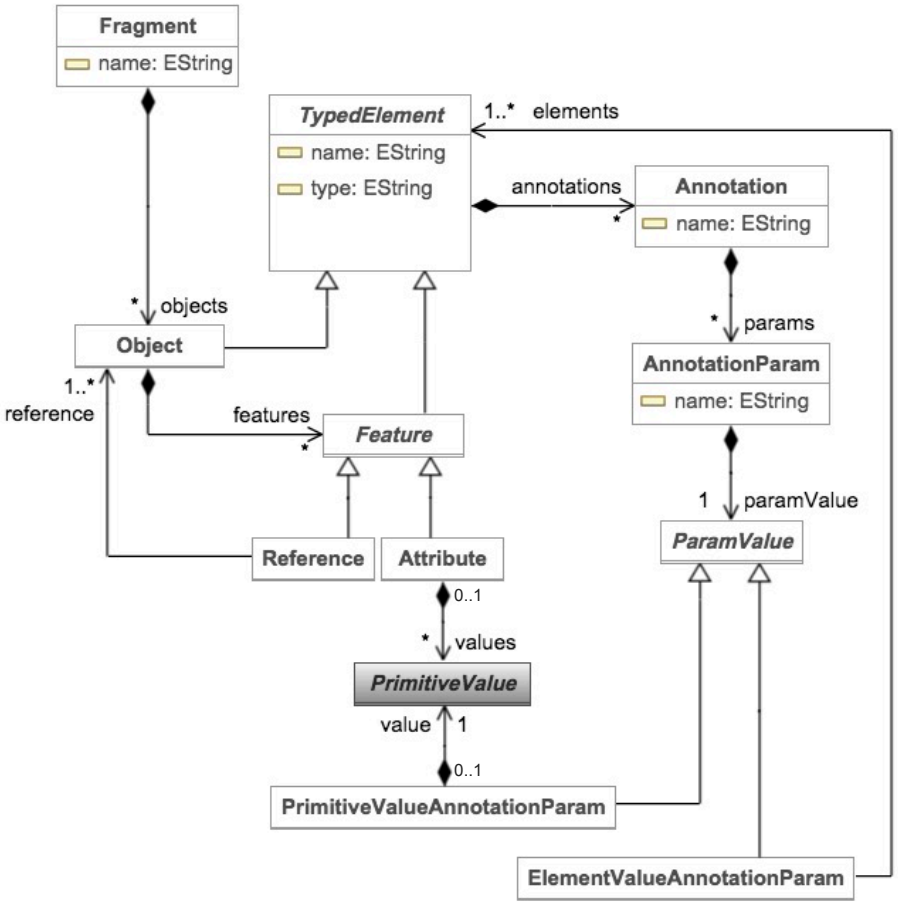


Figure 3.8: Meta-model for representing fragments.

- A `Connection` with key-value labels is transformed into an object annotated with *@connector*, with an attribute for each key-value, and containing two references pointing to the source and target objects.

- Each `KeyValueLabel` is transformed into an `Attribute`. The value found in the sketch is parsed to detect whether it is a decimal value, an integer, a boolean value or a string.

- Each `AnnotationLabel` is transformed into an `Annotation`, provided that the name of the annotation label is valid - that is, registered in the annotation catalogue (see in Sectionsec:annotations). Any parameter that refers to elements in the sketch is resolved to the actual fragment elements (objects and features).

- The first `PlainLabel` (if any) of a `Node` is used as the name for the generated object. We check for duplicate names, and add a numeric postfix if this is the case.

- The first `PlainLabel` (if any) of a `Connection` is used as the name of the generated reference or connector object. If no label is provided, we use the name of the target object to derive the name of the reference or the type of the generated connector object.

Executing the sketch-to-fragment transformation on the sketch model in Figure 3.7 would produce the fragment in Listing 3.1 as output, using a simple textual notation similar to the Human Usable Textual Notation (HUTN) [79].

In the Listing, it can be observed that an Object per Sketch Node was created. Additionally, Connectors are automatically transformed into References (marked with the reserved word *ref*), whereas each Key-Value label pair becomes an Attribute (identified with *attr*).

```
1  fragment yed_sketch {
2    home1 : Home {
3      attr name = "Elliott Smith"
4      @overlapping ref overlapping_cableModem = cableModem1
5    }
6
7    isp : InternetServiceProvider {
8      attr name = "lemon"
9      @style(...)
10     ref "#000000_4_Line_None_Standard" = ispNetwork1
11   }
12
13   cableModem1 : CableModem {
14     attr ipBase = "251.12.210.48"
15     @overlapping ref overlapping_home = home1
16     @style(...)
17     ref "#000000_4_Line_None_Standard" = ispNetwork1
18   }
```

```
19
20   ispNetwork1 : ISPNetwork {
21     attr tier = 3
22     attr location = "BCN"
23   }
24 }
```

Listing 3.1: Text fragment generated from sketch.

In the text version of fragments, style features are represented with annotations, and hence, each reference created from connectors is added a @style annotation including all the aforementioned ConnectorProperties attributes as parameters. Notice that, although these parameters have been omitted for simplicity, we will go through a complete example including style annotations (see Figure 3.13).

Moreover, the overlapping relationship between home1 and cableModem1 is represented by a pair of references pointing to each other and annotated with @overlapping. Again, Section 3.5.1.1, provides detail on how these relationships are detected and added to the text fragment.

At this point, the ME can edit the text fragment before being processed to produce a meta-model, as we detail in Section 3.4.

3.4 From Fragments to a Meta-model

With each user iteration, we obtain a new text fragment. These fragments progressively feed a growing meta-model describing the abstract syntax of a DSML. The format an abstract syntax can take in our system is represented in Figure 3.9, and it includes common elements found in meta-modelling languages, including: meta-classes, references and attributes. The supported primitive types in attributes are: String, Integer, Double and Boolean. Moreover, we enable the use of meta-model annotations, which we explain in detail in this section.

We provide a specific language for representing meta-models for two main reasons: On the one hand, it is intended to be as platform independent as possible. Hence, had we opted to employ ordinary meta-model representations like Ecore or MOF, the framework would have been natively directed to the use of a specific technology, such as EMF. On the other hand, these more standardized meta-modelling languages cover a considerably wider range of features than our framework, which was conceived under the premise of simplicity for DEs and MEs.

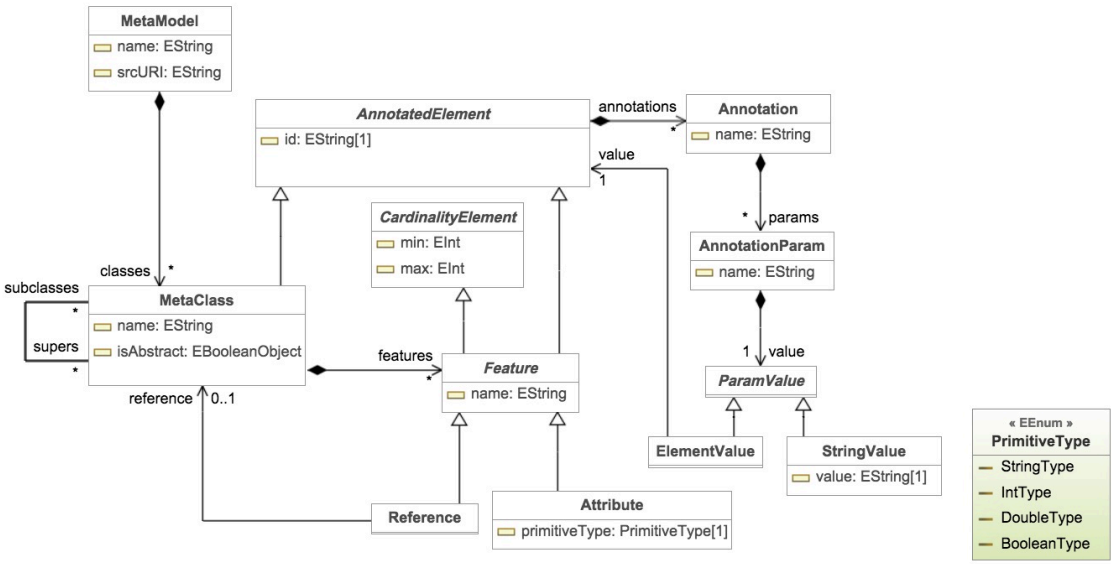


Figure 3.9: Meta-model for representing the abstract syntax of a DSL.

In this section, we describe our meta-model induction algorithm, how meta-model refactorings are applied, the strategy for conflict resolution, and the recommendation system.

3.4.1 The Meta-model Induction Algorithm

Given a fragment, our algorithm (shown in Figure 3.10) proceeds by creating a new meta-class in the meta-model for each object with distinct type (lines 2–4). If a meta-class already exists in the meta-model due to the processing of previous fragments or other objects within the same fragment, then the meta-class is not newly added. Then, for each slot in any object, a new attribute is created in the object's meta-class, if it does not exist yet (lines 6–8). Similarly, for each relation stemming from an object, a relation type is created in its meta-class, if it does not exist. The minimum cardinality of features is set to 0 by default (line 9). The maximum cardinality is set to unbounded in case it refers to multiple objects or primitive values. In case a feature is mono-valued, we take the convention of mapping plural feature names to multivalued references, and singular to mono-valued ones (lines 10–19). Additionally in the case of references, if two relations 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 relation type, with a subclass for the type of each target object (lines 20–29).

```
1: for every object in a fragment do
2:   if a meta-class with the same name as the object type does not exist then
3:     create new meta-class named after the type of the object.
4:   end if
5:   for every feature in an object do
6:     if a meta-class named like the object type does not have a feature with the same
       name and type then
7:       add new meta-class feature.
8:     end if
9:     set minimum cardinality to 0.
10:    if the feature is multivalued then
11:      set maximum cardinality to *.
12:    else
13:      if the feature has a plural name then
14:        set maximum cardinality to *.
15:      end if
16:      if the feature has a singular name then
17:        set maximum cardinality to 1.
18:      end if
19:    end if
20:    if the feature is a reference then
21:      if the reference points to multiple objects then
22:        if the target objects belong to a hierarchy then
23:          make the reference point to the closest-level common super-class.
24:        else
25:          add new abstract meta-class and make it common to all referenced objects.
26:          make the reference point to the new meta-class.
27:        end if
28:      end if
29:    end if
30:  end for
31: end for
```

Figure 3.10: Meta-model induction algorithm.

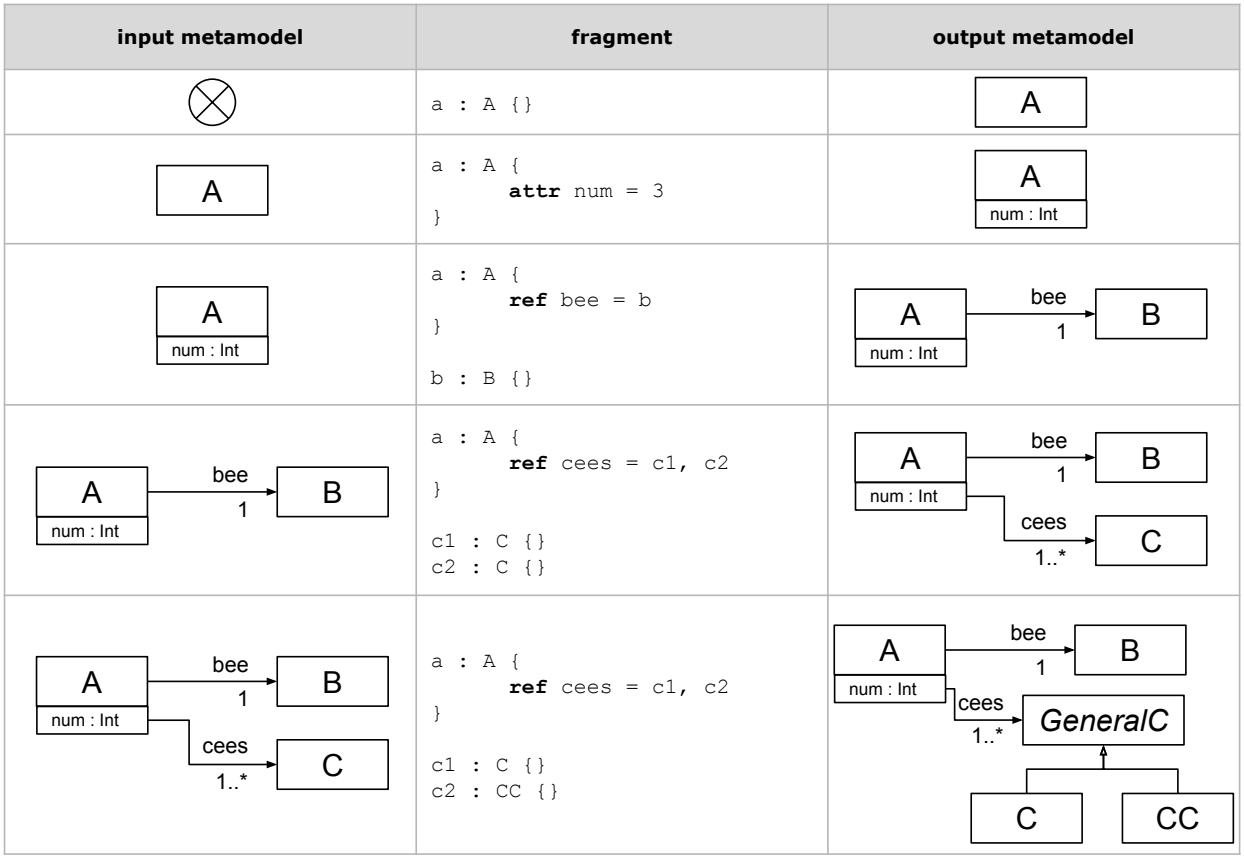


Figure 3.11: Example sequence of the meta-model induction algorithm execution

Figure 3.11 shows an example of the algorithm execution, with the meta-model state prior and after each fragment insertion.

The ME is allowed to configure the defaults for the minimum (0 or the minimum in the fragment) and maximum (unbounded or the maximum in the fragment) cardinality of relations. Once the meta-model has been produced, the user may modify it. As input fragments are automatically stored in a *Fragment* folder and validated in real-time against the current status of the meta-model, the system is able to detect inconsistencies in past fragments as the meta-model evolves.

Figure 3.12 shows the derived meta-model after executing the algorithm over the sketch fragment from Figure 3.7. Notice that the ME has performed a few changes over the originally inferred text fragment. Namely, references have been renamed with names closer to the domain. Moreover, one of the overlapping relationships between CableModem1 and Home1 has been removed, as it has been considered redundant (we only need it to be navigable one-way). In fact, the modeller has decided to make it a @composition association, understanding

that the spatial relationship between the objects implies such modelling reference type.

```
1   fragment yed_sketch {
2     @unique
3     internetServiceProvider1 :
          InternetServiceProvider {
4       attr name = "lemon"
5
6       @style (...)
7       ref networks = ispNetwork1
8     }
9
10    home1 : Home {
11      attr name = "Elliott Smith"
12      @overlapping @composition ref
            modem = cableModem1
13    }
14
15    cableModem1 : CableModem {
16      attr ipBase = "251.12.210.48"
17
18      @style (...)
19      ref network = ispNetwork1
20    }
21    ispNetwork1 : ISPNetwork {
22      attr tier = 3
23      attr location = "BCN"
24    }
25  }
```

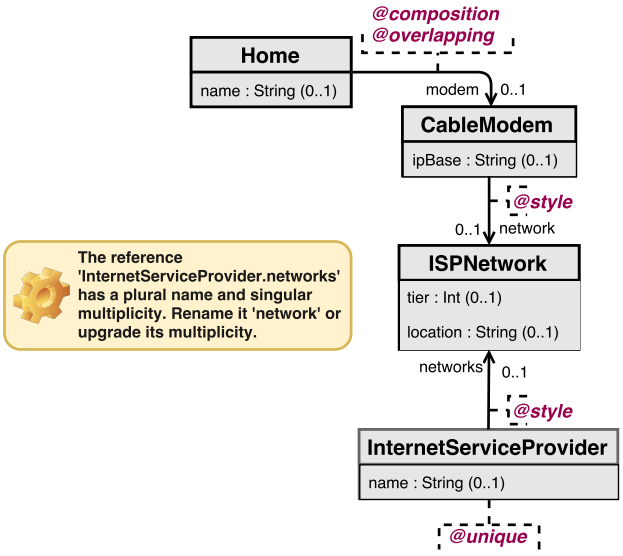


Figure 3.12: Meta-model construction iteration by fragment addition.

We will discuss the `@unique` annotation and the rename suggestion featured in Figure 3.12 in Sections 3.4.2 and 3.4.3 respectively. The latter is an automatic tip triggered by our system after the addition of the indicated fragment.

3.4.2 Annotations

As mentioned before, annotations play a pivotal role in the meta-model induction process. The ME is expected to use them at this stage of the process, as they are expected to complement the information given by the fragments. The use of arbitrary, unregistered annotations is not enabled in our system. Instead, they need to be either natively supported, or user-defined via extension points (see Chapter 5). Finally, it is worth mentioning that user-defined sketch importers are allowed to introduce annotations at the time a fragment is imported, avoiding the ME having to explicitly include them.

Annotations can be classified following two different criteria. On one hand, they can be either *persistent* or *volatile*. The former are transferred to the meta-model. In contrast, volatile

annotations only accompany fragment elements, as they are supposed to add information to the meta-model induction process, and hence can be discarded right after.

An annotation catalog is provided, covering a wide range of common meta-model development scenarios. Table 3.1 shows the supported annotations (first column), the fragment and meta-model elements they can annotate (second and third column, respectively), their possible parameters (fourth column), and a description of their functionality in the fifth column. Please note that, if persistent, these annotations are copied from the fragment to the corresponding elements in the induced meta-model. Additionally, Appendix A delves into the equivalent OCL expression for each constraint annotation.

The categories in which annotations are classified in Table 3.1 are the following:

- **Design**. They refer to meta-modelling design decisions that should be reflected in the meta-model generated from the fragments. Understandably, they are always persistent. Notice that few of these annotations are introduced in the induction process. For example, object references will be annotated as `@composition` in case they represent an association of that kind in a fragment. When the fragment is processed by the meta-model induction process, the annotation is copied into the corresponding reference in the meta-model.

- **Refactoring**. Normally volatile, these annotations are made available in order to introduce alterations in the meta-model induction process. For instance, a set of features annotated as `@general` will alter the meta-model induction process by introducing a new superclass holding all these features, and making every meta-class that formerly owned them be a subclass of the newly created one.

- **Constraints**. Meta-models are frequently complemented with integrity constraints (e.g. in OCL). Because we needed to provide this functionality, we enable the use of (persistent) high-level constraint annotations that permit adding such restrictions, and whose semantics is given in terms of OCL. For instance, one can forbid an object to reference itself by annotating one of its associations `@irreflexive`.

- **Geometry**. As previously discussed, it is our mechanism for indicating that certain fragment elements represent spatial relationships. Because this information will aid the generation of visual environments for DSLs, these annotations need to be persistent.

Annotation	Fragment	Meta-model	Parameter(s)	Meaning
Design				
@abstract	-	Metaclass	-	Marks the annotated metaclass as abstract.
@composition	Reference	Reference	-	Marks the annotated reference as composition.
@opposite	Reference	Reference	Reference	Marks the annotated reference as opposite to the one in the parameter. The other one needs to be annotated @opposite too, parameter-pointing to the first one.
Refactorization				
@general	Object*	-	String?	Takes every annotated object class and pulls common features up to another superclass whose name can be specified via a parameter. The superclass can be either new or existent. If the parameter is omitted, a heuristically inferred name is provided for a new common superclass.
@general	Feature*	-	-	Pulls the annotated elements up to an existing common superclass, or to a new common superclass if none exists.
@inline	Reference	-	-	Turns a reference targeting a class with no features into an int-type attribute with the same name, and removes the target class.
@merge	Object*	-	String?	Combines every annotated object's class features in a single one. If specified in parameter, the new class name is assigned to a new or existent one.
@multiplicity	Feature	-	Integer, Integer	Explicits a feature's minimum and maximum cardinality, independently of metaBup's induction algorithm policy.
@pluralize	Feature	-	-	Switches a feature's name into its plural form, setting max multiplicity to * if formerly 1.
@singularize	Feature	-	-	Switches a feature's name into its singular form, setting min and max multiplicity to 1 if formerly greater.
Constraints				
@acyclic	Reference	Reference	-	A given reference is acyclic.
@covering	Reference	Reference	Reference*	A given set of references $\{ref_i\}$ pointing to the same class A is jointly surjective: each A object receives some reference from the set $\{ref_i\}$.
@cycleWith	Reference	Reference	Reference*	If defined, a given reference must commute with a sequence of references. The sequence of references can be of any length.
@irreflexive	Reference	Reference	-	Forbids self-loops through a reference.
@nand	Reference*	Reference*	-	A given set of references cannot all have value at the same time. All references must start from, or come into, the same class.
@subset	Reference	Reference	-	The values held by a given reference are a subset of those held by another one. Both references must be owned by the same class.
@tree	Reference	Reference	-	A given reference spans a tree.
@unique	Object	MetaClass	-	A given class is unique (it can only be instantiated once).
@unique	Attribute	Attribute	-	An attribute represents an object identifier.
@xor	Reference*	Reference*	-	One and only one of a given set of references should have a value. All references must be owned by the same class.
Geometry				
@containment	Reference	Reference	-	The annotated reference models a containment relationship between the source object (container) and target object (containee).
@overlapping	Reference	Reference	-	The annotated reference models the source object overlaps (lays over) the target object.
@adjacency	Reference	Reference	Position, Position	The annotated reference models an adjacency relationship between two objects. The first parameter indicates the side from which the source object touches the target, whereas the second one flags an alignment side between the two objects, if any. Possible position values are: top, bottom, right and left.

Table 3.1: *metaBup* annotation catalog.

It is important to highlight that the advantage of using annotations instead of directly OCL invariants is twofold: On the one hand, annotations are simpler to use for non meta-modelling experts and/or OCL developers, as they are higher-level than pure OCL. On the other hand, annotations are compiled into different OCL expressions depending on the properties of the annotated element (e.g., the direction of the involved references or their multiplicity), and on the particular compilation platform (as the same element can be compiled differently depending on the target platform).

For example, our ME introduced a @unique annotation in the text fragment from Figure 3.12. This annotation tags objects (and subsequently their meta-classes) indicating that they are allowed to be instantiated at most once. Listing 3.2 shows the equivalent OCL code. More interestingly, Appendix A shows the entire comparison between each one of the annotations from our catalog, and their OCL version.

```
1 context <class> inv unique:
2   <class>.allInstances()->size() <= 1
```

Listing 3.2: Equivalent OCL code for @unique annotation

The second fragment from our running example (the one in Figure 3.13) leads our meta-model to a second increment. Figure 3.14 shows the parsed fragment in textual syntax, edited by the ME (to the left) and the resulting meta-model (to the right). In the imported fragment from Figure 3.13, we notice that, firstly, all elements contained in Home1 have been assigned to a single reference (devices). This decision automatically triggers the creation of the abstract meta-class *Device*, as seen in the meta-model to the right. Secondly, the auto-generated @adjacency annotation originally featured a side parameter that has been removed, indicating that the orientation of the adjacent Ports to Router is irrelevant in the visual domain. In this same context, the opposite references from Port objects to Router have been removed. Style names for references have been renamed and given names closer to the domain (Router.modem).

Finally, notice that all persistent annotations have been copied to their corresponding meta-model elements.

3.4.3 Recommendations

A virtual assistant which continuously monitors the meta-model is provided for detecting places where the meta-model design can be improved and recommending solutions, based on well-known design patterns, refactorings and style guidelines. Table 3.2 shows the recommen-

```
1   fragment yed_sketch {
2     home1 : Home {
3       @overlapping ref overlapping_cableModem = cableModem1
4       @composition @containment ref contains_router = router1
5       @composition @containment ref contains_fixedPhone =
            fixedPhone1
6       @composition @containment ref contains_wifiNetwork =
            wifiNetwork1
7     }
8
9     router1 : Router {
10      @adjacency(side = "bottom") ref adjacent_ports = port1,
            port2
11
12      @style ( color = "#000000",
13            width = 3,
14            line = dashed,
15            source = none,
16            target = crows-foot-many )
17      ref "#000000_3_Dashed_None_CrowsFootMany" =
            cableModem1
18    }
19
20    fixedPhone1 : FixedPhone { }
21
22    wifiNetwork1 : WifiNetwork { }
23
24    port1 : Port {
25      @adjacency(side = "top") ref adjacent_router = router1
26    }
27
28    port2 : Port {
29      @adjacency(side = "top") ref adjacent_router = router1
30    }
31
32    cableModem1 : CableModem {
33      @overlapping ref overlapping_home = home1
34    }
35  }
```

Figure 3.13: A sketch and text fragment, featuring the three main supported spatial relationships.

Name	Element	Condition	Effect
Structural suggestions			
Inline class	class	A class A refers to a class B using a reference with cardinality 1..1.	Classes A and B are merged.
Pullup features	class	A set of classes define common features.	The common features are pulled up to a new or existing common superclass.
Generalize references	class and reference	A set of classes $A_1, ..., A_n$ receive references $r_1, ..., r_n$ from a class B.	A common abstract superclass A is created for $A_1, ..., A_n$, if it does not exist. References $r_1, ..., r_n$ are replaced by a new reference r from B to A, with cardinality *.
Replace class by integer	class	A class with no features or children is target of a reference.	The class is removed. An integer attribute is added to the source class of the reference.
Remove abstract class	class	An abstract class with no features has no incoming references.	The class is removed.
Naming style suggestions			
Number conflict	class attribute reference	(1) a multivalued feature has singular name, or (2) a class has plural name, or (3) a monovalued feature has plural name.	(1) suggests using a plural name, (2) suggests using a singular name, (3) suggests using a singular name or changing the multiplicity to *.
Class prefix	attribute reference	The name of a feature has the form $\langle_{owning\text{-}class\text{-}name}\rangle X$.	Suggests renaming the feature to X.
Class camel case	class	The name of a class is not in upper camel case.	Converts the class name to upper camel case, taking care of underscores and slashes.
Feature camel case	attribute reference	The name of a feature is not in lower camel case.	Converts the feature name to lower camel case, taking care of underscores and slashes.

Table 3.2: Recommendations for meta-model improvement.

dations currently supported, which we categorize into *structural* and *style* suggestions. All recommendations are activated when their condition is met (third column), and if accepted by the ME, they will trigger a certain meta-model refactoring (fourth column).

```
1   fragment yed_sketch {
2     Home1 : Home {
3       @overlapping ref modem =
            CableModem1
4       @composition @containment
5       ref devices = Router1, FixedPhone1,
            WifiNetwork1
6     }
7
8     Router1 : Router {
9       @adjacency @composition
10      ref routerPorts = Port1, Port2
11
12      @style(...)
13      ref modem = CableModem1
14    }
15
16    FixedPhone1 : FixedPhone { }
17    WifiNetwork1 : WifiNetwork { }
18
19    Port1 : Port { }
20    Port2 : Port { }
21
22    CableModem1 : CableModem { }
23  }
```

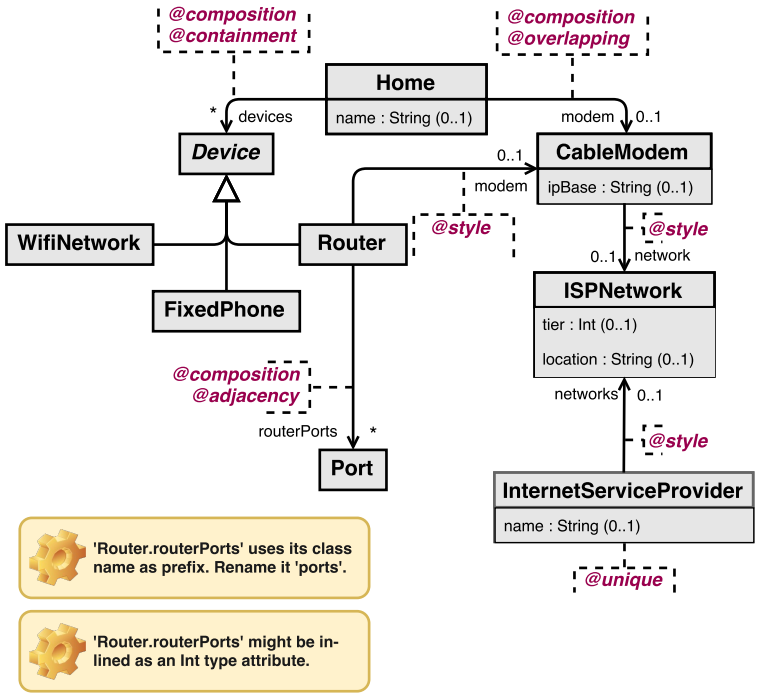


Figure 3.14: Meta-model construction iteration by fragment addition (2).

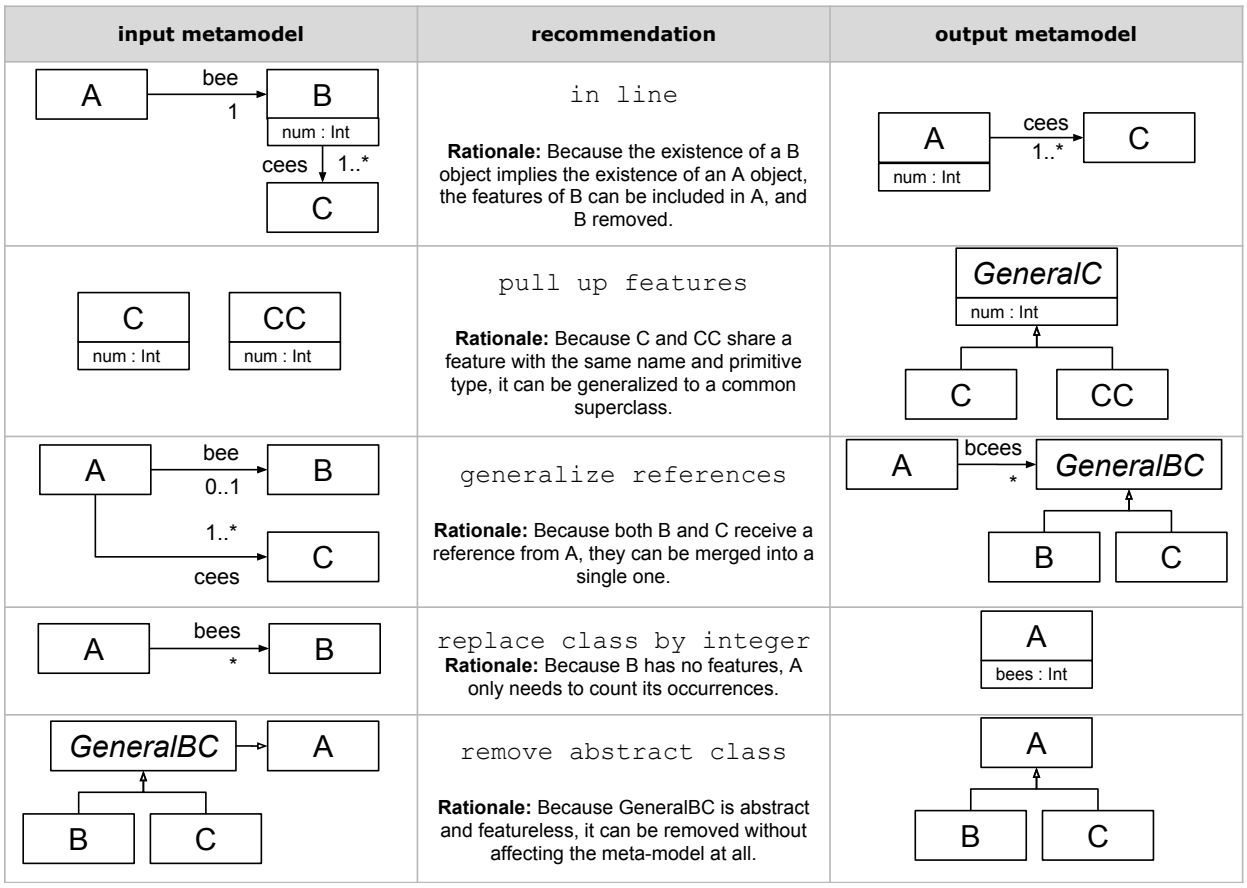


Figure 3.15: Recommendation-triggered refactorings.

Figure 3.15 illustrates the effect that some of these recommendations have on meta-model elements.

The *Inline class* recommendation is given when a class `B` is referenced from another class `A` through a reference with cardinality `1..1`. Accepting the recommendation merges the two classes, i.e., the attributes and incoming/outgoing references of `B` are copied into `A`, and `B` is removed. This well-known refactoring [33], which makes sense if class `B` does not add much value by itself, results in simpler meta-models with less classes.

The *Pullup features* recommendation detects maximal sets of common features and references among the existing classes, and proposes either pulling the features up if a common superclass exists, or creating a common abstract superclass if the affected classes do not share a common parent. This is another well known refactoring [33], which leads to simpler meta-models by removing duplicate fields. Technically, we use the clustering methods of Formal Concept Analysis [26] to detect sets of common features.

The *Generalize references* recommendation proposes the creation of a common abstract superclass A for a set of classes $C = \{A_1, ..., A_n\}$ that receive a set $R = \{r_1, ..., r_n\}$ of references from another class B. In addition, a reference r from B to A is created, "merging" the reference set $\{r_1, ..., r_n\}$, which gets deleted. The cardinality of r is $[\sum_{r_i \in R} a_i, b]$, where $[a_i, b_i]$ is the cardinality of reference r_i, and $b = *$ if some $b_i = *$, else $b = \sum_{r_i \in R} b_i$. This recommendation leads to a better structured meta-model, extracting a common superclass for the $A_1, ..., A_n$ classes that reflects their commonality (all can be accessed from B).

The *Replace class by integer* recommendation appears if a class with no features or children is referenced from only one class. In such a case, it is recommended to replace the class by an integer attribute in the source class of the reference, as this attribute should suffice to count the number of objects in the collection. Applying this recommendation leads to simpler meta-models, with less classes.

The *Remove abstract class* removes an intermediate abstract class with no features from an inheritance hierarchy. This class may have been created due to a generalization of some common features, which at some point have been generalized again to a higher class.

Regarding naming style suggestions, if a reference is multivalued but its name is singular, the assistant suggests changing the name to plural. If a reference is mono-valued but its name is plural, the assistant suggests either changing the name to singular, or increasing the upper multiplicity to *. The default recommendation in this case can be configured by the user. If an attribute name contains the name of the owning class as prefix, the assistant suggests the removal of the prefix (as recommended in [14]). Further suggestions take care of the capitalization of feature and class names, reflecting widely used modelling style guidelines [80].

If we go back to the two input fragments from our running example shown in Figures 3.12 and 3.14, a set of recommendations are displayed right after each iteration. Specifically, after adding the first fragment, it was detected the mismatch between the name of the reference networks in the meta-class InternetServiceProvider and its maximum multiplicity (1). Because one would expect a cardinality greater than one in a plural-named reference, the system suggests the modeller to perform either a renaming operation over it, or a multiplicity upgrade.

When the induction algorithm is fed the second fragment, two more recommendations are triggered: in the first place, the use of a class name (*router*) as prefix in one of its references is detected. Typically, this is considered a bad modelling practice, as it is redundant [4]. Moreover, because the class Port has no features, the system suggests its conversion to an

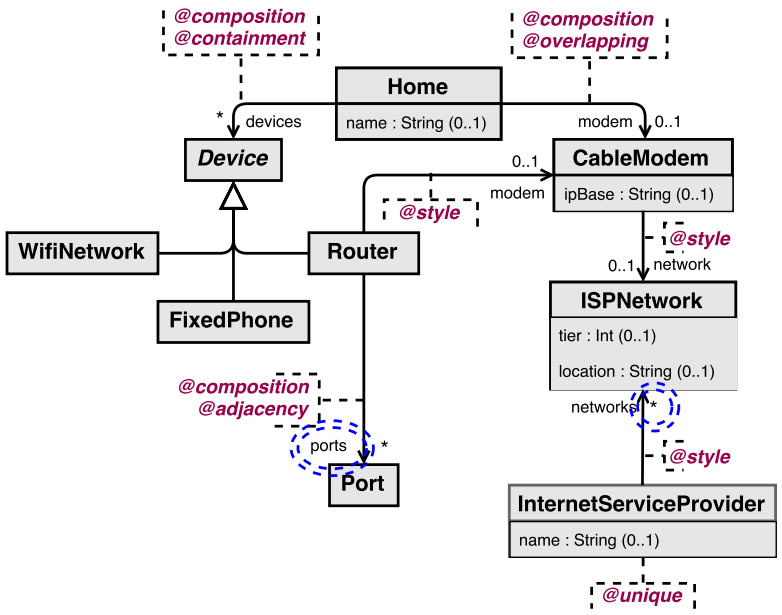


Figure 3.16: Meta-model from the running example after applying recommendations.

integer counter. In this case, the modeller does not apply the recommendation because he knows that the target implementation platform will require visual elements to be represented by a meta-class.

Hence, the modeller opts to apply the first two recommendations. The result is shown in Figure 3.16. Moreover, if a new requirement arises, the meta-model can be changed manually. For example, assume a Router element needs to be granted access to some ISPNetwork. The ME is enabled to edit the minimum multiplicity of that reference as reflected in Figure 3.16, being former and future examples obliged to fulfill this new constraint.

3.5 Example-driven Development of Graphical Domain-Specific Languages

Apart from inducing and helping improve the abstract syntax of the aimed DSL, the proposed system collects information regarding the concrete syntax when parsing the graphical examples. Retrieving that graphical information lets our system automatically derive a concrete syntax close to the DE's conception, minimizing the job of the ME.

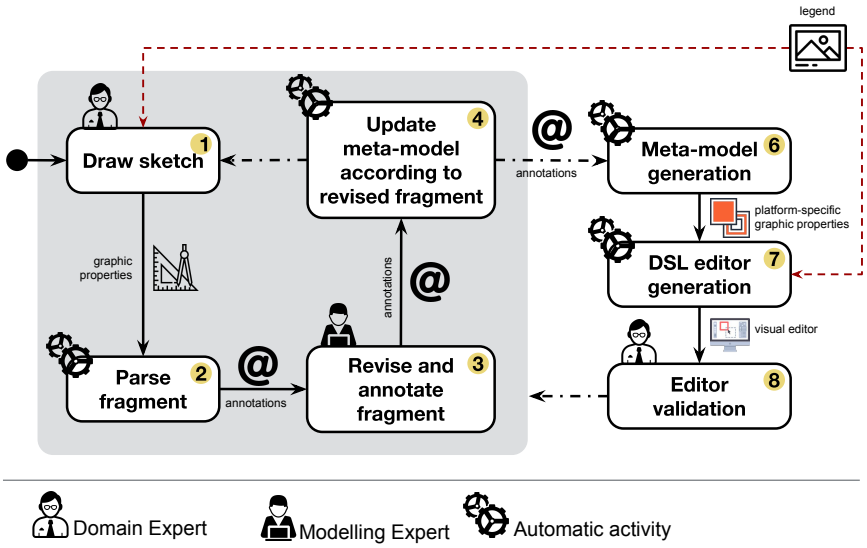


Figure 3.17: Graphical data flow through the example-based process.

As Figure 3.2 showed, the ultimate goal of this work is to provide a system for producing easy-to-develop Domain-Specific Visual Language (DSVL) editors. The final tasks leading to this correspond to the last stages (7, 8) of the described process.

Figure 3.17 shows an adapted version of Figure 3.2, in which the flow of data representing graphical properties is depicted. In this case, the V&V activity (labelled 5 in the original process in Figure 3.2)has been omitted as it is not involved in this part of the process.

As already discussed, every sketch that is added to the system starts an iteration encompassing the grey-shaded activities in Figure 3.17. Each of these sketches (label 1) has some implicit graphical properties that are parsed, together with the rest of the fragment, in the form of annotations (label 2). After the ME has revised the text fragment content (label 3), these annotations are copied to the corresponding elements from the derived meta-model when it is updated (label 4). This iterative process (gray-shaded in the Figure), finishes with the generation of a definitive abstract syntax (label 6) from which the concrete syntax (i.e., the model editor) should be derived (label 7). Finally, the DE is in charge of validating the resulting graphical environment (label 8).

3.5.1 Graphic property processing

In addition to the construction of the abstract syntax, the mechanism described in Section 3.3, provides support for storing graphical properties detected in the sketches. Specifically:

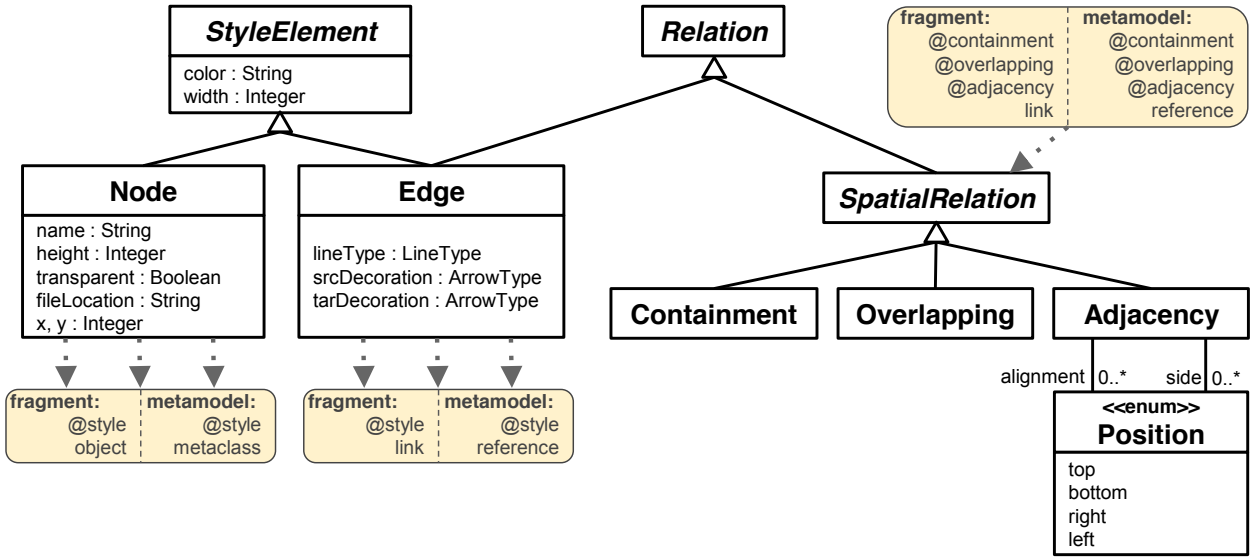


Figure 3.18: Graphical properties inferable from fragments, and corresponding annotations.

icons for the representation of each object, connector style, and existent spatial relationships amongst the different objects. These properties are used for the generation of a DSVL preserving the original aesthetics of the sketches provided by the DE.

Figure 3.18 shows a conceptual model depicting the graphical properties that can be automatically extracted from fragments and used to derive the concrete syntax of a DSVL. Some are explicit features from the elements in drawings, like their colour or size. Other properties are implicit relationships concerning the relative position of icons, like overlapping or adjacency. Both kinds of graphical properties are encoded as annotations of the corresponding objects and link in the textual representation of the fragment. Then, these annotations are transferred to the appropriate domain meta-model classes and references when the fragment is processed. Figure 3.18 also shows the correspondence between the graphical properties and the elements they can annotate.

Although some clues on how graphical properties are extracted and fitted into our Sketch meta-model have already been introduced in this chapter, their detailed extraction and processing are explained next.

First of all, each icon employed in the provided fragments is retrieved, as this is the most relevant aspect of the appearance that the DE expects from the final DSVL. Because most visual editor platforms demand the definition and usage of palettes with all available icons, a directory is purposely made available for storing a copy of the files containing the icons as

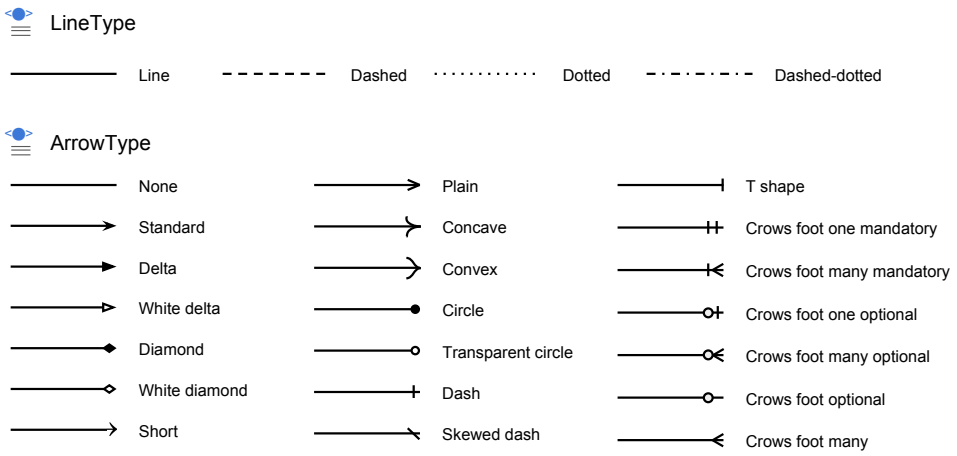


Figure 3.19: Line and arrow type catalog for Sketches.

they are added to the palette. This directory is referred as the *Legend*. The legend files are employed both in the serialization of fragments and in the generation of the concrete syntax, and are named according to the icon they contain.

Secondly, Figure 3.19 shows the list of available line and arrow decorators the system is capable of detecting and classify with regard to their color, line width, style and source and target decorations. As an example, Figure 3.20 contains an edge linking a Router and a Cable modem, and Listing 3.3 shows the textual representation of the improted fragment. When the fragment is imported, the link is annotated with the identified style (lines 26–28 in Listing 3.3), and its name is made of the concatenation of the graphical features of the style. For instance, the name inferred for the link is not modem, but the one struck out (see lines 29–30 in Listing 3.3). Because the text fragments can be edited, the ME has replaced the inferred name with one closer to the domain. What is interesting about this operation is that, from this moment on, each time a link with the same style between a router and a cable modem is imported, it will be automatically named modem. If the ME renames the reference in the future, he will be offered two options: either to replace the previous name modem with the new one, or creating a new reference in class Router which would coexist with the existing reference modem.

By taking the edge style as a source of information, two links between the same two objects will have the same type if their style coincides, and a different type if their style is different. This avoids the *symbol overload* problem for link types [78]. Although this functionality is optional and subjected to the interest of the domain, if deactivated, any link between the same two objects will be assigned the same type, and it will be named using the type of

the link's target object in lowercase. In this way, any graphical information annotation on a link will be transferred to the corresponding meta-model reference, and eventually, to the concrete syntax generator.

Thirdly, inducing the different spatial relationships of objects in a sketch requires a more advanced strategy. Sometimes, spatial relationships between graphical objects have a meaning in the domain and need to be modelled. It is even likely that the DE is unaware of whether layout implies domain requirements or not. Because of this, spatial relationships in sketches are automatically detected, and corresponding relationships between meta-classes declared in the meta-model, annotating these references with the type of spatial relationship they represent in each case.

Finally, notice that Figure 3.20 shows to the right the *legend* folder that contains the files used to represent each domain object in the fragment to the left.

```
1  fragment RouterDirectlyConnected2Modem {
2    home1 : Home {
3      attr phoneNo = 5550225
4      attr name = "Phil Ochs"
5
6      @overlapping
7      @composition
8      ref modem = cableModem1
9
10     @containment
11     @composition
12     ref devices = router1
13
14     @containment
15     @composition
16     ref phones = fixedPhone1
17
18     @containment
19     @composition
20     ref wifiNetworks = wifiNetwork1
21   }
22   router1 : Router {
23     @adjacency(side = bottom)
24     ref ports = port1, port2
25
26     @style ( color = "#000000", width = 3,
27         line = dashed, source = none,
28         target = crows-foot-many )
29     ref '00000_3_dashed_none_crows-foot-many'
30         modem = cableModem1
31   }
32   fixedPhone1 : FixedPhone { }
33   wifiNetwork1 : WifiNetwork {
34     attr name = "myWifi"
35     attr password = "myPw"
36   }
37   port1 : Port { attr portNo = 2 }
38   port2 : Port { attr portNo = 1 }
39   cableModem1 : CableModem {
40     attr ipBase = "251.12.211.16"
41   }
42 }
```

Listing 3.3: Textual representation of the fragment in Figure 3.20

It is up to the ME to keep or discard these relationships by editing the textual fragments, resolving whether space and object arrangement is relevant to the domain or not.

The three kinds of spatial relationships currently supported are:

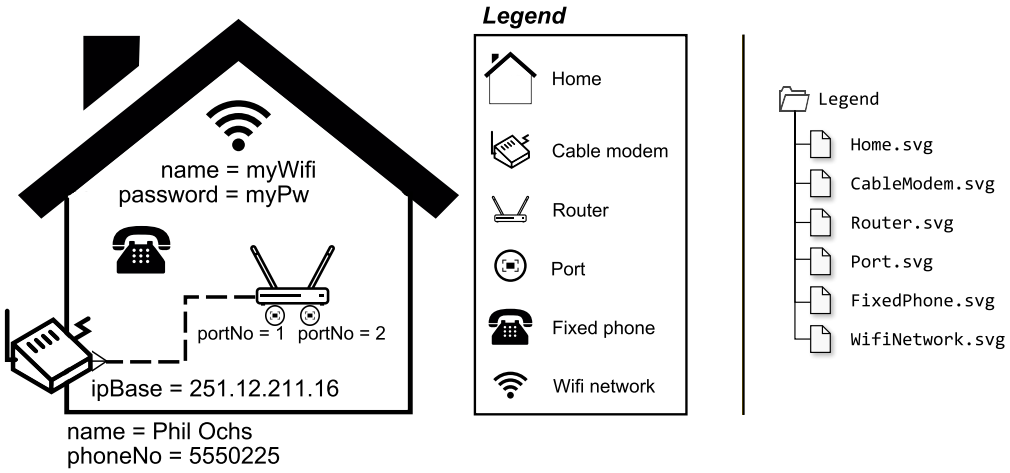


Figure 3.20: Fragment with spatial features (left). Content of the *legend* folder (right).

- *Containment*: a graphical object is within the bounds of another.
- *Adjacency*: two graphical objects are joined or very close. The maximum distance with which adjacency is to be considered is user-defined (*0* by default). Two optional properties are likewise detected: the side(s) from which objects are attached to each other, and alignment, that is, the special case in which one side of each object share a coordinate in the axis with the other.
- *Overlapping*: two graphical objects are superimposed (but not contained).

When one of these spatial relationships is detected, it is explicitly represented as a reference in the meta-model. In the case of containment, the reference is added to the container and points to the containee. For adjacency and overlapping, the following heuristic is applied: if an object `o` overlaps or is adjacent to more than one object of the same kind, the reference stems from `o`'s class; otherwise, the reference stems from the class of the bigger object, and if all objects have the same size, there is the possibility to make the reference bidirectional. The rationale is that, frequently, the different parts of bigger objects are represented as smaller affixed elements (e.g., a component with affixed ports).

The fragment in Figure 3.20 illustrates the three kinds of spatial relationships, which are automatically detected when the fragment is imported (see Listing 3.3). In the fragment, the Home contains a Router, a Fixed phone and a Wifi network. Hence, in the textual representation of 3.3, the Home object has three links annotated as @containment (lines 12, 16 and 20). The Home overlaps with a Cable modem in the fragment, being the Home icon bigger; therefore, the Home object is added a link annotated as @overlapping in the textual representation (line 8). Finally,

the Router has two adjacent Ports to the bottom side in the fragment. Since there are multiple ports, the Router is added a link annotated as @adjacency in the textual representation (line 24). The side parameter of this annotation indicates the side where the adjacency occurs (at the bottom of the router), but it can be removed if this is irrelevant to the domain.

In addition to creating explicit links for the detected spatial relationships, the fragment importer heuristically adds @composition annotations to the created links (see lines 7, 11, 15 and 19 in Listing 3.3). This helps in organizing and realizing only a minimal but *sufficient* set of meta-model references, in the sense that it suffices to capture all inferred spatial relationships. For example, both Ports in the fragment are contained in the Home, but this relation is not made explicit in the textual representation because they are already adjacent to the Router, which is inside the Home. In this case, the @composition annotation of the abstract syntax is used to infer that they are indirectly contained in Home objects via Router objects. The resolution of space conflicts is tackled in depth in Section 3.5.1.1.

Please note that back in Figure 3.6 these relationships were modelled as classes for a more clarifying description of the sketch meta-model. However, because these properties need to persist to the meta-model under development for completing the later concrete syntax generation, their occurrence is automatically inferred by the values of node attributes x, y, width and height in the sketch.

Thus, let A $(w_A,\ h_A,\ x_A,\ y_A)$ and B $(w_B,\ h_B,\ x_B,\ y_B)$ be bounding boxes of two geometric objects such that w is the object's width, h its height, x its position in the X axis, and y its position in the Y axis. Assuming that w and y are greater than 0, then we say that A *contains* B (written $contains(A, B)$) if:

$$contains(A, B) = x_A \leq x_B \ \wedge\ y_A \leq y_B \ \wedge\ (x_A + w_A) \geq (x_B + w_B) \ \wedge\ (y_A + h_A) \geq (y_B + h_B)$$

Likewise, we say that A *overlaps* B if it matches the following condition:

$$overlaps(A, B) = \neg contains(A, B) \wedge x_A < (x_B + w_B) \ \wedge\ (x_A + w_A) > x_B \ \wedge\ y_A < (y_B + h_B) \ \wedge\ (y_A + h_A) > y_B$$

Finally, let k be a constant value. A is *adjacent* to B with k or less separation units (written $adjacent(A, B, k)$) if it does not overlap it and matches at least one of the following conditions:

$$top - adjacent(A, B, k) :$$

$$(y_B \geq (y_A - k) \wedge (y_B + h_B) \leq y_A) \wedge ((x_B \geq x_A \wedge x_B \leq (x_A + w_A)) \vee ((x_B + w_B) \geq x_A \wedge (x_B + w_B) \leq (x_A + w_A))).$$

Figure 3.21: Supported spatial relationships: i) Adjacency ii) Alignment iii) Containment iv) Overlapping.

$right - adjacent(A, B, k)$:

$(x_B \leq (x_A + w_A + k) \wedge x_B \geq (x_A + w_A)) \wedge ((y_B \geq y_A \wedge y_B \leq (y_A + h_A)) \vee ((y_B + h_B) \geq y_A \wedge (y_B + h_B) \leq (y_A + h_A)))$.

$bottom - adjacent(A, B, k)$:

$(y_B \leq (y_A + h_A + k) \wedge y_B \geq (y_A + h_A)) \wedge ((x_B \geq x_A \wedge x_B \leq (x_A + w_A)) \vee ((x_B + w_B) \geq x_A \wedge (x_B + w_B) \leq (x_A + w_A)))$.

$left - adjacent(A, B, k)$:

$((x_B + w_B) \geq (x_A - k) \wedge (x_B + w_B) \leq x_A) \wedge ((y_B \geq y_A \wedge y_B \leq (y_A + h_A)) \vee ((y_B + h_B) \geq y_A \wedge (y_B + h_B) \leq (y_A + h_A)))$.

Hence, the adjacency condition can be summarized as follows:

$adjacent(A, B, k) = \neg contains(A, B) \wedge (top{-}adjacent(A, B, k) \vee right{-}adjacent(A, B, k) \vee bottom{-}adjacent(A, B, k) \vee left - adjacent(A, B, k))$

Figure 3.21 illustrates graphically the supported spatial relationships. We only detect alignment in case of adjacency; this is due to the fact that in most cases, object placement in example models is arbitrary, and hence many non-adjacent elements might be aligned without having any significance to the domain. Finally, notice that the side of A that borders with B is detected depending on which condition above is satisfied: top, right, bottom or left.

Figure 3.13 shows one more sketch, together with its text fragment automatically inferred by our system. Spatial relationships between objects in the sketch are represented by annotated references in the fragment. Thus, in the figure, home1 keeps an overlapping relationship with cableModem1, and containment associations with the remaining elements. By default, a containment relationship is also interpreted as a composition reference, as in most cases, the existence of contained elements is conditioned by their containers.

Also notice that the style parameters omitted in the example from Figure 3.7 are now shown in the reference created for the connector linking router1 to cableModem1, and that these coincide with each of the ConnectionProperties from Figure 3.6.

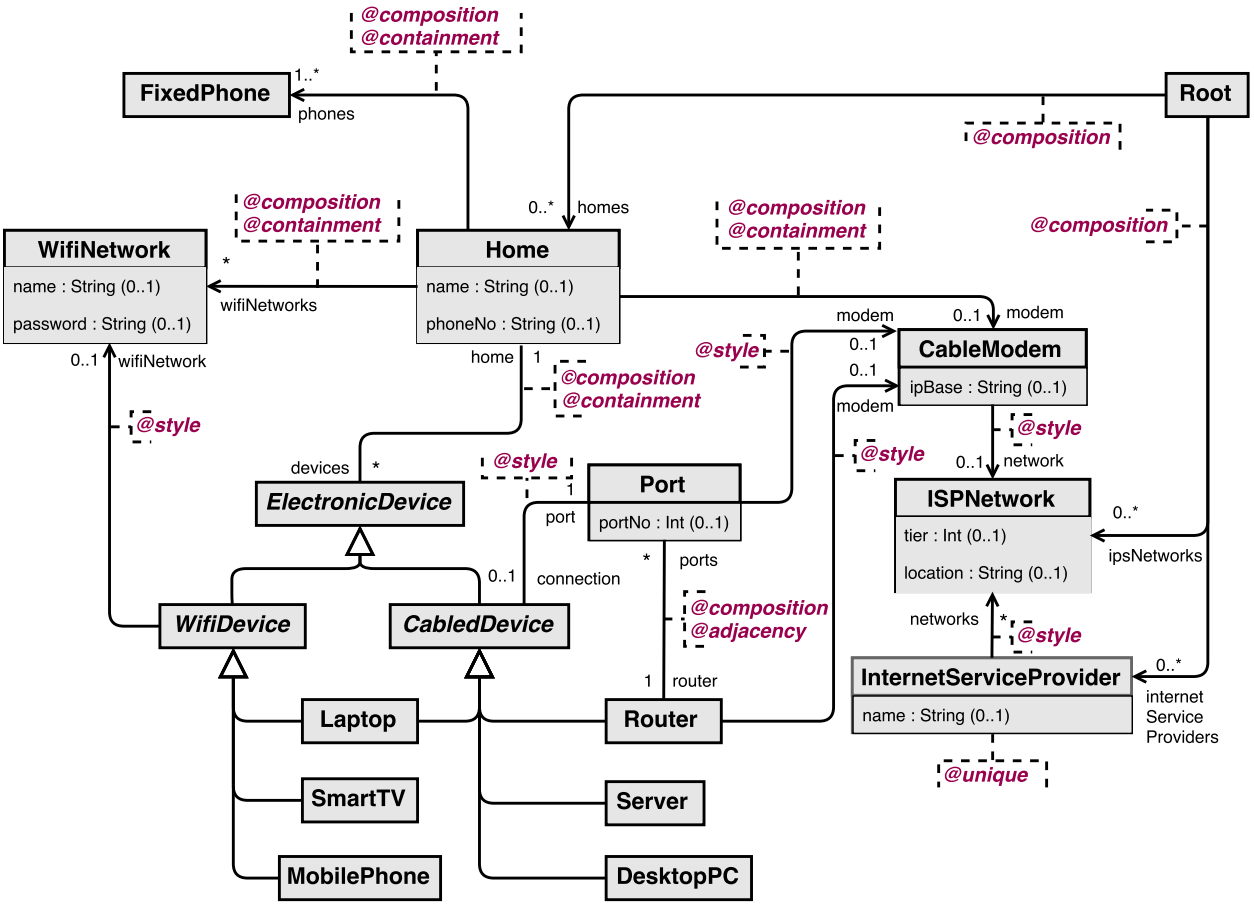


Figure 3.22: State of the abstract syntax to the running example, right before the generation of the graphical environment.

Finally, since port1 and port2 have been placed below router1, top and bottom adjacency relationships between these objects have been automatically inferred. We will see later in this section why port1 and port2 are not contained in home1.

At this point of the process, we have the desired abstract syntax represented by a meta-model, which is annotated with a series of graphical properties, like the one in Figure 3.22. With the only aid of the legend and the technology-neutral meta-model, the system is able to produce a full-fledged editor for the concrete syntax.

3.5.1.1 Resolution of conflicts in spatial relationships

Presumably, multiple spatial relationships converging on the same objects may be found as fragments are added; moreover, some spatial relationships might appear in some fragments

but not in others. The heuristics applied in these cases are illustrated next, showing small excerpts of fragments and the meta-models inferred from them. While the default behavior of some heuristics can be configured, in all cases, the ME can modify by hand the obtained result if it does not fit the domain.

Avoiding redundancy Because the system tries to create the minimum set of references in the meta-model needed to represent all spatial relationships discovered in an imported fragment, it takes advantage of the transitivity of composition relations to decide which ones encode redundant information and can be removed.

Figure 3.23 illustrates the situations where redundant relations can be safely removed. In case a), A contains objects of type B and C, which in their turn overlap. In the generated meta-model to the right, we create a composition reference from A to B due to the containment, and another from B to C due to the overlapping and the fact that the B object is bigger. However, although the C object is also contained in A, we do not add a composition reference between classes A and C as it would be redundant. The situation in cases b) and c) is similar. Actually, case b) occurs in the running example, as the Home object in Figure 3.13 contains a Router with adjacent Ports. In this case, a composition reference is created between Home and Router, and between Router and Port, but not between Home and Port

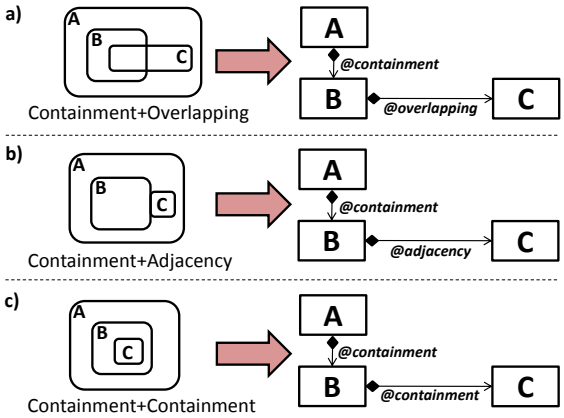


Figure 3.23: Avoiding the creation of redundant references.

Multiple spatial relationships Fragments where a set of objects participate in more than one spatial relationship, may lead to alternative ways to arrange the composition relations in the inferred meta-model. Figure 3.24 illustrates the possible scenarios.

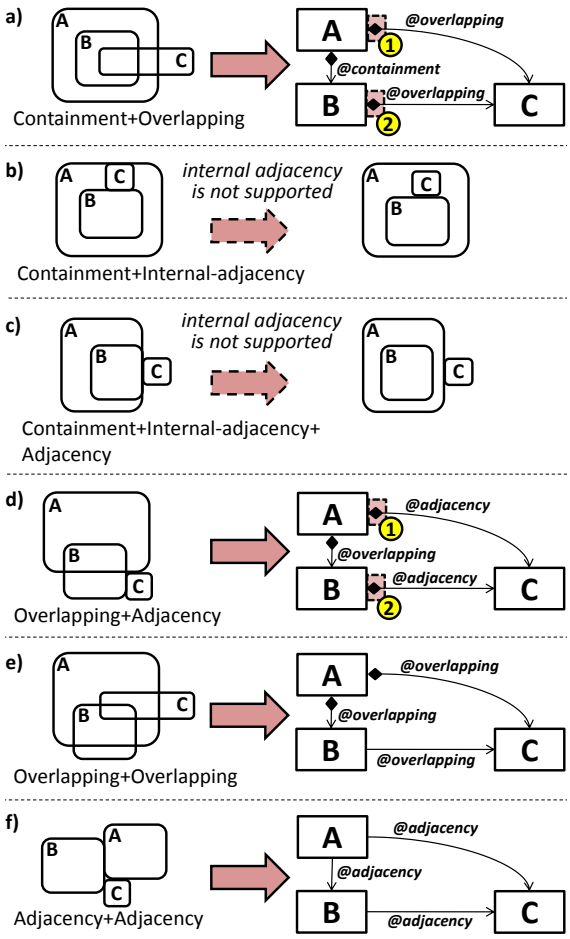


Figure 3.24: Handling multiple spatial relationships converging on the same objects.

In scenario a), containment and overlapping relationships arise for an object of type B. In particular, the B object is contained in an A object and overlaps with a C object, but in contrast to case a) in Figure 3.23, the A object does not contain the C object but both overlap. As a result, the inferred meta-model has two references to represent the different overlappings of C, and the question is which one of them should be a composition, and which one should not. Declaring that a reference is a composition implies a stronger relation between the related classes, as it establishes a "container-containee" dependency between them.

Due to the semantics of composition, both references cannot be compositions because that would not allow an object of type C to be contained in both references at the same type, which is the case shown in the figure (the C object overlaps with objects of type A and B simultaneously). Removing one of the references would not capture this case either. Hence, both references are needed, and the modelling expert should decide whether the composition

corresponds to the reference defined either by class A (option 1) or class B (option 2). The possibility to always use the container (class A) or the containee (class B) as holder of the composition can be configured by default in the system.

In scenario b), objects of types B and C are contained in an object of type A, and moreover, the C object is adjacent to both A and B objects. However, the adjacency of objects A and C is internal. Our system does not currently support this kind of adjacency (see the definition of the adjacency predicate), which is just interpreted as containment, as shown in the fragment to its right. Therefore, this scenario is treated like case b) in Figure 3.23. Similarly, in scenario c) of Figure 3.24, the internal adjacency between objects of types A and B is interpreted as containment, and hence, the fragments shown to the left and to the right are equivalent.

In scenario d), objects of types A and B overlap and are adjacent to a C object. Assuming that a composition reference is created from class A to B (because the A object is bigger than the B object), the question is again which one of the two references inferred from the adjacency relationships should be a composition. As in the case of scenario a), the default behavior can be customized.

In scenario e), three objects overlap with each other. Since the A object is bigger, in the inferred meta-model, we follow the heuristic to make the references stemming from class A compositions, while the overlapping between B and C objects is represented using a regular reference.

Finally, in scenario f), three or more objects may be adjacent to each other. This is represented via non-composition references in the meta-model. Moreover, since the objects of types A and B have the same size, there is the option to make the reference between classes A and B bidirectional. The other two references follow the heuristic to define them on the class of the bigger object (A and B respectively).

Optional spatial relationships Special cases also arise when some spatial relationship is present in some *examples* but not in others, meaning that such relationship is actually optional. This may imply reorganizing the composition references in the inferred meta-model, as Figure 3.25 shows.

In case a) of Figure 3.25, there is a first fragment where the B and C objects overlap and are contained in an A object. Then, in a second fragment, a C object is inside an A object

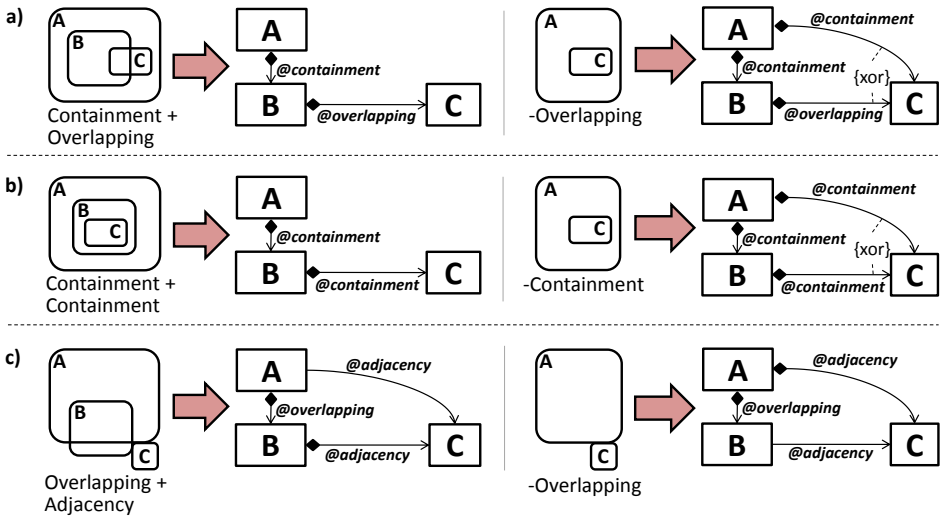


Figure 3.25: Handling optionality of spatial relationships.

without overlapping with any B object. The meta-model obtained from the first fragment does not suffice to represent the containment relationship in the second fragment, since class A does not define a container reference for C objects. Hence, the meta-model is extended with a new composition from A to C, together with a xor constraint to indicate that C objects can be contained either in A objects (due to the containment relationship in the second fragment), or in B objects (due to the overlapping relationship in the first fragment).

Similarly, in case b), C objects can be placed either inside of B or A objects. This is represented by two compositions from classes A and B to C, and a xor constraint.

Finally, in case c), the first fragment includes overlapping objects of types A and B, which in addition are adjacent to a C object. In this case, the modelling expert has selected the second option in case d) of Figure 3.24 to infer the meta-model, and hence, the reference from B to C is a composition. Then, in the second fragment, there is a C object that is not adjacent to any B object. Therefore, the composition is moved from the reference between B and C, to the reference between A and C. This change would not be necessary if the modelling expert had selected this option when processing the first fragment.

3.5.2 Environment generation

The DSL editor generation (label 7 in Figure 3.17), is provided in this framework as an extensible feature. We use as a basis the graphical environment generator proposed in [39], a work that proposes a neutral, platform-independent model for the generation of visual

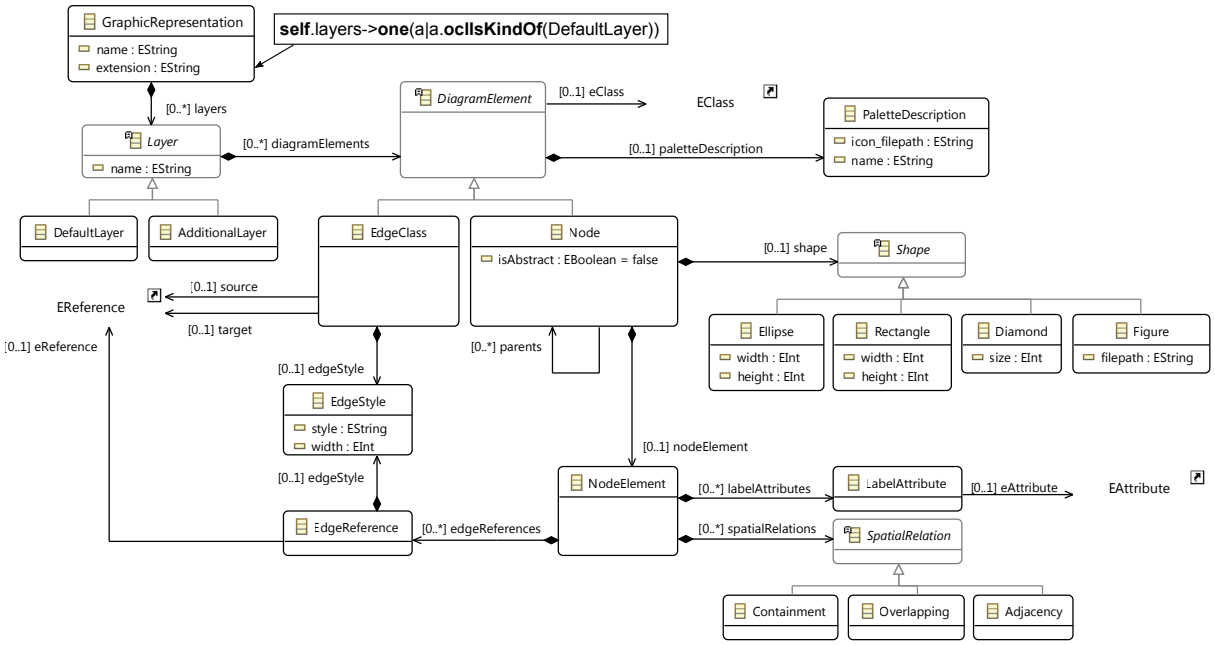


Figure 3.26: Excerpt of the *GraphicRepresentation* meta-model.

editors.

Figure 3.26 shows an excerpt of the neutral meta-model proposed in [39] to represent graphical concrete syntaxes, which is called *GraphicRepresentation*. The concrete syntax information induced from fragments is converted into this intermediate meta-model to be independent from the target technology, but also, to be able to refine this information, e.g., by specifying palette information, organize elements in layers, or select labels for nodes.

A graphical representation in the *GraphicRepresentation* meta-model is organized into layers (abstract class Layer). There is one DefaultLayer where all diagram elements belong by default (as shown by the OCL invariant attached to class GraphicRepresentation), and zero or more AdditionalLayers.

DiagramElements define the graphical representation of the objects of a certain meta-model EClass, and can be visualized either as nodes (class Node) or edges (class EdgeClass). In both cases, they may hold a PaletteDescription with information on how the element is to be shown in the editor palette. Nodes may be represented as geometrical shapes (Ellipse, Rectangle, etc.) or as image figures (class Figure). They can display a label either inside or outside the node, being possible to configure its font style (class LabelAttribute). Moreover, some nodes may need to be displayed in a relative position with respect to other nodes in the diagram, like being

adjacent to (class Adjacency) or being contained in (class Containment) other nodes. On the other hand, as abovementioned, classes can also be visualized as edges using class EdgeClass. In such a case, it is possible to configure their style (class EdgeStyle) and the references of the class acting as source and target of the edge. Regarding the representation of EReferences, they can be visualized as links by means of the class EdgeReference, and can define a style and decorators (omitted in the figure).

The *GraphicRepresentation* meta-model also enables the reuse of graphical property definitions by means of relation parents in class Node, so that graphical properties defined for a node are inherited by its children nodes. If a node is only being used as a placeholder for reusable properties but not for drawing, then it should be marked as abstract.

The generation of the modelling environment requires establishing a correspondence between the abstract syntax meta-model of the DSL and the concrete syntax meta-model in Figure 3.26. Classes in the domain meta-model can be represented either as nodes (class Node) or as edges (class EdgeClass), and are referred to through the reference DiagramElement.eClass. References in the domain meta-model are mapped into EdgeReferences, and their concrete syntax annotations are mapped into an EdgeStyle. In addition, if a reference is annotated with @containment, @adjacency or @overlapping, then it gets assigned a Containment, Adjacency or Overlapping object, respectively. All created graphical elements are included in the DefaultLayer and receive a PaletteDescription.

The quality of the resulting editor is subject to the accuracy of the fragments and examples provided by the DE, to the skills of the ME, and finally to the power of the generator that is employed for building the environment. The validation of the deliverable tool is expected to be carried out by the DE, who shall inspect the produced software in order to assess its validity, and report deficiencies to be corrected. We distinguish three scenarios for failure, each one of them solved in a different way:

- **The editor is incomplete.** If there are node types or relationships missing in the DSL editor, most probably that means more fragments are needed for completing the abstract syntax. In this case, it is suggested that new fragments are provided, aiming to cover the missing features the DE has signaled.

- **The concrete syntax does not match the aesthetics of the sketches.** When some node or connector does not have the same look than those employed in the original drawings, it should be checked whether the target platform actually supports the graphical features

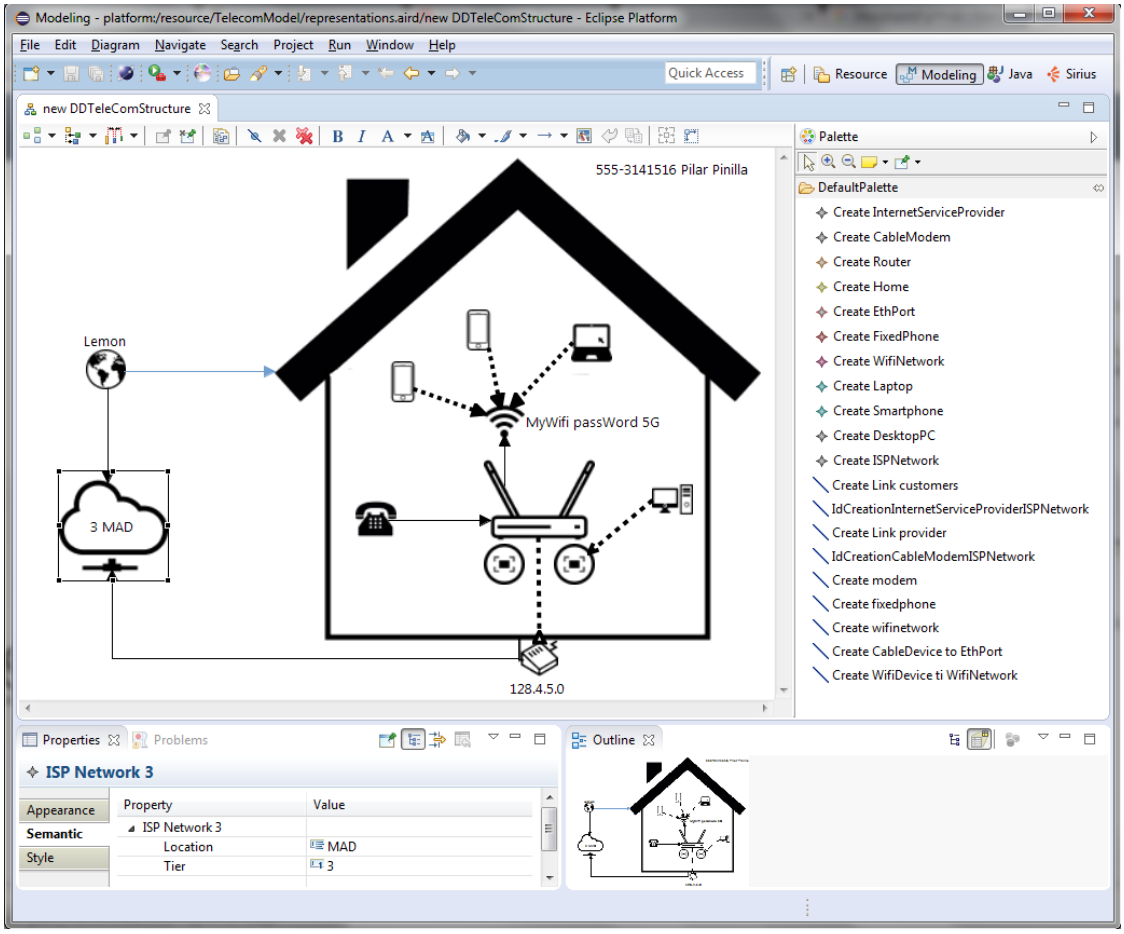


Figure 3.27: Resulting DSVL editor from the running example.

covered by our framework. If it does, then the parameters of annotations `@style` and the legend folder should be checked and repeat the generation process.

- **The editor is semantically incorrect.** Errors in the abstract syntax might cause the unexpected functioning of the editor, e.g., objects that can be placed or connected to some other objects that they should not or unexpected attributes. This case demands the ME to examine the meta-model to assess changes to be applied on the meta-model.

Figure 3.27 shows the resulting editor to our running example, obtained from exporting the abstract syntax from Figure 3.22 together with the graphic properties derived from examples. It can be noticed that the concrete syntax of the editor includes the same elements added in the fragments with the exact same graphic representation of nodes and spatial relationships amongst them. Notice, for instance, that the Home contains a series of ElectronicDevices, or that there is an adjacency relationship between Ports and the Router. A palette has been created

including all the object types and available connections between them, also preserving the edge style utilized in the fragments, and finally, induced attributes can be added to objects in the properties view.

In conclusion, in this chapter we have presented a process for developing DSML visual editors by example, in which a Domain Expert provides sketches of the desired models, and these are transformed into processable fragments which automatically build a meta-model. A Modelling Expert is in charge of ensuring that the abstract syntax is being built correctly, and is enabled to perform modifications over it. This iterative process results in a visual DSML editor mimicing the graphical properties used by the Domain Expert for the sketches.

4

Meta-Model Validation and Verification of DSMLs

This chapter presents a framework for meta-model testing which complements the proposed *bottom-up* process introduced in Chapter 3. The aim is to have means to Validate and Verificate (V&V) the quality and adequacy to the domain of a DSML under construction. This is accomplished with three different languages: one for unit-testing (Section 4.2), one for specification-based testing (Section 4.3) and one for reverse testing (4.4).

There are scarce tools and methods supporting DSML Validation and Verification (V&V) [19], which are essential activities for the proper engineering of meta-models. More specifically in our framework, there is an actual need for establishing well-defined means for the Domain Expert to ascertain the fidelity of the language under development to the original requirements, as well as ways for the Modelling Expert to formulate assertions to evaluate the quality and precision of the current abstract syntax.

In order to fill this gap, three complementary features to the system, in the form of V&V languages are proposed in this chapter:

- The first one, named *mmUnit*, has similar philosophy to the *xUnit* [15] framework, as it enables the definition of meta-model unit test suites comprising model fragments and assertions on their (in-)correctness.
- The second one (*mmSpec*) is directed to express and verify expected properties of a meta-model, including domain and design properties, quality criteria and platform-specific requirements.

- The third one (*mmXtens*) aims for the automatic generation of example models that take into account both the abstract and concrete syntax of a DSL. 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.

This chapter is partially based on [71, 72, 73, 74], although certain details have been extended or modified for reflecting later additions and improvements performed over the originally published work.

4.1 Overview

Figure 4.1 outlines the proposed approach to V&V. It comprises our three complementary approaches and languages for meta-model V&V: *mmUnit*, for unit testing; *mmSpec*, for meta-model property specifications; and *mmXtens*, for the generation of example models.

The first approach, realized in the *mmUnit* language, allows performing unit testing on meta-models with respect to valid and invalid test models, likely provided by the Domain Expert in the form of graphical sketches (label 1 in Figure 4.1) in the exact same way it is explained in Chapter 3, relying on the core system features for their transformation into text fragments (label 2).

mmUnit incorporates an assertion language for describing expected errors in test models and make explicit why a certain test model is incorrect. It is the Modelling Expert in this case the one in charge of formulating those assertions based on the fragments provided by the Domain Expert (label 3) and his own knowledge on the abstract syntax. In the end, the fragment and the attached assertions are evaluated to check the fragment conformance with respect to the meta-model under test (label 4), producing the system the pertinent test results.

Note that in EMF, meta-models can be tested using the generic JUnit framework for unit Java code. However, unit testing of incorrect models is problematic because EMF models need to conform to their meta-models, and hence cannot include erroneous features. Moreover, JUnit assertions are suitable for Java code (e.g., to assess whether the result of an operation is null), while for meta-model testing, specific assertions to indicate expected

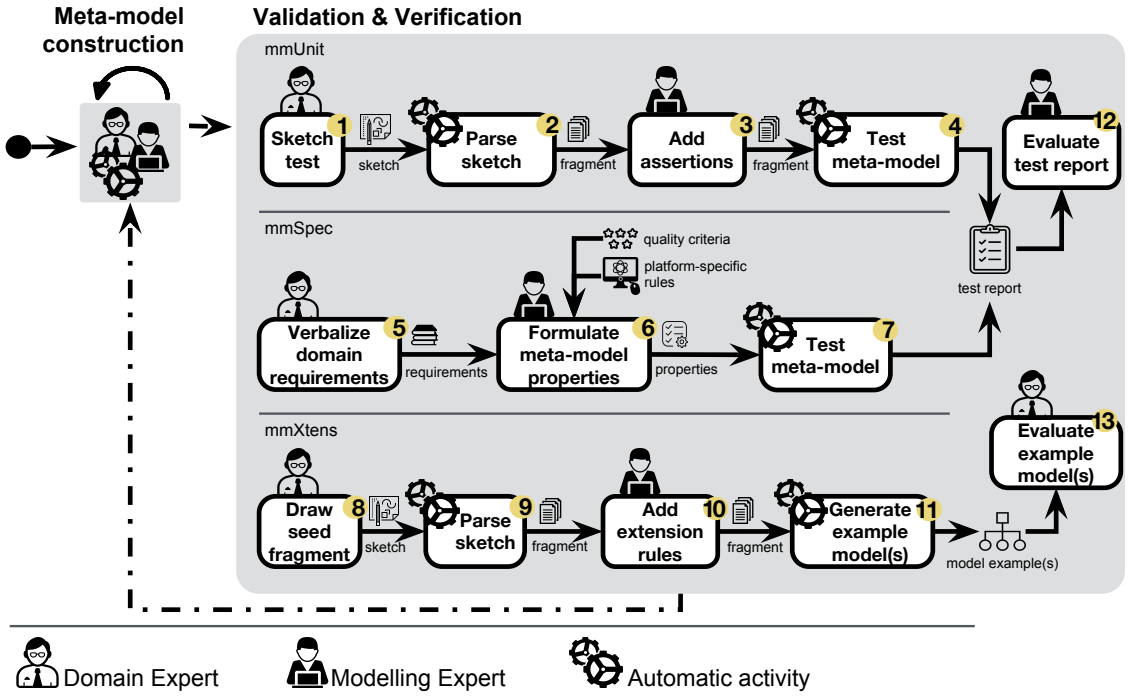


Figure 4.1: Overview of the framework for meta-model V&V.

conformities/disconformities might be appropriate. Section 4.2 will present all details of this language, whereas Section 6.2.1 features an evaluation with respect to using EMF for unit testing.

Secondly, the V&V of meta-model properties is approached. These properties can emerge from domain requirements elicited by domain experts, quality criteria adapted from object-oriented metrics [13], conceptual schema quality rules [3], naming style conventions [5], and platform-specific rules [95]. While properties can come either from the Domain (label 5) or the Modelling Expert (input of the activity labelled 6), they will necessarily be encoded by the latter (main activity in label 6), and run over the meta-model under development (label 7).

Thus, a DSL, named *mmSpec*, has been developed to describe meta-model properties in a compact and meaningful way. For this purpose, it includes high-level primitives for recurrent meta-model checks, like the existence of a navigation path between two given classes. The features of this language will be explained in Section 4.3. Although we could employ OCL to specify meta-model properties, it was appreciated in the realization of this work that the resulting expressions may get cumbersome, for which a DSL able to express these common properties was created. A comparison between *mmSpec* and OCL is given in Section 6.2.2.

Finally, the goal of the third example-based validation approach, called *mmXtens* and presented in Section 4.4, is to automatically produce interesting example models that can be easily inspected (even by non-meta-modelling experts) to validate the correctness of a meta-model, or counterexamples signalling meta-model flaws.

The system is able to produce the output example models either in text format, or, more interestingly, in their visual syntax, if previously specified. The arrangement of the different graphical elements in the produced visualization preserves the one in the seed model (if it was provided), and moreover, it may take into account domain-specific layout rules regarding adjacency, containment and (non-)overlapping of graphical elements.

Occasionally, the *mmXtens* solver might be unable to produce valid examples. In that event, the Modelling Expert would be committed to exploratorily find out whether this happens due to inconsistencies in the abstract syntax, or due to the use of a search space too narrow. In that event, the user can parametrize the number of objects and relationships that an output model is expected to include.

Normally, at the finalization of any of these three approaches, the delivered output shall be reviewed either by the Modelling Expert (*mmUnit* and *mmSpec*) or by the Domain Expert (*mmXtens*), although deciding whether or not these results motivate a change in the abstract syntax before starting the next iteration of the meta-model construction cycle, relies chiefly on the former. These final activities are referred to in labels 12 and 13, respectively.

4.2 mmUnit: Meta-model Unit Testing

In the approach to unit-test a DSML here introduced, 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 (AnnotationLabels in our Sketch Meta-model from Figure 3.6) that identify incorrect or missing elements. Sketches are imported into the system and used to evaluate the current meta-model version. Thus, this approach involves domain experts in the validation of the DSML, which is vital to build correct, useful DSMLs [50].

Internally, sketched fragments are parsed and converted into test cases expressed with the defined textual DSL for unit testing. Annotations indicating errors in the provided example sketches are translated into assertions in the created textual test cases. The Modelling

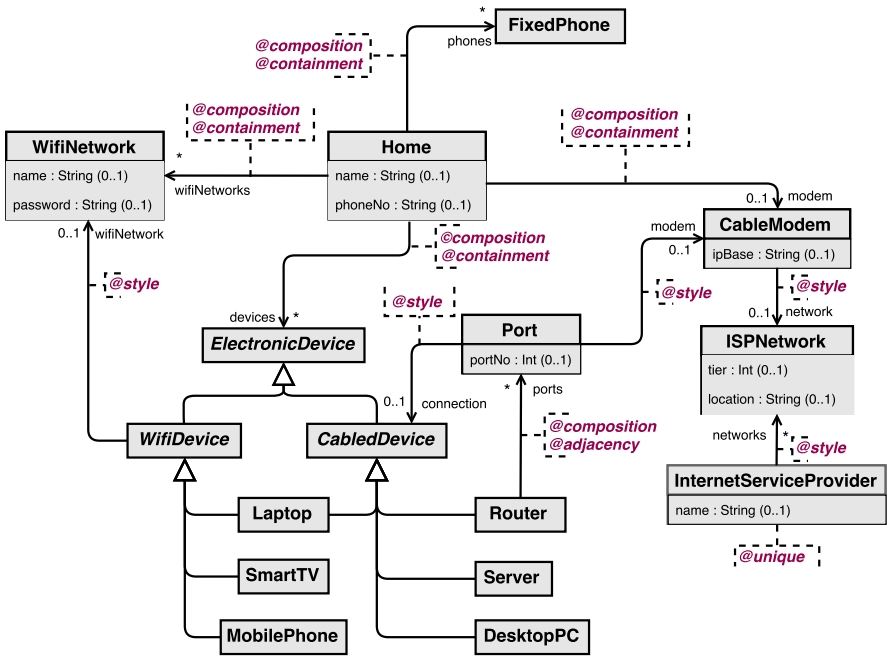


Figure 4.2: Resulting abstract syntax for the running example.

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 [16].

As an example, assume that, after a series of fragment additions and meta-model refactorings, we have come to the abstract syntax shown in Figure 4.2 of our running example.

At this point, we would be interested in testing whether, at this stage of the meta-model development process, a given model fragment would fit our meta-model under construction. Figure 4.3 shows a test case for the running example. Lines 1-26 in the listing to the left have been automatically derived from the sketch to the right. Line 1 indicates that the test includes a model fragment and therefore may lack mandatory elements, but still be assessed valid provided it satisfies the defined assertions. Lines 2–26 describe the fragment object configuration using a textual notation. Lines 28–32 include assertions describing some of the reasons why the example is invalid, expressed in the proposed DSL to define test cases, named *mmUnit*.

Each test case is made of a configuration of objects, and in case the configuration is

```
1   fragment Device2PortConnection {
2     server1 : Server {
3       ref port = port2
4     }
5
6     desktopPC1 : DesktopPC{
7       ref port = port3
8     }
9
10    laptop1 : Laptop{
11      ref port = port4
12    }
13
14    port1 : Port{ }
15    port2 : Port{ }
16    port3 : Port{ }
17    port4 : Port{ }
18
19    router1 : Router{
20      attr name = "myRou"
21
22      @adjacency
23      @composition
24      ref ports = port1, port2,
25            port3, port4
26    }
27
28    fails at least because:
29      unknown attribute Router.name,
30      missing attribute isWifi from router1,
31      missing composition from some Home to
              router1,
32      unknown type of server1
33  }
```

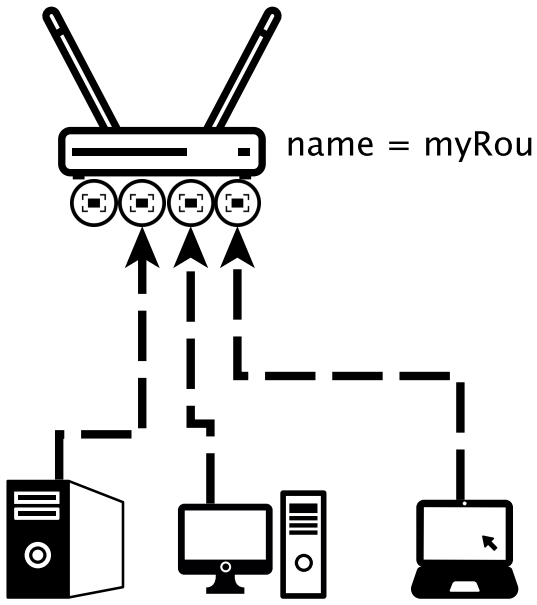


Figure 4.3: mmUnit test case for a sketch.

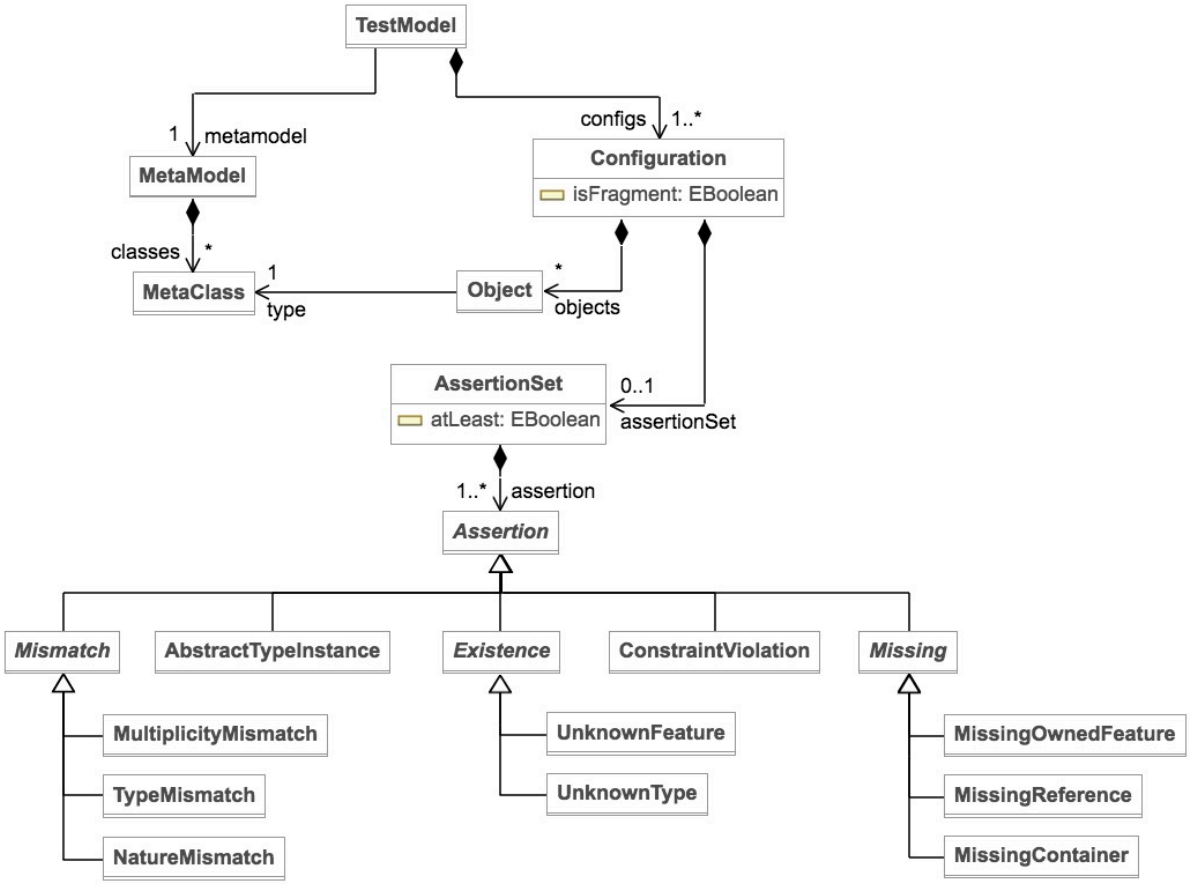


Figure 4.4: Excerpt of the *mmUnit* meta-model, including the hierarchy of assertions.

deemed invalid, a set of assertions stating why it is incorrect. In order to allow building more intensional tests, for the structural part, both *examples* of full-fledged models and model *fragments* are supported. 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 4.4 shows an excerpt of *mmUnit*'s meta-model. It contains elements to explicitly reference model and meta-model elements in the format of our own meta-model syntax, 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.

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, two possibilities are contemplated: 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, an appropriate OCL invariant is generated. This assertion kind points to violations of such annotations. The list of supported annotations is detailed in Appendix A.

Supported by the same mechanism that imports fragments for meta-model development, test fragments consist of a model fragment (or example) coming from a sketch that can be enriched with the assertions in the enumeration above. Moreover, assertions can be inspected in two ways, controlled by the keywords 'fails' and 'fails at least'. While the former is 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.

Syntactically, all the fragments/examples aimed to *mmUnit* evaluation present the following structure:

['example' | 'fragment'] <name> ':'
    <configuration of objects> ['fails' ('at least')? 'because']
        <list of assertions>

In this way, the assessment of the four assertions in Figure 4.3 is as follows: the assertion in line 29 asserts the erroneous inclusion of a name attribute associated to a Router type element, to a positive output (the attribute hasn't been added to the meta-model so far). Also regarding this same object, line 30 tests and matches the absence of a Boolean attribute isWifi. Thirdly, in the assertion from line 31 it is outlined the lack of a Home object containing the Router in the sketch, although in this case the assertion is evaluated false, since the Home.devices reference in the meta-model is not bidirectional. Finally, line 32 obtains a negative evaluation as well, because it requires an object to lack a type, which is indeed present in the meta-model (Server).

Eventually, the fragment fulfils 50% featured assertions and therefore, the test is not passed. At this point, the Modelling Expert is enabled to apply the necessary changes for the meta-model to meet the identified disconformities.

Additionally, sketches may include annotations indicating the fault in the case of incorrect configurations. These annotations become translated into equivalent assertions in the derived test case. The aim is to simplify their formulation taking them to a visual context closer to the one that handles the Domain Expert.

For instance, the word "missing" in the Sketch from Figure 4.5 to the right, yields a *missing reference* assertion in the generated test case that has been added to the sketch in order to signal the lack of a linking reference from the CableModem to the ISPNetwork in the example. According to the meta-model from Figure 4.2, this reference is not compulsory, which implies that the assertion evaluates to false.

The Modelling Expert can specify additional assertions in the textual test case generated from the sketch, or directly in the sketches. Moreover, we also enable the Domain Expert to customize the literals for expressing the *assertion* annotations.

Altogether, *mmUnit* permits the definition and evaluation of test cases integrated in the meta-modelling process, and is particularly well suited for example-driven DSML development.

```
1   fragment MissingModem2ISPConnection {
2     internetServiceProvider1 : InternetServiceProvider
            {
3       attr name = "lemon"
4       ref networks = ispNetwork1
5     }
6
7     ispNetwork1 : ISPNetwork{
8       attr tier = 3
9       attr location = "MAD"
10    }
11
12    home1 : Home{
13      attr name = "Damien Jurado"
14
15      @overlapping @containment
16      ref modem = cableModem1
17    }
18
19    cableModem1 : CableModem{
20      attr ipBase = "251.12.210.56"
21    }
22
23    fails because:
24      missing reference from cableModem1 to
            ispNetwork1
25  }
```

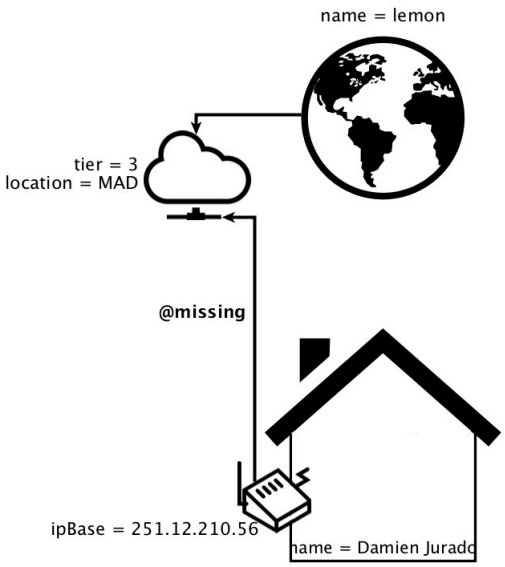


Figure 4.5: mmUnit test case with an assertion generated upon an annotation.

4.3 mmSpec: 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. 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.

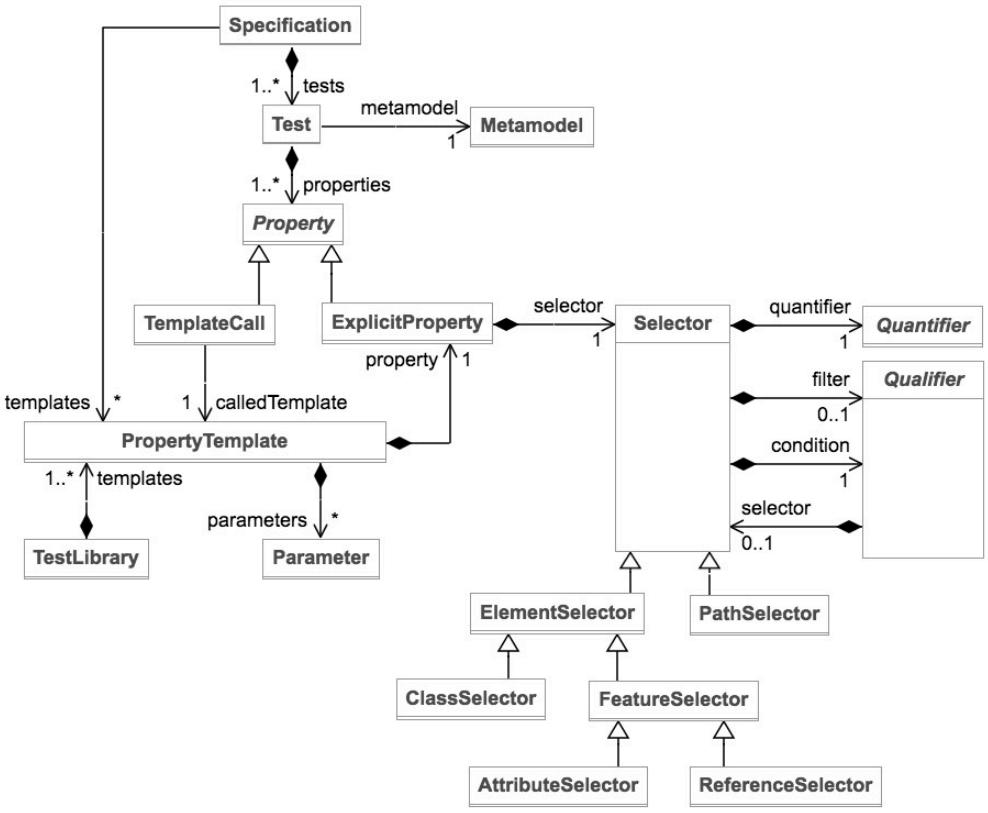


Figure 4.6: Excerpt of *mmSpec* meta-model, including the structure of properties.

To facilitate the specification and testing of expected meta-model properties, a DSL, called *mmSpec*, was created. This language aims to cover 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 4.6 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, enabling recursive checks. Table 4.1 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.

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, camelphrase	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.
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 4.1: Main qualifiers for selectors and conditions.

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 [77]. 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).

To illustrate this approach, we take advantage of *mmSpec* for checking the fulfillment of properties like the ones in Listing 4.1 in the meta-model of our running example, at its state in Figure 4.2. The Listing shows a specification with the domain requirements Rq1 to Rq3.

```
1 -- "domain"
2 test (metamodel "/networkReference/NetworkReference.mbup"){
3   -- "Rq1: Routers cannot have direct access to their Network, but through a Cable Modem"
4   every path{and{
5       from a class{name=Router},
6       to a class{name=ISPNetwork}}}
7   => through a class{name=CableModem}.
8
9   -- "Rq2: Every Home should contain at least one telephone"
10  a class {name=Home}
11  => collect{cont} [1,*) of a class {name{suffix} = synonym{Telephone}}.
12
13  -- "Rq3: We can navigate from any device to a Network"
14  every class{and{!abstract, sub-to a class{name = ElectronicDevice}}}
15  => reach a class{name{suffix} = Network}.
16 }
```

Listing 4.1: *mmSpec* test suite (domain requirements).

The meta-model under test is referenced in line 2. The property for Rq1 (lines 3–7) states that every path from class Router to class ISPNetwork should go through class CableModem. 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 CableModem, and succeeds, since the only existing path in the meta-model under test goes exactly through the class Modem before reaching ISPNetwork.

The property for Rq2 (lines 9–11) checks whether the meta-model grants the presence of at least one telephone, either fixed or mobile, in every Home. The property uses the collect primitive to calculate the composed cardinality, with modifier cont for containment, and uses the predefined primitives suffix and synonym for stating that we are looking for any class name ending with a word that means *telephone*. This second property fails to be fulfilled by the meta-model, since both fixed and mobile phones may or may not be included in a Home, being the cardinality of both containment references, phones and devices, set to *. The meta-model can be fixed in several ways to meet this requirement, the easiest being to set the minimum cardinality of Home.phones to 1.

Finally, the property for Rq3 (lines 13–15) first selects every non-abstract class inheriting from ElectronicDevice and checks whether they reach either a WifiNetwork or an ISPNetwork. The test fails this one too, as no path exists that connects a Server and DesktopPC to a network. Making Port.connection and Router.ports bidirectional as well, would solve this.

In addition to domain-specific properties, which normally need to be defined anew for each DSML, *mmSpec* allows the creation of libraries of reusable parameterized property templates. In the current implementation, a library with typical design quality criteria [30] and style guidelines [5] (see Appendix D) is provided. As an example, lines 1–4 in Listing ?? show an excerpt 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. Specifications can import libraries and reuse their templates as in lines 8 and 16 respectively. 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
6  ------------------------------------------------------------
7
8  import "/quality.mbm"
9
10 -- "quality" @warning
11 test (metamodel "/networkReference/NetworkReference.mbup"){
12   -- "Rq4: No class is included in two containers"
13   no class => reached-from{cont, jumps=[1,1]} 2 class.
14
15   -- "Rq5: No inheritance hierarchy has a 5-level or greater depth"
16   depthOfInheritanceTree(threshold=5).
17
18   -- "Rq6: Every class name is a noun, possibly qualified, written in upper camel-case"
19   every class => name = upper-camel-phrase{ends{noun}}.
20 }
21
22 -- "EMF" @warning
23 test (metamodel "/dataServices/DataServices.mbup"){
24   -- "Rq7: The meta-model needs to define a root class"
25   strictly 1 class => cont-root{absolute}.
26 }
```

Listing 4.2: *mmSpec* specifications (style guidelines, and quality and platform requirements).

Lines 10–19 in Listing 4.2 show a specification with further properties, accounting for requirements Rq4 to Rq6. The specification 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 13 states that no class can be directly contained (modifiers jumps and cont) in two classes. The property in line 16 calls the depthOfInheritanceTree template with 5 as threshold value. The property in line 19 checks that every class has a name in upper camel-case, the last part being a noun. The

meta-model in Figure 4.2 satisfies all these quality properties.

Lines 22-25 contain one property specific for EMF which checks that there is strictly one absolute root class. This property ensures the existence of a single non-abstract, uncontained meta-class from which every remaining concrete meta-class in the meta-model can be reached following containment associations. It is a compulsory requirement in specific meta-model implementations like those provided by EMF. In this case, the property fails because neither Home, InternetServiceProvider, or ISPNetwork are contained in any other class. This can be solved, for example, by adding a new meta-class that contains the three of them.

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. The possibility of automated checking is useful for regression testing, as the properties can be rechecked every time the meta-model changes.

4.4 mmXtens: Example-based Validation of Meta-models

In the two previously introduced languages, the Modelling Expert is the one best suited to evaluate the results of all the developed tests. However, the Domain Expert still has no way to individually evaluate the correctness of the DSL under development. Hence, considering the example-based nature of the proposed framework, it should allow the Domain Expert to build examples, that is, as independently as possible to the abstract syntax, and preferably in the same format originally employed for drawing the sketches, as in this way, the evaluation will be more approachable and accurate.

Hence, a third V&V component was designed for automatically producing instance models of the meta-model under development. These examples can be delivered either in a text format representation (more likely to be inspected by the Modelling Expert), or in graphical format (regularly for their validation from the Domain Expert side). The final purpose is detecting possible flaws on them and, in that event, performing refactorings over the abstract syntax (a task which would rely on the Modelling Expert).

Although this is a very typical way to validate meta-models, doing it manually tends to be wearisome, with users instinctively producing very basic examples that accommodate any preconception they may have on the language. Because of this, this work enables the automatic generation of examples.

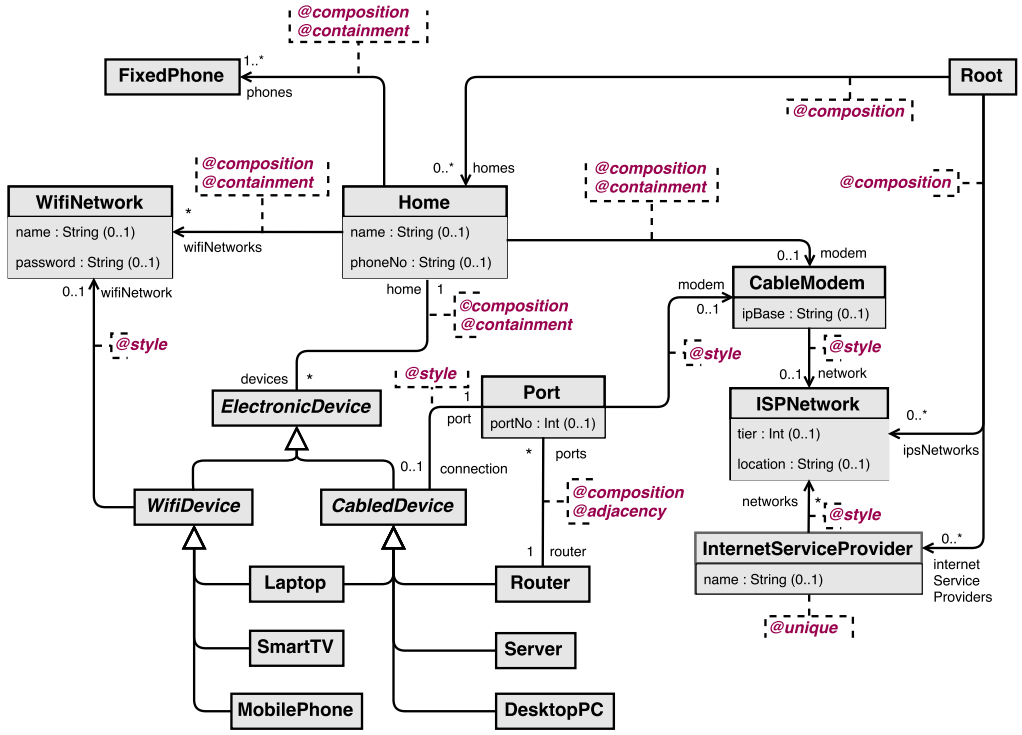


Figure 4.7: Running example resulting abstract syntax after changes made upon test results.

The process can be fine-tuned using a textual DSL called *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. Moreover, in order to produce the concrete syntax for the generated examples, in addition to the typical node-arch graphical representation, the same spatial relations detected at the Fragment import stage (see Section 3.5.1) are supported.

We illustrate a first approximation to *mmXtens* following our running example. Assume our meta-model presents the state from Figure 4.7 after evolving throughout the iterative process described in Chapter 3 and after the changes made upon the test results from Sections 4.2 and 4.3. At a sufficiently developed stage of its construction, we would be comprehensively interested in checking whether or not this abstract syntax can accept in valid models. We can take advantage of *mmXtens* for this purpose.

Figure 4.8 illustrates the testing approach proposed. In step 1, a seed model fragment is provided by the Domain Expert. This is optional, as it is always possible to start from an empty model. In step 2, *mmXtens* would be used to specify rules for extending the seed fragment. Although these rules might come from the Domain Expert, they are expected to

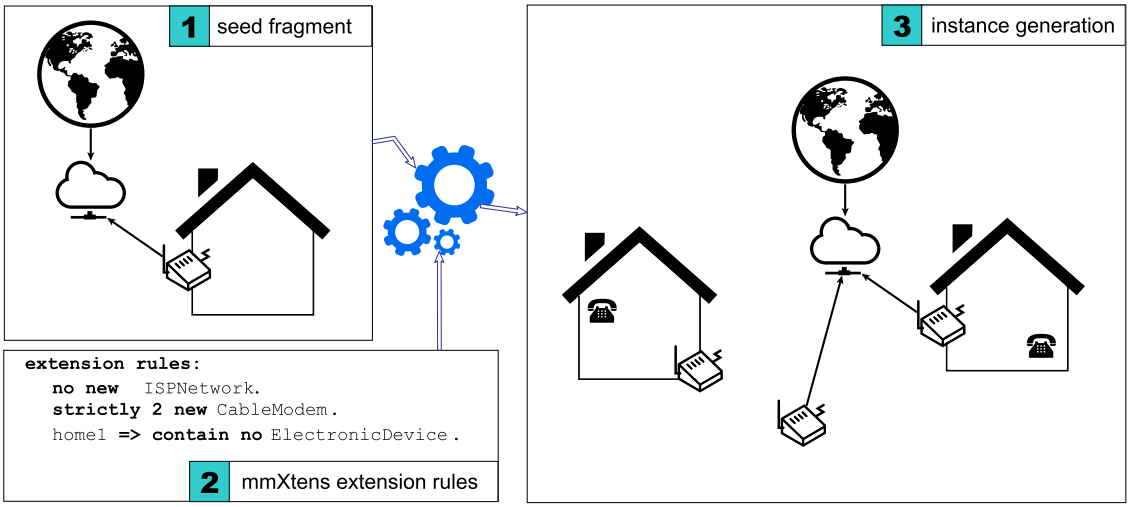


Figure 4.8: An example of instance generation with mmXtens.

be encoded by the Modelling Expert as he might be more familiar with their syntax. These rules encode properties that the generated model should meet. Then, in step 3, our engine translates the *mmXtens* specification into OCL constraints, and uses an OCL-based finder [67] to produce a model satisfying the specification and conformant to the meta-model under test. This produced model (if any exists that satisfies the extension rules) can be inspected to validate whether it complies with the requirements, the expectations about the meta-model, or the intuitions about the domain.

In the figure, the Domain Expert provided a seed model with an ISPNetwork which reaches the unique InternetSeviceProvider and, connected to the former, a single CableModem belonging to a Home node. The extension rules in step 2 are encoded by the Modelling Expert and express that the output model should add no new ISPNetwork, exactly two new CableModem, and that the Home in the seed fragment, named home1 in its text version, should contain no ElectronicDevice. Using this information, the system delivers the example model shown in step 3. This model satisfies all the extension rules, and all the spatial relationships annotated in the meta-model (which are containment and overlapping). From this generated model we identify two issues whose suitability to the domain needs to be discussed.

First of all, the generated example has an isolated Home (with a FixedPhone and a CableModem), that is, unconnected to an ISPNetwork. In this case, we assume the Domain Expert accepted this as an admissible model configuration. Secondly, there is a CableModem unattached to any Home node. This happens because in the meta-model in Figure 4.7, the reference between these two classes is unidirectional, and hence, the existence of CableModems

is not conditional on that of a Home. Rejected this model configuration by the Domain Expert, the Modelling Expert shall proceed to apply the necessary changes over the meta-model for avoiding its occurrence. In this case, making the reference bidirectional, setting the Home edge cardinality to 1, would solve the problem.

Notice that, despite being able to import/export models from/to their concrete syntax, *mmXtens* operates with fragments and examples represented in text format. Thus, for uniformity, the syntax of an *mmXtens* seed fragment is exactly the same as any other fragment in the system. Extension rules are simply added at the end of it, in the same style that *mmUnit* fragments were added an assertion code block.

Hence, *mmXtens* specifications have the structure shown in Listing 4.3 (lines 1–5). They 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 rules, 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 rules with conditions that the model to be generated should satisfy (line 4). These rules 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.

```
1  SPECIFICATION ::= [ positive | negative ]? example <name> {
2      SEED_FRAGMENT
3      extension rules:
4          EXTENSION_RULES
5  }
6
7  SEED_FRAGMENT ::= [ <object-id> : <type> {
8      [ <slot-id> = value ]*
9  } ]*
10
11 EXTENSION_RULES ::= [
12     <object-id> => CONDITIONER . |
13     QUANTIFIER [new]? <type> [FILTER]? [=> CONDITIONER ]? .
14 ]*
15
16 CONDITIONER ::=
17     [ reference | contain ] [ {via <refer-name>} ]? SELECTOR |
18     attr <att-name> = value
19
20 QUANTIFIER ::= <n>..<n> | strictly <n> | <n> | every | no | some
21
```

```
22 SELECTOR ::= <object-id> |
23     QUANTIFIER [ new ]? <type> [ FILTER ]?
```

Listing 4.3: Simplified grammar of *mmXtens* specifications.

Extension rules may refer either to specific objects in the seed fragment (line 12), or to arbitrary objects which can exist in the seed fragment or may need to be created new (line 13). In both cases, we can use a *CONDITIONER* stating required properties for the object. For example, the extension rule used in Fig. 4.8:

home1 => **contain no** ElectronicDevice.

requires the existing home1 object not to be added any ElectronicDevice type object when the seed fragment is extended. As we see in this example, a *CONDITIONER* may require an object to be - or not to be, in this case - contained in some other using the keyword **contain**. If we use the keyword **reference** instead, then, the reference should be an association. Optionally, we may specify a reference name using the keyword **via**. Finally, we can use a *SELECTOR* (lines 22-23) 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.

Alternatively, a *CONDITIONER* may define requirements on the attribute values of the selected object (line 18 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** and **not** (omitted in the listing for simplicity).

Rules can also require properties on sets of objects of a certain type, without referring explicitly to an existing object (line 13). In such a case, we use a *QUANTIFIER* to select the objects. Syntactically equal to *mmSpec*, 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.

For instance, valid object selections include **every new** ElectronicDevice (for all newly created ElectronicDevice objects, where ElectronicDevice is an abstract class), **some** Home, **no new** FixedPhone or **0..4** ISPNetwork.

In this latter rule 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.

Each *mmXtens* specification may have zero or more extension rules, all of which must be fulfilled. In addition, the system permits customizing:

- Minimum and maximum number of objects of each type to generate. In the example from Figure 4.8, the minimum was set to 0, and that explains why, for example, no Port elements were created. Normally, the maximum number of objects to create is small (e.g., 3), as bigger numbers might provoke a combinatorial explosion in the solver, resulting in unnaceptable runtimes.
- Minimum and maximum default reference cardinality. Alluding to the number of elements each association or composition reference ought to reach. In our example, this property was set to 0..50.
- Whether or not new attribute values for slots not considered in the seed fragment should be added to objects from the seed fragment. Set to false in the example.
- Whether or not new (non-containment) references should be added to objects from the seed fragment. Set to false in the example.
- Whether or not objects from the seed fragment should contain new objects. This property was set to true in our example, having the solver added a new FixedPhone to home1.
- Whether or not objects from the seed fragment should be reachable/contained by new objects. This property is accepted in the example, and hence, a new CableModem gets to reach the object ispNetwork1.

Figure 4.9 shows to the left a text version of the same visual seed fragment featured in Figure 4.8, with new extension rules appended at the end. The first one is asking for a fixed number of two Port objects, regardless of the unconstrained minimum set in the preferences. The second demands the solver to add some ElectronicDevice to home1, and this new device should reach a new CableModem, which virtually means demanding the creation of a new object of that kind too. Finally, the assertion no new ISPNetwork indicates the solver not to create elements of such a type apart from the one included in the seed fragment. We can see the produced example model to the right of the figure.

```
1   fragment SeedFragment {
2     internetServiceProvider1 : InternetServiceProvider {
3       ref networks = ISPNetwork1
4     }
5
6     ispNetwork1: ISPNetwork { }
7
8     home1 : Home {
9       @overlapping @composition
10      ref modem = cableModem1
11    }
12
13    cableModem1 : CableModem {
14      ref network = ispNetwork1
15    }
16
17    extension rules:
18      2 new Port.
19      home1 => contain some ElectronicDevice{reference a new
              CableModem}.
20      no new ISPNetwork.
21  }
```

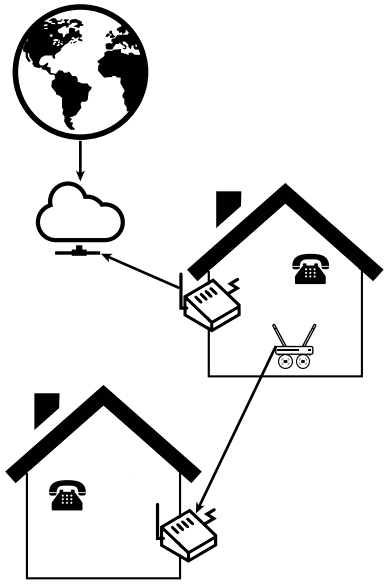


Figure 4.9: Text fragment with extension rules and its resolved graphical example.

From the output that the system produces, we find a very visible flaw in our design: Routers can be connected to whatever CableModem, not necessarily in the same Home. Once the Domain Expert rejects this model, the meta-model could be fixed enriching Router.modem with a restriction. Fortunately, our annotation library accounts one that solves it. Annotating Router.modem with @cycleWith(Router.home, Home.modem) forces every CableModem instance referenced by a Router to be also contained in the same Home. See Appendix A for further detail on the specifics of this annotation.

To sum up, the approach for the example-based validation of meta-models is enriched by means of a DSL, *mmXtens*, which permits describing seed fragments and extension rules to complete those examples with the aid of a model finder. The seed examples may come from informal drawings, which are parsed, being their spatial relations likewise represented, and layout is hence traced in the generated examples. The usefulness of the approach has been tested on a set of meta-models built by students, as explained in Section 6.2.6.

In this Chapter, the need for a complete environment for meta-model testing has been fulfilled. Particularly, by introducing three languages (*mmUnit*, *mmSpec* and *mmXtens*) which serve as mechanisms for testing a DSML from three perspectives: unit, specification-based and reverse testing. The approach has been placed in the overall example-based process

introduced in this PhD, although the three languages adapt to meta-models built using any other strategy.

5

Tool support

After having introduced the two main contributions from this PhD in Chapters 3 and 4, this Chapter goes through the tool that supports them technically. This is made through the same running example used in previous Chapters, and illustrated with snapshots from the original Eclipse plug-ins that implement the tool.

The process described in previous sections comes supported by a technical prototype offering coverage of all the mechanisms needed to build an editor for supporting the abstract and concrete syntax of a DSML (as described in Section 3), and of the techniques for meta-model testing (detailed in Section 4). The former are all comprised in the *metaBup* framework, while the latter are presented as a complementary suite to named *metaBest*.

The two technical contributions are implemented over EMF (Eclipse Modeling Framework), the most widespread solution for MDE development. However, their functionality has been enhanced so as to provide platform-independent development, i.e. not based on specific meta-model implementations, like Ecore. Moreover, the framework accepts (via Eclipse extension points) the addition of new fragment input formats, annotations, and meta-model importers, editing actions, and exporters.

The following sections go through these two tools and their relationship with the proposed methodology.

5.1 metaBup

Figure 5.1 shows the main software components of *metaBup*. They are separated into four big blocks according to their functionality: the **project manager** takes care of the general aspects and utilities involved in the DSML development life-cycle, including the legend, the meta-model update history and the available preferences for configuring customizable aspects of the process; the **fragment manager** deals with the import (from visual to text format) and - text - editing of fragment and example models; the **meta-model manager** holds all the abstract syntax related functionality, from its increment by fragment addition, to its export to a platform-specific format; finally, the **annotation manager** interfaces with the remaining components when they need to access the annotation catalogue.

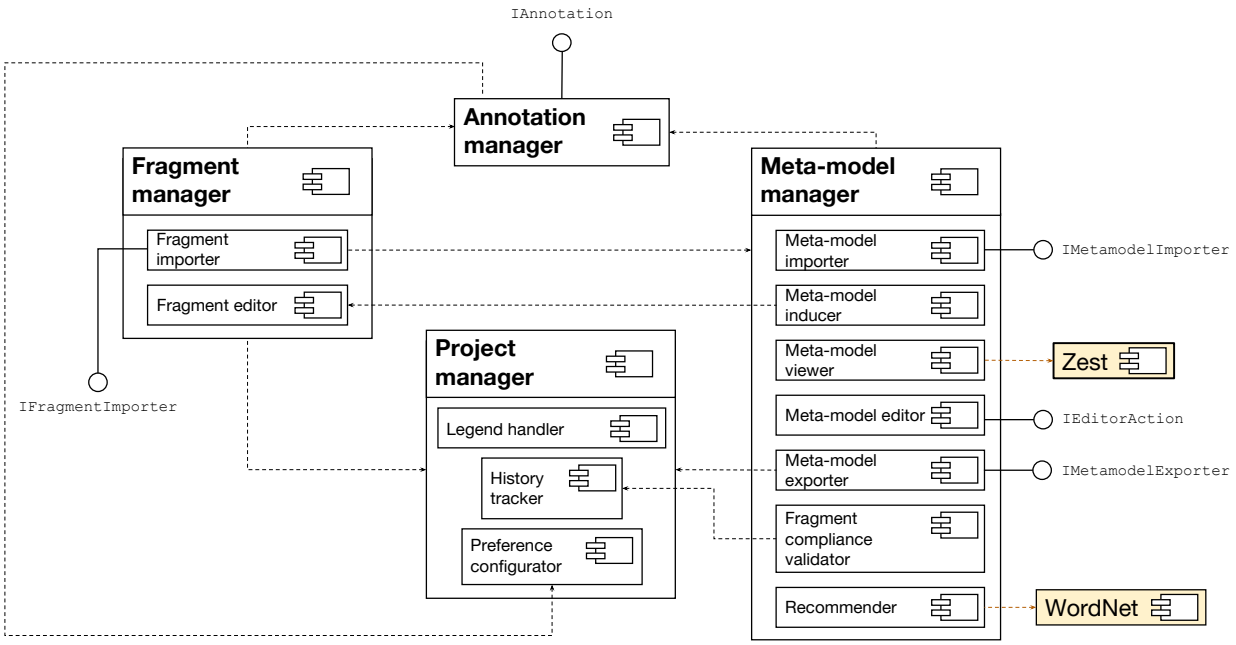


Figure 5.1: *metaBup* component diagram.

Reproducing the process proposed in Section 3 with *metaBup* involves the following steps:

Firstly, the approach needs an Eclipse project to be created, with the *metaBup* nature (see Figure 5.2). These projects, handled by the **project manager**, contain the following structure:

- **Meta-model file**. A `mbup` extension file containing the meta-model to be constructed. Initially containing no elements, it will grow as fragments are appended to the project.

- **Legend folder**. A folder containing the files used for drawing the elements included in the sketches. Unless a specific importer makes it otherwise, the elements in the folder shall receive the name that their abstraction in the meta-model is expected to receive. The *legend handler* tracks these files and gives programmatic access to them for those programmers wanting to contribute to the framework through any of its extension points.

- **Fragment folder**. A directory in which a copy of each fragment that has contributed to growing the meta-model is stored, with the extension `mbupf`. This is our implementation of a *history tracker*.

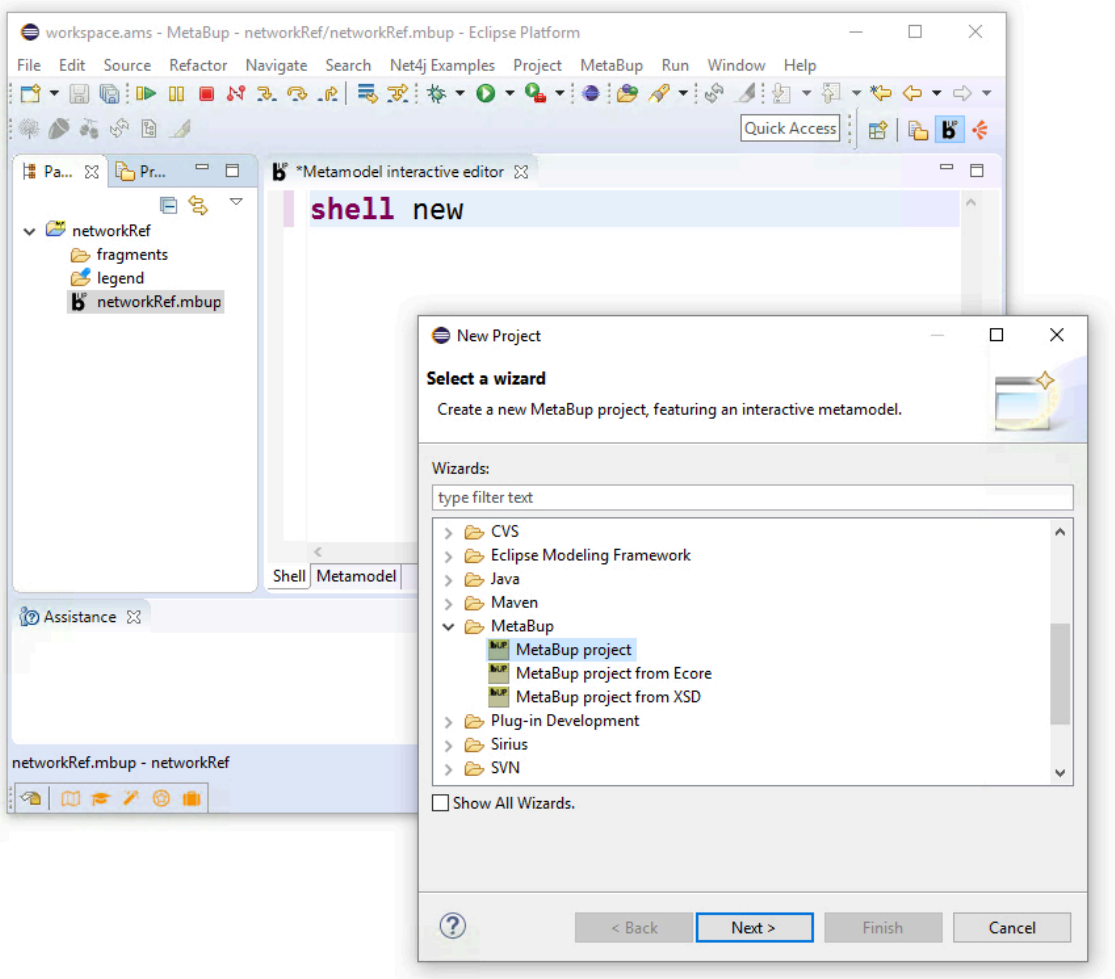


Figure 5.2: New *metaBup* project.

Several features in the environment, including annotations and extension points, can define properties, for which a *preference configurator* is enabled. The preferences can be set through

the preference window that Eclipse provides for these purposes.

Secondly, the **fragment manager** takes care of the whole fragment life cycle. Sketches are a concept that only exists outside the framework, as it specifically means drawings made using an external editor. These are provided as input to the **importer** of the fragment manager, which is able to process them and turn them into fragments (or examples). The tool gives programmatic access to the items in the *Legend folder*, in case that they need to be used for transforming the sketches into fragments.

Moreover, importers can interact with the meta-model manager, being granted access to the abstract syntax, as they need to explore its structure in order to produce a fragment that matches the already existent properties of the meta-model. The prototype provides *Dia* and *yED* importers with the processes and heuristics explained in Section 3.3.

Figure 5.3 shows a *yED* fragment to the right and its automatic derivation into text in the editor to the left. In the Eclipse browser, the used files for creating the sketch have been added to the legend folder.

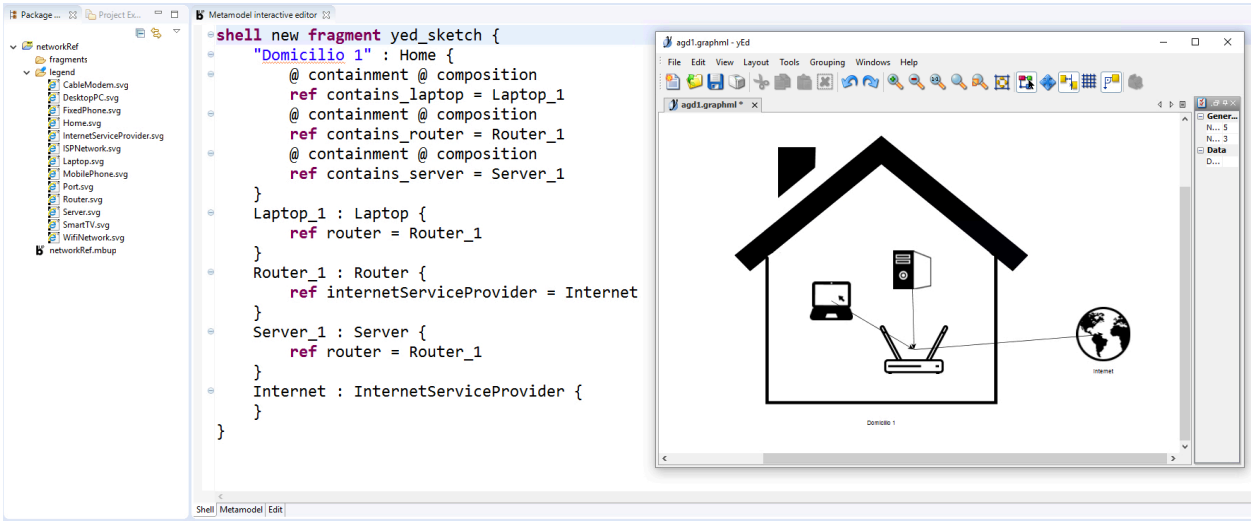


Figure 5.3: Graphic sketch imported and converted into text fragment.

The output of importers is an in-memory *metaBup* fragment (or example). These are automatically given a text syntax that is editable by users (normally the ME) in the *fragment editor*.

The **meta-model manager** is in charge of the abstract syntax and all its files and components involved during the development process. Meta-models can start either from scratch, or be

imported from files in other compatible formats. Meta-model *importers* are defined by means of the extension point made available in the framework. The prototype provides Ecore and XSD importers (refer back to Figure 5.2).

Thus, an editor like the one shown in Figures 5.2 and 5.3 is assigned for *mbup* files, and the meta-model is evolved with the aid of the *inducer*, which takes the fragment or the example present in the "shell" tab of the editor, and applies the logic described in Section 3.4 so as to increment it. The result is displayed graphically in the "metamodel" tab of the environment. This viewer is implemented with Zest. With each increment, the changes and refactorings introduced in Section 3.4.3 are suggested by the *recommender* view, which uses WordNet [77] for the calculation of syntactic assumptions like plural/singular name detection.

Since the meta-model is stored as an XMI model, it is possible to edit it with the tree editor EMF provides for such files, embedded in the third tab of the environment ("edit"). Moreover, developers are given the chance to add automatic refactoring actions to the environment that ease their work with the meta-model. For instance, the tool natively incorporates the insertion of a *root* class containing all the meta-classes in the meta-model, as this is a typical requirement in the EMF framework. Actions are automatically added to the editor's menu. The upper side of Figure 5.4 shows the meta-model inferred from adding the fragment in Figure 5.3, while the result of applying the root class refactoring and some recommendations is shown to the bottom.

A *compliance validator* ensures the fragments in the *fragment folder* are still consistent with the evolved state of the meta-model after each change or increment, signaling the elements in each fragment that do not match the current abstract syntax.

Furthermore, because annotations play a key role in the entire development process - either enriching fragments and meta-models, describing concrete syntax properties or triggering refactorings -, the **annotation manager** is connected to all the remaining components of the environment. Annotations can also define preferences (number, string or boolean variables) whose value is used for taking specific decisions during the meta-model induction process in what refers to how model elements transform and modify meta-model elements.

Meta-models can be exported either in their abstract or in their concrete syntax, any time in the development process. The result of the former case is normally a single file, while the latter is a DSML editor. In any case, an extension point is provided for implementing DSML *exporters* which are automatically added to the exporter section of the editor. An

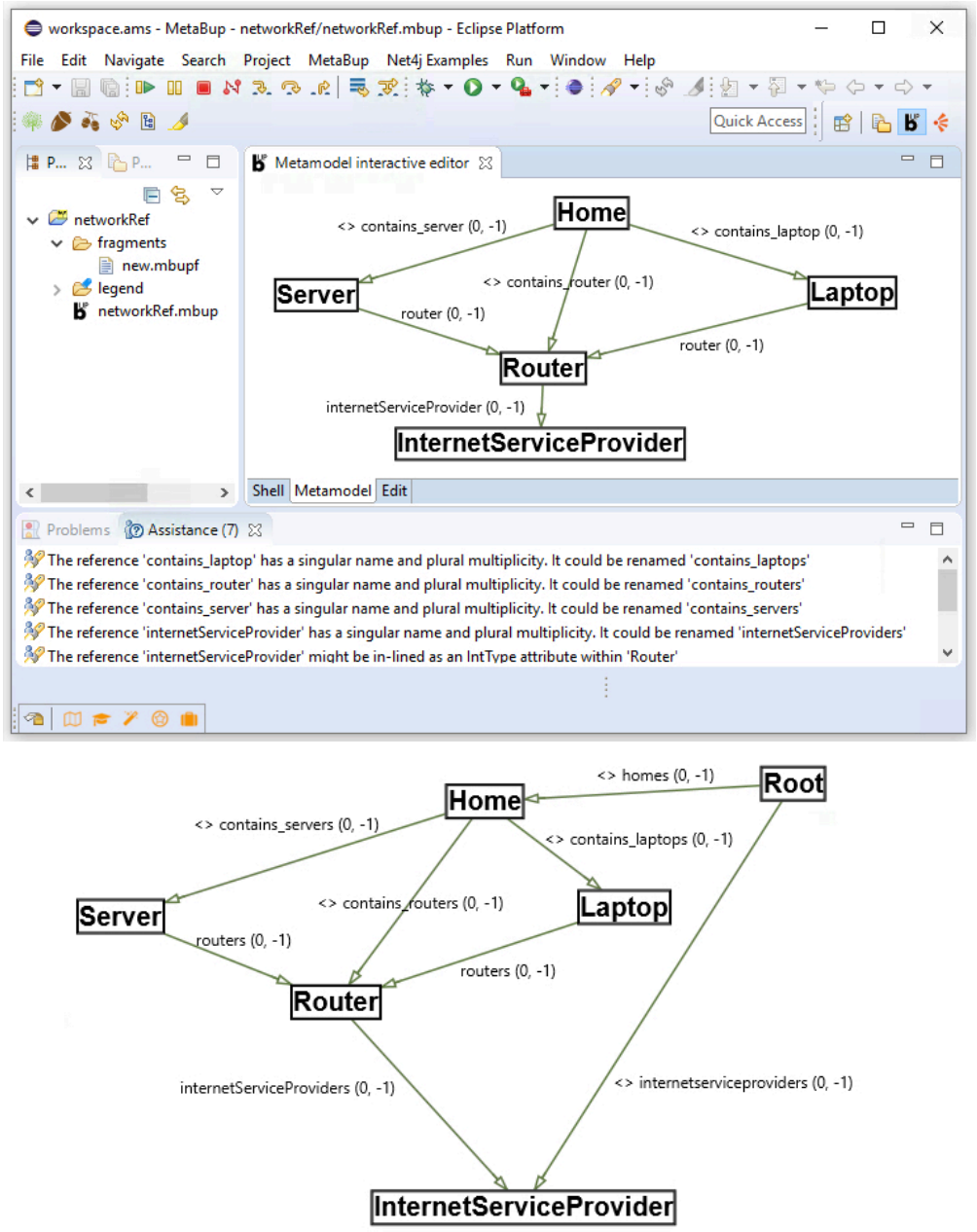


Figure 5.4: Meta-model before (up) and after (down) applying changes.

exporter shall normally need access to the legend handler for extracting the concrete syntax of objects; it can also, like any extension in the extension point set, define preferences; and finally, it is granted access to the fragment history, as one could be interested in representing a copy of the input fragments in the target platform.

This work provides an abstract syntax exporter for Ecore, and a concrete syntax exporter for Sirius. The latter is the feature that gives shape to the last part of the process described in this dissertation (DSVL generation). With this one-click option, the user can have a fully operating editor for the DSML that is being designed. Figure 5.5 shows the output of that extension, generated after applying the changes shown in Figure 5.4. As it can be noted, the resulting editor includes the same example that was contained in the original *metaBup* project (the one in Figure 5.3, from which the meta-model was inferred), preserving the original node representation, attribute values, spatial relationships and connections.

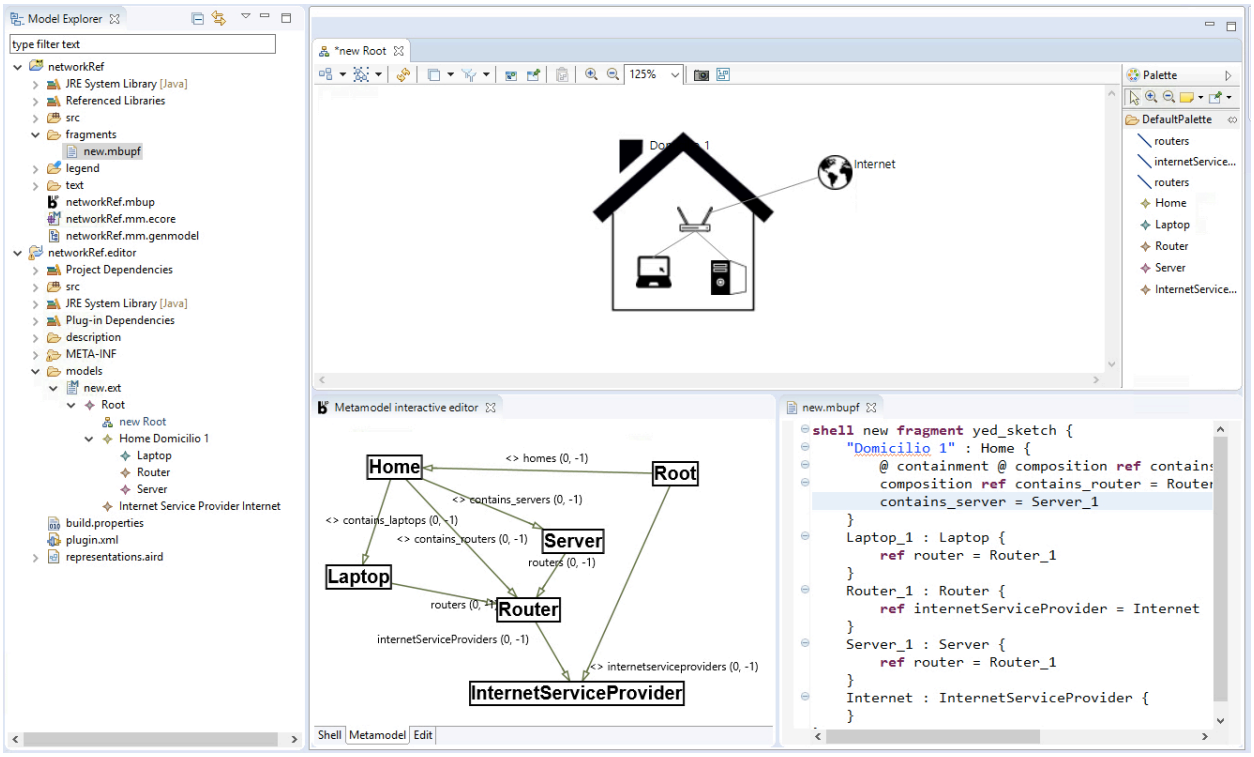


Figure 5.5: DSVL editor generated with *metaBup*.

Applying changes to the *metaBup* project shall refresh the layout and behavior of the editor, complying with the iterative character of the defined process.

5.2 metaBest

Figure 5.6 shows the architecture of *metaBest*. It is organized in the three blocks which correspond to the testing approaches described in Section 4. These are described in the next subsections.

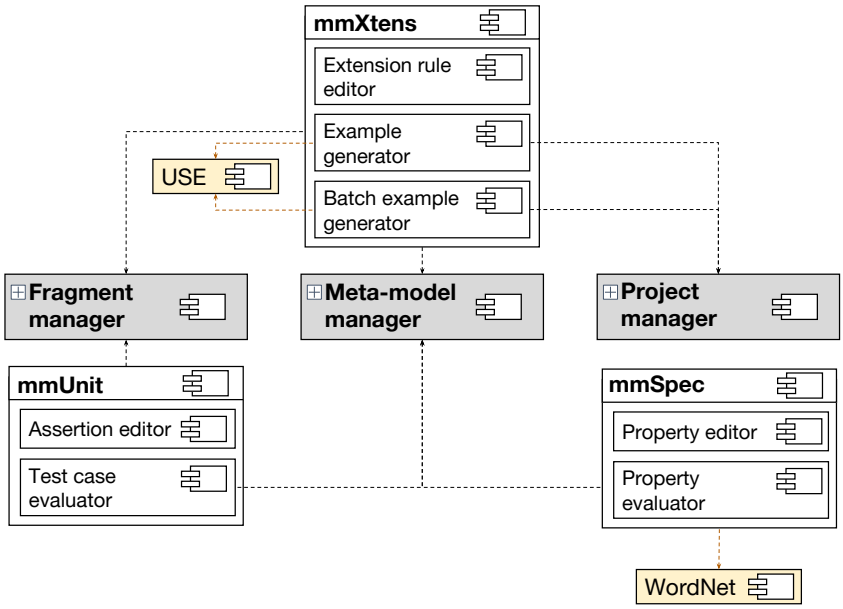


Figure 5.6: *metaBest* component diagram.

5.2.1 mmUnit

As described in Section 4.2 *mmUnit* allows the definition of test cases. Each test case includes a configuration of objects, which can be defined either using a dedicated textual syntax, or a sketching tool. In the latter case, sketches are imported and automatically translated into this textual format to facilitate their subsequent processing, following the same approach as when adding sketches to the *metaBup* project. The editor used for editing test cases is similar to the one for editing fragments; the only difference is that these fragments need to be declared with a different extension (.mbf) under a dedicated testcase package in the project.

As an example, Figure 5.7 shows the same sketch from Figure 5.3, in its textual format, declared as a test case. The test includes assertions at the end of the fragment, in the syntax

defined in Section 4.2. These state reasons why the fragment could be invalid. Once the test is ready, the tool verifies its validity against the meta-model from Figure 5.4, and provides a report view of the results (bottom of the Figure).

Additionally, other incompatibilities not contemplated in the assertion set are shown (below *'more errors found'* in the Figure), just in case one would be interested in going through them.

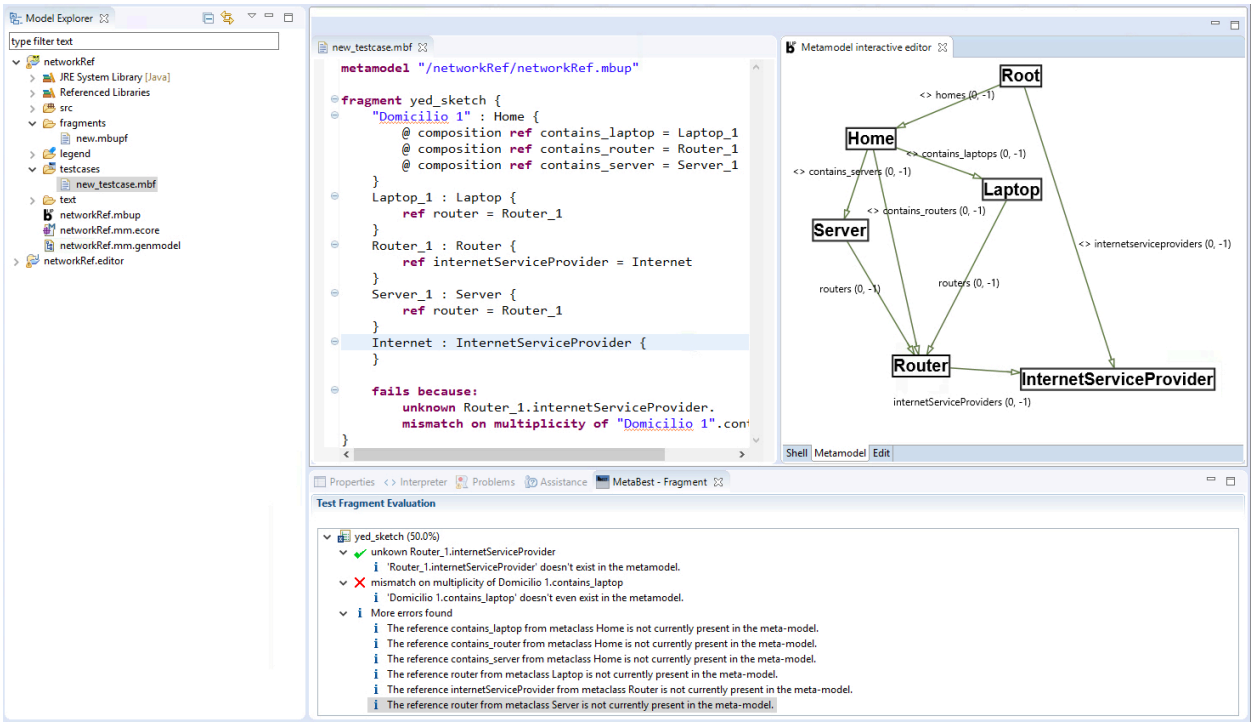


Figure 5.7: *mmUnit* test fragment evaluated.

5.2.2 mmSpec

As Section 4.3 details, *mmSpec* is targeted to express and evaluate expected meta-model properties. Its use within *metaBest* is similar to *mmUnit*. First, a new test file, with .mbm extension, needs to be created in the testcase package. The libraries described in Section 6.2.4 are provided as a baseline property catalog (see the source code of these libraries in Appendix D). Users (usually MEs) can either define their own properties or call the ones in the libraries. They can also define their own libraries and reuse them at their convenience.

Property tests can be validated against the working meta-model, and the results are shown

in the view shown at the bottom of Figure 5.8. Besides assessing them true or false, faulty elements (those not meeting the property) are signaled in the view, while matching elements (those meeting the conditions in each property) are listed if the property is assessed valid.

For example, Figure 5.8 shows the evaluation of three properties (one user-defined and two calls to library-defined properties). While two of them (the first and the third) are true, the second one gives a negative output as a result, meaning that the tested meta-model does not comply with it. The results view to the bottom of the figure shows the expanded result tree for the faulty property, which signals the concrete elements not fulfilling the checked condition. In particular, the property checks that all the elements in the meta-model account a string attribute whose name ends with an "id" suffix.

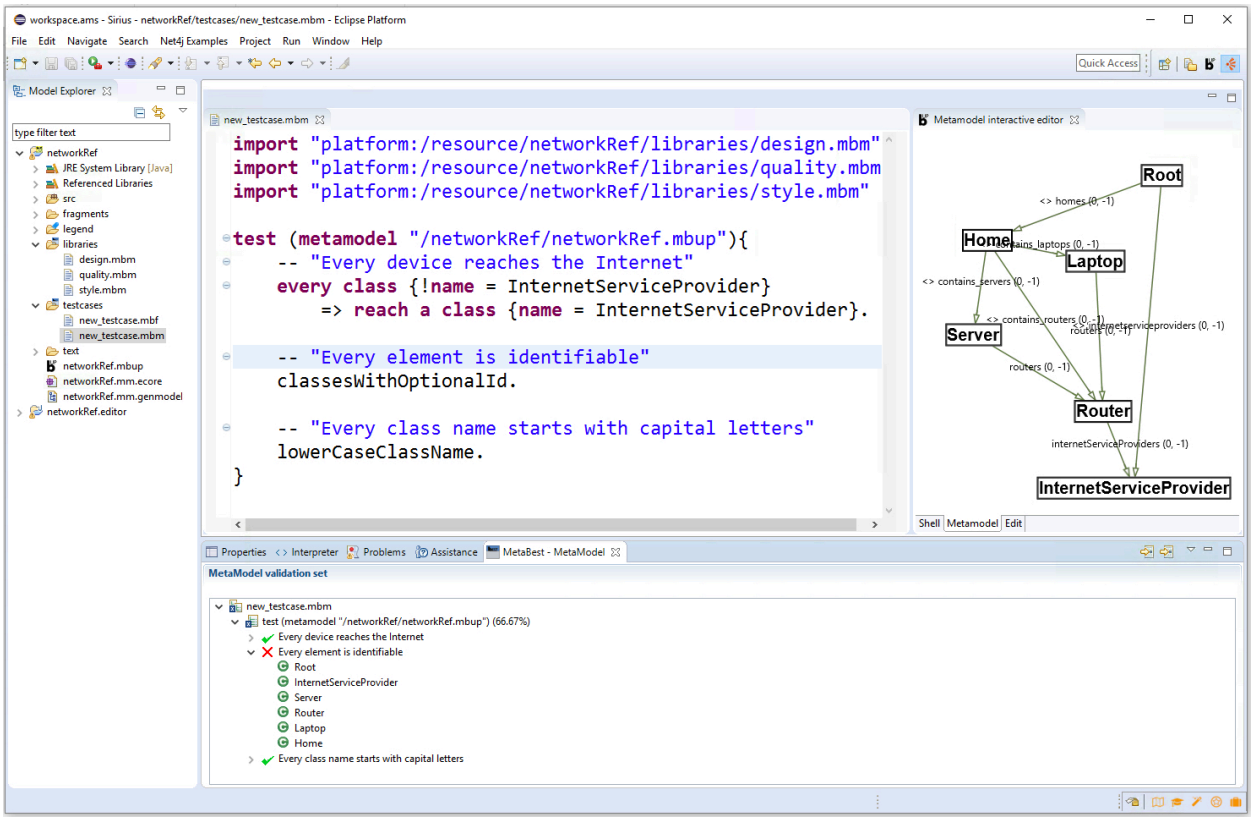


Figure 5.8: Evaluation of an *mmSpec* test.

5.2.3 mmXtens

The third *metaBest* component, *mmXtens*, generates valid instances of the meta-model under construction so as the DE has means to validate the DSML that is being built. These

instances are generated both in text and visual syntax. The power of this feature is that the DE receives examples for validation expressed in the same format employed for drawing the sketches from which the meta-model has been built.

In order to generate an example, first, a test fragment with .mbf extension needs to be created and stored in a testcase package in the project. This will be the seed fragment, as described in Section 4.4. Then, the extension rules need to be added at the end of the fragment using the same syntax introduced in Section 4.4.

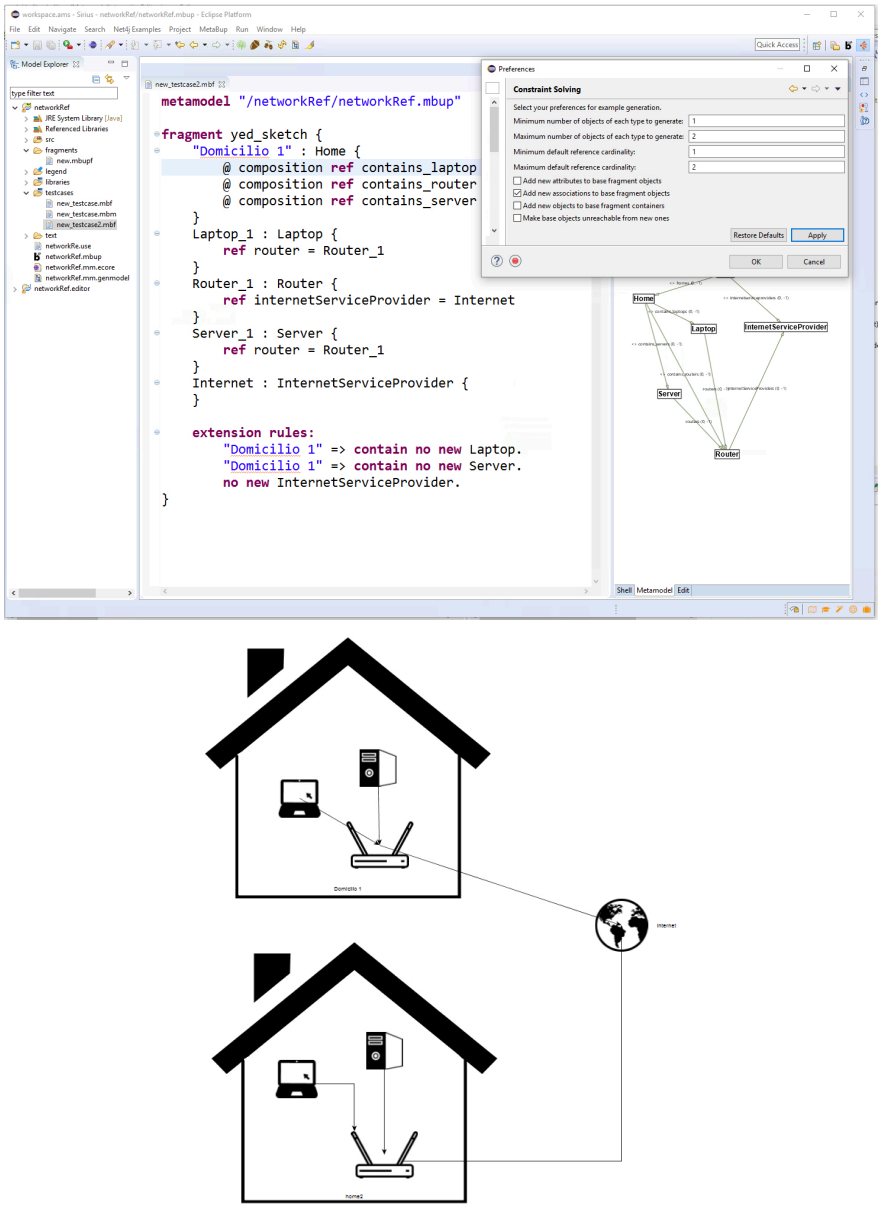


Figure 5.9: Representation of a model example (bottom) generated with *mmXtens* (top).

The preferences explained in Section 4.4, as well as the minimum and maximum object and reference cardinality can be customized at the Eclipse preference wizard. Figure 5.9 shows the previous steps before having a valid example generated: the seed fragment with three extension rules open at the editor in the middle, and the preference wizard to the right. The resulting example from that configuration is shown below.

Finally, *mmXtens* includes a *Batch example generator*. With it, one can generate multiple examples from a meta-model selecting the elements that would be interesting to check. This is achieved with the only aid of a wizard listing all the elements in the meta-model. The user only needs to drag & drop those considered of interest, configure minimum and maximum cardinalities desired for each one, and a set of new examples is placed in the testcase package of choice. The criteria applied for this example generation is established following the one proposed in [32].

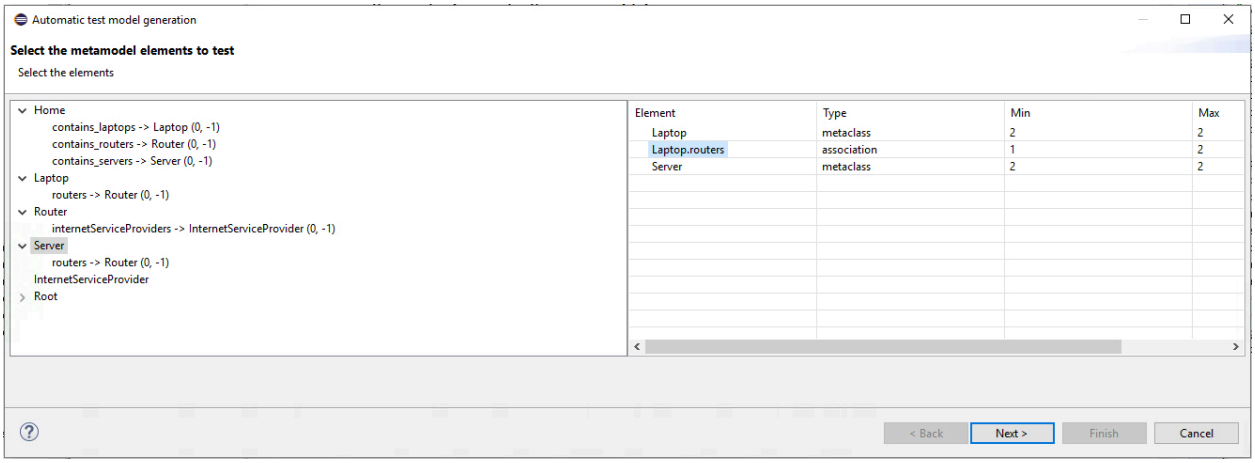


Figure 5.10: Batch example generation with *mmXtens*.

5.3 Support material

A set of support materials on the technical contribution of this dissertation can be consulted at: `www.jesusjlopezf.com/metabup`, including a gallery of examples, demonstration videos, tutorials and the installable version of *metaBup*, in the form of an Eclipse plug-in.

6

Evaluation

This Chapter assesses the validity of this work. In Section 6.1, we present the results of the evaluation carried out with real users generating editors for the DSML from the running examples following the process from Chapter 3. And in Section 6.2 the three languages introduced in Chapter 4 are subjected to a series of experiments to test their usefulness, conciceness, performance, expressiveness and usefulness.

6.1 Evaluation of DSVL development

In order to evaluate our example-based approach to generate graphical modelling environments, a user study was conducted. Since one of the goals of the proposal is enabling the active involvement of Domain Experts in the DSL environment construction process, the study was performed from the point of view of the DE. Hence, the participants in the study played the role of Domain Experts, whereas the author played the role of the Modelling Expert. As DEs, the participants were asked to provide fragments, as well as to evaluate the environments generated from them. In this way, the goal of this evaluation is two-fold: first, to assess whether the example-based approach here introduced is perceived as useful to generate graphical environments, and second, to explore to what extent the generated environments fulfill the DEs' expectations regarding their devised DSL. Hence, the study explores the following two research questions:

- **RQ1**: How useful is the approach to create graphical environments?
- **RQ2**: How well do the generated environments reflect the devised DSLs?

From a technical perspective, it is also interesting to assess the quality of the artifacts

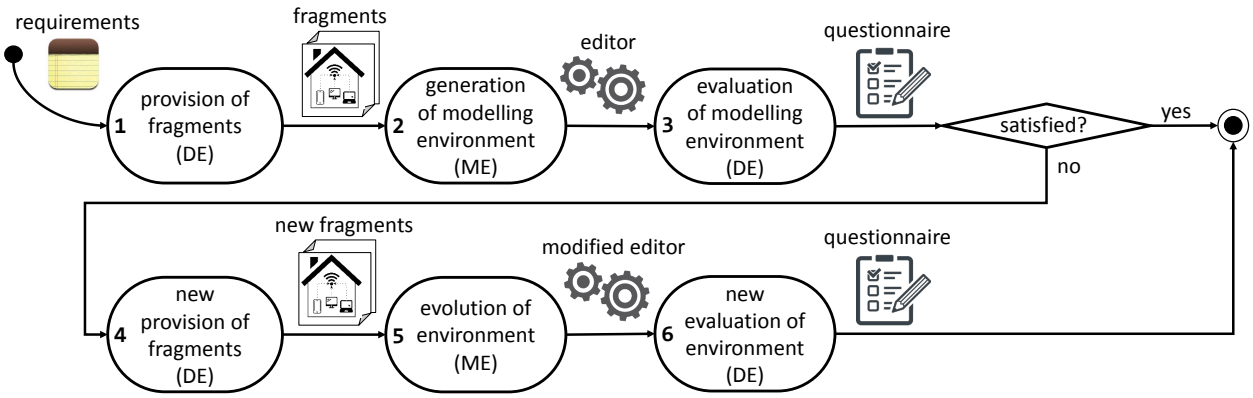


Figure 6.1: Evaluation process.

produced by the approach, which leads to a third research question:

- **RQ3**: How is the quality of the induced domain meta-models perceived?

6.1.1 Evaluation setup

To evaluate these research questions, the user study design emulates a typical example-based workflow, with remote participants playing the role of Domain Experts, and the author playing the role of the Modelling Expert. This workflow is summarized in Figure 6.1. It includes the following six steps:

Step 1 (DE): provision of fragments. First, the participants were given an on-line textual description of the requirements of a DSL, and a drawing tool installation (*yED*) which contained a palette with admissible icons for the DSL. It was decided to use the same DSL as our running example (i.e., home networks), as this would allow analysing whether the same requirements might lead to different graphical representations. The DSL requirements were very general, similar to the description given in Chapter 3. Then, the participants used yED to draw as many examples as necessary to represent all desired aspects of the expected DSL, and uploaded these examples via a web application together with the time employed to complete them.

Step 2 (ME): generation of modelling environment. Starting from the fragments, a graphical modelling environment was generated for each participant. Then, each participant was sent the environment generated out of his/her fragments. At this stage, occasional little formal corrections were performed over the fragments, always ensuring a minimal intervention (see

Section 6.1.4).

Step 3 (DE): evaluation of modelling environment. The participants were allowed to use the generated environment freely with no time restrictions. Then, they replied an online questionnaire rating different aspects, like resemblance of the generated DSL to their expectations, and remarkable or missed features in the modelling environment. Both Likert scales (with scores from 1 to 5) and free-answer questions were used. In case the participants had developed meta-models or modelling environments in the past, they were asked additional technical questions related to the quality of the generated meta-model, and could comment on their preferences on using an example-based meta-model or graphical editor construction process instead of using the typical top-down approach. The complete questionnaire is available in the Appendix E.

Step 4 (DE): new provision of fragments. Participants were given the opportunity to provide a new set of examples complementing those in the first iteration. This was optional, only in case they wanted to refine the generated environment, e.g., because they had spotted some defect on the environment, or because they had failed to represent some DSL requirement in the first iteration.

Step 5 (ME): evolution of environment. The new examples were used to evolve the initial version of the modelling environments.

Step 6 (DE): new evaluation of environment. The participants in this second iteration evaluated the new version of their editors, answering whether their quality had improved and which DSL aspects still remained uncovered. The complete questionnaire is available in the Appendix E.

Thirty people with different backgrounds and ages were invited to participate in the study. In total, 11 replied to the petition, 3 female and 8 male, with ages ranging from 24 to 46 years old. Amongst the different respondents, 8 were university employees (either in an academic or a technical position), 2 worked in the private sector (one in the IT field and the other in a different sector), and 1 was unemployed.

6.1.2 Evaluation results

This section shows the results of the evaluation. First, we analyse some features of the fragments provided by the participants. Then, the information as well as the questionnaire

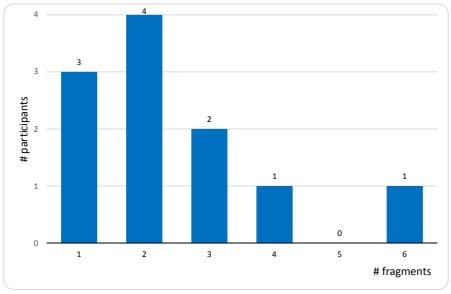


Figure 6.2: Number of fragments per participant.

replies are used to give an answer to our three initial research questions.

6.1.2.1 Diversity of fragments

Each participant could provide as many fragments as desired. Eventually, the number of provided fragments per participant ranges from 1 to 6, with a median of 2. Figure 6.2 shows how many participants (y axis) provided each number of fragments (x axis).

We can examine the structure of the provided fragments to assess the extent of use of the capabilities of the framework. First of all, we study the scope of each fragment. Similar to unit tests in Test-Driven Development [16], in this methodology, each fragment is meant to identify a situation of interest (ideally one DSL requirement) using the minimal number of elements to convey the given meaning. The DSL palette for this experiment had 13 different element types, and the average number of element types per fragment was 9 (see Figure 6.3). The three participants that provided a single fragment used all 13 element types in the fragment, which is understandable as, otherwise, their editors would have resulted incomplete.

Concerning size, fragments had an average of 12 objects and 9 edges, though their size strongly differ from 2 to 30 objects and from 0 to 29 edges. Considering that the average number of object types per fragment is 9, it may be concluded that there is low redundancy (i.e., few repeated objects of the same type).

If we compare the number of spatial relationships and edge-based relationships used in fragments, it is noticeable that objects are connected through edges 2.3 times more frequently than they are using spatial relationships (overlapping, adjacency or containment). While every participant used at least 1 edge per fragment, 4 participants did not employ any of

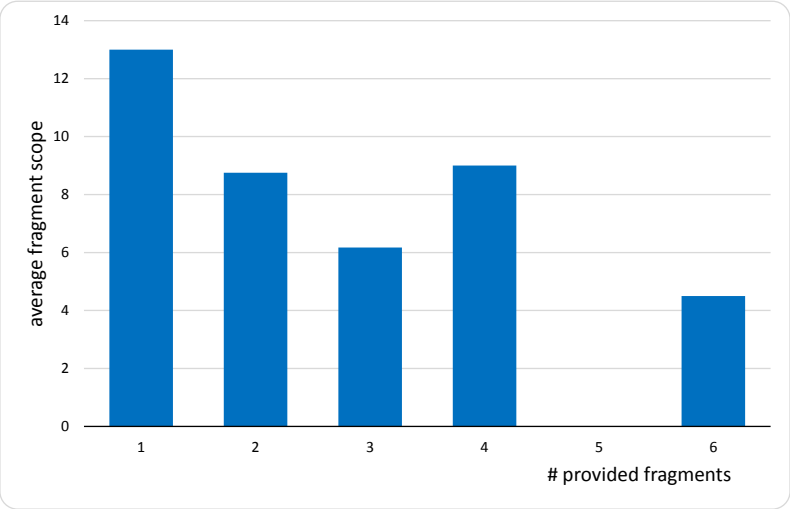


Figure 6.3: Average fragment scope (i.e., number of element types) w.r.t. number of provided fragments.

the detectable spatial relationships. However, if we do not consider these 4 participants, the ratio spatial relationship/edge decreases to 1.3, which puts the average use of both kinds of relationships at a closer level. Still, the general shape of the DSLs was very much graph-like.

Similarly, although the documentation that accompanied the DSL requirements detailed the possibility of using different edge styles, edge styling was seldom used. Only 2 out of the 11 participants exploited this option to discriminate different ways to connect pairs of the same object types. Just as illustration, Figure 6.4 shows to the left the fragment of a user who made heavy use of most of the graphical features supported by our framework, namely spatial relationships, edge styling and attribute labelling for nodes. The fragment to the right belongs to another user who merely connected the objects with non-styled edges and made no use of text labels.

Altogether, the use of graphical features by the participants can be summarized as follows:

- 100% used edges for connecting objects.
- 63% used spatial relationships.
- 27% used object or edge labelling.
- 18% gave style to edges for distinguishing different types of connections between pairs of objects.

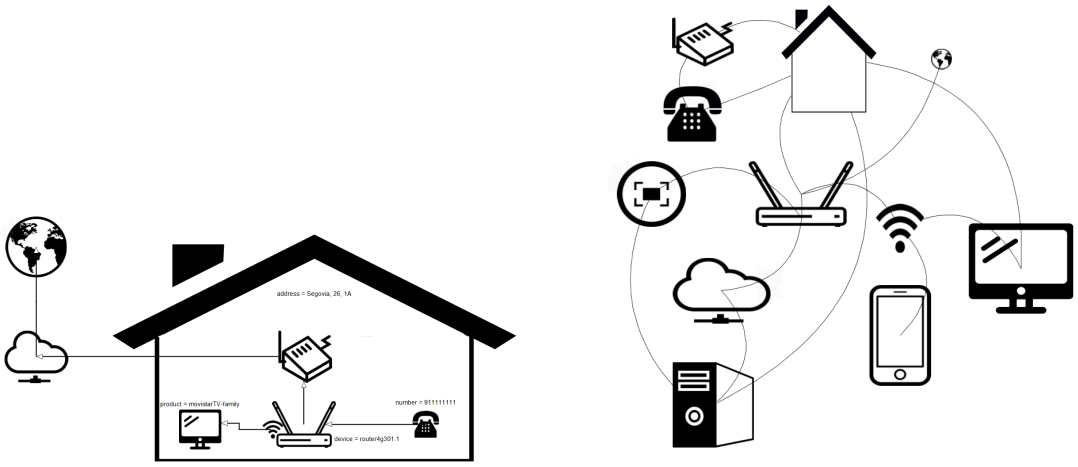


Figure 6.4: User fragments with heavy (left) and meager (right) use of the supported graphical features.

Despite the DSL requirements document encouraged the use of edge styling and layout in fragments, and although the proposed problem really fostered their usage, the results evidence that, in the future, the possibilities of the environment should be further emphasized to potential users.

Once analyzed the features generally present in fragments, the three research questions can be answered.

6.1.2.2 RQ1: How useful is our approach to create graphical environments?

The approach shall be considered useful if it speeds up the construction of graphical environments and it promotes the active involvement of Domain Experts. To evaluate this, it needs to be first analyzed the creation time of the environments in this study, and then, their usability assessed.

Using the proposed approach, the time to create a graphical environment is the sum of the time employed to draw the DSL examples plus the time to generate the environment from them. Since the latter is automatic and negligible compared to the former, it can be assumed, with a minimum error range, that the time to create a graphical environment is roughly the time to draw the examples. Figure 6.5 shows the time employed in this task by the 11 participants. The times range from 15 to 120 minutes, with a median of 37 minutes and an average of 43.8 minutes, while the time per fragment is between 3 and 60 minutes. Hence, the average time to create an environment for the DSL in the study was 43.8 minutes, with 72%

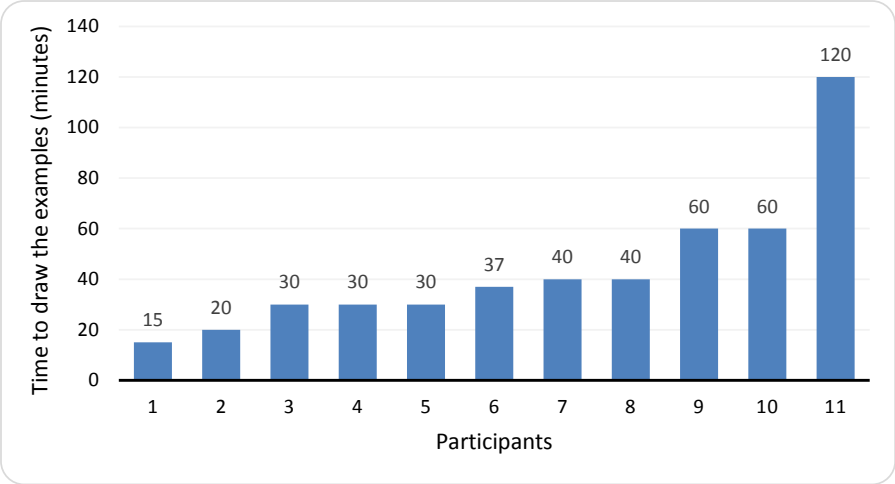


Figure 6.5: Time employed to draw the fragments.

of the participants employing even less. This time can be considered short, as developing a similar environment by hand would require implementing the following artifacts (average numbers over all generated editors) in the context of the EMF framework: an *ad-hoc* built meta-model with 14 classes, 14 attributes, 22 references and 1.5 inheritance relationships; and a platform-specific visual editor generator model with 232 objects. Moreover, it would require having deep technical knowledge on all of these technologies.

In the study, 6 participants had experience on developing meta-models and modelling environments. Surprisingly, three of them were the slowest (60, 60 and 120 minutes), while the other three were the fastest to build the examples (15, 20 and 30 minutes); hence, there is no correlation between time and MDE experience. The only 2 participants that had no experience on modelling or meta-modelling dedicated 37 and 40 minutes on drawing 3 and 1 fragments, respectively. This demonstrates that non-modelling experts can actively contribute to developing graphical editors by providing DSL examples, as our approach synthesizes working editors out of them.

Regarding the usability of the generated editors, Figure 6.6(b) shows how easy to use they are according to the participants. The answers range from average (3) to very easy (5), with a median of 4 (easy), and average of 4.1. These numbers suggest a good usability of the final environments, although the participants also mentioned some aspects to improve, which are summarized in Table 6.1. In particular, one participant suggested reducing the high number of edge types that appear in the palette, e.g., by having one button for all of them, and deducing the type of any created edge from the types of the objects it connects.

While this is a good strategy for large DSLs, the default drawing mode of Sirius is using a palette button per edge type, and hence, we plan to study the feasibility of this proposal in the future. The rest of suggestions are limitations of the target platform employed, not feasible to overcome by this study. As an example, two participants stated that containment seemed "odd" in the generated editor, and that objects placed in a container could not be moved to any other container. One of them also signaled that the action of creating new models was not "intuitive", alluding to a menu option in the prototype enabling the creation of new model files that he didn't manage to find.

As for the editor aspects the participants liked the most (question Q9 in the questionnaire, see Appendix E), they mentioned flexibility, simplicity, being easy to use, and the support of many edge styles. Two participants explicitly mentioned that they found the approach very useful.

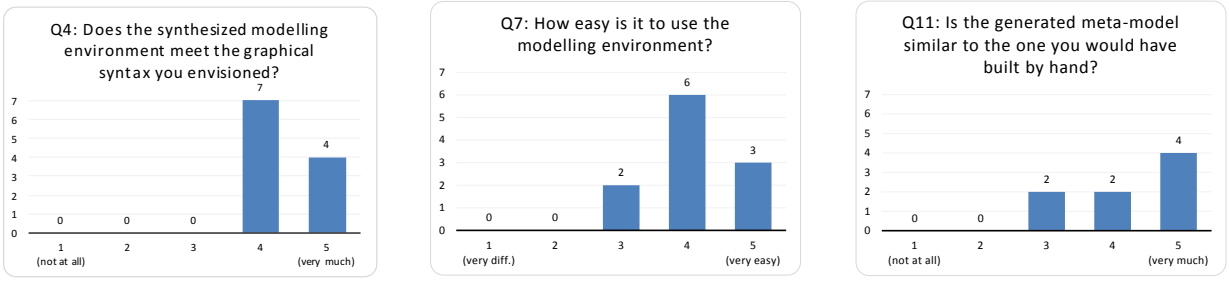


Figure 6.6: Scores to different aspects of the generated environments and their underlying domain meta-models.

Table 6.1: Answers to question Q8: *Which aspects of the (generated) environment would you improve?*

There are too many edge types in the palette.
Creating new models is not intuitive.
Objects are hard to resize.
Handling of containment is intricate.
Moving an object between containers is not possible.
Difficulty to draw edges and layout.

In summary, the participants found the approach to create graphical modelling environments useful. More importantly, participants with no modelling background were able to design a graphical DSL and, by providing examples, creating an editor for the DSL. Nonetheless, the participants have also proposed some improvements to the usability of the generated environments, which have been added to this work's future improvements schedule.

6.1.2.3 RQ2: How well do the generated environments reflect the devised DSLs?

To answer this research question, the responses to questions Q4, Q5 and Q6 in the questionnaire are examined (see Appendix E). Question Q4 requests a score for grading how precisely the approach was capable of producing a graphical syntax that resembles the original drawings. Figure 6.6(a) summarizes the given scores, which went from much (4) to very much (5), with a median of 4 and average of 4.36. Questions Q5 and Q6 in the survey (both free-answer questions) provide more elaborate answers concerning the accuracy of the generated graphical DSLs.

First, one participant complained that fixed and mobile phones could be placed both inside and outside homes in the generated environment. It is significant that this participant provided fragments in which phones were inside homes, and fragments in which they were not. Because the ME incorrectly interpreted these fragments as examples (i.e., as complete versions of models), in the generated environment phones could be placed in two different kinds of containers: homes and "the canvas". Interpreting the drawings as fragments (i.e., as possibly incomplete models) solves the problem. Alternatively, the environment could have been refined in a second iteration, though the participant deemed it was not necessary.

Another participant stated that objects originally painted superimposed in the fragments had been substituted by containment relationships in the final editor. This is a limitation of Sirius, which does not support overlapping relationships between objects. In order to handle this, the implemented exporter for the prototype offers the possibility to choose which of the two remaining spatial relationships (adjacency or containment) should substitute overlapping in the editor. During the experiment, this preference was set to containment for all examples.

Finally, two participants reported differences on the size and position of the model elements with respect to the original fragments, but they did not report mismatches regarding the expected DSL itself.

Among the aspects best captured by the generated DSLs, the users mentioned the edge styles and the containment and adjacency relationships. Anecdotally, one participant liked that the position of elements had been preserved in the migrated models, which surprisingly, was mentioned as an aspect to improve by another participant.

Notably, no participant requested a second iteration to refine the generated environments. This fact, and the average score 4.36 (over 5) given by the participants when asked if the

DSLs met their expectations, indicate that the environments reflect reasonably well the devised DSLs.

6.1.2.4 RQ3: How is the quality of the induced domain meta-models perceived?

All solutions led to similar meta-models, with particular differences lying in how participants chose to graphically represent certain aspects of the domain.

Next, to answer RQ3, we analyze the replies to questions Q11, Q12 and Q13 in questionnaire 1, which were only available to those participants with meta-modelling experience (8 out of 11 participants). Question Q11 provides a measure of the degree in which the induced meta-model matches the user's expectations. As Figure 6.6(c) shows, the similarity between the expected meta-model and the generated meta-model ranges from average to very much, with a median of 4.5 and average of 4.25. These numbers indicate that the induced meta-model was found similar to what a modelling expert would build by hand.

Question Q12 identifies aspects incorrectly captured by the induced meta-model, hence giving an indication of the perceived meta-model quality. This is of special interest in the case of participants that assigned lower scores to Q11. Table 6.2 summarizes the identified issues. Two participants commented that they would have created one abstract class holding common references to other classes. Interestingly, the proposed solution supports this refactoring and recommended its application in these cases, though it wasn't applied because of the premise to not to pervert the evaluation with manual modifications to the induced meta-model. Another participant complained that the name of some inferred references was strange, e.g., containment; as before, these names could have been modified by the ME in the fragment revision phase. This same participant also missed some meta-model attributes to represent the object locations; however, this ought to be regarded as concrete syntax information which should not belong to the domain meta-model. The remaining 5 participants (including one that ranked 3 to Q11) did not find errors in the induced meta-model. Overall, these results show that the participants perceived the quality of the induced meta-models as high.

Table 6.2: Answers to question Q12: *Which aspects does the (generated) meta-model not capture correctly?*

Common features are not generalised.
Missing attributes to represent object locations.
Some features have strange names, e.g., containment.

Regarding Q13, only 2 participants stated that they would prefer building the meta-model by hand, even though they had rated the induced and expected meta-models as very similar (maximum score in Q11). The remaining 6 participants would prefer using an example-based approach, 4 of them requiring the ability to modify the resulting meta-model by hand, and the other 2 considering this unnecessary.

Altogether, the participants ranked the quality of the generated meta-models as 4.25 out of 5 in average, hence, the perceived quality of the generated meta-models is high. Regarding the few detected issues (like the generalization of common features), the system assists the ME in their correction by recommending suitable refactorings.

6.1.3 Threats to validity

Next, we analyse the different threats to the validity of the study. The selection bias was minimized by promoting the participation of people with different background, ranging from computer science students with a shallow knowledge of modelling techniques, professors both with and without expertise on DSLs, workers on technology companies without a specific training on modelling, and people working on non-technological companies. This is representative of very different Domain Expert profiles. However, the participants were not real experts on the selected home network domain, and therefore, they might have been less demanding when evaluating the expressiveness of the generated editors. This effect was tried to minimize by asking the participants whether the final DSL was the one they had in mind, for which the domain expertise is not relevant. Similarly, no incentive (monetary or of any other kind) was offered to the participants, which may have hindered their engagement leading to less accurate scores or answers, and preventing their participation on a second iteration of the editor [62].

Another threat to the internal validity of the results is the 6-7 days elapse since the participants provided the examples until they evaluated the generated environment, as in the meantime, some of their initial expectations regarding the DSL may have been distorted. This elapse was due to the variety of participant profiles, and in order to promote the

participation, 7 days were granted to draw and submit the examples off-line, and another 7 to evaluate the generated editor and fill in the questionnaire. Moreover, 3 participants delayed their evaluation 7 extra days due to professional commitments.

On the other hand, having performed an off-line double-blind user study has eliminated any possible experimenter bias that could have been inadvertently introduced.

Regarding the generalizability of the results across people, as mentioned before, we selected participants with different backgrounds on modelling and DSLs in order to make our results as general as possible. However, considering the nature of the proposed problem (in the field of computer network configuration), all of them had some training or working experience on computer science and programming. Hence, it remains as a threat to the external validity of this study taking into account other kinds of domain experts with a low technological profile.

Moreover, the study emulated a workflow where only one DE contributed DSL examples. Hence, the findings here concluded cannot be generalized to situations where contributions come from several experts who might even provide different representations for the expected DSL. In particular, in a project setting, one would probably deal with teams of domain experts providing fragments.

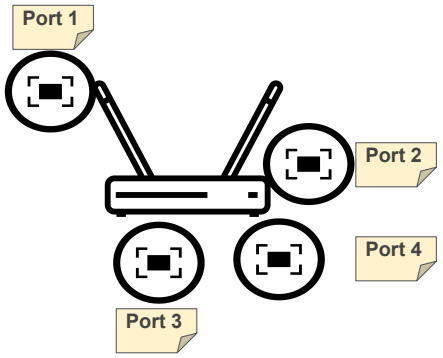
Finally, regardless the number of people invited to participate in the study, only 11 participants completed the evaluation. As stated in [62], recruiting participants in tool studies is difficult, but performing further studies with more participants, working in teams, is in the scope of the future work of this study.

6.1.4 Discussion

Finally, several interesting details of the experiment, lessons learnt, and open challenges for future work are discussed.

Playing the Modelling Expert role, it was tried to interfere as least as possible in the editor generation process. However, in some cases, little adjustments needed to be performed to the imported fragments to correct evident mistakes made by accident when drawing the examples, or to perform fixes that did not affect the semantics of the domain. Figure 6.7 shows an example. In the tool used for drawing the sketches, each element in the palette is contained in an invisible bounding box, and spatial relationships are calculated in accordance to this box. Thus, in Figure 6.7, although the intention of the DE was to draw all ports

adjacent to the router, the two ports that are superimposed to the bounding box of the router (i.e., Port1 and Port2) get classified as overlapping references by the system. Hence, in this case, the imported fragment had to be manually modified so as to delete the overlapping reference and add Port1 and Port2 to the adjacency reference.




```
1  fragment
       importedFragment
       {
2    router : Router {
3      @overlapping
4      ref
           overlapping_ports
           =
5        port1, port2
6
7      @adjacency
8      ref adjacent_ports
           =
9        port3, port4
10   }
11
12   port1 : Port {}
13   port2 : Port {}
14   port3 : Port {}
15   port4 : Port {}
16 }
```

Figure 6.7: Common faux pas in the drawing of fragments (left) and its automatic parsing into text fragment (right).

The following list itemizes the adjustments that needed to be performed, indicating in parenthesis the frequency of changes with respect to the total number of participants:

- **Ignore invisible bounding box (6/11).** The above-mentioned case, in which a participant is ignorant of the transparent bounding box of objects.
- **Rename reference (11/11).** Because fragments do not include reference names, these were created in the format <*source*>2<*target*>.
- **Rename auto-generated superclass (1/11).** The meta-model induction algorithm is able to infer abstract superclasses for common features, assigning as class name a common substring of the children class names (e.g., Phone if the children classes are FixedPhone and MobilePhone). If the subclasses have no common morpheme, the ME is prompted to set a domain-significant name for the new class.

It is valued positive that, although the process entitles the ME to perform deep changes over the fragments and the meta-model, little manual editing was necessary to obtain well-valued editors.

Roughly half the participants expressed some disconformity with the selected tool (*yED*) for drawing the sketches in the prototype. Most criticisms signalled the complexity to draw edges as the main reason to delay the completion of fragments. However, when analyzing the results of the survey, the time to draw each fragment does not seem significant in the assessment of the generated editors. Looking at Q4, which asked whether the synthesized environment met the envisioned graphical syntax, the average score given by the participants who took 24 minutes (the average time per fragment) or more to complete each fragment is 4.25 out of 5, while the score given by the participants who took less than 24 minutes to draw each fragment is 4.43, which is not a significant difference. Probably, a more usable or popular drawing tool would have led to shorter drawing times. Anyhow, the system can be extended with importers for other drawing tools, not being the particular selection of drawing tool a limitation of the approach.

Similarly, most identified deficiencies regarding the usability of the generated editors are due to limitations of some Sirius features. Although the framework is likewise extensible in the export stage as it is in the import, Sirius is one of the most powerful tools nowadays for developing graphical modelling editors.

Regarding the fragment provision process, we have detected that there is a need to better instruct Domain Experts in some features of the framework, in particular concerning the usage of spatial relationships and edge styling. Moreover, the difference between fragment and example was not well understood by some participants in the study. A better understanding would have helped in clarifying certain ambiguities and misinterpretations, contributing to the utter completion of more qualified editors. In the framework, examples are built in the same way as fragments, but only the former represent complete models. Hence, the meta-model induction algorithm does not have to apply heuristics or prompt disambiguation tasks to the ME when processing examples, as it may happen for fragments. Ideally, fragments should contain minimal sets of objects representing portions of domain information, whereas examples are more widespread. In both cases, they can be used as test cases [74] to identify conflicts that should be resolved before altering the domain meta-model.

Finally, the experiment has purposely omitted some advanced features of the system for simplicity. For instance, drawings can be annotated to introduce some domain restrictions, which get compiled into OCL meta-model constraints (see A for more details). Actually, one participant stated in the questionnaire that the editor should have incorporated an OCL constraint.

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

6.2 Evaluation of V&V techniques

In this section, the proposed V&V framework is subjected to a series of tests and experiments in order to prove its value addition to currently available alternatives. In this sense, the next subsections aim to evaluate *mmUnit*, *mmXtens* and *mmSpec* in terms of usefulness. *mmSpec* is also evaluated regarding its conciseness, performance and expressiveness.

6.2.1 Evaluating the Usefulness of *mmUnit*

In this section, *mmUnit* is compared with the available means for unit testing in EMF [95], 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 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 analyze 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, every *mmUnit* primitive was tried to be implemented using JUnit and EMF. Appendix B includes part of the Java implementation code, while Table 6.3 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.

```
1  fragment MissingCableModem2Network {
2    internetServiceProvider1 : InternetServiceProvider {
3      attr name = "lemon"
4      ref networks = ispNetwork1
5    }
6
7    ispNetwork1 : ISPNetwork{
8      attr tier = 3
9      attr location = "MAD"
10   }
11
12   home1 : Home{
13     attr name = "Damien Jurado"
14
15     @overlapping @containment
16     ref modem = cableModem1
17   }
18
19   cableModem1 : CableModem{
20     attr ipBase = "251.12.210.56"
21   }
22
23   fails because:
24     missing reference from cableModem1 to ispNetwork1
25 }
```

```
1  public class InternetConnectionTests {
2
3    private String uri = "TestCase2.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("
           NetworkReference.impl.",
12       NetworkReferenceFactory.eINSTANCE,
13       loader.getErrors());
14   }
15
16   @Test
17   public void testAssertion(){
18     assertion.assertMissingFeature( "CableModem", "
           network");
19   }
20 }
```

Listing 6.1: Example of *mmUnit* test (left) and corresponding EMF test (right).

As an example, Listing 6.1 shows to the left the *mmUnit* test case already shown in Figure 4.5, and to the right an attempt towards a similar test in EMF. We had to develop utility classes to load models (class MmUnitModelLoader) and to emulate to some extent some of the *mmUnit* assertions (class MmUnitAssertion). Appendix B provides details of the developed library of assertions.

In this example, we can see that the model fragment and the test are specified together in *mmUnit* (lines 2-21 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 the content of the file was intentionally omitted). This separation of the model under test and the assertions hinders understanding the rationale and objective of the test. Regarding *mmUnit* assertions (line 24 in the left listing), they are emulated in JUnit by using our library of assertions for EMF inside test methods (lines 17-19 in the Listing to the right). 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 cableModem1 and ispNetwork1 to refer to objects, this is not possible in EMF, where the assertion just checks that there is a missing feature named network from an object of type CableModem. 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, the following lessons learned must be enlisted:

- 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 6.3, by the fact that 7 out of 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 edit the 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 this experiment, a library of model-specific assertions which inspect the error messages produced by the Diagnostic class needed to be manually encoded.
- 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.

Therefore, 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.

6.2.2 Evaluating the Conciseness of *mmSpec*

mmSpec has been designed to facilitate the definition of meta-model properties by making available high-level primitives like path, inh and collect. In order to evaluate to which extent *mmSpec* is concise, its primitives were compared with their equivalent representation in OCL. The comparison is made with OCL because this is the standard language proposed by the OMG for model queries [81], and a meta-model is just a model.

Appendix C includes the translation of most primitives in *mmSpec* into OCL (36 cases in total). In this section, the most relevant observations are commented.

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 C): 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, the OCL expression shown in Listing 6.3 checks Rq1 from Listing 4.1 (replicated in Listing 6.2), which requires that Routers do not have direct access to their ISPNetwork, but only through a Cable Modem.

```
1  -- "Rq1: Routers cannot have direct access to their Network, but through a Cable Modem"
2  every path{and{
3     from a class{name=Router},
4     to a class{name=ISPNetwork}}}
5  => through a class{name=CableModem}.
```

Listing 6.2: *Rq1* from Listing 4.1, expressed in *mmSpec*.

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

Listing 6.3: Encoding of an *mmSpec* property in OCL

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 Listing 4.1) is more comprehensible and maintainable, due to the use of primitives for path quantification and analysis (e.g., through).

Reasonably, OCL results in more complex property expressions than *mmSpec* since 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, it must be acknowledged that OCL is a richer, more expressive constraint language than *mmSpec*. Some of the features of OCL that *mmSpec* does 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, the goal of *mmSpec* 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.

6.2.3 Evaluating the Performance of *mmSpec*

Another interesting facet to look into is whether the *mmSpec* interpreter that supports *mmSpec* is capable of delivering runtime figures analogous to the standard OCL. To evaluate this aspect, this section presents the results of measuring the evaluation runtime of a set of properties expressed with *mmSpec* and their equivalent expressions in OCL. In doing this, the native org.eclipse.ocl 3.4.2 OCL library was employed, as it is based on Eclipse and therefore it has the same resources at its disposal as the *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 the experiment, a test set of 33 meta-model properties was evaluated 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, the properties shown in the Appendix C were evaluated, 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 6.8 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. The upper graphic from Figure 6.8, shows that the runtime increases in absolute terms as the size of the meta-model does. In the case of *mmSpec*, the lower chart from Figure 6.8 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.

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 Appendix C). 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 *mmSpec* in these particular cases.

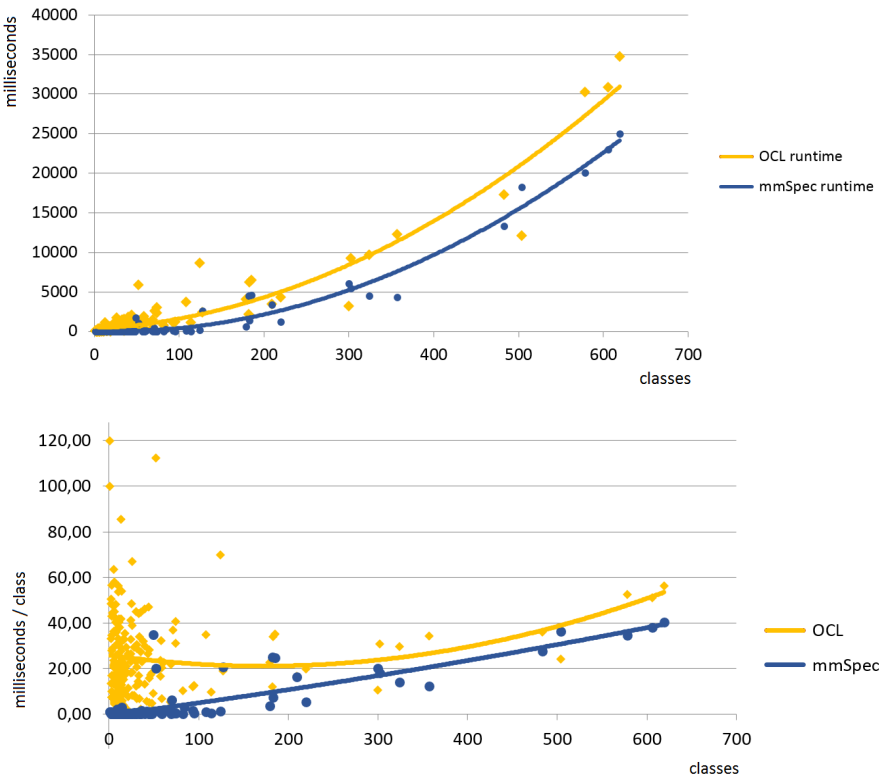


Figure 6.8: Comparison of performance in OCL and *metaBest*: runtime vs meta-model size.

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

To conclude, a threat to the validity of the results here presented is that it depends on the actual OCL encoding of the properties, and the study might not have come up with the most efficient expressions. To mitigate this risk, the expressions were carefully designed, trying to follow performance optimization patterns for OCL [25], 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*.

6.2.4 Evaluating the Expressiveness of *mmSpec*

To demonstrate the expressiveness of *mmSpec*, a library is reported 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

[4, 5] or have been derived by the experience of the author and supervisors. The library has four categories of issues, depending on their nature and relevance:

Design. Properties signaling 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 [22].

Table 6.4 lists the properties from these categories, whose complete enconding can be consulted in the Appendix D.

To illustrate *mmSpec*'s expressiveness, next we show the formulation of one property from each category. Notice that Listing **??** showed the encoding of BP03 (in Rq7), and a similar property to N04 (Rq6).

- *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.
  ```

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 6.4: Library of meta-model quality properties.

- *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:

```
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 the library, some of the properties in the aforementioned works [4, 5] were discarded, 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 expressiveness for its practical use, as all naming conventions suggested for classes, attributes and binary associations in [5] could be encoded, as well as 34 out of 44 (77.2%) quality issues for conceptual schemes presented in [4] (excluding 21 UML-specific or non-automatable checks 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 this example-based meta-modelling framework, like derived attributes, user-defined data types, or arbitrary OCL expressions.

As a threat to the validity of these 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 the *mmSpec* language with these primitives. In order to mitigate the risk, a set of properties developed by a third-party [4, 5] were chosen when building the meta-model quality property library.

6.2.5 Evaluating the Usefulness of *mmSpec*

To evaluate the need for the *mmSpec* technique and have a measure of the quality of current meta-modelling practice, the library of quality properties was applied to a test set of 338 meta-models of varying sources, size and format. The main 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 meta-model zoo (295 meta-models) and specifications of the Object Management Group (OMG[1], 43 meta-models). In particular, the 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, only those specifications that make available a meta-model implementation in a format that can be imported by *mmSpec* (UML, EMOF, CMOF and Ecore) were included. Note that the fact that a variety of meta-model formats was considered in this analysis, shows the re-usability of the approach, as the same library of properties was applied to all cases regardless their format.

The reason why meta-models from two different repositories were selected, 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 analyze – whereas the degree of maturity and completeness of the meta-models in the ATL zoo greatly varies. 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 the goals of this work is to check whether V&V is needed for both large and small meta-models.

Figure 6.9 shows the number of quality issues detected in both analyzed repositories. The ATL zoo 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 analyzed OMG meta-model raised some potential quality error or warning.

Figure 6.10 depicts the distribution of detected issues with respect to the meta-model size.

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

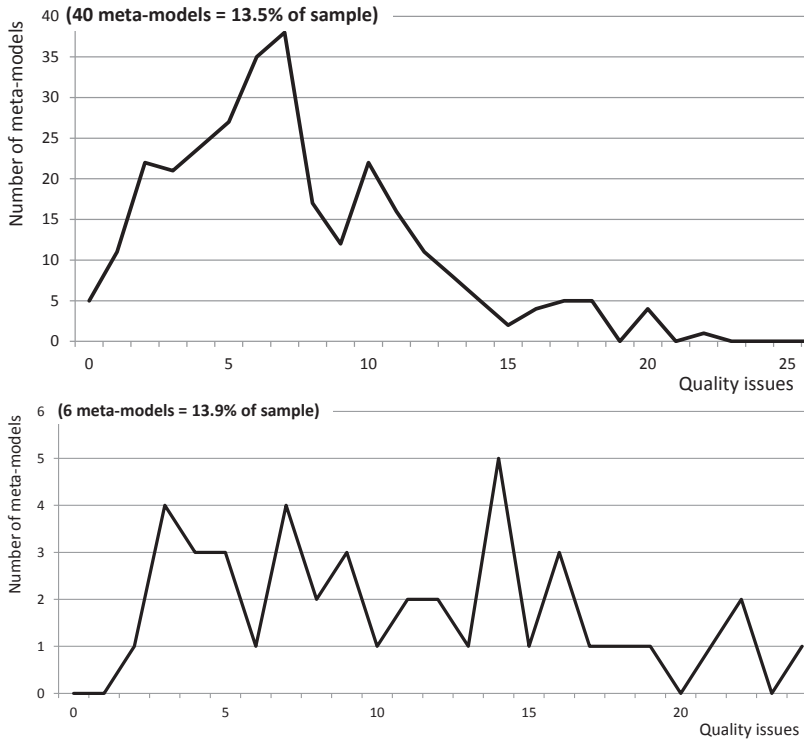


Figure 6.9: Number of meta-model quality issues in ATL Zoo (upper chart) and OMG specifications (lower).

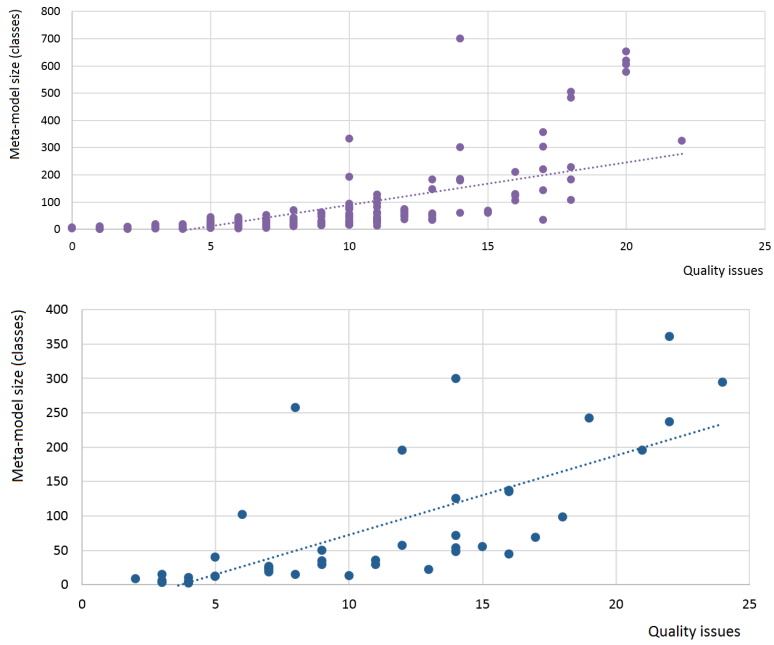


Figure 6.10: Number of meta-model quality issues, with respect to the meta-model size, in ATL Zoo (upper chart) and OMG specifications (lower).

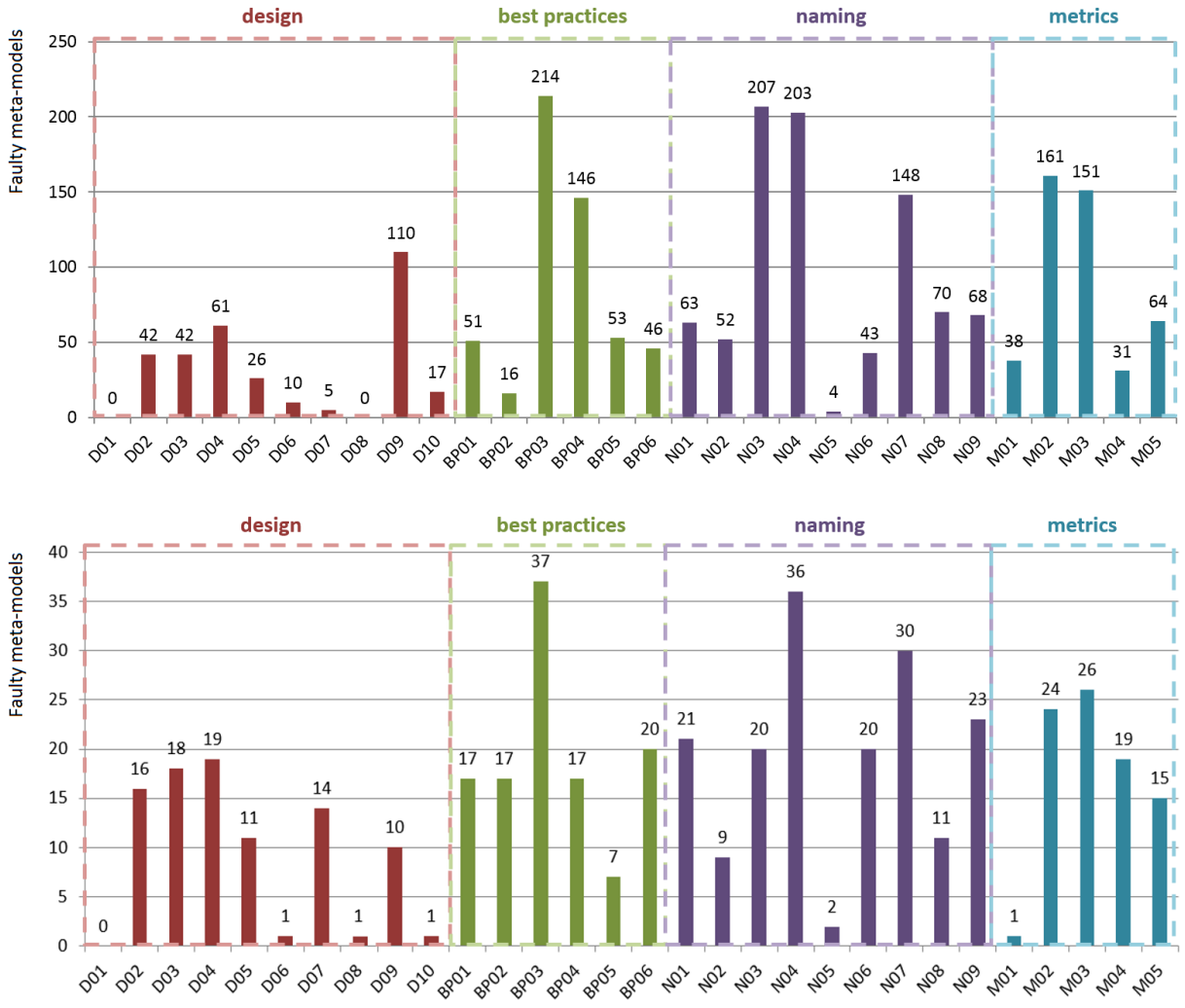


Figure 6.11: Number of meta-models that contain issues of a certain type.

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.

Regarding the distribution of issues according to their kind, Figure 6.11 shows how many meta-models fail each property from Table 6.4. *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 (upper graphic in Figure 6.11): `D01` and `D08`. `D01` checks the absence of repeated attributes

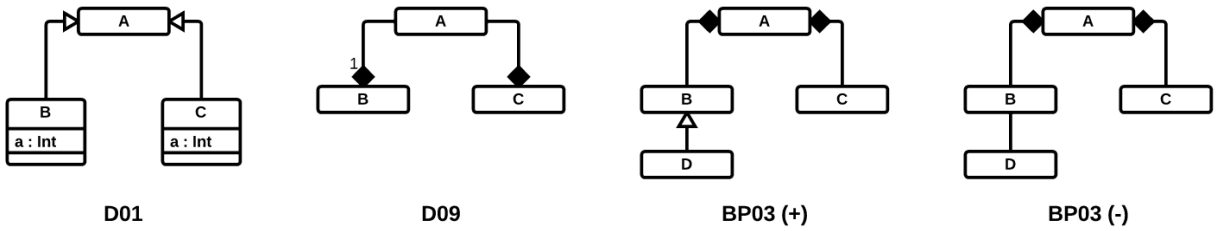


Figure 6.12: Some quality issues analyzed by the library.

in a whole hierarchy (see *D01* in Figure 6.12 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 (lower graphic in Figure 6.11), D01 is not present likewise, and there is only one 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.

However, 110 meta-models from ATL (37.2%) and 10 meta-models from OMG (23.2%) fail property D09. As illustrated in Figure 6.12, 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 necessarily 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% analyzed 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 6.12 shows an example meta-model that fulfills 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.

Amongst the naming conventions, N03, N04 and N07 are scarcely followed. N03 demands the names of binary associations to be verbs (e.g., reaches), while N04 and N07 check conventions

for naming class and boolean attributes (see Table 6.4).

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). Certainly, a way to improve the quality of meta-models might be 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 this study, there are a number of issues that should be mentioned. First, the number of analyzed meta-models (338) could be considered insufficient, and indeed, the number of analyzed meta-models will expectedly be increased in future studies. Nonetheless, even if the number of analyzed meta-models could be higher, the point is that a high number of issues were detected 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 analyzed 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 sub-classified by the meta-model users. For this kind of meta-models, the featured best practice `BP02` which checks whether there are abstract leaf classes is not needed. Because this study has not removed such cases from the results, some of them therefore raised issues that may not be actual errors. It is up to future work to extend this work in order to exclude such cases from the results.

6.2.6 Evaluating the Usefulness of *mmXtens*

In the following experiment, the usefulness of *mmXtens* is assessed by validating the meta-models built by 26 undergraduate students as solution to a deliverable exercise. In particular, *mmXtens* was used to generate instances of the delivered meta-models satisfying certain requirements, which were encoded as *mmXtens* extension rules. The manual inspection of the generated models exposed different kinds of meta-model issues. The exercise is described below:

> Design a DSVL for house blueprints that should consider several types of rooms, like entries, living rooms, bedrooms, kitchens, baths, gyms and balconies. Any house should have at least one bath and one living room, at most one entry, and there are no restrictions concerning the number of rooms for the rest of types. In addition, all rooms may have any number of plug outlets and switches for lights.
>
> The blueprint should allow designing the disposition of rooms, which, for simplicity, are rectangular and have the same size. Rooms can adjoin other rooms in any of the four cardinal points (north, east, south and west), and there cannot be isolated rooms (i.e., without any adjacent room). Rooms can adjoin at most one other room in each direction, balconies can be adjacent to one room at most, and entries can be adjacent to three rooms at most.
>
> It should be possible to place between zero and two windows in each exterior wall (i.e., in non-adjacent walls to other rooms). Balconies are the only exception to this condition, as their windows can only be placed in interior walls (i.e., walls adjacent to other rooms).
>
> Regarding doors, they can only be placed in interior walls. Rooms may have either zero or one door in each wall, and at least one door in total. Additionally, a blueprint should include exactly one entrance door, which, in this case, should be in an exterior wall. If the blueprint has an entry room, the entrance door should be there.[2]

Figure 6.13 shows a valid blueprint of a hypothetical solution to the problem. Students developed their solutions with standard EMF tools (Ecore and OCL editors).

[2]Text originally in Spanish in the exercise.

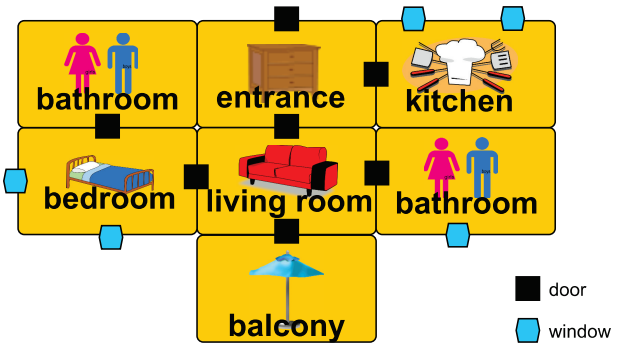


Figure 6.13: Example of valid house blueprint.

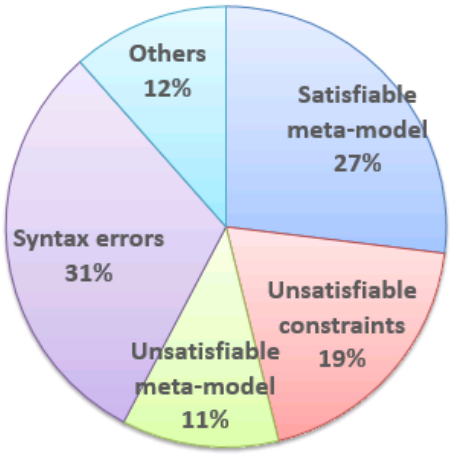


Figure 6.14: *mmXtens* evaluation results

Then, *mmXtens* was used to test to what extent the evaluated meta-models are able to produce valid models meeting the requirements. Basic *mmXtens* rules, minimally adapted for matching the features of each solution, were used to generate a model with only one house, being the reason for failure checked in case of failing.

Figure 6.14 shows the results to the evaluation: 11% meta-models presented satisfiability problems, meaning severe flaws like unfeasible association cardinalities or containment relationships that make impossible having instances, even if their OCL constraints are not considered.

There are two main types of errors for OCL constraints: syntax errors (31%) and unsatisfiable constraints (19%). The former include errors like missing brackets or wrong parameter types. The latter refer to incompatible constraints, so that making satisfiable one would make the other(s) unsatisfiable. These issues are hard to detect in a traditional testing approach, since one would have to produce models and detect errors manually. A small per-

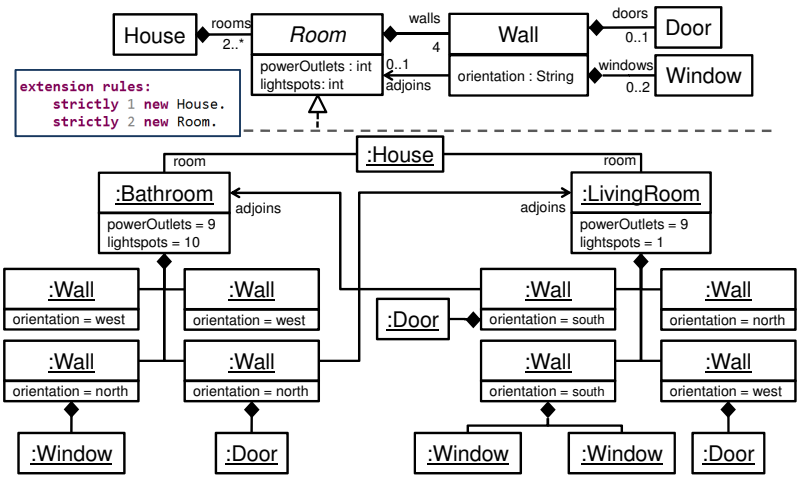


Figure 6.15: A student solution and a generated model.

centage (12%) of solutions had complex and numerous issues, demanding profound changes for being acceptable. Only 27% were able to produce examples. Those 7 solutions were used to make a series of tests using *mmXtens* to check their correctness regarding the domain of the problem.

As an example, Figure 6.15 shows a student solution with two extension rules seeking the smallest valid house, and the generated model. The model has some flaws: both rooms have walls with the same orientation, while lacking others. Moreover, both rooms keep a logical north/south adjacency, but the involved walls hold two doors instead of sharing one. Finally, the living room has two windows on its south wall, and adjoins the bathroom in that direction, hence the windows are incorrectly placed.

The requirements of the DSVL were systematically encoded as *mmXtens* extension rules. Then, they were used on the 7 instantiable meta-models to generate models that satisfy the requirements. The results are shown in Table 6.5. While most achieved the most basic requirement (Rq1), they fail for more complex ones. Particularly, none of the students were able to give a successful solution to balconies being adjacent to one room, as well as the topological coherence. The rate of meta-models satisfying the remaining requirements is poor.

It is also worth mentioning that, in the experiment, *mmXtens* and the invoked constraint solver had good performance, producing models within seconds.

The validity of this experiment might present certain threats, though. First students did

Requirement	Correct solutions
1. Houses have at least a living room and a bathroom.	6
2. Balconies can only be adjacent to a single room.	0
3. Entries can be adjacent to three rooms at most.	1
4. Only balconies can have windows in interior walls.	1
5. A room at north/east of another is at south/west of the other.	3
6. Houses have an entry door (at the entry if there is one).	2
7. Rooms are topologically coherent.	0

Table 6.5: Solutions that fulfil the original requirements

not use *mmXtens* to validate their meta-model, and hence we cannot conclude that their meta-models would have been better if they would have done so. Moreover, the model generation has been delegated to a constraint solving facility [67], which performs a bounded search. In the cases that *mmXtens* was not able to produce a model, it means that no model with the given requirements exists within the search scope. Hence, we have used wide search scopes to minimize the risk of having missing a possible meta-model instance due to performing the search in a too narrow scope.

In conclusion, we have seen that with *mmXtens* we could evaluate the conformance of meta-models with regret to the problem statement. The advantage is that we did not have to create tentative examples manually. Furthermore, as most students failed in some way, it could be argued that the availability of a tool like *mmXtens* would have helped in obtaining higher quality solutions. However, this cannot be concluded from the present experiment, and we leave it to future work.

7

Conclusions and Future Work

In Chapter 1, the main gaps in the traditional approach for meta-model and DSML development, and the available environments that support it, were identified. This final chapter goes through all of them and concludes whether and how this dissertation has overcome these limitations. Finally, future work to be developed after its completion is described.

7.1 Conclusions

This PhD has presented a process for the example-based development of Domain-Specific Modelling Languages (DSMLs) and their modelling environments. Like most approaches promoting this kind of development, it involves a Modelling Expert (ME) and a Domain Expert (DE). Moreover, a whole framework for the Validation & Verification (V&V) of DSMLs has been introduced, and integrated in the proposed process.

This work has proposed *metaBup*, a novel approach to the development of meta-models to make MDE more accessible to non-experts. We have introduced a bottom-up approach where a meta-model is induced from model fragments, which may be specified using informal sketching tools. Such sketches are transformed into a specialized textual notation that can be directly used by MEs. Model fragments can be annotated to guide the automatic induction of the meta-model and document the intention of certain elements. The process is iterative, as fragments are added incrementally, causing updates in the meta-model, which can be refactored in the process based on the recommendations provided by a virtual assistant. To involve the Domain Experts in the meta-model validation process, an example-based approach to the testing of meta-models, called *metaBest*, is followed. Finally, the meta-model is compiled for specific platforms and usage purposes, and a fully-working modelling environment mimicing the concrete syntax from fragments is obtained.

In addition, the main goal of this work was proposing means to overcome the identified drawbacks in traditional approaches thus carried by the supporting technology, mainly operating following a *top-down* strategy. Next, we go through the aforementioned challenges and analyze how this work has contributed to overcome them with the proposed *bottom-up* process.

Excessive focus on MDE practitioners Most approaches do not foster the active participation of Domain Experts in the DSML design process. This dissertation has presented a process and tool support for a style of development of DSMLs where Domain Experts are engaged and play an active role along the building process. This is accomplished by letting them express their requirements in the form of fragments, using a graphical syntax with the aid of a drawing tool. The induction of meta-models as of these fragments is automated so the MDE practitioner is still able to perform meta-model editions and refactorings. These tasks are made easier to the Modelling Expert by means of annotation and recommendation systems that automate the application of typical modifications to the meta-model.

The Domain Expert is actively involved in the Validation & Verification of the DSML as well, as the proposed framework enables defining test cases in a graphical format which coincides with the same open syntax used for defining fragments, and also because the engine accounts a system to automatically produce visual examples of the DSML under development, which can be validated by the Domain Expert.

Centralization of the development process over the meta-model *Top-down* DSML construction approaches require building the meta-model upfront, and only then it can be used for building instance models. Even though Modelling Experts get used to work in this way, it often results too demanding for Domain Experts.

As opposed to this strategy, in *metaBup* the DSML development is driven by the examples provided by the Domain Expert, being the meta-model automatically derived from them.

Lack of interaction promotion between Domain Experts and Modelling Experts The proposed approach promotes the interaction of both roles by enabling each one of them to work in a format that they are familiar with (drawings in the case of the Domain Expert, and meta-models in the case of the Modelling Expert). The engine takes care of the subsequent transformations (sketch to fragment and fragment to meta-model), hence reducing the gap

between the two agents.

The interaction is active in testing too. In the case of *mmUnit*, Domain Experts can provide their own test cases, while the Modelling Expert codes the properties to check, hence being both implicated in the inspection of the DSML features. As for *mmXtens*, the Domain Expert can validate meta-model instances shown in a visual syntax, while the Modelling Expert encodes the rules that will constrain the example generation.

All these activities involve both roles and promote active participation in the modelling and testing tasks. The quality of the resulting editor greatly benefits from this interaction, as it is validated from both the domain and the modelling side.

Technology for building graphical DSMLs has a steep learning curve The fact that Domain Experts have to become familiar with DSML development tools, highly difficults working with currently existent solutions. One of the core benefits from the presented approach is that our Domain Experts do not have to become familiar with new technology. Instead, they only need to work with general purpose drawing tools to which most computer users are familiar with. The case study from Section 6.1 proves that users are capable of generating visual editors by sketching fragments with a drawing tool. Furthermore, the environment facilitates adding new drawing tools by means of an extension point.

Lack of V&V mechanisms Chapter 2 explained the absence of integral solutions for DSML Validation & Verification. In this work, V&V is approached from its three perspectives: unit testing (through *mmUnit*), specification-based testing (*mmSpec*) and reverse testing (*mmXtens*). Altogether, these three mechanisms conform the DSML testing framework *metaBest*, which can be used both in a *bottom-up* strategy like *metaBup*, or in any other, as meta-models can be imported to the system any time. These three approaches have been validated from different perspectives (conciseness, performance, expressiveness and usefulness) in Section 6.2.

In conclusion, this work has presented a complete process and tool support for the creation of environments for Domain-Specific Modelling Languages. Making it example-driven has facilitated overcoming the most significant impediments usually found in traditional approaches, mainly due to the fact that Domain Experts are enabled to express their examples using drawings. Modelling Experts also benefit from the approach as they do not need to translate domain requirements into a meta-model; instead, they systematically obtain it from

the examples made by the Domain Expert.

The environment is completed with a testing framework which allows Domain Experts defining test cases and obtain automatically generated examples of the DSML under construction to validate. A third mechanism lets Modelling Experts define properties that the meta-model should fulfill.

Finally, despite overcoming the all the identified challenges, the scope of this work presents a series of **limitations**, being the first of them that the kind of supported DSMLs that can be built using the approach is limited by their size, as no experimental tests have been conducted using big-size fragments (i.e., to the tune of hundreds of elements).

Moreover, although *metaBup* is the only identified solution in the field capable of detecting spatial relationships from sketches, DSMLs of great visual complexity are not supported. The presented approach is capable of detecting the most widespread spatial relationships in MDE practice, but is limited when it comes to complex graphical features with onerous logic behind.

7.2 Future Work

A set of tasks that could extend and improve the proposed approach of this dissertation have been identified:

Real-time validation. A future task is synchronizing the example provision and the visual editor generation. If the visual editor could be systematically updated as of the insertion of fragments, Domain Experts could validate the tool themselves, only reporting necessary meta-model fixes to the Modelling Expert if necessary.

Improve support for the editor evolution. A common scenario might be the manual modification of the Sirius editor model. To avoid overriding these manual changes, techniques similar to [63] may be employed, where manual changes are described as a program that is reapplied when re-generation occurs. In this sense, mechanisms for assessing the quality of the created DSL within the process, in the style of [45, 54, 78, 91], could be integrated.

Extend the annotation and recommendation catalogs. In the future, we will give support to a broader range of refactorings.

Extend testing languages and functionalities. *mmXtens* could be enriched with more sophisticated primitives, e.g. expressions regarding attribute values, or conditions on paths between objects. As for the output of *mmUnit* and *mmSpec*, *quick fixes* or recommendations suggested upon test failures could be added.

Assess quality in meta-model repositories. We will exploit *mmSpec* so as to test the quality in meta-models found at other repositories (like *GitHub*). This can be accomplished by means of a similar experiment to the one conducted in Section 6.2.5.

Asses the quality of concrete syntax. The quality in concrete syntax from DSMLs can be evaluated following a similar approach to the one introduced in Section 4.3. For this purpose, we will have to produce a fourth mechanism for the V&V environment enabling to express and check properties in visual models and/or examples, and conduct a study on quality criteria for concrete syntax, similar to the one carried in Section 6.2.5.

8

Conclusiones y Trabajo Futuro

En el Capítulo 1, hemos identificado los principales inconvenientes del método tradicional para el desarrollo de meta-modelos y DSMLs, así como de los entornos disponibles que dan soporte a los mismos. Este último capítulo vuelve sobre dichos inconvenientes para concluir en qué modo esta tesis ha resuelto dichas limitaciones. Finalmente, se propone el trabajo futuro que se desarrollará tras haber completado esta investigación.

8.1 Conclusiones

En esta tesis, se ha presentado un proceso para el desarrollo de Lenguajes de Modelado de Dominio Específico (DSMLs) basado en ejemplos, y para sus entornos de modelado. Como la mayoría de métodos para este tipo de desarrollo software, el proceso incluye a un Experto en Modelado (ME) y a un Experto en el Dominio (DE). Además, se ha presentado un framework completo para la Validación y Verificación (V&V) de DSMLs.

Así, *metaBup* supone un nuevo método *bottom-up* de desarrollo de meta-modelos para hacer el MDE más accessible a usuarios que no son expertos en la materia. En él, los meta-modelos se deducen partiendo de fragmentos que pueden ser especificados mediante el uso de herramientas informales de dibujo. Dichos dibujos son transformados en texto, conforme a una notación específica que puede ser empleada directamente por el ME. Asimismo, los fragmentos pueden ser anotados para contribuir al proceso de deducción del meta-modelo, así como complementar información en ciertos elementos. El proceso es iterativo, ya que los fragmentos son añadidos al entorno de manera incremental, a la vez que el meta-model se acutaliza. El meta-modelo puede también ser modificado, a partir de una serie de recomendaciones que provee el asistente virtual.

Para implicar a los DEs en el proceso de validación del meta-modelo, se presenta un

método de *testing* también basado en ejemplos.

Finalmente, el meta-modelo se compila a la plataforma específica en la que se quierea utilizar, generando un entorno de modelado plenamente funcional que respeta la sintaxis concreta empleada para dibujar los fragmentos.

El objetivo principal de este trabajo era proponer técnicas que solventaran los principales inconvenientes identificados en las técnicas tradicionales de desarrollo de DSMLs, arrastrados por la tecnología que les da soporte. Dichas técnicas siguen, principalmente, estrategias *top-down*. A continuación, repasamos dichos inconvenientes, analizando cómo este trabajo ha contribuido a superarlos mediante el proceso *bottom-up* propuesto.

Focalización excesiva en expertos en MDE La mayoría de las técnicas estudiadas, no promueven la participación activa de los DEs en el proceso de diseño de los DSMLs. Este trabajo de tesis ha presentado un proceso y una herramienta para un estilo de desarrollo de DSMLs en el que los DEs juegan un papel activo a lo largo de todo el proceso. Esto se consigue permitiéndoles expresar los requisitos software en forma de fragmentos, utilizando una sintáctica gráfica con la ayuda de una herramienta de dibujo. La deducción de meta-modelos a partir de estos fragmentos se automatiza para que el experto en MDE siga pudiendo editar el meta-modelo y reestructurarlo, si lo ve conveniente. El trabajo se presenta además de un modo más sencillo para el ME mediante la introducción de anotaciones y de un sistema de recomendaciones que automatizan la aplicación de modificaciones típicas sobre el meta-modelo.

El DE se involucra de forma activa en la Validación y Verificación del DSML, ya que el framework que proponemos permite definir casos de prueba en formato gráfico, coincidiendo con la misma sintaxis empleada para definir los fragmentos, y también gracias a que el sistema cuenta con un mecanismo que produce automáticamente ejemplos visuales del DSML que se está construyendo, lo que permite que el DE pueda validarlos.

Centralización del proceso sobre el meta-modelo Las técnicas *top-down* requieren construir el meta-modelo por adelantado, y sólo a partir de éste pueden instanciarse modelos. Incluso aunque los MEs estén acostumbrados a trabajar de este modo, a menudo esta mecánica resulta muy complicada para los DEs.

Por el contrario, en *metaBup*, el desarrollo del DSML está dirigido por los ejemplos que

provee el DE, siendo el meta-modelo derivado de forma automática a partir de éstos.

Falta de interacción entre el Experto en Dominio y el Experto en Modelado Nuestra propuesta promueve la interacción de ambos roles permitiendo a cada uno de ellos trabajar en un formato que les sea familiar (dibujos en el caso del DE, y meta-modelos en el caso del ME). Las transformaciones necesarias entre los distintos artefactos (dibujo a fragmento y fragmento a meta-modelo) es automática, lo que reduce la posibilidad de error en la comunicación entre los dos perfiles.

La interacción es activa también en lo que respecta al *testing*. En el caso de *mmUnit*, los DEs proveen sus propios casos de prueba, mientras que los MEs codificará las propiedades que se quieran comprobar, siendo ambos parte de la revisión del DSML. En cuanto a *mmXtens*, el DE puede validar instancias del meta-modelo siéndole éstas presentadas en una sintaxis visual, mientras que el ME será el encargado de codificar las reglas que definirán qué tipo de ejemplo se quiere generar.

Todas estas actividades, implican a los dos agentes del proceso, poniendo los medios para que haya colaboración tanto en el modelado como en las pruebas. Esta interacción beneficia significativamente la calidad del editor resultante, ya que de este modo, la herramienta es validada desde el punto de vista del dominio, y desde el del modelado.

La curva de aprendizaje de la tecnología disponible para construir DSMLs gráficos es muy pronunciada El hecho de que los DEs tengan que familiarizarse con las herramientas de desarrollo de DSMLs, dificulta enormemente trabajar con las soluciones existentes. Uno de los principales beneficios del proceso que aquí se presenta es que los DEs no tienen que pasar por el proceso de familiarizarse con herramietna alguna. En su lugar, pueden trabajar con herramientas de dibujo de propósito general, a las que la mayoría de usuarios de computadoras está más que acostumbrado. El caso de estudio de la Sección 6.1, prueba que los usuarios son capaces de general editores visuales simplemente dibujando los fragmentos con una de estas herramientas de dibujo. Además, el entorno facilita añadir nuevas herramientas de dibujo mediante un punto de extensión.

Ausencia de mecanismo de V&V En el capítulo 2, mencionábamos la ausencia total de soluciones integrales para la Validación y Verificación de DSMLs. En este trabajo, se trata el V&V desde las tres perspectivas conocidas: *unit testing* (mediante *mmUnit*), testing basado en

especificaciones (*mmSpec*) y *reverse testing* (*mmXtens*). Juntos, los tres mecanismos conforman el framework de pruebas para DSMLs *metaBest*, que puede usarse independientemente de si se sigue una estrategia de desarrollo *bottom-up* como *metaBup*, como si se sigue cualquier otra, ya que el sistema permite la importación de meta-modelos. Estas tres perspectivas han sido validadas en cuanto a concisión, rendimiento, expresividad y utilidad. Los resultados pueden consultarse en la Sección 6.2.

En conclusión, en este trabajo hemos presentado un proceso completo junto a una herramienta que soportan la creación de entornos para Lenguajes de Modelado de Dominio Específico. Su planteamiento dirigido por modelos ha solventado los impedimentos más significativos que suelen encontrarse en los enfoques tradicionales, fundamentalmente gracias a que los DEs pueden expresar ejemplos mediante dibujos. Los MEs también se benefician del proceso, ya que no necesitan traducir los requisitos del dominio a un meta-modelo, sino que éste se obtiene sistemáticamente a partir de los ejemplos que provee el DE.

El entorno se completa con un framework de pruebas que permite a los DEs definir casos de prueba y obtener automáticamente ejemplos del DSML que se está construyendo, para su validación. Un tercer mecanismo permite a los MEs definir propiedades que el meta-modelo debería cumplir.

Finalmente, a pesar de haber superado todos los impedimentos que se habían identificado, el ámbito de este trabajo presenta una serie de limitaciones, siendo la primera de ellas que el tipo de DSMLs que pueden construirse están limitados por su tamaño, no habiéndose llevado a cabo experimento alguno en el que se utilizasen fragmentos de gran tamaño (del orden de cientos de elementos distintos).

Además, aunque *metaBup* es la única solución capaz de detectar las relaciones espaciales de los dibujos, los DSMLs de gran complejidad visual no se han validado. El sistema es capaz de detectar las relaciones geométricas más comunes del MDE, pero está limitado en lo que se refiere a propiedades gráficas de lógica compleja.

8.2 Trabajo Futuro

Se han identificado una serie de tareas que podrían extender y mejorar el trabajo que aquí se propone, incluyendo:

Validación en tiempo real. Una tarea futura sería sincronizar la provisión de ejemplos con la generación del editor visual. Se el segundo pudiese ser actualizado sistemáticamente cada vez que se introdujese uno de los primeros, los DEs podrían validar la herramienta generada por sí mismos, solicitando únicamente cambios en el meta-modelo al ME si es necesario.

Mejorar el soporte para la evolución del editor. Podemos esperar la modificación manual del editor de Sirius. Para evitar este tipo de cambios manuales, se podrían emplear técnicas similares a las propuestas en [63], donde se describen cambios manuales como un programa que se re-aplica al regenerar el editor. En este sentido, se podrían también incorporar al proceso mecanismos de validación de la calidad del DSML.

Extender los catálogos de anotaciones y recomendaciones. En el futuro, el sistema dará soporte a un mayor número de refactorizaciones.

Extender los lenguages de *testing* y su funcionalidad. *mmXtens* podría enriquecerse con primitivas más avanzadas, como expresiones sobre el valor de los atributos, o condiciones sobre los caminos entre objetos. En cuanto a la salida que producen tanto *mmUnit* como *mmSpec*, se podrían añadir *quick fixes* y recomendaciones que se sugieresen a partir de éstas.

Evaluar la calidad en repositorios de meta-modelos. Tras la publicación de este trabajo, podríamos emplear llevar a cabo un experimento similar al de la Sección 6.2.5 para comprobar la calidad de los meta-modelos utilizando *mmSpec*.

Evaluar la calidad de la sintaxis concreta. La calidad de la sintaxis concreta de un DSML podría evaluarse procediendo de forma similar a como lo hemos hecho en la Sección 4.3. Para ello, tendríamos que producir un cuarto mecanismo para el entorno de V&V que permitiese expresar y comprobar propiedades en modelos visuales, y llevar a cabo un estudio de criterios de calidad en la sintaxis concreta, similar al que se ha presentado en la Sección 6.2.5.

Appendices

A

OCL equivalence with metaBup constraint annotations

@acyclic

```
1   context <class> inv unique: <class>.allInstances()->size() <= 1
```

@covering

Case 1: the upper bound of the reference is 1.

```
1       context <ref.src> inv acyclic:
2         not self.ref.oclIsUndefined() implies
3         self->closure(ref)->excludes(self)
```

Case 2: the upper bound of the reference is bigger than 1.

```
1       context <ref.src> inv acyclic: self->closure(ref)->excludes(self)
```

@cycleWith

Let assume a reference $ref1$ which has to commute with the sequence of references $ref2$ and $ref3$.

Case 1: the upper bound of all references is 1.

```
1     context <ref1.src> inv cycleWith:
2       if self.ref1.oclIsUndefined() then true
3       else
4           if self.ref1.ref2.oclIsUndefined() then false
5           else self.ref1.ref2.ref3 = self
6           endif
7       endif
```

Case 2: the upper bound of all references is bigger than 1.

```
1     context <ref1.src> inv cycleWith:
2     self.ref1->forAll(r1 | r1.ref2->exists(r2 | r2.ref3->includes(self)))
```

@irreflexive

Case 1: the upper bound of the reference is 1.

```
1       context <ref.src> inv irreflexive: self.ref<>self
```

Case 2: the upper bound of the reference is bigger than 1.

```
1       context <ref.src> inv irreflexive: self.ref->excludes(self)
```

@nand

Let assume two references $ref1$ and $ref2$. The upper bound of $ref1$ is 1, and the upper bound of $ref2$ is > 1.

Case 1: both references have the same source.

```
1       context <ref1.src> inv nand:
2         self.ref1.oclIsUndefined() or self.ref2->isEmpty()
```

Case 2: both references have the same target

```
1       context <ref1.tar> inv nand:
2         (not <ref1.src>.allInstances()->exists(o | o.ref1 = self)) or
3         (not <ref2.src>.allInstances()->exists(o | o.ref->includes(self)))
```

@subset

Case 1: upper bound of annotated reference $(ref1) = 1$.

```
1       context <ref1.src> inv subset:
2         not self.ref1.oclIsUndefined() implies self.ref2->includes(self.ref1)
```

Case 2: upper bound of annotated reference $(ref1) > 1$.

```
1       context <ref1.src> inv subset: self.ref2->includesAll(self.ref1)
```

@tree

```
1       context <ref.src> inv tree:
2         self->closure(ref)->excludes(self) and <ref.src>.allInstances()
3         ->collect(ref)->flatten()->count(self) <= 1
```

@unique

```
1     context <class> inv unique:
2       <class>.allInstances()->size() <= 1
```

@xor

Let assume two references $ref1$ and $ref2$. The upper bound of $ref1$ is 1, and the upper bound of $ref2$ is > 1.

```
1   context <ref1.src> inv xor:
2     Sequence{self.ref1}->one(not self.oclIsUndefined()) xor
3     Sequence{self.ref2}->one(not self->isEmpty())
```

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 B.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 B.2: Excerpt of Library of EMF-based assertions

C

OCL encoding of mmSpec primitives

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:	
	1 **some class** => exist.	1 EClass.**allInstances**()−>**notEmpty**()
	(2) Some attribute exists:	
	1 **some attribute** => exist.	1 EAttribute.**allInstances**()−>**notEmpty**()
	(3) Some reference exists:	
	1 **some reference** => exist.	1 EReference.**allInstances**()−>**notEmpty**()

	mmSpec	**OCL**
Element name	(4) There is an element named *x*:	
	1 **some element** => **name** = x.	1 ENamedElement.**allInstances**() 2 −>**exists**(e | e.name = 'x')
	(5) The name of some element starts with *a*:	
	1 **some element** => **name**{prefix} = **a**.	1 ENamedElement.**allInstances**() 2 −>**exists**(e | 3 **if** e.name.**oclIsUndefined**() **then** false 4 **else** e.name.**indexOf**('a') = 1 **endif**)
	(6) The name of some element contains *a*:	
	1 **some element** => **name**{infix} = **a**.	1 ENamedElement.**allInstances**() 2 −>**exists**(e | 3 **if** e.name.**oclIsUndefined**() **then** false 4 **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**
	1 **some element** => **name**{suffix} = **a**.	1 ENamedElement.**allInstances**() 2 −>**exists**(e | 3 **if** e.name.**oclIsUndefined**() **then** false 4 **else** (e.name.**indexOf**('a') <> 0 **and** 5 e.name.**indexOf**('a') = 6 (e.name.**size**() − 'a'.**size**() +1)) **endif**)
	(8) The name of some element is a verb / noun / adjective:	
	1 **some element** => **name** = verb. 2 **some element** => **name** = **noun**. 3 **some element** => **name** = adjective.	*not supported*
	(9) The name of some element is a synonym of *x*:	
	1 **some element** => **name** = **synonym**{x}.	*not supported*
	(10) The name of an elem. is a camel/pascal phrase ("likeThis" or "LikeThis"):	
	1 **some element** => **name** = **camel**−**phrase**. 2 **some element** => **name** = **pascal**−**phrase**.	*not supported*

	mmSpec	**OCL**
Class abstractness	(11) Some class is abstract:	
	1 **some class** => **abstract**.	1 EClass.**allInstances**() 2 −>**exists**(c | c.abstract)

	mmSpec	**OCL**
Class features	(12) Some class contains a feature (attribute or reference) named *x*:	
	1 **some class** => **with a feature**{**name** = x}.	1 EClass.**allInstances**() 2 −>**exists**(c | 3 c.eStructuralFeatures−>**exists**(f | 4 f.name = 'x'))

	mmSpec	**OCL**
Class inheritance	(13) Some class is subclass of another:	
	1 **some class** => **sub**−**to some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | c.eSuperTypes−>**notEmpty**())
	(14) Some class is subclass of another in at most *n* steps in the hierarchy:	
	1 **some class** 2 => **sub**−**to**{**depth**=[1,n]} **some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | **Sequence**{1..n} 3 −>**iterate**(i:**Integer**; super:**Set**(EClass)=**Set**{} | 4 super−>**union**(super−>**including**(c) 5 −>**select**(c2 | 6 c2.eSuperTypes−>**notEmpty**()) 7 −>**collect**(c2 | 8 c2.eSuperTypes)−>**asSet**())) 9 −>**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**
	1 **some class** => **cont**−leaf.	1 EClass.**allInstances**() 2 −>**exists**(c | 3 **not** c.eReferences−>**exists**(ref | 4 ref.containment = true) **and** 5 EClass.**allInstances**()−>**exists**(contclass | 6 contclass.eReferences−>**exists**(ref | 7 ref.containment = true **and** 8 ref.eType = c)))

	mmSpec	**OCL**
Class reachability	(21) Some class contains another:	
	1 **some class** 2 => **reach**{**cont**} **some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | 3 c.eAllReferences−>**reject**(ref | 4 ref.containment = false) 5 −>**notEmpty**())
	(22) Some class is contained in another:	
	1 **some class** 2 => **reached**−**from**{**cont**} **some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | 3 EReference.**allInstances**()−>**exists**(ref | 4 ref.eType = c **and** 5 ref.containment = true))
	(23) Some class reaches another:	
	1 **some class** 2 => **reach some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | c.eAllReferences−>**notEmpty**())
	(24) Some class reaches another in up to *n* jumps:	
	1 **some class** 2 => **reach**{**jumps**=[1,n]} **some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | **Sequence**{1..n} 3 −>**iterate**(i:**Integer**; reaches:**Set**(EClass)=**Set**{} | 4 reaches−>**union**(reaches−>**including**(c) 5 −>**select**(c2 | 6 c2.eAllReferences−>**notEmpty**()) 7 −>**collect**(c2 | 8 EClass.**allInstances**() 9 −>**select**(c3 | 10 c2.eAllReferences−>**exists**(ref | 11 ref.eType = c3)))−> **asSet**()))) 12 −>**notEmpty**())
	(25) Some class is reached from another:	
	1 **some class** 2 => **reached**−**from some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | EReference.**allInstances**() 3 −>**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**
	1 **some class** 2 => **reached**−**from**{**jumps**=[1,n]} 3 **some class**.	1 EClass.**allInstances**() 2 −>**exists**(c | **Sequence**{1..n} 3 −>**iterate**(i:**Integer**; targets:**Set**(EClass)=**Set**{} | 4 targets−>**union**(targets−>**including**(c) 5 −>**select**(c2 | 6 EClass.**allInstances**()−>**exists**(tar | 7 tar.eAllReferences−>**exists**(ref | 8 ref.eType = c2))) 9 −>**collect**(c2 | 10 EClass.**allInstances**()−>**select**(tar | 11 tar.eAllReferences−>**exists**(ref | 12 ref.eType = c2)))−>**asSet**())) 13 −>**notEmpty**())

	mmSpec	**OCL**
Feature class	(27) Some class has a feature:	
	1 **some class** => **with some feature**.	1 EClass.**allInstances**() 2 −>**exists**(c | c.eAllStructuralFeatures−>**notEmpty**() ↪)
	(28) Some class has a feature, not inherited from its superclasses:	
	1 **some class** => **with**{!**inh**} **some feature**.	1 EClass.**allInstances**() 2 −>**exists**(c | c.eStructuralFeatures−>**notEmpty**())

	mmSpec	**OCL**
Feature multiplicity	(29) Some feature has a minimum/maximum multiplicity between *n* and *m*:	
	1 **some feature** => **multiplicity**{**min**=[n,m]}. 2 **some feature** => **multiplicity**{**max**=[n,m]}.	1 EStructuralFeature.**allInstances**() 2 −>**exists**(f | f.lowerBound >= n **and** 3 f.lowerBound <= m) 4 5 EStructuralFeature.**allInstances**() 6 −>**exists**(f | f.upperBound >= n **and** 7 f.upperBound <= m)

	mmSpec	**OCL**
Attribute type	(30) Some attribute has a primitive type *t*:	
	1 **some attribute** => type = t.	1 EAttribute.**allInstances**() 2 −>**exists**(attribute | attribute.eType.name = 't')

	mmSpec	**OCL**
Reference ends	(31) Some reference starts in a class:	
	1 **some reference** => **from a class**.	1 EReference.**allInstances**() 2 −>**exists**(ref | EClass.**allInstances**() 3 −> **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))

D

An mmSpec library of meta-model quality properties

```
1  define hierarchySpreadAttribute : not every class{and{!abstract, sub−to a class{and{abstract, name = <?:className, "the
       class name">}}}} => with{!inh} 1 attribute{name = <?:attName>}.
2
3  define overridenAttribute : no class{with{!inh} 1 attribute{name=<?:attName, "the attribute name">}} => sub−to a
       class{with{!inh} 1 attribute{name=<?:attName, "the attribute name">}}.
4
5  define synonymClassNames : no class {!name=<?:synClass, "the class name">} => name = synonym{<?:synClass, "the
       class name">}.
6
7  define attNameContainsClassName : no attribute {owned−by{!inh} a class{name = <?:className, "the class name">}} =>
        name{infix} = <?:className, "the class name">.
8
9  define classDataType : no attribute => type = <?:className, "the class name">.
10
11 define doubleCompulsoryContainment : no class{name = <?:className, "the class name">} => and{
12                                          reached−from{strict, !inh, cont, jumps=[1,1]} 2 class,
13                                          with{!inh} a reference{and{
14                                                      multiplicity{min=[1,∗)},
15                                                      opposite−to a reference{cont}
16                                                    }}}.
17 define doubleContainment : no class{name = <?:className, "the class name">} => and{
18                                          reached−from{strict, cont, jumps=[1,1]} 2 class,
19                                          with a reference{opposite−to a reference{cont}}}.
20
21 define containmentAndInheritance :
22   no class{name = <?:className, "the class name">}
23     => super−to a class{with{!inh} a reference{and{cont, to a class{name = <?:className, "the class name">}, opposite
           −to a reference{multiplicity{min=[1,∗)}}}}}}.
24
25
26
27 define mutualReference : no class{name = <?:className, "the class name">} => with{!inh} a reference{and{!@opposite, !
       cont, to a class{and{!name = <?:className, "the class name">, with{!inh} a reference {and{!@opposite, !cont, to a
       class{name = <?:className, "the class name">}}}}}}}.
28
29
30 define noContCycle : no class {name = <?:className, "the class name">} => reach{strict, cont, !inh} a class{name =
       <?:className, "the class name">}.
```

```
31
32 define everyClassContained : a class{and{name = <?:className, "the class name">, !cont−root }} => reached−from{
       cont} some class.
33
34
35 −− "Design"
36 test(project){
37
38   −− "D01: An attribute is not repeated among all specific classes of a hierarchy"
39   hierarchySpreadAttribute(className = every class{super−to [2,∗) class}, attName = every attribute{owned−by{!inh} a
         class{sub−to some class}}).
40
41   −− "D02: There are no isolated classes"
42   no class => and{sub−to no class, super−to no class, reach no class, reached−from no class}.
43
44   −− "D03: No abstract class is super to strictly one other class"
45   no class{abstract} => super−to strictly 1 class.
46
47   −− "D04: There are no composition cycles"
48   noContCycle(className = every class).
49
50   −− "D05: There are no irrelevant classes (abstract sub to non−abstract)"
51   no class{abstract} => sub−to a class{!abstract}.
52
53   −− "D06: No binary association is composite in both member ends"
54   no reference{cont} => opposite−to some reference{cont}.
55
56   −− "D07: No overriden inherited attributes"
57   overridenAttribute(attName = every attribute).
58
59   −− "D08: Every feature has a greater than 0 max multiplicity"
60   no feature => multiplicity{max=[0,0]}.
61
62   −− "D09: No class is contained in two classes, compulsorily in one of them"
63   doubleCompulsoryContainment(className = every class).
64
65   −− "D10: No class contains at least one of its superclasses"
66   containmentAndInheritance(className = every class).
67
68 }
69
70
71 −− "Best practices"
72 test(project){
73
74   −− "B01: No redundant generalizations"
75   no class => sub−to{width=[2,∗)} some class.
76
77   −− "B02: Every class can be instanced"
78   no class{abstract} => super−to no class{!abstract}.
79
80   −− "B03: There is a root class that contains all others"
81   strictly 1 class{cont−root{absolute}} => exist.
82
83   −− "B04: No class is contained in two classes"
```

```
84    doubleContainment(className = every class).
85
86    -- "B05: No class that's not involved in any association is super to any other"
87    no class{and{reach{strict, !inh} no class, reached-from{strict, !inh} no class, sub-to no class, !abstract}} => super-to
          some class.
88
89    -- "B06: Two classes don't refer each other"
90    mutualReference(className = every class).
91
92  }
93
94
95  -- "Naming"
96  test (project){
97
98    -- "N01: Attributes are not named after their feature classes"
99    attNameContainsClassName(className = every class).
100
101   -- "N02: Attributes are not potential associations (named after a class)"
102   no attribute => name = some class.
103
104   -- "N03: Every binary association is named with a verb phrase"
105   every reference {opposite-to some reference} => or{ name = camel-phrase{start{verb}},
106                                 name = pascal-phrase{start{verb}}}.
107
108   -- "N04: Every class is named in pascal-case, with a singular-head noun phrase"
109   every class => name = pascal-phrase{end{noun{singular}}}.
110
111   -- "N05: Element names are not too complex to process"
112   every element => name{size} = [0, <?:maxNamesize, "the max name size", 50>].
113
114   -- "N06: Features are named using camel-case"
115   every feature => name = camel-phrase.
116
117   -- "N07: Every non-boolean attribute has a noun-phrase name"
118   every attribute{!type=BooleanType} => name = camel-phrase{end{noun}}.
119
120   -- "N08: Every boolean attribute has a verb-phrase name"
121   every attribute{type=BooleanType} => name = camel-phrase{start{verb}}.
122
123   -- "N09: No class is named with a synonym to another class name"
124   synonymClassNames(synClass = every class).
125
126 }
127
128
129 -- "Metrics"
130 test(project){
131
132   -- "M01: No class is overloaded with attributes"
133   every class => with{!inh} [0, <?:maxAttCount, "the max attributes allowed", 10>] attribute.
134
135   -- "M02: No class reaches too many classes"
136   every class => reach{strict, jumps=[1,1], !inh} [0, <?:maxRefCount, "the max attributes allowed", 5>] class.
137
```

```
138   -- "M03: No class is reached from too many classes"
139   every class => reached-from{strict, jumps=[1,1], !inh} [0, <?:maxRefCount, "the max attributes allowed", 5>] class.
140
141   -- "M04: No hierarchy is too deep"
142   no class => sub-to{depth= [<?:d, "Min forbidden inheritance depth", 5>, *)} some class.
143
144   -- "M05: A class accounts too many direct children"
145   no class => super-to{depth=[1,1]} <?:c, "Unallowed subclass number", 11> class.
146 }
```

E

Questionnaire for user validation of *metaBup*

This appendix contains the questionnaires used in the evaluation. Questionnaire 1 was answered by all the participants, in order to convey their opinions on the first version of the generated modelling tool. Questionnaire 2 was only answered by those participants that opted for generating a second version of the modelling tool. Mandatory questions are marked with an asterisk.

Questionnaire 1:

Q1 *Indicate your age * :* _____

Q2 *Indicate your gender * :*

- ○ *Male*
- ○ *Female*

Q3 *Indicate your current workplace * :*

- ○ *University*
- ○ *Information technology company*
- ○ *Different sector*
- ○ *Unemployed*

Q4 *Does the synthesized modelling environment meet the graphical syntax you envisioned when providing the examples? **
Not at all ○*1* ○*2* ○*3* ○*4* ○*5* *Very much*

Q5 *Which aspects of the graphical language are not correctly captured by the modelling environment?* _____

Q6 *Which aspects of the graphical language are best captured by the modelling environment?*

Q7 *How easy is it to use the modelling environment? **
Very difficult ○*1* ○*2* ○*3* ○*4* ○*5* *Very easy*

Q8 *Which aspects of the environment would you improve?* _____

Q9 *Which aspects of the environment do you like the most?* _____

Q10 *Which is your higher level of expertise with modelling? **

- ○ *I have developed both meta-models and graphical domain-specific languages.*
- ○ *I have developed meta-models, but not graphical domain-specific languages.*
- ○ *I have used domain-specific languages like UML or BPMN.*
- ○ *I have never used or developed models or meta-models.*

Answer the following questions only if you selected one of the first two choices in the previous question (Q10).

Q11 *Is the meta-model generated from the examples similar to the one you would have built by hand? **
Not at all ○*1* ○*2* ○*3* ○*4* ○*5* *Very much*

Q12 *Which aspects does the meta-model not capture correctly? **_____

Q13 *Which approach would you prefer to build the meta-model? **

- ○ *I would prefer designing the meta-model myself.*
- ○ *I would prefer using examples, and then being able to modify the meta-model manually.*
- ○ *I would prefer using examples, and I do not think necessary to modify the meta-model manually.*

Questionnaire 2:

Q1 *Has the editor quality been improved with respect to the first iteration?*

Not at all ○*1* ○*2* ○*3* ○*4* ○*5* *Very much*

Q2 *Which aspects of the language are still not reflected in the editor?*

- ○ *None*
- ○ *Other:* _____

Bibliography

[1] Lukman Ab.Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015.

[2] David Aguilera, Raúl García-Ranea, Cristina Gómez, and Antoni Olivé. An eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops*, volume 6999 of *LNCS*, pages 323–327. Springer, 2011.

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

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

[5] David Aguilera, Cristina Gómez, and Antoni Olivé. Enforcement of conceptual schema quality issues in current integrated development environments. In *CAiSE*, volume 7908 of *LNCS*, pages 626–640. Springer, 2013.

[6] Thomas Allweyer. *BPMN 2.0*. BoD, 2010.

[7] Xavier Amatriain and Pau Arumí. Frameworks generate domain-specific languages: A case study in the multimedia domain. *IEEE Trans. Software Eng.*, 37(4):544–558, 2011.

[8] Marco Autili, Antonia Bertolino, Guglielmo De Angelis, Davide Di Ruscio, and Alessio Di Sandro. A tool-supported methodology for validation and refinement of early-stage domain models. *IEEE Trans. Software Eng.*, 42:2–25, 2016.

[9] Omar Bahy Badreddin, Andrew Forward, and Timothy C. Lethbridge. A test-driven approach for developing software languages. In *MODELSWARD*, pages 225–234. SciTePress, 2014.

[10] Islem Baki and Houari A. Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Trans. Softw. Eng. Methodol.*, 25(3):20, 2016.

[11] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*, volume 1. The MIT Press, 2000.

[12] Zoltán Balogh and Dániel Varró. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2009.

[13] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, January 2002.

[14] Richard Barker. *Case*Method: Entity Relationship Modelling*. Addison-Wesley Professional, 1990.

[15] Kent Beck. Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, 1994.

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

[17] Manuel F. Bertoa and Antonio Vallecillo. Quality Attributes for Software Metamodels. In *13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010)*, 2010.

[18] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[19] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.

[20] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[21] 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.

[22] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[23] Hyun Cho, Jeffrey G. Gray, and Eugene Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE 2012, Zurich, Switzerland, June 2-3, 2012*, pages 22–28, 2012.

[24] Antonio Cicchetti, D.D. Ruscio, Dimitrios S. Kolovos, and Alfonso Pierantonio. A test-driven approach for metamodel development. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 319–342. IGI Global, 2012.

[25] Jesús Sánchez Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin. Deriving OCL optimization patterns from benchmarks. *ECEASST*, 15, 2008.

[26] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[27] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *FASE*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[29] Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12), December 2004.

[30] Maged Elaasar, Lionel C. Briand, and Yvan Labiche. Domain-specific model verification with QVT. In *ECMFA*, volume 6698 of *LNCS*, pages 282–298. Springer, 2011.

[31] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[32] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2):185–203, 2009.

[33] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

[34] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests.* Addison-Wesley Professional, 1st edition, 2009.

[35] Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey. Verifying modelling languages using lightning: a case study. In *MODEVVA'14*, volume 1235 of *CEUR Workshop Proceedings*, pages 19–28. CEUR-WS.org, 2014.

[36] Antonio García-Domínguez, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Inmaculada Medina-Bulo. EUnit: A unit testing framework for model management tasks. In *MoDELS*, volume 6981 of *LNCS*, pages 395–409. Springer, 2011.

[37] J. García, Félix Óscar' García', Vicente Pelechano, Antonio Vallecillo, Juan Manuel Vara, and Cristina Vicente-Chicote. *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas*. Ra-Ma Editorial, 2013.

[38] Ana Gabriela Garis and Alejandro Sanchez. Verification and validation of domain specific languages using alloy. In *XXI Congreso Argentino de Ciencias de la Computación (Junín, 2015)*, 2015.

[39] Antonio Garmendia, Ana Pescador, Esther Guerra, and Juan de Lara. Towards the generation of graphical modelling environments aided by patterns. In *SLATE*, volume 563 of *CCIS*, pages 160–168. Springer, 2015.

[40] Adnane Ghannem, Ghizlane El-Boussaidi, and Marouane Kessentini. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process*, 26(7):692–713, 2014.

[41] GMF. `http://www.eclipse.org/modeling/gmp/`.

[42] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398, 2005.

[43] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *MoDELS*, pages 24–34. ACM, 2016.

[44] J. J. C. Gomez, B. Baudry, and H. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 131–140, April 2012.

[45] David Granada, Juan M Vara, Veronica A Bollati, and Esperanza Marcos. Enabling the development of cognitive effective visual dsls. In *International Conference on Model Driven Engineering Languages and Systems*, pages 535–551. Springer, 2014.

[46] Graphiti. `https://eclipse.org/graphiti/`.

[47] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009. See also `http://www.eclipse.org/modeling/gmp/`.

[48] Nicolas Hili. A metamodeling framework for promoting flexibility and creativity over strict model conformance. In *FlexMDE @ MoDELS*, volume 1694 of *CEUR Workshop Proceedings*, pages 2–11. CEUR-WS.org, 2016.

[49] Javier Luis Cánovas Izquierdo and Jordi Cabot. Community-driven language development. In *MISE'12*, pages 29–35. IEEE CS, 2012.

[50] Javier Luis Cánovas Izquierdo, Jordi Cabot, Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Engaging end-users in the collaborative development of domain-specific modelling languages. In *CDVE13*, volume 8091 of *LNCS*, pages 101–110. Springer, 2013.

[51] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Automatically reasoning about metamodeling. *Software & Systems Modeling*, 14(1):271–285, 2015.

[52] JetBrains MPS. `https://www.jetbrains.com/mps/`.

[53] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1–2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[54] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4):1505–1526, 2015.

[55] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[56] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Model transformation by-example: A survey of the first wave. In *Conceptual Modelling and Its Theo. Foundations*, volume 7260 of *LNCS*, pages 197–215. Springer, 2012.

[57] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. Design guidelines for domain specific languages. In *DSM'09*, pages 7–13, 2009.

[58] Lennart C.L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: Enabling test-driven language development. In *OOPSLA '11*, pages 139–154, New York, NY, USA, 2011. ACM.

[59] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

[60] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[61] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

[62] Andrew Jensen Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015.

[63] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255, 2017.

[64] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. *The Epsilon Transformation Language*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[65] Dimitrios S. Kolovos, Richard F. Paige, and Fiona 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.

[66] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77 – 91, 2016.

[67] Mirco Kuhlmann and Martin Gogolla. From UML and OCL to relational logic and back. In *MODELS*, volume 7590 of *LNCS*, pages 415–431. Springer, 2012.

[68] Ye Liu, Sören Höglund, Ali Hanzala Khan, and Ivan 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.

[69] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Example-driven meta-model development. *Software and System Modeling*, 14(4):1323–1347, 2015.

[70] Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra, and Juan de Lara. Example-based generation of graphical modelling environments. In Andrzej Wąsowski and Henrik Lönn, editors, *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, LNCS, pages 101–117, Cham, 2016. Springer International Publishing.

[71] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. Assessing the quality of meta-models. *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, page 10, 2014.

[72] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. Meta-model validation and verification with metabest. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 831–834, New York, NY, USA, 2014. ACM.

[73] Jesús J López-Fernández, Esther Guerra, and Juan de Lara. Example-based validation of domain-specific visual languages. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 101–112. ACM, 2015.

[74] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. Combining unit and specification-based testing for meta-model validation and verification. *Inf. Syst.*, 62:104–135, 2016.

[75] Nicolas Mangano, Alex Baker, Mitch Dempsey, Emily Navarro, and André van der Hoek. Software design sketching with calico. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 23–32, New York, NY, USA, 2010. ACM.

[76] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[77] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[78] Daniel L. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009.

[79] OMG. HUTN version 1.0. `http://www.omg.org/spec/HUTN/`.

[80] OMG. UML 2.4.1 specification. `http://www.omg.org/spec/UML/2.4.1/`.

[81] OMG. OCL 2.4. `http://www.omg.org/spec/OCL/`, 2014.

[82] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Specification-driven development of an executable metamodel in eiffel. *In Proc. Workshop in Software Model Engineering 2004, co-located with UML*, 2004.

[83] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 162–171, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Roly Perera. First-order interactive programming. In *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*, PADL'10, pages 186–200, Berlin, Heidelberg, 2010. Springer-Verlag.

[85] Ana Pescador and Juan de Lara. Dsl-maps: From requirements to design of domain-specific languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 438–443, New York, NY, USA, 2016. ACM.

[86] QVT. `http://www.omg.org/spec/QVT/`.

[87] Louis M. Rose, Dimitrios S. Kolovos, and Richard F. Paige. Eugenia live: A flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 15–20, New York, NY, USA, 2012. ACM.

[88] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. Constructing models with the human-usable textual notation. In *MODELS*, volume 5301 of *LNCS*, pages 249–263. Springer, 2008.

[89] Daniel A. Sadilek and Stephan Weißleder. Testing metamodels. In *ECMDA-FA*, volume 5095 of *LNCS*, pages 294–309. Springer, 2008.

[90] Ugo Braga Sangiorgi and Simone D. Junqueira. SKETCH: Modeling using freehand drawing in eclipse graphical editors. In *FlexiTools @ ICSE*, 2010.

[91] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software & Systems Modeling*, pages 1–36, 2015.

[92] Stefan Sobernig, Bernhard Hoisl, and Mark Strembeck. Requirements-driven testing of domain-specific core language models using scenarios. In *QSIC*, pages 163–172. IEEE, 2013.

[93] Stefan Sobernig, Bernhard Hoisl, and Mark Strembeck. Extracting reusable design decisions for uml-based domain-specific languages: A multi-method study. *Journal of Systems and Software*, 113:140–172, 2016.

[94] Jean-Sébastien Sottet and Nicolas Biri. JSMF: a javascript flexible modelling framework. In *FlexMDE @ MoDELS*, volume 1694 of *CEUR Workshop Proceedings*, pages 42–51. CEUR-WS.org, 2016.

[95] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008.

[96] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, June 1974.

[97] Yu Sun, Jeff Gray, and Jules White. A demonstration-based model transformation approach to automate model scalability. *Software & Systems Modeling*, 14(3):1245–1271, 2015.

[98] Albert Tort and Antoni Olivé. An approach to testing conceptual schemas. *Data Knowl. Eng.*, 69(6):598–618, 2010.

[99] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238, July 2014.

[100] Markus Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace, 2013.

[101] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

[102] Hao Wu, Rosemary Monahan, and James F. Power. Exploiting attributed type graphs to generate metamodel instances using an smt solver. In *2013 International Symposium on Theoretical Aspects of Software Engineering*, pages 175–182, July 2013.

[103] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw., Pract. Exper.*, 38(10):1073–1103, 2008.

[104] Dustin Wüest, Norbert Seyff, and Martin Glinz. Flexisketch team: Collaborative sketching and notation creation on the fly. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 685–688, May 2015.

[105] Dina Zayan, MichałAntkiewicz, and Krzysztof Czarnecki. Effects of using examples on structural model comprehension: A controlled experiment. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 955–966, New York, NY, USA, 2014. ACM.

[106] Athanasios Zolotas, Dimitris S. Kolovos, Nicholas Drivalos Matragkas, and Richard F. Paige. Assigning semantics to graphical concrete syntaxes. In *XM @ MoDELS*, volume 1239 of *CEUR Workshop Proceedings*, pages 12–21. CEUR-WS.org, 2014.