Engineering Scalable Modelling Languages

Antonio Garmendia Jorge

Supervisors: Esther Guerra, Ph.D
Juan de Lara, Ph.D

Department of Computer Science
Autonomous University of Madrid

This dissertation is submitted for the degree of
Doctorado en Ingeniería Informática y Telecomunicaciones

August 2019

Abstract

Model-Driven Engineering (MDE) aims at reducing the cost of system development by raising the level of abstraction at which developers work. MDE-based solutions frequently involve the creation of Domain-Specific Modelling Languages (DSMLs). While the definition of DSMLs and their (sometimes graphical) supporting environments are recurring activities in MDE, they are mostly developed ad-hoc from scratch. The construction of these environments requires high expertise by developers, which currently need to spend large efforts for their construction.

This thesis focusses on the development of scalable modelling environments for DSMLs based on patterns. For this purpose, we propose a catalogue of modularity patterns that can be used to extend a modelling language with services related to modularization and scalability. More specifically, these patterns allows defining model fragmentation strategies, scoping and visibility rules, model indexing services, and scoped constraints. Once the patterns have been applied to the meta-model of a modelling language, we synthesize a customized modelling environment enriched with the defined services, which become applicable to both existing monolithic legacy models and new models.

A second contribution of this thesis is a set of concepts and technologies to facilitate the creation of graphical editors. For this purpose, we define heuristics which identify structures in the DSML abstract syntax, and automatically assign their diagram representation. Using this approach, developers can create a graphical representation by default from a meta-model, which later can be customised.

These contributions have been implemented in two Eclipse plug-ins called EMF-Splitter and EMF-Stencil. On one hand, EMF-Splitter implements the catalogue of modularity patterns and, on the other hand, EMF-Stencil supports the heuristics and the generation of a graphical modelling environment. Both tools were evaluated in different case studies to prove their versatility, efficiency, and capabilities.

Keywords: Model-Driven Engineering, Meta-Modelling Patterns, Domain-Specific Modelling Languages, Meta-Modelling, Graphical Modelling Environments, Scalability, Modularity

Resumen

El Desarrollo de Software Dirigido por Modelos (MDE, por sus siglas en inglés) tiene como objetivo reducir los costes en el desarrollo de aplicaciones, elevando el nivel de abstracción con el que actualmente trabajan los desarrolladores. Las soluciones basadas en MDE frecuentemente involucran la creación de Lenguajes de Modelado de Dominio Específico (DSML, por sus siglas en inglés). Aunque la definición de los DSMLs y sus entornos gráficos de modelado son actividades recurrentes en MDE, actualmente en la mayoría de los casos se desarrollan ad-hoc desde cero. La construcción de estos entornos requiere una alta experiencia por parte de los desarrolladores, que deben realizar un gran esfuerzo para construirlos.

Esta tesis se centra en el desarrollo de entornos de modelado escalables para DSML basados en patrones. Para ello, se propone un catálogo de patrones de modularidad que se pueden utilizar para extender un lenguaje de modelado con servicios relacionados con la modularización y la escalabilidad. Específicamente, los patrones permiten definir estrategias de fragmentación de modelos, reglas de alcance y visibilidad, servicios de indexación de modelos y restricciones de alcance. Una vez que los patrones se han aplicado al meta-modelo de un lenguaje de modelado, se puede generar automáticamente un entorno de modelado personalizado enriquecido con los servicios definidos, que se vuelven aplicables tanto a los modelos monolíticos existentes, como a los nuevos modelos.

Una segunda contribución de esta tesis es la propuesta de conceptos y tecnologías para facilitar la creación de editores gráficos. Para ello, definimos heurísticas que identifican estructuras en la sintaxis abstracta de los DSMLs y asignan automáticamente su representación en el diagrama. Usando este enfoque, los desarrolladores pueden crear una representación gráfica por defecto a partir de un meta-modelo.

Estas contribuciones se implementaron en dos plug-ins de Eclipse llamados EMF-Splitter y EMF-Stencil. Por un lado, EMF-Splitter implementa el catálogo de patrones y, por otro lado, EMF-Stencil implementa las heurísticas y la generación de un entorno

de modelado gráfico. Ambas herramientas se han evaluado con diferentes casos de estudio para demostrar su versatilidad, eficiencia y capacidades.

Palabras clave: Desarrollo Dirigido por Modelos, Patrones de Meta-modelado, Lenguajes de Modelado de Dominio Específico, Entornos Gráficos de Modelado, Escalabilidad, Modularidad

Table of contents

List of figures

List of tables

Abbreviations

ADM Architecture-Driven Modernization. 10

ATL ATLAS Transformation Language. 20

CDO Connected Data Objects. 23, 24, 93, 94, 96, 121

CMOF Complete MOF. 10

DSL Domain-Specific Language. 20, 21, 117

DSML Domain-Specific Modelling Language. 1–4, 7–10, 12, 17, 18, 20–22, 25, 27, 29–32, 36–38, 42, 51, 52, 54, 55, 60, 62, 65–68, 76, 78, 110, 116, 117, 120, 122

EMF Eclipse Modeling Framework. 2, 18–20, 24–26, 39, 68, 74, 79, 91, 97, 104, 111, 114, 116, 121

EMOF Essential MOF. 10, 11, 18

EMP Eclipse Modeling Project. 18–20

ETL Epsilon Transformation Language. 20

GMF Graphical Modeling Framework. 25, 39

JDT Java Development Tools. 42

MDA Model-Driven Architecture. 9, 10

MDE Model-Driven Engineering. 1, 2, 7, 9, 10, 12, 13, 15, 17, 18, 20, 21, 24, 25, 29, 120

MOF Meta-Object Facility. 10, 11, 20

OCL Object Constraint Language. 12–14, 19, 20, 22, 33, 36, 39, 44, 45, 50, 53, 66, 67, 71, 122

OMG Object Management Group. 9–11, 18, 20, 113

OOP Object-Oriented Programming. 9

SDF Syntax Definition Formalism. 20

UML Unified Modelling Language. 2, 10, 11, 17, 19, 91

XMI XML Metadata Interchange. 11, 12, 19, 22, 86, 91, 95

XML Extensible Markup Language. 11, 12, 26

Chapter 1

Introduction

The purpose of this chapter is to provide a general overview of the dissertation. Section 1.1 describes the motivation of this work, regarding difficulties that have been detected in software development when the Model-Driven Engineering (MDE) paradigm is applied, and also the shortcomings addressed in this thesis. Next, Section 1.2 summarizes the publications derived from this work. Section 1.3 briefly reports on a research stay performed during the thesis. Section 1.4 describes the funding received for the realization of this thesis and finally, Section 1.5 explains the organization of the rest of this document.

1.1 Motivation

The MDE paradigm proposes software development by the use of models of a higher level of abstraction than code [117]. Hence, models are used to automate many activities, like code generation, system simulation or testing. One of the important concepts within MDE are Domain-Specific Modelling Languages (DSMLs), which enable modelling using primitives of a particular domain [67].

The creation of DSMLs is recurrent in MDE, for which one needs to describe their abstract and concrete syntax, their semantics, and developing a suitable modelling environment for them. The domain concepts are reflected in the abstract syntax and formally described by defining a meta-model. The concrete syntax is defined by mapping each element of the meta-model with one or several representations, either graphical, textual or a mix of both. The semantics define the reality represented by the model, which is heavily domain-specific, for example: the network configuration of a computer system or a relationship of debts between banks [47]. The availability of the corresponding modelling environment facilitates the creation of valid models of

the DSML and other basic functionalities like model persistence or model consistency checking.

Software applications are becoming increasingly complex, and MDE aims to reduce their production costs [67, 131]. However, while models have a higher level of abstraction than code, for large systems, models may become large and unwieldy as well. There are different enabling technologies for MDE, being the Eclipse Modeling Framework (EMF) [118] a popular framework with a set of compatible plug-ins to facilitate the creation of DSMLs. However, the existing technologies, including EMF, lack of appropriate mechanisms for fragmenting models. Hence, models are generally monolithic, making them costly to process by tools and difficult to understand by people.

With respect to models, there is an absence of systems facilitating (meta-)model modularity. Therefore, the composition, extension and re-utilization of (meta-)models, becomes difficult. Although some environments provide ad-hoc modularization services for specific modelling languages (e.g.,the Unified Modelling Language (UML)) [5, 39, 61, 85, 90, 102], these are not readily available to developers of new DSMLs. Instead, developers need to program the required services manually for the platform where the DSML environment is being built. Since this is a complex task that requires expert knowledge, most environments for DSMLs end up lacking these features, which hinders the scalability of modelling in practice. Modularity in DSMLs would bring as a benefit a scalable approach to the construction of software models through the composition of smaller model components which can be implemented separately in a simpler way [103]. Another advantage of modularity includes increased flexibility and reuse possibilities, facilitating distributed teamwork and version control.

Regarding tools, there are software frameworks that facilitate the development of textual and graphical environments [13, 67, 115], but the creation of DSMLs is mostly an ad-hoc process lacking the ability to build on existing knowledge coming from the construction of similar DSMLs. A further difficulty is the fact that graphical frameworks do not scale well and do not support scalability mechanisms, beyond layers and hierarchical drill down to describe the different aspects of a model.

Altogether, current approaches to DSML development present the following shortcomings:

- Lack of modularity mechanisms for models, which prevents the creation of very large models fragmented in smaller chunks.
- Graphical modelling environments do not scale well.

- The development of modelling environments for DSMLs is an ad-hoc process, which requires deep expertise in the specific language development framework used, sometimes requiring manual programming.
- The definition of services for the modelling environments (including modularity services) needs to be created ad-hoc by manual programming.

In order to alleviate the identified problems, in this thesis, we propose an approach to define modularity services for DSMLs, allowing the definition of complex models from sub-models which are easier to process and reuse. The approach is based on a catalogue of patterns that contribute services to the DSMLs, including modularity and concrete syntax services. This way, the construction of DSMLs environments becomes a systematic process. The patterns currently implemented are:

- **Fragmentation**: this pattern provides modularity for models, so that models can be fragmented, and their parts organized into folders and files in the file system.
- **Reference scoping**: this pattern applied to a reference reduces the candidate objects from other fragments to be referenced.
- **Visibility**: this pattern provides access control to the elements of a fragment from the outside.
- **Indexing**: it enables the creation of object indices by selected fields. These indices can be used for efficient object access [46].
- **Scoped validation**: it allows the efficient evaluation of integrity constraints upon a change in a fragmented model, considering only the subset of objects within the model that has changed.

This systematic approach has been evaluated by implementing case studies such as Wind Turbines [53], Henshin [4] and Computer Aided Engineering Exchange (CAEX) [87]. In addition to this, performance tests were performed on the use of the fragmentation and the scoped validation patterns. On one hand, we measure the time of fragmenting on realistic and synthetic models, also comparing its usefulness with third party tools. On the other hand, we evaluate the use of constraints in fragmented models and compare their evaluation time with the standard evaluation of constraints.

As an addition to the previous approach, we developed two patterns which support the graphical and tabular concrete syntax. We implemented a set of heuristics to simplify the creation of DSMLs for the graphical representation pattern. In this way, users can obtain a concrete syntax through the definition of a meta-model. These patterns are compatible with monolithic and fragmented models, but the latter provides a graphical modelling environment that scales better. We evaluated the approach creating modelling environments that mimic existing, monolithic ones, such as Henshin, CAEX and Wind Turbines.

1.2 Technical Contributions

In addition to the above mentioned fundamental contributions, this thesis also makes the following technical contributions:

1. The development of an Eclipse plug-in called EMF-Splitter for the development of DSMLs. EMF-Splitter implements the described catalogue of modularity patterns for fragmentation, reference scoping, visibility, indexing and scoped validation.

2. The development of an Eclipse plug-in called EMF-Stencil that implements the patterns and heuristics related to the graphical syntax of DSMLs.

3. The integration of EMF-Splitter and EMF-Stencil into DSL-tao, which proposes a pattern-based approach to construct meta-models. The pattern catalogue was added to DSL-tao and dedicated wizards were implemented. DSL-tao plug-in can be downloaded at: http://www.miso.es/tools/DSLtao.html.

4. As proof of concept, we develop scalable graphical environments for Henshin [4], CAEX [87] and Wind Turbine control systems. The latter is an industrial case study implemented during the MONDO EU project [91] in which part of this thesis was developed.

The tools EMF-Splitter and EMF-Stencil, that supports the systematic approach proposed in this thesis, can be downloaded at: http://www.miso.es/tools/EMFSplitter.html.

1.2.1 Publications

The contributions of this thesis have led to the following publications. These publications have been organized into five groups: journals, journals under review, international conferences and workshops, national conferences and book chapters.

Journals:

1. A. Jiménez-Pastor, Antonio Garmendia, and J. de Lara, "Scalable model exploration for model-driven engineering," *Journal of Systems and Software*, vol. 132, pp. 204–225, 2017, Elsevier Science Inc. (JCR: 2.278, **Q1 Computer Science, Software Engineering**)

2. J. J. López-Fernández, Antonio Garmendia, E. Guerra, and J. de Lara, "An example is worth a thousand words: Creating graphical modelling environments by example," *Software and Systems Modeling (Springer)*, vol. 18, no. 2, pp. 961–993, 2019.Invitation to participate in special issue with best papers of ECMFA'16. (JCR: 1.722, **Q2 Computer Science, Software Engineering**)

3. Antonio Garmendia, E. Guerra, J. de Lara, A. García-Domínguez, and D. Kolovos, "Scaling-up domain-specific modelling languages through modularity services," *In Press of Information and Software Technology*, 2019, Elsevier Science Inc. (JCR: 2.921, **Q1 Computer Science, Software Engineering**)

Journals under review:

1. A. Gómez, X. Mendialdua, K. Barmpis, G. Bergmann, J. Cabot, X. de Carlos, C. Debreceni, Antonio Garmendia, D. S. Kolovos, J. de Lara, and S. Trujillo, "Scalable modeling technologies in the wild: An experience report on wind turbines control applications development," *Software and Systems Modeling (Springer)*, 2019.Invitation to participate in special issue with best papers of ECMFA'17. (Second round of review, JCR: 2.660, **Q1 Computer Science, Software Engineering**)

International Conferences and Workshops:

1. Antonio Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara, "EMF Splitter: A structured approach to EMF modularity," in *Proceedings of the 3rd Workshop on Extreme Modeling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (XM@MoDELS), Valencia, Spain, September 29*, 2014, pp. 22–31.

2. Antonio Garmendia, A. Jiménez-Pastor, and J. de Lara, "Scalable model exploration through abstraction and fragmentation strategies," in *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations federation of conferences (STAF), L'Aquila, Italy, July 23*, 2015, pp. 21–31

3. Antonio Garmendia, A. Pescador, E. Guerra, and J. de Lara, "Towards the generation of graphical modelling environments aided by patterns," in *Proceedings of 4th International Symposium on Languages, Applications and Technologies (SLATE), Madrid, Spain, June 18-19, Revised Selected Papers*, ser. Communications in Computer and Information Science 563, Springer, 2015, pp. 160–168

4. A. Pescador, Antonio Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara, "Pattern-based development of domain-specific modelling languages," in *Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Ottawa, ON, Canada, September 30 - October 2*, 2015, pp. 166–175. **Core 2015: B, Acceptance Rate: 26.7% (46/172)**

5. Antonio Garmendia, "Constructing scalable domain-specific graphical modelling languages," in *Proceedings of the Doctoral Symposium at the 19th ACM/IEEE International Conference of Model-Driven Engineering Languages and Systems (MoDELS), Saint Malo, France, October 2*, 2016. **Core 2016: B**

6. D. Vaquero-Melchor, Antonio Garmendia, E. Guerra, and J. de Lara, "Towards enabling mobile domain-specific modelling," in *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26*, 2016, pp. 117–122

7. J. J. López-Fernández, Antonio Garmendia, E. Guerra, and J. de Lara, "Example-based generation of graphical modelling environments," in *Proceedings of the 12th European Conference on Modelling Foundations and Applications, (ECMFA), Held as Part of STAF, Vienna, Austria, July 6-7*, ser. LNCS 9764, Springer, 2016, pp. 101–117

8. A. Gómez, X. Mendialdua, G. Bergmann, J. Cabot, C. Debreceni, Antonio Garmendia, D. S. Kolovos, J. de Lara, and S. Trujillo, "On the opportunities of scalable modeling technologies: An experience report on wind turbines control

applications development," in *Proceedings of the 13th European Conference on Modelling Foundations and Applications (ECMFA), Held as Part of STAF, Marburg, Germany, July 19-20*, ser. LNCS 10376, Springer, 2017, pp. 300–315

National Conferences:

1. Antonio Garmendia, E. Guerra, and J. de Lara, "Building scalable graphical modelling environments with EMFSplitter (tool demo)," in *Proceedings of the XXIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD), September 17-19, 2018, Seville, Spain.*

Book Chapters:

1. D. Vaquero-Melchor, Antonio Garmendia, E. Guerra, and J. de Lara, "Domain-specific modelling using mobile devices," in *Software Technologies, 11th International Joint Conference (ICSOFT), Lisbon, Portugal, July 24-26, Revised Selected Papers.*, 2016, pp. 221–238

1.3 Research Visits

During the realization of this PhD, an external research stay was conducted in collaboration with the software engineering group of the University of Marburg, Germany, supervised by Dr. Gabriele Taentzer. The stay was made in that university for a period of three months from April 9 to July 9, 2018. As a result of the collaboration, a graphical editor for the Henshin tool was developed, which will be presented in Section 7.5.2.

1.4 Support

The first years of the realization of this PhD were supported by the European project MONDO (http://www.mondo-project.org/). The objective of this project was to address the scalability issues in MDE, which among other aspects implies the construction of large models in a systematic manner that can facilitate collaboration. When MONDO finished, I received an FPI grant from the project "*FLEXOR*" (TIN2014-52129-R) with the grant number BES-2015-073098. This project was funded by the Spanish Ministry of Economy and Competitivity.

1.5 Organization

The following is a brief summary of what each of the 7 chapters of this thesis describes.

- **Chapter 2** presents an overview of the MDE paradigm and the construction of DSMLs. In addition, different tools used in the field of MDE are described and compared.
- **Chapter 3** provides a detailed description about the use of patterns in the creation of DSMLs. This chapter shows examples of how to specify a pattern and describes a proposal of patterns to assist in the definition of DSMLs.
- **Chapter 4** describes the proposed approach for providing modularity to DSMLs. The approach is based on a catalogue of modularity patterns and services.
- **Chapter 5** introduces the proposed approach to assist developers in obtaining a graphical and tabular concrete syntax for DSMLs. In addition, it explains how to improve the scalability of graphical representations of fragmented models.
- **Chapter 6** provides a general description of the tools developed in this thesis, which are EMF-Splitter and EMF-Stencil. The first tool generates a scalable environment using information from instantiated patterns at the meta-model level. EMF-Stencil assists developers in the concrete syntax instantiation and automatically generates graphical editors.
- **Chapter 7** describes the evaluations carried out using the proposed tools. Among the different experiments explained in this chapter, an evaluation was carried out with large models and two modelling environments were created for Henshin and CAEX. This chapter also describes applications built in collaboration with other developers using the technologies presented in this thesis.
- **Chapter 8** discusses the conclusions obtained with our research and proposes lines for future research.

Chapter 2

Background and Related Work

This chapter is divided in two parts. The first one (Section 2.1) provides a general perspective on MDE and basic concepts, such as meta-models, code generators, integrity constraints, and DSMLs. This part (Section 2.1.1) includes a discussion about open challenges that have been tackled in this PhD. The second part (Section 2.2) outlines some technologies and related research, regarding scalability, modularity and the creation of graphical editors.

2.1 Model-Driven Engineering: Basic Concepts

Programming paradigms have evolved over time, proposing innovative solutions to challenges in software development, such as re-usability, modular software design, and improved productivity by increasing the abstraction level of programming languages. One of the most significant paradigms was Object-Oriented Programming (OOP) [45], which proposed the use of classes and objects and notions like message passing, inheritance, polymorphism and encapsulation.

The use of models can be seen as a further step in this direction. What laid the foundation of using models in software development was the publication of the Model-Driven Architecture (MDA) by the Object Management Group (OMG) [95]. The OMG consists mainly of researchers and industry professionals who are dedicated to developing technological standards. Until that moment, models were only used to document applications, but the MDA paradigm made them first-class citizens [71], becoming actively used in all stages of the software development life cycle.

MDE is a software engineering paradigm that promotes a model-centric approach to software development, where models are used to specify, design, test and generate code for the final application [19]. MDE was a later concept defined to encompass

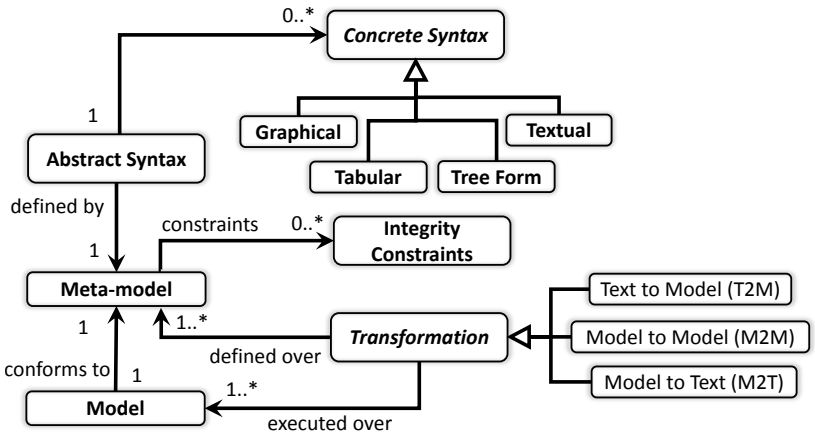


Fig. 2.1 Model-Driven Engineering overview.

other more specific model-driven paradigms such as MDA, the Architecture-Driven Modernization (ADM) [2] and others that use models created using DSMLs as the main development artefact [19]. This paradigm aims at reducing the production costs of software development by enabling the generation of a family of applications for a given domain instead of focusing on the construction of single applications. Nowadays, many organisations use MDE to develop their systems [8, 53, 60, 130] or to migrate legacy code, being supported by model management tools [101].

Many approaches to MDE use DSMLs instead of general-purpose modelling languages like the UML. DSMLs focus on specific concepts for a particular domain [67], and their creation is recurrent within the MDE paradigm. Figure 2.1 shows an overview of the main MDE concepts, which includes models, modelling languages and transformations. The **abstract syntax** of a DSML is defined by a **meta-model** which specifies the domain elements and their properties and relations [70]. **Models** should conform to their meta-models, and specify the structure and behaviour of the systems they represent [19]. The **concrete syntax** describes the representation of models and can be **graphical**, **textual**, **tabular**, and **tree form**, among others. To ensure the correctness of an instance model, **integrity constraints** are frequently attached to meta-models.

Figure 2.2 shows the 4-layer infrastructure proposed by the OMG, which comprises the description of languages to represent meta-models, down to the real system. Layer three includes the definition of a language to represent meta-models, called Meta-Object Facility (MOF) [96]. This is divided into two parts: Complete MOF (CMOF) and Essential MOF (EMOF). CMOF is considered a very complex language and, in

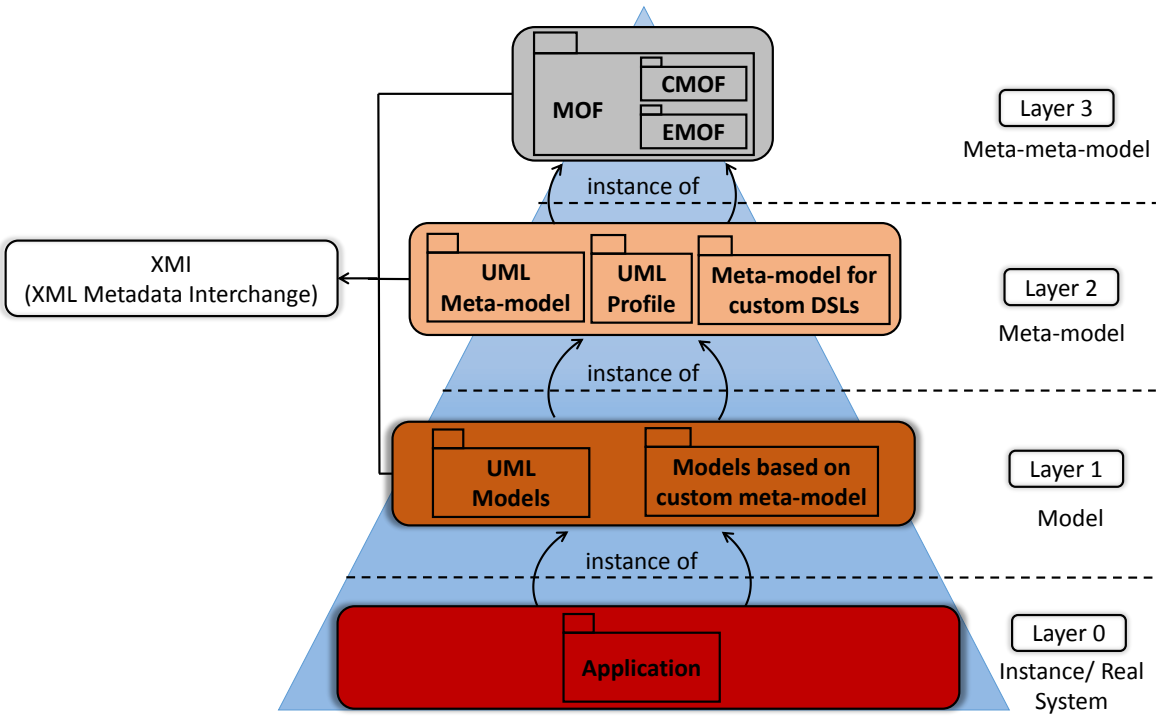


Fig. 2.2 OMG 4-layer architecture.

general, meta-models can be described using a subset of concepts such as those defined in EMOF.

The layer number two of the OMG architecture corresponds to the meta-models created with MOF. This layer includes domain meta-models like the one shown in Figure 2.3 (to be described below), as well as meta-models for general purpose modelling languages and standards like the Unified Modelling Language (UML) [126]. UML is a standard frequently used in software development that defines a set of diagrams to describe the structure and behaviour of systems [114]. UML is a very extensive language and for certain domains, some features are too generic or useless. That is why domain meta-models are built, which specifically target certain application domains.

Layer number one is where we find the models that are instances of meta-models. Figure 2.4 shows an example of a model that will be described later. Models represent the system reality in a precise way to fit a certain purpose. By being domain specific, models typically suppress technical implementation details and specific platform aspects where the system is to be deployed.

For the serialization of models and meta-models, the OMG proposes the XML Metadata Interchange (XMI) standard (Figure 2.2) [97]. XMI is based on the Extensible Markup Language (XML), a specification for storing documents highly used on the

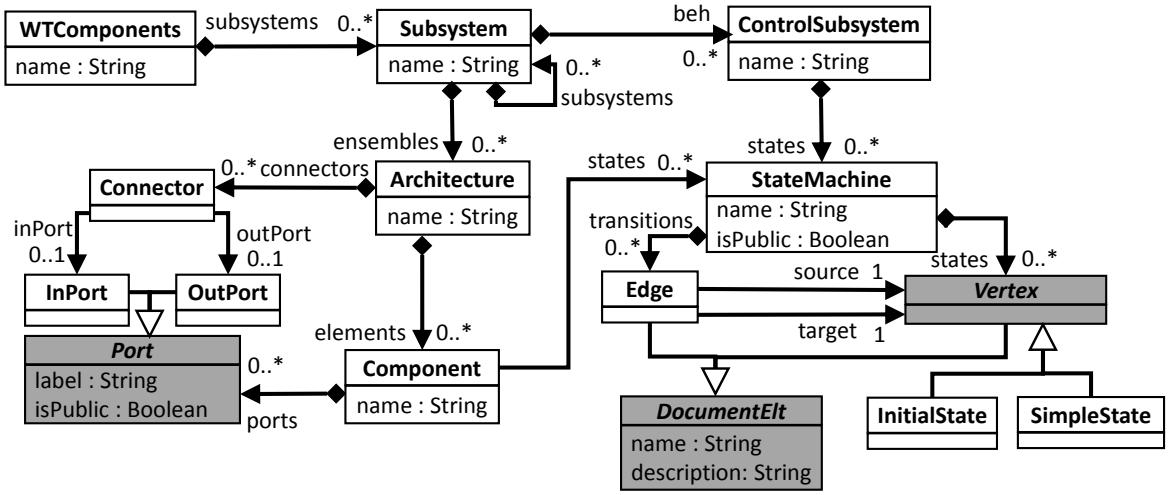


Fig. 2.3 Meta-model excerpt for Wind Turbines.

World Wide Web (WWW). XML supports a very flexible definition of serialization. However, because the objects can be serialized in different ways in XML schemas, this still makes the exchange of information between tools a complicated task. In order to fill this gap, XMI defines a set of object-oriented rules, which makes the exchange of models between modelling frameworks easier [47]. As a consequence, XMI has become a widespread standard to facilitate the interoperability between MDE tools.

Finally, layer zero corresponds to the real application that is created from the models or the systems the models represent. Based on models, it is possible to create databases, user interfaces, configuration files, application code and other artefacts relying on the target platform. In this way, it is possible to generate a family of applications for different deployment environments based on models.

As seen in Figure 2.1, the abstract syntax of DSMLs is defined by a meta-model. An example of a meta-model is presented in Figure 2.3. It is used to describe the controller software of Wind Turbines (WTs) [53]. The language defines a class WTComponents, which includes a set of Subsystems. The WT meta-model supports the definition of components, which contain input and output ports that are connected through connectors. In addition, the behaviour can be specified using state machines. The gray background in some classes means that they are *abstract* classes, and hence cannot be instantiated.

Generally, there may be some requirements in the domain that can not be fully expressed in a meta-model. Therefore, as a complement to meta-models, additional constraints are usually written using the Object Constraint Language (OCL) [23]. OCL is a declarative language statically typed. It can be used to define meta-model invariants,

that is, boolean expressions stating conditions that the instances of a meta-model should satisfy. OCL invariants are declared in the context of a class, and are evaluated on all objects of that type. A model typed by a meta-model and satisfying all its OCL invariants is said to *conform* to the meta-model.

As an example, Listing 1 shows an OCL invariant for the WT meta-model to guarantee that each ControlSubsystem contains at least two objects of type StateMachine with their attribute isPublic equals to false. In this case, the context class is ControlSubsystem, because the invariant must check each one of its instances.

Line 3-4 define the expression that all instances of type ControlSubsystem must satisfy. In this expression, self is used to refer to the instance where the expression is evaluated. The navigation expression self.states obtains all state machines contained by the ControlSubsystem object. To manage collections, OCL offers a set of operations, like select. This operation filters a collection to return only the objects that fulfil a certain condition. In this example, it selects the state machines whose attribute isPublic is false. Next, the operation size() returns the number of elements in the filtered collection, which should be greater than or equal to two.

```
1 context ControlSubsystem
2 inv minNumberOfSMInCSubsystem :
3 self.states→select(stateMachine |
4   stateMachine.isPublic = false)→size() >= 2
```

Listing 1 Example of OCL constraint for objects of type ControlSubsystem.

Listing 2 shows another example of OCL constraint. Its goal is to verify that the attribute name is different for objects of type Subsystem. In this case, we use allInstances() to retrieve all objects of type Subsystem, and the forAll operation to compare the name of every two objects of type Subsystem. The forAll operation returns true if the given condition is fulfilled by all instances of the collection.

```
1 context Subsystem
2 inv SubsystemDifferentNames :
3 Subsystem.allInstances()→forAll(sub1, sub2 |
4    sub1 <> sub2 implies sub1.name <> sub2.name)
```

Listing 2 Example of OCL constraint for objects of type Subsystem.

Another key concept within MDE is the creation of models (Figure 2.1). These artefacts must be constructed as instances of the meta-models and to ensure that they are valid, they must satisfy all OCL constraints in the meta-model. Figure 2.4 shows a model that is typed by the WT meta-model. In the example model, the object of type WTComponents is the root element, which directly contains two subsystems.

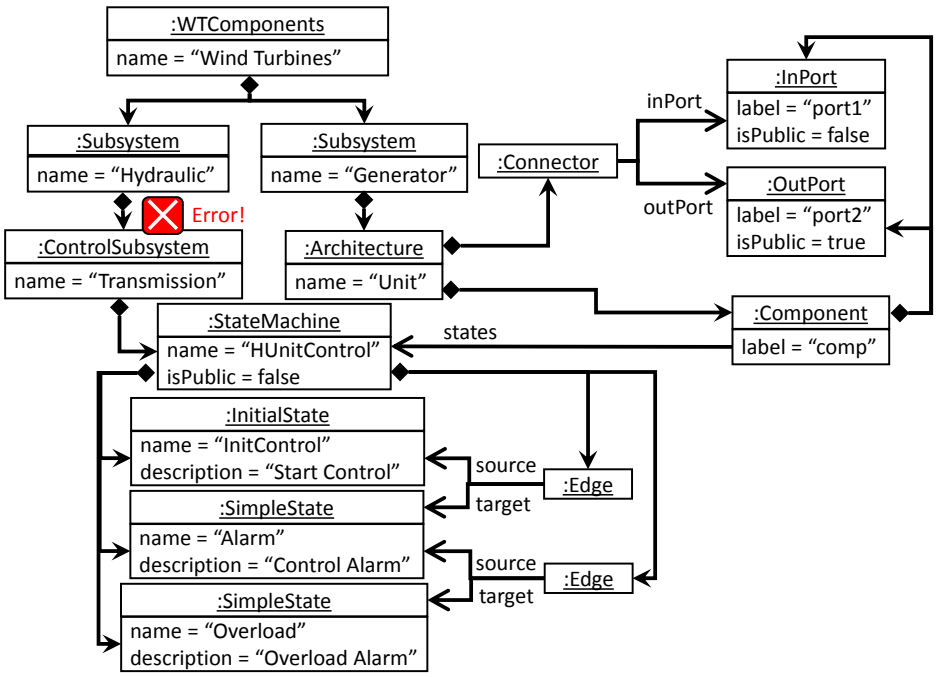


Fig. 2.4 Abstract syntax of a Wind Turbine model.

The Subsystem called *Hydraulic* includes a control subsystem with a state machine to describe its behaviour. The Subsystem called *Generator* contains an Architecture object which includes one component with two ports connected by a Connector element. The model does not satisfy the constraint in Listing 1, because there is only one state machine where the isPublic attribute is false. The constraint shown in Listing 2 is satisfied, because all elements of type Subsystem have different names. Hence, altogether the model does not conform to the WT meta-model.

Figure 2.4 shows the model using abstract syntax, as an object diagram. However, for proper editing and visualization, languages typically define a concrete syntax, either graphical or textual. Figure 2.5 shows a graphical representation of the WT model depicted in Figure 2.4, where the violated OCL invariant is signalled as well. That is why the ControlSubsystem object appears with an error in the figure. This diagram shows how objects are related using containment, edges, and adjacency. The containment relationship is established when one object is displayed inside another, as shown between objects Hydraulic and Transmission, Generator and Unit, and Transmission and StateMachine. Adjacency is configured when the border of objects overlap, usually, one object is selected as the main one and the others are shown as smaller, annexed parts. The adjacency is shown in the diagram when objects of type Port are annexed to the main object of type Component. Finally, the edges relate some states with others,

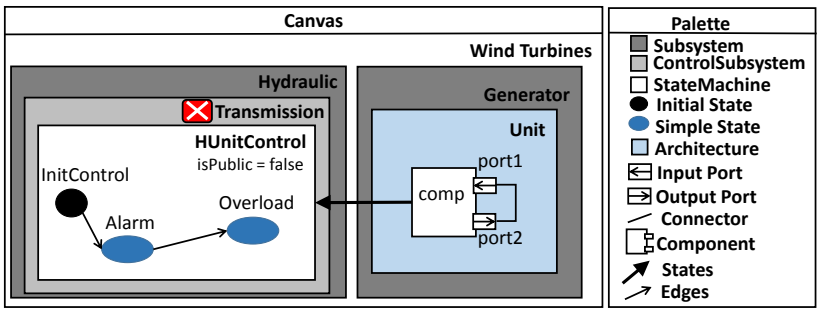


Fig. 2.5 Graphical representation of a Wind Turbine model.

connect ports, and components with state machines. This way, some relationships in the meta-model are graphically represented as spatial relations. Graphical editors offer a palette from which the user can create objects on the canvas and facilities to edit the object properties, as shown to the right of Figure 2.5.

The other concept involved in MDE is the **Transformation** (Figure 2.1). Transformations are used to modify existing models or create other artefacts, such as other models or source code. Transformations may be classified in different types, among them we can find *Model-to-Model (M2M)*, *Text-to-Model (T2M)* and *Model-to-Text (M2T)* [29]. *M2M* transformations can be classified into two types based on the relationship between the input and output models. If the input model is updated directly, then the transformation is called *in-place*, otherwise, if a new output model is created, it is an *out-place* transformation.

An example scenario for *M2M* transformations is when a language needs to be migrated to a new one, either to optimize a certain operation or to modify the program structure. When this type of change is performed, old models are no longer compatible and must be updated to the new version of the language. Figure 2.6 shows an example of a *M2M* transformation of type *out-place*. The source meta-model is an excerpt of the WT meta-model shown in Figure 2.3, were we have extracted the classes related to Architecture. The variation of the target meta-model is that the class Connector disappears, and then, to show the relationship between ports, we added a reference target from the class OutPort to InPort. The transformation results in a compatible model where Connector objects are replaced by a target reference.

In *M2T* transformations, the resulting text may be source code, application documentation or a configuration file, among other artefacts. When a *M2T* transformation generates code, it is frequently called code generator. The code generation can be *partial*: when some program details are not represented at the meta-model level and it is necessary to program those manually, or *full* when all artefacts are generated.

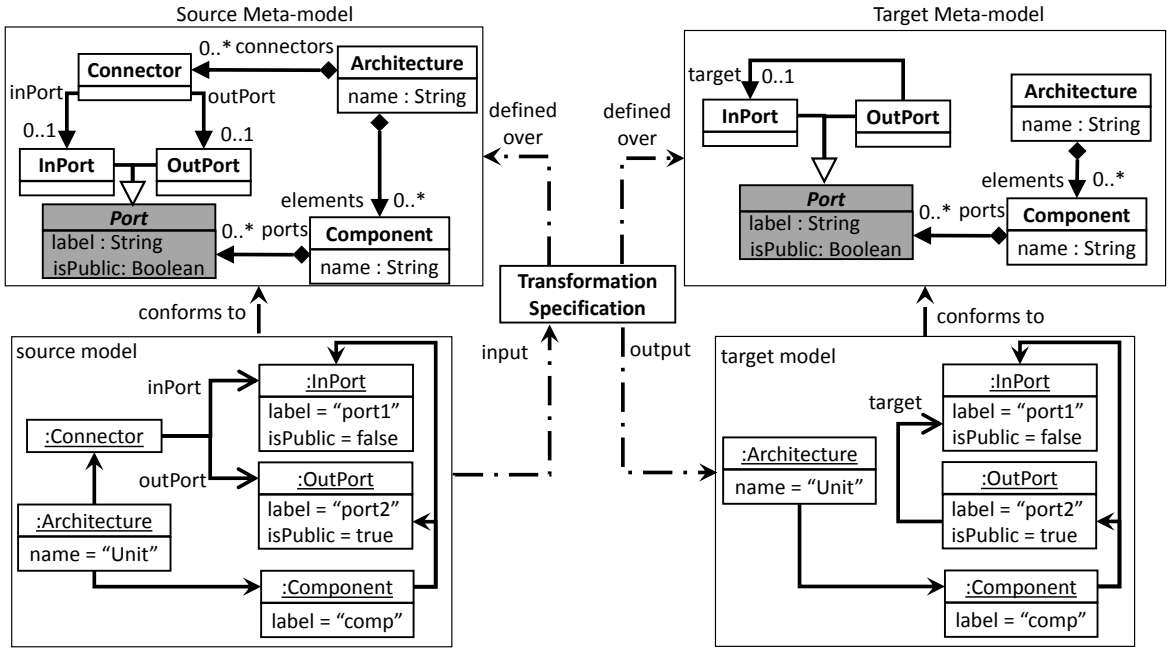


Fig. 2.6 Example of M2M transformation.

Figure 2.7 shows an example of code generation from a meta-model to Java source code.

The last type of transformation is *T2M*, which is generally used to do reverse engineering tasks [19]. A field within MDE that offers reverse engineering solutions is called Model-Driven Reverse Engineering (MDRE) [22]. Companies use this type of solutions when they need to migrate existing applications to more modern technological platforms. Its rationale is that some programming language may have been discontinued over time, but companies still own business processes implemented in these languages, and so migration to a new language is necessary.

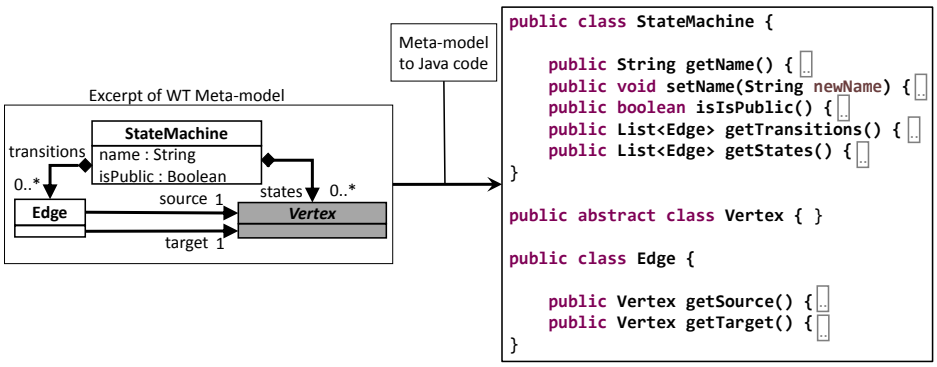


Fig. 2.7 Example of *M2T* transformation.

2.1.1 Challenges

Despite the benefits of MDE, and the availability of techniques to build DSMLs, there are certain limitations regarding the visualization and construction of models, which motivates this work:

- **Scalability**: The current modelling practice in MDE is characterized by the construction of monolithic models. These models may become large and unwieldy when the systems they represent are complex. Therefore, modelling environments struggle when development and visualization of large models is needed. At the same time, the persistence mechanisms of current technology is usually based on monolithic files, which is inefficient when the files reach large dimensions [77].
- **Modularity**: Software development using modular design facilitates collaboration between developers separating the application functionality into modules. In addition, modularity is used as a solution to solve scalability issues. Within MDE, there are no standard techniques enabling model modularity for arbitrary modelling languages, but existing proposals are specific for a language, like the UML [34]; therefore adding to new languages capabilities to compose, extend or reuse their models is difficult. The availability of modularity techniques would have the advantage of enabling the creation of libraries of model abstractions, providing standard solutions for problems known, solved and tested.
- **Systematic engineering of modelling environments**: Building modelling editors is expensive since they have to adapt to the particularities of the modelling language for which the editor is defined. Building editors for graphical languages is particularly challenging since their construction typically requires deep knowledge of the language development framework, and there are no predefined graphical components that can be reused in different environments to reduce the development time, or heuristics that guide and facilitate the construction of those environments.

Therefore, in this PhD, we propose the definition of meta-model patterns to define services for DSMLs [26] and a semi-automatic approach to generate graphical editors. The modelling environment that is generated will be scalable following the guidelines of a modularity pattern, and services like scoping, visibility, indexing and concrete syntax will be defined using patterns as well. In the next section, we will describe some works related to these topics.

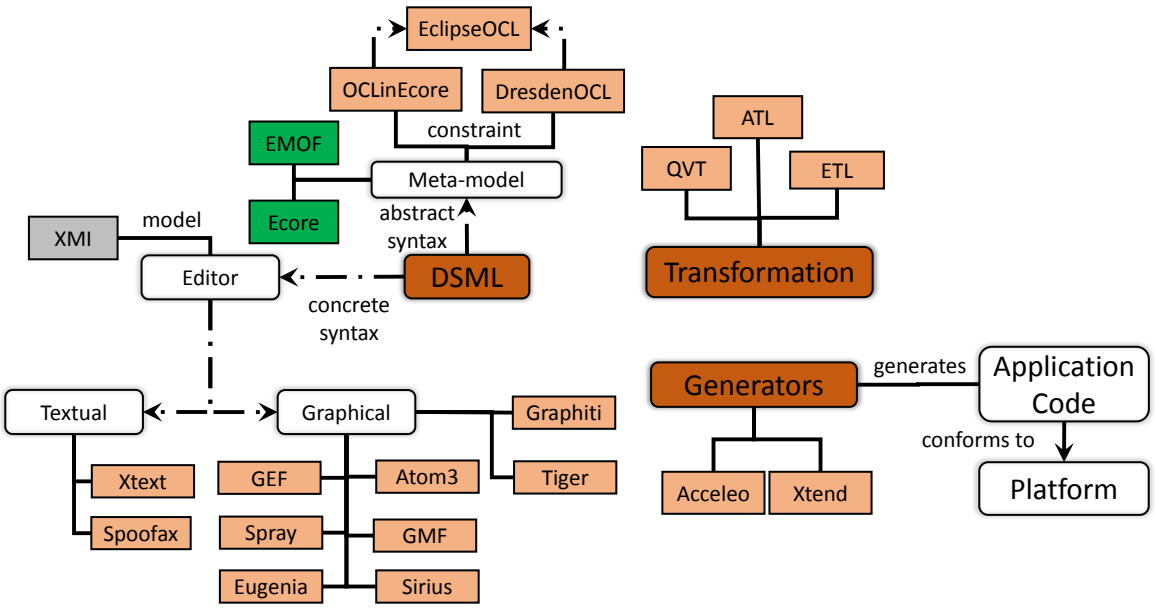


Fig. 2.8 Model-Driven Engineering technologies.

2.2 Related Work

In this section, first we revise current modelling technologies, and then, we focus on existing works dealing with model fragmentation, large models, the use of patterns in MDE, and technologies and frameworks to create graphical DSMLs.

2.2.1 Modelling Technologies

Figure 2.8 shows a summary of tools as outlined in this section. There are different tools that give support to MDE, but the advent of the Eclipse Modeling Project (EMP) [36] has made popular all the proposed technologies for the Eclipse ecosystem. The Eclipse Modeling Framework (EMF) [118] is the core of the EMP project. It has become a widespread technology, and many compatible plug-ins have been developed to facilitate the creation of DSMLs atop EMF. The EMF project proposes Ecore as the language to describe meta-models. Nevertheless, EMF supports also the EMOF specification defined by the OMG, which is very similar to Ecore.

In this thesis, we use Ecore as the technology to create meta-models. Figure 2.9 shows its main concepts. The EPackage class contains EClasssifiers as well as other EPackages. The packages are used to modularize the resulting meta-model, grouping the EClasses that are related. There are two types of EClasssifiers: EClasses and EDataTypes. Classes may have EReferences and EAttributes. EReferences can be tagged as containment,

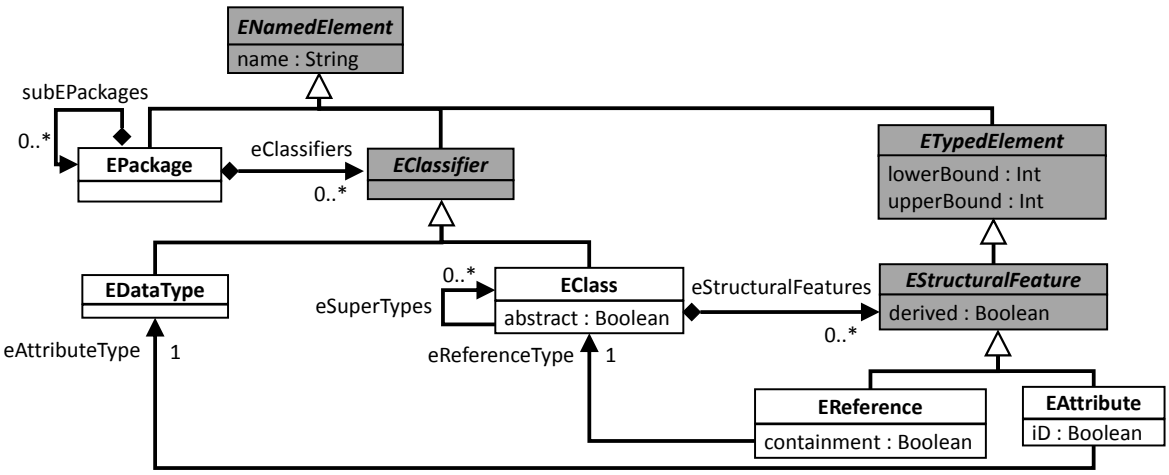


Fig. 2.9 Meta-model excerpt with the basic concepts of the Ecore meta-modelling language.

with a semantics similar to UML composition associations. This means that they contain objects of a certain type, otherwise they will only reference them. EClasses can also hold attributes of primitive data type, like EFloat or EString. EClasses support multiple inheritance through reference eSuperTypes, so that an EClass inherits the attributes and references from the declared supertypes.

The EMF framework relies on the XMI format for model and meta-model persistence. XMI provides mechanisms to link objects from different model files, by means of so-called cross-references. Splitting a very large XMI file into smaller ones improves the performance of model management tools (editors, transformations, etc.), especially if these files can be processed separately [56]. However, split automation is not supported out of the box in standard EMF.

As part of EMP and based on EMF, the Eclipse OCL framework provides a core implementation that facilitates the construction of tools to define OCL integrity constraints for meta-models. For example, tools such as OCLinEcore [99] and Dresden OCL [32] (Figure 2.8) are Eclipse plug-ins based on this framework. OCLinEcore [99] is a tool that integrates the OCL language within Ecore, providing an editor to embed OCL constraints as annotations within the meta-model. Dresden OCL [32] is another tool supporting the definition of OCL constraints, which is basically a parser for OCL. Dresden OCL separates the OCL constraints from the Ecore file.

There are several tools supporting the definition of textual concrete syntaxes for modelling and programming languages, such as the Spoofax Language Workbench [66] and Xtext [13]. Both tools are based on the definition of a grammar from which an editor and a parser are generated, but they use different technologies for their

implementation. On the one hand Spoofax [66], uses the Syntax Definition Formalism (SDF) to define the grammar and generate a partial textual editor, which can be further customised to add e.g., code completion, new scopes, and syntax highlighting, among other features. This tool is integrated in Eclipse, and at this time, there is a plan to build a prototype of Spoofax for the web. On the other hand, Xtext [13], belongs to the EMP. Xtext allows the synthesis of grammars from Ecore meta-models, which are used as a basis to generate the parser and an editor for the Eclipse IDE, although it can also be embedded in a web application.

In MDE, models are frequently used to generate text documentation or code. Frameworks such as Acceleo [1] and Xtend [133] are languages for code generation based on templates (Figure 2.8) integrated in the Eclipse IDE.

Acceleo is a code generator for EMF models that started as an independent project, but was later included in EMP. The working scheme of this tool relies on templates that generate text files. Acceleo templates support OCL queries which are executed on the models, although it is possible to call external operations in Java. Xtend is another framework that provides multi-line template expressions to facilitate code generation, which is nearly always used together with Xtext, but can also be used independently.

Many of M2M transformation languages have been proposed, such as the ATLAS Transformation Language (ATL) [7], the Query/View/Transformation language (QVT) [107] and the Epsilon Transformation Language (ETL) [74] (Figure 2.8). These technologies introduce Domain-Specific Languages (DSLs) based on rules to define M2M transformations. ATL bases its language on OCL and supports unidirectional transformations, in which the source model is not modified, but it is queried to create the target model [47]. The ETL language also supports this type of transformations and is also integrated into the Epsilon family of languages for model management. Finally, QVT is the OMG proposal to perform M2M transformations. The abstract syntax of QVT is based on MOF, which consists of three languages: *QVT Operational Mappings*, *QVT Relations* and *QVT Core*.

As this thesis deals with the engineering of graphical DSMLs, we will describe and compare in detail the most prominent frameworks for their creation in Section 2.2.4.

2.2.2 Systematic Development of Domain-Specific Modelling Languages

As discussed in Section 2.1, the creation of DSMLs is recurrent in the MDE paradigm. In order to improve productivity in the construction of DSMLs, several researchers

have devised dedicated engineering processes for this. For example, Strembeck and Zdun [119] identify four main activities in DSL development based on different projects in which they have participated. These activities do not have a defined order as it can vary according to the application context, but they may serve as a guide for DSL developers [106].

Another work related with the development of DSL is [88]. Mernik et al. identify patterns for decision, analysis, design, and implementation phases of DSL development. In this article, the authors extend and improve earlier works on DSL design patterns, with the aim to help developers in the construction of DSLs. Patterns are structures that have been previously tested and proved successful to solve a problem. In this way, a number of studies also deal with the definition and classification of DSL patterns [42, 116].

Due to the complexity of DSL design, works such as [65] describe guidelines to achieve better quality of the language design. This paper proposes 26 guidelines, of which we would like to draw the attention to two. The first one states that we should look at the abstract syntax of existing languages to identify patterns that can be applied to the new languages. The second guideline recommends providing modularity mechanisms to new languages, by which the systems can be divided into small pieces that can refer to each other. Following the same idea, this thesis defines a catalogue of patterns that are instantiated at the meta-model level to enable modularity.

Several works formalize patterns and automate their application for some notation. Bottoni and collaborators [17] propose a language for the specification of patterns using some of them to show their applicability. Another example of pattern specification is the one realized by France and collaborators [44]. In this approach, a UML-based pattern specification is described by two class diagrams: one for pattern specification and the other one for pattern interaction.

With respect to DSMLs, some works encourage the use of patterns for their construction. For instance, in [26], the authors define meta-modelling design patterns directed to design decisions and provide a few examples. They also analyse which type of structures defined in the meta-model give as a result common instantiation of the concrete syntax. In this sense, three meta-modelling patterns are proposed to represent lines and boxes, containment and several types of relationships. In [111], the authors detect the lack of engineering processes for DSML construction, and propose documenting DSMLs using use cases and design patterns. In [116], Spinellis defines a set of architectural patterns for DSMLs design, like composing DSMLs through pipelines. Altogether, in this branch of works, design patterns improve the inner

quality of DSMLs but are normally low-level, which entails a greater effort of the developer.

Other papers describe the use of domain patterns to capture domain knowledge and speed up the construction of meta-models [43, 123]. Similarly, [104] argues on the benefits of building DSMLs by composing domain concepts. A domain concept is a meta-model and its semantics is given as a model transformation. The authors define some composition operators, and aim at composing the respective transformations.

2.2.3 Model Scalability and Modularity

Due to the need to process large models, some authors have proposed to split them to facilitates different tasks. For instance, Scheidgen and collaborators [113] propose a persistence framework called EMF-Fragments that allows automatic and transparent fragmentation to add, edit and update EMF models. This process is executed at runtime, with considerable performance gains. The approach has been used for the analysis of large code repositories [112], so that code projects are parsed into a model representation, and then analysed using OCL queries. To guide the fragmentation, the composition references in the meta-model that are aimed at producing fragments need to be annotated. EMF-Fragments stores fragmented models in memory, and for persistency it primarily relies on distributed file-systems and key-value stores like MongoDB and HBase. For traditional XMI persistence, fragmentation is currently not reflected in the file system. For this reason, these technologies can not benefit from traditional version control systems (like SVN or Git), since the fragmentation is not reflected in files and folders.

Other works [69, 120] decompose models into submodels for enhancing their comprehensibility. For example, Kelsen and collaborators propose an algorithm to fragment a model into submodels (actually they can build a lattice of submodels), where each submodel is conformant to the original meta-model [69]. The algorithm considers cardinality constraints but not general OCL constraints, and there is no tool support. Other works use Information Retrieval (IR) algorithms to split a model based on the relevance of its elements [120]. This research resulted in the creation of Splittr as a tool to split models. Therefore, splitting models that belong to the same meta-model can produce different structures. Customizable graph clustering techniques, with the purpose of meta-model modularization, have also been proposed [121]. The techniques are based on several clustering algorithm operations on a distance matrix. This matrix is obtained by weighting different meta-model relations (generalization, composition, association) according to their relevance. While the approach proposed in this thesis

is applicable to existing large models, a distinctive feature is that we also generate a modelling environment that enforces the defined modularization strategy when creating a new model.

Other approaches are based on search techniques guided by quality criteria [41, 93]. Moody and Flitman [93] use genetic algorithms to cluster a data model into a multi-level structure, using principles of human information processing. More recently, inspired by the work presented in [28], Fleck and collaborators present an approach for model modularization applicable to arbitrary modelling languages [41]. The approach is based on mapping concrete meta-model elements to modularization concepts like "module". Then, a generic transformation is used to actually split the model elements into modules based on quality criteria like cohesion and coupling. The transformation rules are applied according to multi-criteria optimisation algorithms. The modularization proposed in this thesis is richer, as it supports projects, packages, and nested packages, while Fleck's approach lacks hierarchical decomposition (modules within modules).

Other works directed to define model composition mechanisms [58, 68, 122] are intrusive. Heidenreich et al. [58] and Struber et al. [122] present techniques for model composition and realise the importance of modularity in models as a research topic to minimise the effort. Strüber et. al [68] present a structured process for model-driven distributed software development which is based on split, edit and merge models for code generation.

Amálio et al. [3] developed a theory of model fragmentation, upon which the practical solution of this thesis is based. The theory is based on graphs and morphisms, and was developed using the Z formal language with the help of proof assistants. It describes possible model organizations, based on fragments (units), clusters (packages) and models (projects). The theory describes fragmentation devices, like proxy nodes and cross-links, which are available in EMF; and explains model de-composition and composition, showing correctness of the obtained model.

Instead of using fragmentation, other persistence options to handle large models have been proposed, like Connected Data Objects (CDO) [25], Morsa [40], NeoEMF [11], or Hawk [46]. Table 2.1 summarizes their main features. CDO is a widely used model repository and persistence back-end. It supports different technologies for data storage such as Hibernate [59] or Objectivity/DB [98], although the recommended one is DB Store [30] because it supports all CDO functionalities. Another example is Morsa [40], which makes use of a NoSQL database to provide scalable access to large models. Morsa is integrated with EMF, and there is a prototype based on MongoDB [27] as the database engine. On the contrary, Neo4EMF allows storing the models in graph-based

Tools	Integrated with EMF	Back-end Technology	Types of Databases
CDO [25]	✓	DB Store /Hibernate/Objectivity	Graph/Object Oriented /NoSQL databases
Morsa [40]	✓	MongoDB	NoSQL database
NeoEMF [11]	✓	Neo4j	Graph database
Hawk [46]	✓	Neo4j/OrientDB	Graph databases

Table 2.1 Summary of tools to handle large models based on databases.

back-ends and its prototype relies on using Neo4j [10]. The last tool is Hawk, which is a model indexer that allows the efficient execution of queries.

In Table 2.1 it is noticeable that CDO provides drivers for different types of databases. Indeed, CDO is the most mature technology to persist the models. Most tools are compatible with EMF, which makes this framework a standard for the development of MDE tools.

2.2.4 Frameworks to Create Graphical Editors

Environments to develop visual languages exist since the end of the 90s. There are many tools like KOGGE [35], DOME [12], GME [82], Diagen [89], MetaEdit+ [124] and AToM3 [81], who laid the foundations for the development of graphical modelling environments.

The emergence of Eclipse has boosted model-driven approaches to software development. Eclipse has made available modelling technologies and has joined efforts for the construction of frameworks, to create visual editors. Graphical language development tools like Tiger [14], the Graphical Modelling Framework (GMF) [52], EuGENia [76], Spray [51], Graphiti [55], and Sirius [115] (Figure 2.8) are part of the Eclipse ecosystem. In the following, we revise the main features of these tools.

Graphiti is a graphical modelling tool based in GEF and Draw2D [109], which provides a Java API for coding. Graphiti requires manual programming, so building graphical editors can be a huge effort. There are tools atop Graphiti to reduce construction time, such as Spray [51] and Xdiagram [110]. Spray uses a textual DSL to define the concrete syntax, reducing the lines of code needed to develop a graphical modelling tool and avoiding the need to know the Graphiti Java API. Similarly, Xdiagram [110] allows defining diagrammatic representations based on Eclipse technologies. One of the most significant differences between these two tools, is that Xdiagram is based on model interpretation, and Spray generates Graphiti code.

Another graphical framework that belongs to the Eclipse ecosystem, and is based on EMF is the Graphical Modeling Framework (GMF). In order to define the concrete syntax of a DSML with GMF, the following models must be provided: *.gmfmap*, *.gmfgraph* and *.gmftool*. The characteristics of graphic elements such as nodes and edges are defined using the *.gmfgraph* model. The *.gmfmap* model relates the elements of the Ecore meta-model with the graphical ones. Finally, the *.gmftool* model is used to specify the palette of the editor. To facilitate the creation of these three models, EuGENia permits annotating the meta-model elements with the expected graphical syntax, and automatically generates a GMF editor from the annotations. EuGENia is part of the Epsilon project supported by the University of York.

Besides the frameworks previously described we should mention Sirius [115]. This framework was created by Obeo and Thales as a commercial tool. Since 2013, it became part of Eclipse and began to be very popular in the MDE community [72]. The definition of concrete syntaxes in this tool is by constructing an *.odesign* model that describes the shapes for nodes, the style for edges, the mappings of graphical elements to meta-model elements, the elements in the palette, and the actions to be performed when palette elements are selected. The *.odesign* model encompasses all characteristics of the three GMF models (*.gmfmap*, *.gmfgraph* and *.gmftool*). Sirius is interpreted while GMF, which needs the generation of Java code. In this PhD, we use Sirius as the technology to create graphical modelling editors.

The development of graphical editors demands developers with experience in the use of MDE technologies. In order to isolate developers from the complexity of the tools, López-Fernández and collaborators [83, 84] proposed an iterative process to build meta-models driven by examples created by domain experts using informal drawings tools such as yEd or Dia. Starting from these examples, a meta-model and its associated modelling environment are automatically generated. The approach was implemented in a tool called metaBup.

Outside of the Eclipse ecosystem, we can create graphical environments using tools like MetaEdit+ [124], OpenFlexo [57], or FlexiSketch [132]. MetaEdit+ offers two tools to develop DSMLs: MetaEdit+Workbench to design the language and MetaEdit+Modeler to configure the modelling tool. MetaEdit+ employs the Graph-Object-Property-Port-Role-Relationship (GOPPRR) for the abstract syntax definition, which is a proprietary meta-modelling language. Openflexo provides an infrastructure for multifaceted modelling and the generation of artifacts. The resulting tools are the Viewpoint Modeler and View Editor. For the abstract syntax definition, Openflexo provides it own language based on XML, but it is also compatible with EMF. The

OpenFlexo and MetaEdit+ tools have desktop versions. FlexiSketch provides a desktop version, but also an Android application for mobile devices implemented with the 2D gaming framework Corona. FlexiSketch permits the creation of free-form sketches to semi-automatically create a simple meta-model. With respect to the abstract syntax, FlexiSketch is compatible with GOPPRR and also provides its own language based on XML.

There are other tools like FlexiSketch, that permits the creation and edition of models from a remote location. That is the case of a prototype tool called DSL-comet [127] to enable mobile domain-specific modelling. This tool allows also collaboration by using the short-range communication capabilities of mobile devices like Bluetooth or WiFi. In addition, DSL-comet supports a combined scenario in which a model can be created using a desktop environment and then it can be used in a mobile context.

In addition to tools that allow the construction of modelling environments there are widely used visualization tools to explore and analyse graphs. Perhaps the most well-known is Gephi [9], which includes a force-based layout, enables the calculation of metrics, and dynamic graph analysis. This tool aims to visualize large graphs and according to its web page, it can handle up to 100.000 nodes and 1.000.000 edges.

None of the revised tools to develop graphical editors, are able to evaluate the quality of the visual notations defined. In order to fill this gap, CEViNEdit [54] is a tool that assesses the quality of graphical notations taking into account the Moody's principles [92]. Moody proposes nine principles to explain and predict why some visual notations are more accurate than others, being able to evaluate and improve some graphical notations. Using the Moody's criteria, CEViNEdit provides some metrics to assess whether the concrete syntax has adequate perceptual features so that the models can be easily understood and built. This tool is based on Eugenia and provides a tree-based editor to annotate the Ecore meta-model.

Table 2.2 summarizes the existing graphical frameworks. Many of them use Eclipse as an IDE and Ecore to define the abstract syntax. This entails that EMF is being widely used for the creation of modelling environments. The table is organized by the IDE on which the tool is based and by the year in descending order.

Models are usually created monolithically and may reach millions of elements depending on the system complexity, the domain represented or a combination of both. The frameworks described above do not provide natively any type of modularity mechanisms for models, and have scalability issues to show large models. Adding modularity mechanisms to the DSMLs created with these frameworks would require manual programming. Another shortcoming of these tools is that they do not provide

any guidance or heuristic to define the concrete syntax. In this thesis, we propose heuristics to define graphical editors and a mechanism to generate modular modelling environments based on patterns.

Graphical Frameworks	**Base Technology**	**IDE**	**Abstract Syntax**	**Concrete Syntax**	**Last Update**
AToM^3 [81]	Python/ Tcl/Tk	Own	Python	Forms/ graphical editors	2008
Diagen/ DiaMeta [89]	Java 2D	Own	Ecore	Hypergraph grammars/ Ecore	2009
FlexiSketch [132]	Corona	Own	XML/ GOPPRR	XML	2013
Gephi [9]	OpenGL 3D	Own	-	-	2017
OpenFlexo [57]	EMF	Own	XML	XML	2018
MetaEdit+ [124]	Own	Own	GOPPRR	Forms / graphical editors	2018
Spray [51]	Graphiti	Eclipse	Ecore	Textual DSL	2013
Xdiagram [110]	Graphiti	Eclipse	Ecore	Textual DSL	2017
MetaBup [83, 84]	EMF/ Stencil	Eclipse	Ecore	Examples as informal drawings	2017
GMF [52]	GEF	Eclipse	Ecore	Model-based (*.gmfgraph*, *.gmftool* and *.gmfmap*)	2018
Eugenia [76]	GMF	Eclipse	Ecore	Annotations	2018
Graphiti [55]	GEF & Draw2D	Eclipse	Ecore/ Java-based	Java API	2018
CEViNEdit [54]	Eugenia	Eclipse	Ecore	XMI	2018
Sirius [115]	GMF	Eclipse/ Obeo designer	Ecore	Model-based (.odesign)	2019

Table 2.2 Summary of tools to develop graphical editors.

2.3 Summary and Conclusions

In this chapter, we introduced the basic concepts of MDE and identified challenges that this thesis will address in the next chapters. In addition, the existing modelling technologies were described, with a particular attention to the frameworks for creating graphical editors. Furthermore, we discussed approaches to the systematic development of DSMLs and advances made regarding scalability and modularity.

In the next chapter, we will introduce patterns as a key to develop DSMLs, and describe their types, variants and the steps to instantiate these patterns at the meta-model level.

Chapter 3

Patterns

This thesis uses patterns to describe services for DSMLs, including both, modularity and concrete syntax services. Therefore, this chapter describes in Section 3.1 different types of patterns to assist in the definition of DSMLs. Next section (Section 3.2) describes how patterns are applied and instantiated at the meta-model level. Section 3.3 presents how to add functionality to patterns. Finally, Section 3.4 introduces the notion of pattern variants to enable a more flexible pattern application.

3.1 Types of Patterns

The definition of a DSML encompasses several aspects [105]. The first one concerns its abstract syntax, which should gather the primitives of the domain, realized in a high-quality meta-model. The second aspect deals with the representation of the DSMLs, either textually or graphically, through a concrete syntax. In the case of a graphical syntax, aspects like layouting or zooming (e.g., through filters or hierarchical grouping) may also be specified. Third, the DSML semantics specifies the meaning of models, e.g., through simulation, execution, model transformation or code generation. Finally, the editing of models of a DSML is usually performed using a dedicated modelling environment which provides services like model persistence and model conformance checking. One may define patterns to address all these aspects:

- **Domain patterns**. These patterns characterize a family of DSMLs, gathering requirements of similar languages within a domain, and documenting their variability. For example, there may be patterns for workflow languages, expressions (e.g., arithmetic, logical), variants of state machines, query languages, and component/connector architectural languages. A single DSML may use several of

these patterns, customized for a given need, and probably extended with other domain-specific concepts.

- **Design patterns**. These lower-level patterns are concerned with the meta-model design. Some examples include patterns describing different options to realise tree-like structures [15], lists, containment relations [26], connectors, or the type/instance relation [86].

- **Concrete syntax patterns**. They characterize families of DSMLs with similar representation [16]. For example, graph-based languages depict concepts using nodes and arrows; hierarchical graph languages represent in addition hierarchy; and tabular languages use columns and rows. Moreover, some domain patterns may attach a predefined instantiation of a concrete syntax pattern, customized for the domain. For example, state machines may be represented with a particular instantiation of a hierarchical graph pattern, where states are depicted as ovals with the name (and maybe other states) inside, and transitions are shown as arrows. Similarly, workflow languages may attach a graph pattern instance that represents each gateway type as a rhombus with a different decoration.

- **Dynamic semantics patterns**. These patterns describe the participant roles in different styles of semantics [18]: Petri-net like, variations of state machines, event-based semantics, data-flow semantics, etc. Alternatively, the semantics could be given by transformations into a semantic domain, or via code generation.

- **Infrastructure patterns**. These patterns identify services typically provided by modelling environments, but which need to be configured for a particular DSML. Some examples of this kind of patterns include model fragmentation strategies allowing the hierarchical decomposition of large models into folders and model fragments [48], model abstraction services to obtain a simpler view of a model containing the subset of elements of interest [79], and different layouts for graphical DSMLs.

Patterns can be used in two ways. First, as a means to raise the productivity, repeatability and reliability of the meta-model construction process, by incorporating the pattern elements (i.e., a meta-model fragment) to the DSML meta-model. Some elements of the pattern may already exist in the meta-model, in which case, only the missing ones are added. This usage is typical for domain and design patterns. In the second way, patterns are a means to configure functionality for the DSML by identifying the pattern elements (or roles) with existing meta-model elements. This is typically the

case for concrete syntax, dynamic semantics and infrastructure patterns. In practice, one often has intermediate situations. Once a pattern is applied, the meta-model elements identified or created by the pattern are "annotated" with the pattern roles. In addition, patterns may offer services to be used together with other patterns, like a layout pattern for certain concrete syntax pattern. To define this interaction, patterns can publish the services they provide or require as pluggable components through suitable interfaces.

This thesis focuses on infrastructure and concrete syntax patterns. Chapter 4 covers the identification and description of a catalogue of modularity patterns for the creation of scalable modelling environments for DSMLs, and Chapter 5, presents the definition of the graphical and tabular representation pattern in order to customise concrete syntaxes. Next section, proposes a specific approach to specify and apply patterns to meta-models.

3.2 Pattern Specification

Our notion of pattern is meta-level independent, as patterns can be applied to models or meta-models. For simplicity of presentation, we assume that they are applied at the meta-model level only. In this section, the main ingredients of patterns (structure, instantiation and application) are introduced.

The structure of a pattern is specified by a meta-model, and its elements (classes, references, and attributes) are called *roles* [78]. Each role defines a cardinality interval which governs how many times the role can occur in a pattern application. If a role does not define a cardinality explicitly, then it is assumed to be of cardinality [1..1]. Class roles can be tagged with the stereotype abstract, in which case the class role cannot be instantiated but is a placeholder for attribute or reference roles that get inherited by children class roles. Since roles tagged as abstract cannot be instantiated, they do not have any cardinality.

Class roles may have two kinds of fields[1]: *field roles* and *configuration fields*. On one hand, field roles must be mapped to fields with a compatible type in the domain meta-model, while their name can be different. Just like class roles, field roles define a cardinality range ([1..1] by default). On the other hand, configuration fields are not mapped but need to receive a value when the pattern is applied. Inspired by deep characterization in multi-level modelling [6, 80], we tag the configuration fields with "@1" (potency 1) as they receive a value when the pattern is instantiated one meta-level

[1]We uniformly refer to attributes and references as fields.

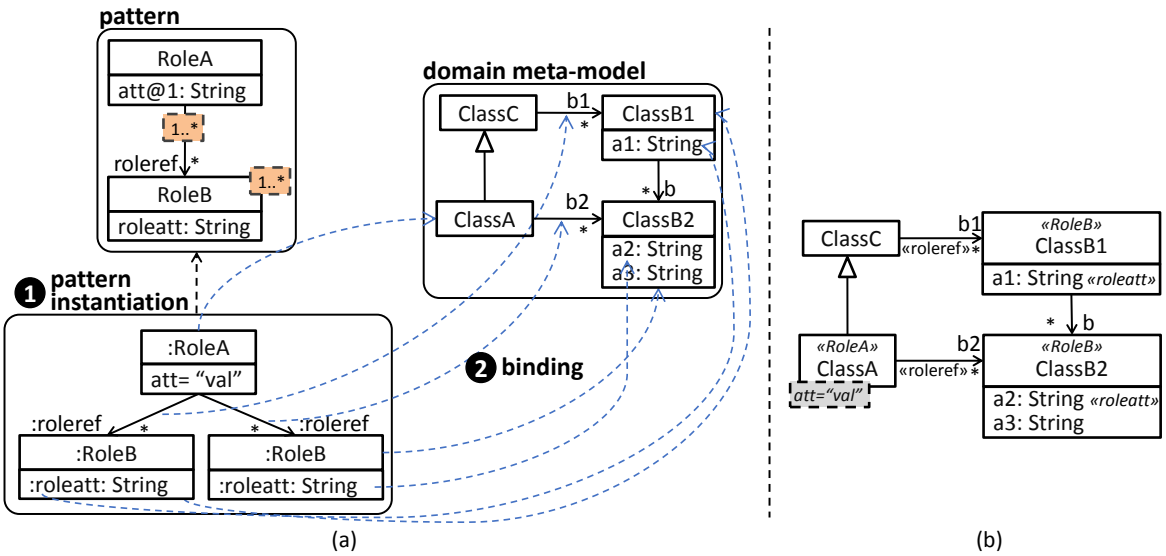


Fig. 3.1 (a) Example of pattern application to domain meta-model. (b) Visualization of applied pattern.

below, while the field roles have potency 2 as they receive a value two meta-levels below. We omit the inscription "@2" in the field roles for readability reasons.

Figure 3.1(a) exemplifies the application of a pattern (in the upper-left corner) to a meta-model (in the upper-right corner). We use a synthetic example to better illustrate all features of our patterns and their application, and refer to Section 4.3 for the catalogue of proposed patterns. The example pattern has two class roles, one reference role, one attribute role, and one configuration attribute (marked with "@1"). RoleA and roleatt do not specify a cardinality, hence they are assumed to have [1..1] interval; RoleB and roleref define cardinality [1..∗]. Note that field roles have two cardinalities: a role cardinality which governs the number of instances of the role ([1..∗] for roleref), and a field cardinality which must be compatible with the field mapped in the domain meta-model (∗ for roleref).

The application of a pattern to a domain meta-model proceeds in two steps. First, the language designer instantiates the pattern as a regular meta-model, respecting its role cardinalities. Then, he/she needs to bind the elements in the pattern instance to elements in the domain meta-model (class roles to domain classes, attribute roles to domain attributes, and reference roles to domain references). This binding allows structural matching, i.e., if a class role r is mapped to a domain class c, then the field roles inside r must be bound to fields owned or inherited by c. In addition, some patterns may define extra conditions expressed in OCL or Java to restrict the bindings considered correct [105].

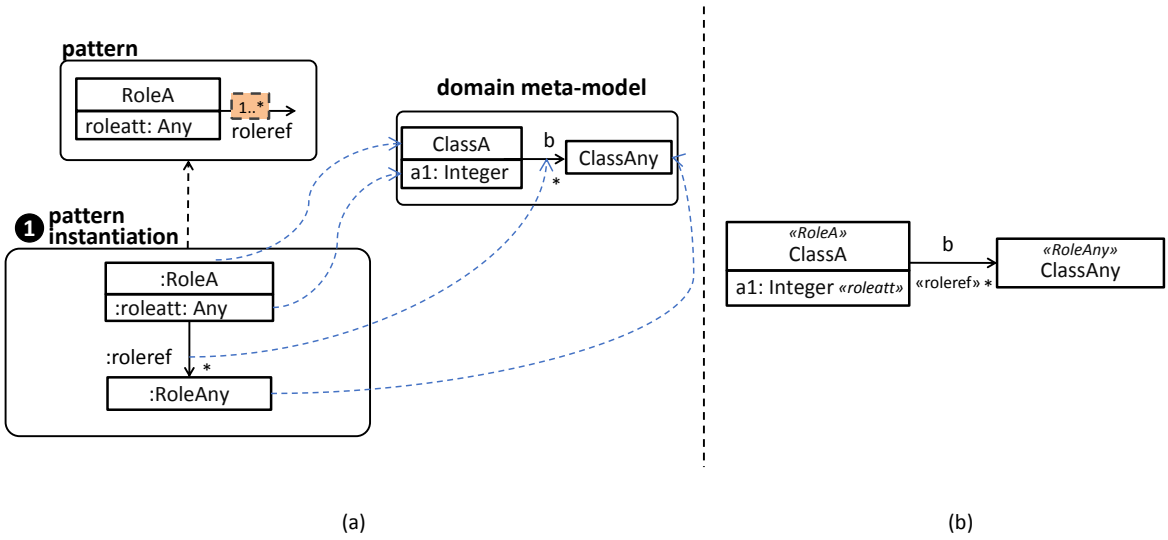


Fig. 3.2 (a) Example of application of pattern with field roles without type. (b) Visualization of applied pattern.

As a first step, the pattern is instantiated. In the example, the created pattern instance contains one instance of RoleA, two instances of RoleB, and the configuration attribute att receives the value "val".

In a second step, the pattern instance elements are bound to elements of the meta-model. In Figure 3.1(a), the binding (depicted as dotted arrows) maps :RoleA to ClassA; one :RoleB to ClassB1 and the other to ClassB2; one :roleatt to a1 and the other to a2; and the two :roleref to two references of ClassA, one owned and the other inherited. In general, one element in the domain meta-model is allowed to receive several roles. For example, if ClassA had a self-reference r and a String attribute a, then both :RoleA and :RoleB could be mapped to ClassA, with :roleref mapped to r, and :roleatt to a.

Figure 3.1(b) visualizes the domain meta-model with the applied pattern. Roles are depicted as stereotypes on the mapped class or field, and configuration fields (like att) are shown in a box attached to the stereotyped class.

Note that a pattern definition also admits reference roles, with no target class role and attributes with any data type. Figure 3.2(a) shows a pattern with a class role RoleA which contains an attribute role roleatt that can be mapped to an attribute of any data type. The figure also shows a reference role roleref that can point to any target class.

Domain meta-models can be defined instantiating one or several patterns. As the previous examples show, if an instance of a role is mapped to a domain element, then the element will get tagged by the role. However, if some role instance is left unmapped, then a new element will be added to the domain meta-model playing that role. In this

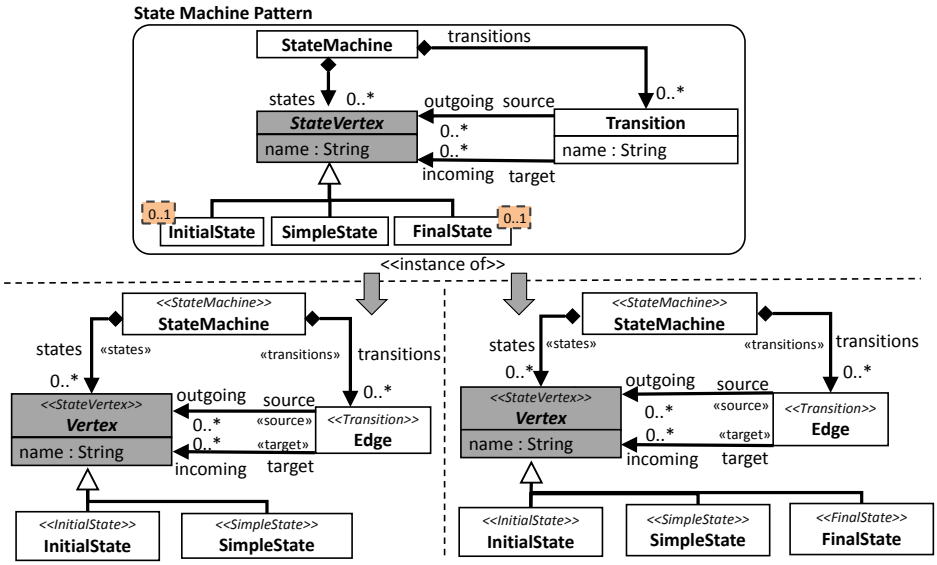


Fig. 3.3 State machine pattern and some valid instantiations.

sense, it is possible to bind existing meta-model elements to roles in the pattern, but not every pattern role of the chosen instance will be necessarily created new in the meta-model.

Figure 3.3 illustrates the definition of a domain pattern and its application to our running example. Specifically, the pattern corresponds to the definition of a state machine. In this case, the pattern is used in the WT meta-model (Figure 2.3) to control the system behaviour, through the definition of states and transitions. The cardinality of all roles in this pattern is 1, except the InitialState and SimpleState roles which have cardinality [0..1]. In this sense, the variability results from the instantiation or not of these optional roles.

The lower right and left corners of Figure 3.3 show two valid instantiations of the state machine pattern. The lower left corner, shows a valid instantiation which correspond to the classes that define a state machine in the WT meta-model. In this case, all roles with cardinality 1 have been mapped and also the SimpleState class role. The lower right corner shows a meta-model that maps all the roles of the pattern.

The pattern roles elements may contain additional constraints defined, for example, using the OCL language. Listing 3 shows an invariant to force the Transition class to have exactly two references. This constraint use eAllReferences which return all outgoing references of the class.

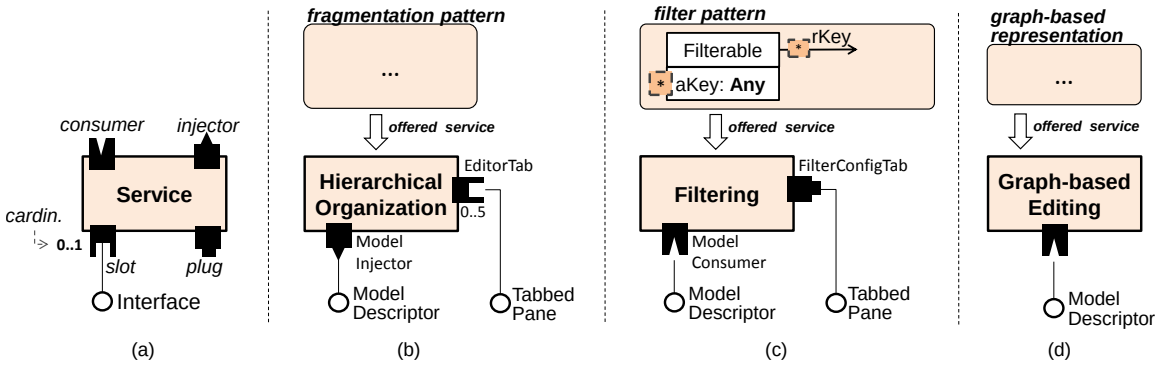


Fig. 3.4 (a) Schema of pattern services. (b,c,d) Services in the running example. (e) Service composition.

```
1 context Transition
2 inv checkNumberOfReferences :
3 self.eAllReferences.size() = 2
```

Listing 3 Example of OCL constraint for the State Machine pattern.

3.3 Pattern Services

Patterns may include services contributing functionality to the environment generated for a DSML. Figure 3.4(a) shows the general schema of a pattern service. A service is encapsulated as a component, which may define any number of ports. Each port declares an interface and can be of four different types: *slot*, *plug*, *injector* and *consumer*. Services can be connected through their ports if their types and interfaces are compatible. Regarding type compatibility, slots are compatible with plugs, and injectors are compatible with consumers.

A slot represents a functionality "hole" in a service, to be provided by another service that declares a plug with a compatible interface. Moreover, slots have a cardinality which constrains the allowed number of plugs that can be connected to them. Typically, the functionality provided by a plug needs to be deployed in the context of a service with a compatible slot, while slots with a minimum cardinality 1 need to be connected to a compatible plug to obtain a proper behaviour. On the other hand, an injector port is an emitter of information populated by a service. This information can be used by a consumer port with a compatible interface. In this way, a connection between an injector and a consumer induces a dependency injection from the service defining the injector to the service defining the consumer.

For example, Figures 3.4(b) and (c) show two patterns that define two services that we would like to have in the modelling environment for the running example. The

first one is the fragmentation pattern that will be explained in Section 4.3.1. This allows generating a modelling environment where models are organized hierarchically into projects, packages and files of different types, similar to the organization of Java software projects. The associated service has a slot named EditorTab that allows other services to extend the environment with tabs that contribute further functionality. The service also defines an injector which supplies to consumers a ModelDescriptor object with information of the model unit selected in the environment.

The pattern in Figure 3.4(c) allows customizing model filters. The classes amenable to be filtered should be bound to Filterable, while the attributes and references to be considered in the filtering conditions should be bound to aKey and rKey, being possible to have any number of them. The actual type of the attributes bound to aKey and the references bound to rKey are unimportant. This pattern defines a service that is compatible with the previous HierarchicalOrganization service: on one hand, it has a plug named FilterConfigTab that contributes a tab to control the model filter behaviour; on the other hand, it has a consumer port named ModelConsumer from which the service will obtain the model unit to filter, selected in the environment.

The service in Figure 3.4(d) is associated to the concrete syntax pattern graph-based representation which will be described in Section 5.2. This service generates an editor to build models using the defined graphical concrete syntax. The service needs a model descriptor, which will be provided by the HierarchicalOrganization service.

When a pattern is applied, its services become available. Most of the times, these are realize via code generation. Hence, a generator synthesizes segments contributing functionality to the final environment.

3.4 Patterns Variants

The cardinality of pattern roles allows their fine-grained customization for a context. In addition, a pattern may have variants accounting for coarse-grained alternatives in the pattern structure. For example, transitions in a state machine may be represented as references between two states, as hyperedges connecting multiple source to multiple target states, or with an intermediate class storing properties of the transition. Only the latter two cases allow associating a trigger to transitions. This variability in the pattern structure would be challenging to express with role cardinalities alone.

For this reason, we can equip patterns with a feature model [64] that defines the coarse-grained variability of patterns. As an example, the left of Figure 3.5 shows the feature model attached to the pattern for state machines. It defines three alternatives

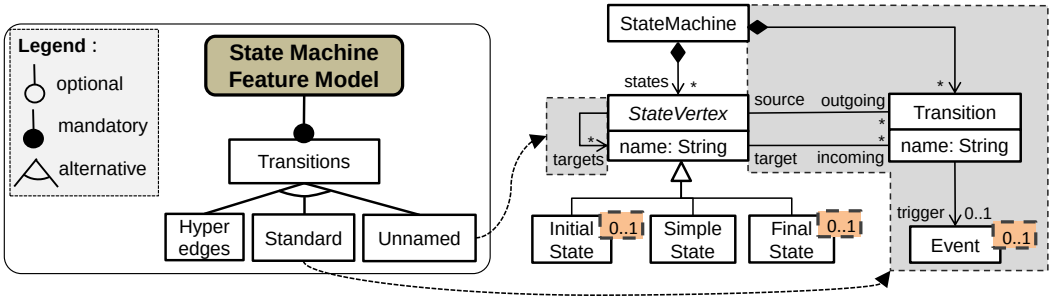


Fig. 3.5 Feature model for state machine pattern variants.

for the design of transitions (Hyperedges, Standard and Unnamed), and each alternative is exclusive (i.e., only one can be selected at a time). Hence, transitions can be either a class like in Figure 3.3 (variant Standard), a hyperedge, or a reference (variant Unnamed, in which case there are no classes for the roles Transition and Event, but there is an association starting and ending in StateVertex instead).

3.5 Summary and Conclusions

This chapter has described how patterns assist developers in the creation of new DSMLs. The application of patterns and their variants allow the definition of a wide range of meta-models. In this thesis, we use patterns to create meta-models, but also to associate different services to their classes. For example, the pattern of state machines may have associated a concrete syntax by default. Altogether, patterns not only encourage a well-designed abstract syntax, because they also can be used in other aspects concerning the development of DSMLs.

The next chapter presents a catalogue of modularity patterns and its associated services to provide scalability to DSML environments.

Chapter 4

A Pattern-based Approach to Language Modularity

This chapter presents a detailed description of the proposed patterns to define modularity services for DSMLs. Section 4.1 presents the running example and identifies some difficulties in creating scalable environments. Section 4.2 provides an overview to build these environments through patterns. Finally, Section 4.3 presents our catalogue of modularity services and their associated patterns with a detailed description of each one using our running example.

4.1 Motivation and Running Example

In this section, we introduce a motivating running example from which we elicit a number of requirements for scalable modelling environments.

As an example, we will be using the language meta-model shown in Figure 2.3, and described in Section 2.2.1. This meta-model allows describing two aspects of wind turbine control systems: (i) the constituent types of system components and how they can be connected (classes Component, Port, InPort, OutPort and Connector); and (ii) the admissible states and state changes of those component types (classes StateMachine, DocumentElt, Vertex, InitialState, SimpleState and Edge). In addition, the meta-model provides classes to organize components into hierarchies of subsystems (class Subsystem), as well as to group the state machines in each subsystem (class ControlSubsystem).

To be able to define models using the architectural language, we would like to have a customised environment with typical modelling facilities like model editing, conformance checking, model search, etc. Since we expect control systems to consist of many components, the environment should be optimized to deal with large models

from the tool perspective (performance) and the user perspective (usability). Building this environment by hand is possible but costly. Instead, using a meta-modelling-based language development framework for its construction is faster.

As we discussed in Sections 2.2.1 and 2.2.4, examples of graphical and textual language development frameworks include GMF [52], Sirius [115] and Xtext [13]. However, these frameworks typically yield environments for editing monolithic models, i.e., models with all their elements included in the same file/resource. As a consequence, these environments have performance problems when managing big models [129]. Moreover, having monolithic models is not optimal in our example, as the language clearly identifies two different concerns (structure and behaviour) and so a mechanism that enables their separation is desirable [33]. Additionally, the language provides primitives (nested subsystems, control subsystems) that may be used to organize the model content in packages according to the subsystems structure. Even though modelling frameworks like EMF [118] permit cross-referencing elements across files, they lack a native way to define and enforce fragmentation policies, or to organize a model into packages (the latter is available for meta-models but not for models). While some language frameworks like Sirius support the definition of diagram types, they do not provide mechanisms to combine several model fragments into a unified model, or to map parts of the model structure to the file system (like packages in Java). Moreover, the fragmentation strategy should be intrinsic to the abstract syntax, not tied to specific concrete syntaxes.

Once a model is fragmented, it is desirable to control which elements can be cross-referenced from other model fragments. For example, we may wish to restrict a Component to reference only those StateMachines located in ControlSubsystems within the same Subsystem the Component belongs to. While it is possible to write an OCL constraint that checks this, an advanced modelling environment would filter out all StateMachine objects that are out of the scope. Frameworks like Xtext support scopes, but these are tied to the concrete syntax, are normally defined in low-level programming languages, and require deep knowledge of the framework.

Meta-models may include integrity constraints, typically specified using OCL, to be satisfied by models. As an example, Listing 4 shows two constraints demanding that every input port is connected to some output port, and vice versa. Constraints are defined in the context of a class, and evaluated on every instance of the class that is contained in a model, which is time-consuming for big models. Instead, we may take advantage of the fragmentation of models to scope the evaluation of constraints to smaller fragment units. In the running example, this means that whenever a component

is changed, only the ports in that component should get their constraints re-evaluated, but not the rest of ports.

```
1 context InPort
2 inv inputPortConnected :
3 Connector.allInstances()→exists(c |
4   c.inPort = self and not c.outPort.oclIsUndefined())
5
6 context OutPort
7 inv outputPortConnected :
8 Connector.allInstances()→exists(c |
9   c.outPort = self and not c.inPort.oclIsUndefined())
```

Listing 4 OCL constraints for the WT meta-model.

Finally, searches on big models can be slow. One way to tackle this problem is the use of model indexers and indices of relevant attributes to speed up the search [46]. However, since building an index incurs a time overhead, it should be possible to customize the subset of attributes to be indexed (only those used in recurrent searches).

Altogether, the modelling environment for the proposed architectural language has to fulfil the following requirements, which indeed are general requirements of scalable modelling environments:

R1 Ability to define fragmentation strategies for models, to enforce the separation of concerns.

R2 Ability to organize the model content hierarchically into packages or folders, to improve usability.

R3 Ability to control the visibility of elements across fragments, and the scope of cross-references, to manage complexity through information hiding.

R4 Ability to customize the scope of integrity constraints, to improve their validation performance.

R5 Ability to define model indices, to improve the performance of recurrent searches.

4.2 A Pattern-based Approach to Modularity of DSMLs

Figure 4.1 shows a scheme of our approach to the development of modelling environments with support for modularity services. The approach provides a catalogue of modularity services expressed as meta-model patterns, which can be applied to the

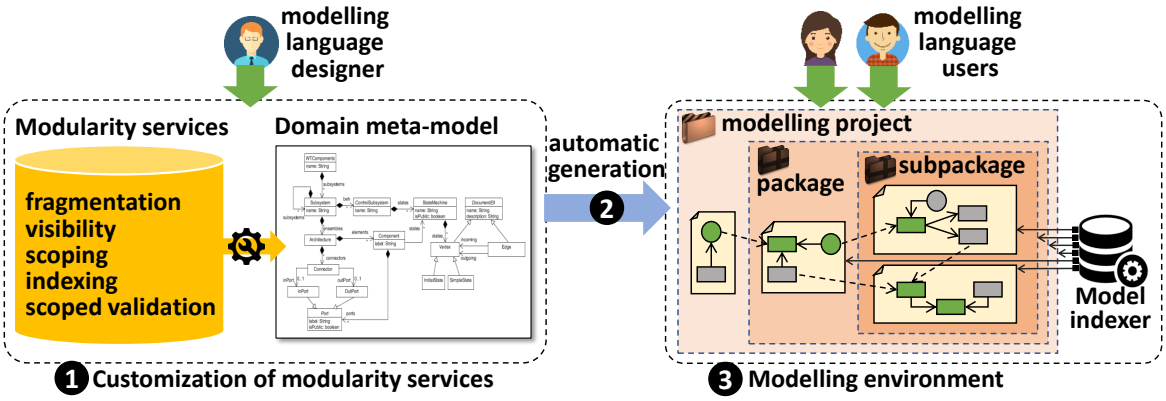


Fig. 4.1 Overview of our approach to build modelling environments with modularity services.

domain meta-model for which the environment is being developed, as explained in Chapter 3 (label 1). The modelling language designer can apply as many patterns as required to the domain meta-model. Each pattern application produces a customization of the corresponding service for the domain meta-model. Section 4.3 will present our catalogue of modularity patterns, which covers services for model fragmentation, object visibility, reference scoping, attribute indexing and scoped validation.

Technically, our modularity patterns have the form of a meta-model and can be applied to the domain meta-model, following the process described in Section 3.2. From this definition, a modelling environment that integrates modularity services configured in compliance with the instantiated patterns is automatically synthesized (labels 2 and 3 in Figure 4.1). This environment features model indexers that improve the efficiency of searches across model fragments and manage inconsistent cross-references when fragments are moved to a different location [108]. The following section describes the supported modularity patterns, and Chapter 6 will describe their tool support.

4.3 Catalogue of Modularity Patterns and Services

In this section, we present our catalogue of modularity services and their associated patterns, which are illustrated using the running example.

4.3.1 Model Fragmentation

Programming languages offer techniques for dividing a program into smaller building blocks. This helps developers to tame the system complexity, and increments the

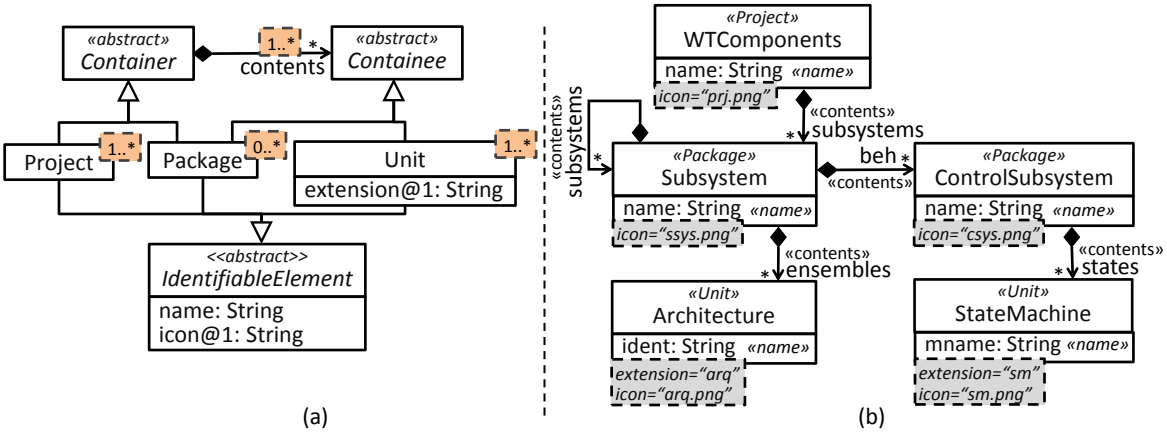


Fig. 4.2 (a) Fragmentation pattern. (b) Applying the fragmentation pattern to the running example.

tooling efficiency to perform tasks such as syntax and type checking, compilation and linking.

Similarly, to scale modelling to larger systems, we propose transferring the concept of fragmentation to DSMLs. This provides benefits in terms of model comprehensibility and processing efficiency (see Section 7.4). Moreover, model fragments can be reused more easily, and teams of engineers working collaboratively on a fragmented model will potentially have fewer conflicts in version control systems than when working on a monolithic model.

To support model fragmentation, we take inspiration from the modularity concepts of languages like Java and its Java Development Tools (JDT) Eclipse programming environment [62]. Eclipse JDT allows creating Java *projects*, and the language permits breaking programs physically into compilation *units* (classes, interfaces and enumerations residing in different files) which are organised into hierarchies of *packages*. Projects, folders and files are located in the *workspace* organized into a tree structure with projects at the top, and folders and files underneath.

We have adapted these ideas to DSMLs by defining the *fragmentation* pattern shown in Figure 4.2(a). The pattern defines Project, Package and Unit as class roles. Designers can configure which classes of a DSML will play those roles. This way, the instances of the DSML will not be monolithic, but they will be structured into projects, packages and units according to the given strategy.

Typically, the root class that contains directly or indirectly all other classes of a DSML should be mapped to Project. The pattern declares a condition (omitted in the figure) checking that all classes bound to Project are root. As a result, each time the root class is instantiated in the generated environment, a new modelling project

(i.e., a folder that will hold all fragments of a model) is produced. To account for meta-models that have several root classes, role Project has cardinality [1..∗]. Similarly, when a class with role Package is instantiated, the environment creates a folder in the file system, together with a hidden file storing the value of the class attributes and non-containment references. Finally, instantiating a class c with Unit role results in the creation of a file that holds instances of the classes that can be directly or indirectly reached from c by means of containment relations.

In the pattern, all concrete class roles inherit the attribute role name to be used as the project, folder or file name, and the configuration attribute icon to specify the icon file to be used as a decorator in the generated environment. Units can also indicate a file extension using the configuration attribute extension. Containers (i.e., Projects and Packages) and Containees (i.e., Packages and Units) are related through the reference role contents. This must be mapped to a containment reference in the DSML meta-model, modelling the inclusion of files in folders, and these in projects.

As explained in Chapter 3, classes in the domain meta-model are allowed to play several roles. For example, a class C can be both Package and Unit. In such a case, upon creating a C object, the user decides whether a package or a file should be created. Similarly, a class can be both Project and Package (which is implicit for Project classes that have self-containment references); as well as both Project and Unit (to allow the creation of monolithic models, if desired).

Example Figure 4.2(b) depicts the application of the fragmentation pattern to the meta-model of Figure 2.3. The role Project is assigned to the class WTComponents. Two types of packages are defined: Subsystem, which can recursively contain other Subsystem packages, and ControlSubsystem. There are also two unit types: Architecture and StateMachine. These classes will be physically persisted as files with extensions "arq" and "sm" respectively, and may store objects of the classes contained in them. For instance, a unit of type Architecture can hold objects of types Architecture, Component, Connector, InPort and OutPort (see Figure 2.3). As an example, Figure 4.3(c) shows a model organised according to the defined fragmentation strategy.

4.3.2 Reference Scoping

When a model is monolithic, its objects can refer to other objects within the same file, according to the reference types defined in the model's meta-model. However, when a model is fragmented, some references may need to cross fragment boundaries. In this context, there is the need to control the fragments a reference can reach, that

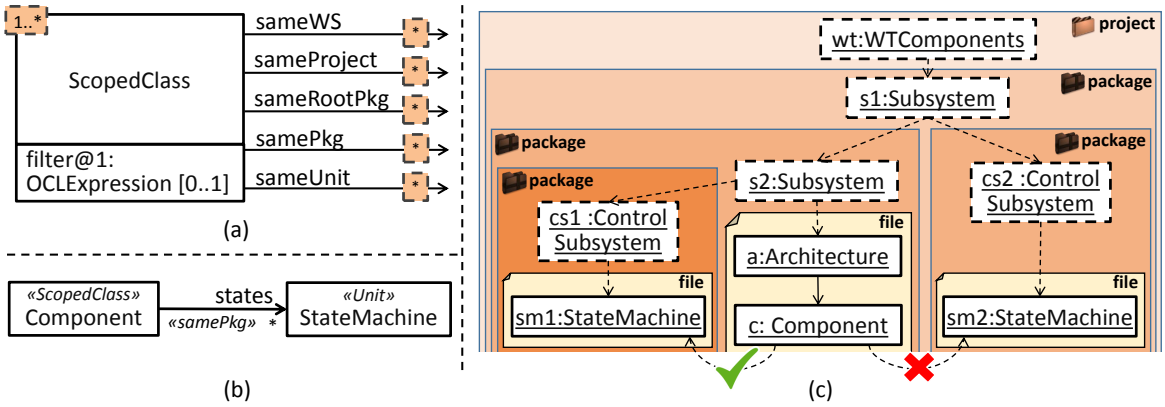


Fig. 4.3 (a) Scoping pattern. (b) Applying *same package* scope to reference Component.states. (c) Effect of pattern application on a fragmented model.

is, its *scope*. This control mechanism is useful to manage the access modifiers for the classes of a given DSML, and to reduce the set of potential candidate objects for a given reference.

We allow customizing this information using the *scoping* pattern in Figure 4.3(a). This declares a single class role ScopedClass, which should be mapped to the domain class owning or inheriting the reference to be scoped. In its turn, this reference should be mapped to one of the five reference roles in the pattern, which represent five different scoping policies. The least restrictive policy is sameWS, which allows a reference to refer to objects in the same workspace, i.e., anywhere in any project. This is the default option for references with no scope. The sameProject policy permits referring to objects within the same project. The policies samePkg and sameRootPkg allow referencing objects within the same package, or that have the same root package, respectively. Finally, sameUnit restricts a reference to the objects residing in the same file. In any of the cases, it is possible to further constrain the scope of the reference by providing an OCL expression to filter out additional unwanted objects.

It can be noted that since our notion of pattern supports reference roles with no target class role, there is no need to map the latter to any domain class. Moreover, the reference roles in the scoping pattern can be mapped multiple times to meta-model references, but they do not impose any cardinality to those references (i.e., they do not specify any reference cardinality).

Example Figure 4.3(b) assigns the *same package* scope to the reference Component.states, and Figure 4.3(c) shows the effect of this scope on a fragmented model. The scope allows connecting the Component object c to any StateMachine located in

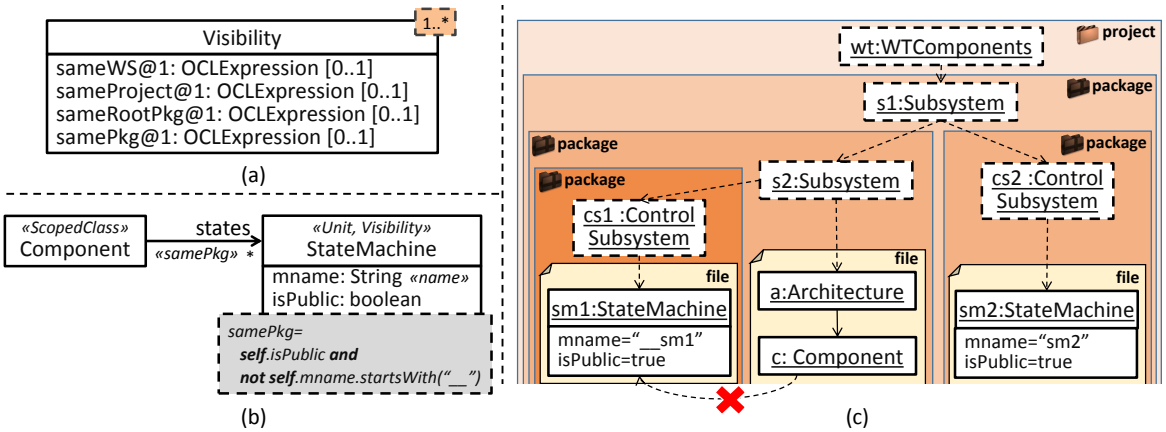


Fig. 4.4 (a) Visibility pattern. (b) Applying *same package* visibility to class StateMachine. (c) Effect of pattern application on a fragmented model.

the container package of c (i.e., in the Subsystem s2). Therefore, c can refer to sm1 as this is (indirectly) contained in s2, but not to sm2 because it is in a different package. Changing the reference scope to *same root package* would allow connecting c to sm2.

4.3.3 Visibility

Programming languages like Java allow classes to control whether other classes can use a particular field or method by means of access level modifiers. Similarly, we allow model fragments to define an interface to expose only a subset of its elements to other fragments, while hiding the rest.

By default, objects are accessible from any other fragment, but this can be constrained by using the pattern in Figure 4.4(a). This pattern allows defining whether the instances of a class are visible to other fragments in the same workspace, project, root package or package. An object is always visible within its file. The visibility is configured by means of an OCL expression which is evaluated on every object of the class, and only the objects satisfying the expression become visible to other fragments in the given scope.

A class can define visibility conditions for different scopes, e.g., *same project* and *same package*. In such a case, if the visibility conditions of several scopes are satisfied, the more general scope is selected. For example, if the visibility conditions for both scopes *same package* and *same project* evaluate to true, then the object is visible at the project level; otherwise, the object is only visible at the level of the expression that evaluates to true.

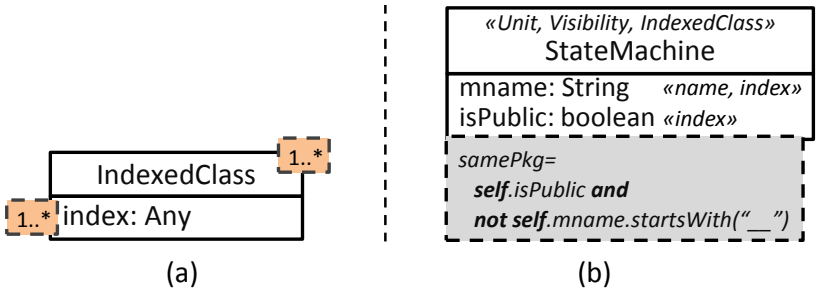


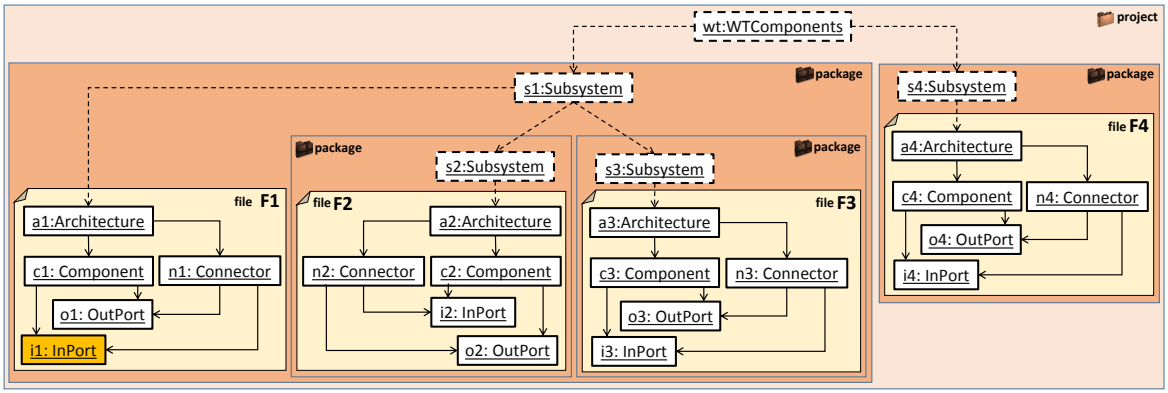
Fig. 4.5 (a) Indexing pattern. (b) Applying pattern to class StateMachine.

The visibility and scoping patterns are complementary. While scoping restricts the content of a reference, visibility restricts the usage context of an object. If a certain class of objects should be visible to all fragments in its package or project, then specifying class visibility is equivalent to specifying the corresponding scope for every cross-reference pointing to the class; however, in this case, the first option is less costly as it is done once for the class instead of once per reference.

Example Figure 4.4(b) applies the visibility pattern to class StateMachine. The OCL condition, which is defined for samePkg, appears in a grey box. In this way, StateMachine objects will only be visible from other fragments in the same package when they are public and their name does not start by a double underscore. Figure 4.4(c) shows the effect of this pattern on a fragmented model. In the figure, the StateMachine sm1 is not visible from other fragments in the same package because its name starts by a double underscore; therefore, it cannot be referenced from the Component c, even if sm1 is in the reference scope. On the other hand, sm2 is not visible to Component c because the visibility level is samePkg and sm2 is in a different package. Moreover, sm2 is not reachable from c because the reference states has scope samePkg.

4.3.4 Indexing

To speed up the computation of common queries over models, we support the creation of indices of objects by selected fields, backed by a model indexer. The pattern in Figure 4.5(a) allows selecting the fields to index. It is made of the class role IndexedClass and the field role index, both with role cardinality [1..∗]. This way, for each domain class mapped to IndexedClass, it is possible to specify one or more fields (attributes or references) to be used as indices. The field role defines the data type Any and no field cardinality; this means that the indexed fields in the domain meta-model can have any type and cardinality.




```
context InPort inv inputPortConnected:
  Connector.allInstances()→exists(c | c.inPort = self and not c.outPort.oclIsUndefined())
```

Fig. 4.6 A fragmented model where a scoped constraint is to be evaluated on the InPort i1.

Example Figure 4.5(b) applies the indexing pattern to select the attributes mname and isPublic as indices for class StateMachine. The motivation to index these attributes is their use in the expression samePkg specified in a previous application of the visibility pattern. This index permits improving the retrieval of StateMachine objects [46].

4.3.5 Scoped Validation

Meta-models may define integrity constraints that are evaluated on monolithic models, have access to all objects within the model, and need to be re-evaluated upon any model change. However, once a model is fragmented, it is natural to incorporate the notion of scope into the integrity constraints, so that each constraint only accesses the objects within the defined scope, and is re-evaluated just when there is a change within the scope. We call an integrity constraint that defines a validation scope a *scoped constraint*. As we will show in Section 7.4, scoped constraints can be evaluated more efficiently than standard constraints, as they consider a subset of model objects instead of the whole model, and the need of re-evaluation is less frequent. Moreover, scoped constraints decouple the constraint from the objects affected by it, while standard constraints need to explicitly select the affected objects by means of filters in the constraint expression.

As in the previous patterns, we consider five scopes for constraints: *same unit*, *same package*, *same root package*, *same project* and *same workspace*. To illustrate their implications, consider the fragmented model in Figure 4.6 and the constraint inputPortConnected shown in the lower part of the same figure.

Depending on the validation scope assigned to the constraint, expression Connector.allInstances() returns a different set of objects when it is evaluated on object i1 of file F1:

- if the scope is *same unit*, the result is Set{n1} because n1 is the only Connector object within the same file as i1, even though other Connector objects exist in the whole model.
- if the scope is *same package*, the result is Set{n1, n2, n3} because the three objects are contained in the same package as i1 (s1) directly or indirectly. Instead, evaluating the expression on i2 returns Set{n2}, as this is the only Connector within i2's container package s2.
- if the scope is *same root package*, the result is Set{n1, n2, n3} as the root package of i1 is s1. Evaluating the expression on i2 yields the same result.
- if the scope is *same project*, the expression returns Set{n1, n2, n3, n4} independently of the object where it is evaluated. If this is the only project in the workspace, we obtain the same set of Connectors when we assign the scope *same workspace* to the constraint.

Listing 5 shows the algorithm to validate scoped constraints. It considers two scenarios: *(i)* the validation of all constraints within a project, which is useful, e.g., when a project is loaded in the workspace; *(ii)* the re-evaluation of constraints upon model changes, where we efficiently restrict the re-evaluation to the subset of objects where the constraint value may have changed. We refer to scenario *(i)* as *full validation*, and to scenario *(ii)* as *incremental validation*.

The entry point of the algorithm is the scopedValidation method in lines 1–23. This receives as parameters the resource r that has changed (a unit, a package or a project), and a boolean flag full (true to execute scenario *(i)*, and false for scenario *(ii)*). For each scoped constraint in the meta-model, the method first obtains its validation context, i.e., the instances of the constrained class where the constraint needs to be evaluated (lines 5–14). In case of scenario *(i)*, the constraint needs to be evaluated on all instances of the constrained class defined in the project (line 7); in case of scenario *(ii)*, the instances are retrieved from the scope specified by the constraint (lines 8–14), and this scope is computed by the methods in lines 25–56. As an example, the method package is invoked when the constraint defines the scope *same package*, and it returns

```
1  def scopedValidation (r : Resource, full : boolean)
2      for each scoped constraint c:
3
4          -- obtain objects over which the constraint will be evaluated (context)
5          var validationContext = nil
6          if full = true and type(r) = Project then
7              validationContext = objects of type c.constrainedClass within r
8          else if c.scope = samePkg then
9              -- jump to root pkg to avoid validating c in each intermediate container pkg
10             validationContext = objects of type c.constrainedClass
11                                 within resource4scope (r, sameRootPkg)
12         else
13             validationContext = objects of type c.constrainedClass
14                                 within resource4scope (r, c.scope)
15
16         for each object o in validationContext:
17             -- obtain model fragment where the OCL expression will be evaluated (scope)
18             var validationScope = resource4scope (o.resource, c.scope)
19
20             -- validate constraint and report detected errors
21             if validationScope <> nil then
22                 var result = o.validate (c.statement, validationScope)
23                 if result = false then o.report (c.error)
24
25 def resource4scope (r : Resource, s : Scope) : Resource
26     var r4s = nil
27     if s = sameUnit then r4s = unit(r)
28     else if s = samePkg then r4s = package(r)
29     else if s = sameRootPkg then r4s = rootPackage(r)
30     else if s = sameProject then r4s = project(r)
31     else if s = sameWS then r4s = workspace(r)
32     loadResource(r4s)
33     return r4s
34
35 def unit (r : Resource) : Resource
36     if type(r) = Unit then return r
37     return nil
38
39 def package (r : Resource) : Resource
40     if type(r) = Package then return r
41     if type(r.container) = Package then return r.container
42     return nil
43
44 def rootPackage (r : Resource) : Resource
45     var root = r
46     while type(root.container) = Package: root = root.container
47     if type(root) = Package then return root
48     return nil
49
50 def project (r : Resource) : Resource
51     var root = r
52     while type(root) <> Project and root <> nil: root = root.container
53     return root
54
55 def workspace (r : Resource) : Resource
56     return $WORKSPACE
```

Listing 5 Algorithm for the validation of scoped constraints.

the received resource if it is a package, or its container if it is a unit (lines 39–42). Before the obtained resource can be used, we make sure that all other resources that it contains directly or indirectly are loaded as well (line 32). In practice, we use a cache to avoid reloading previously loaded resources. The following steps in the algorithm are the same independently of the scenario. Specifically, for each object in the validation context, the algorithm obtains its validation scope (line 18). This is calculated similarly to the validation context, but considering the resource that contains the object, instead of the modified resource. Then, the algorithm validates the constraint over each object of the validation context considering only the validation scope (line 22), and reports an error if the validation returns false (line 23).

Please note that given a constraint c with scope sameRootPkg, the algorithm avoids validating c recursively on each container package until reaching the root package. Instead, it directly jumps to the root package (line 11), and then considers the validationScope of each context object when evaluating c (line 18).

For efficiency reasons, we do not maintain all model fragments in memory, but we load them on demand. Therefore, our algorithm needs to load and merge any model fragment that is necessary to perform the validation. Line 32 in Listing 5 takes care of this. In practice, we rely on EMF proxies and their lazy loading mechanism to achieve this behavior. This way, fragments can refer to elements in other fragments by means of proxies, and only when there is the need to access to these other fragments, we load them and resolve the proxies. This can be seen as a kind of model merge.

This algorithm performs an efficient re-evaluation of scoped constraints upon model changes, as constraints are evaluated only over the objects in the validation context, which are those objects that may have been affected by the change. This kind of incrementality is coarse-grained, in the sense that it only considers the resources that have changed to identify the constraints to re-evaluate, but not the actual model changes. Other approaches like [24, 37, 125] keep track of those changes as well. This permits reducing further the set of constraints to re-evaluate, at the cost of having to memoise the objects/fields accessed during the last evaluation of each constraint.

We support the definition of scoped constraints by means of the *scoped validation* pattern in Figure 4.7(a). A scoped constraint is defined by mapping a domain class (the context of the constraint) to the class role ConstrainedClass. Then, the configuration fields in the pattern allow defining the constraint name (name), the OCL expression (statement), the evaluation scope (scope), and the error message to be reported when the objects of the constrained class violate the constraint (error).

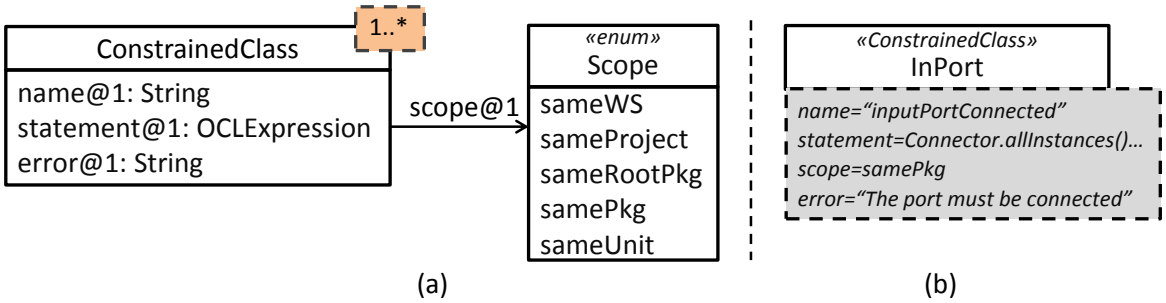


Fig. 4.7 (a) Scoped validation pattern. (b) Defining a scoped constraint for class InPort.

Example Figure 4.7(b) assigns the scope samePkg to the constraint in Figure 4.6. Now, assume file F1 in Figure 4.6 changes. Then, according to the algorithm in Listing 5, the validationContext is the package where F1 resides (i.e., s1). Hence, the constraint is evaluated in all InPort objects within s1 (i.e., {i1, i2, i3}) using the scope of the constraint (*same package*) as validationScope. For i1, the validation scope is the package s1 where i1 is located. Hence, Connector.allInstances() yields {n1, n2, n3}. In case of i2, the constraint is evaluated on the package that contains i2 (i.e., s2), where the only Connector is n2. In case of i3, it is evaluated on its container package s3, where the only Connector is n3. With this scoped validation strategy, we do not need to evaluate the constraint on i4 or consider Connector n4. If the change happened in file F4, then the constraint would be evaluated on i4 considering only the Connector n4.

4.4 Summary and Conclusions

This chapter has first identified the requirements that scalable DSML environments have to fulfil. Subsequently, it has proposed a pattern-based approach to facilitate building these environments satisfying the identified requirements. Finally, five modularity patterns were presented: Model Fragmentation, Reference Scoping, Visibility, Indexing and Scoped Validation. The chapter used the WT meta-model as a running example to show how to instantiate these patterns at the meta-model level.

The next chapter describes our approach to facilitate the definition of graphical and tabular concrete syntaxes for DSMLs.

Chapter 5

Support for Graphical and Tabular Concrete Syntax

This chapter describes our proposal to define graphical and tabular syntaxes. First, Section 5.1 identifies missing requirements in the current technology, from which it derives a number of desirable requirements in frameworks for the creation of graphical modelling tools. Next, Section 5.2 proposes a platform-independent meta-model to represent graphical syntaxes and a set of heuristics and strategies to automatically assign a graphical syntax representation to meta-model classes and features. Subsequently, Section 5.3 describes the meta-model to represent tabular concrete syntaxes. Finally, Section 5.4 describes an approach to combine the graphical representation with the model fragmentation pattern to obtain a scalable environment.

5.1 Motivation

The existing technologies for the creation of DSMLs were described in Section 2.2.4. These technologies require highly specialized technical skills from user developers in order to create a graphical modelling environment. The construction of new DSMLs in any of these frameworks is an ad hoc process, lacking the ability to build on existing knowledge coming from the construction of similar DSMLs.

While for the creation of meta-models, different researchers have identified common structures to represent recurrent modelling solutions (e.g., design patterns), there is no similar support to assist in the construction of graphical modelling editors.

To simplify the creation of DSMLs, we propose several heuristics to detect structures in meta-models that are frequently used for the representation of graphical elements. In

particular, this work proposes analysing the meta-model to obtain a default graphical representation. These heuristics will be presented in Section 5.2.1.

Section 4.1 identified scalability as one of the requirements of a modelling environment. Usually, the frameworks to create graphical editors only allow the creation of monolithic models, making it difficult to visualize them when the models become large. To alleviate these problems, Section 5.4 proposes splitting the graphical representation of models following the fragmentation strategies applied.

In this chapter, we will propose solutions to the following identified requirements when creating graphical editors:

R1 Automated assistance in the detection of graphical representations and styles for meta-model elements, to speed up development.

R2 Isolate developers from complex technical or implementation details related to graphical frameworks, to simplify development.

R3 Ability to fragment the concrete syntax representation of models, to improve scalability and separation of concerns.

5.2 Graphical Concrete Syntax

Our approach to the definition of graphical editors is the creation of an intermediate meta-model to represent the graphical concrete syntaxes in a neutral form. This meta-model is independent from the target implementation technology. This way, from a same model of the graphical concrete syntax, a family of editors in different graphical frameworks can be created.

Figure 5.1 shows the neutral meta-model we have developed to represent graphical concrete syntaxes, which is called *GraphicRepresentation*. Graphical elements are organized into layers (class Layer). A graphical representation has one defaultLayer where all graphical elements belong by default, and zero or more additional layers. Layers contain graphical elements, which can be either Node-like or Edge-like. In both cases, they may hold a PaletteDescription with information on how the element is to be shown in the palette. Nodes may be represented as geometrical shapes (Rectangle, Ellipse, etc.) or images (class Figure). They can also display a label either inside or outside the node, being possible to configure its font style (class Label). The label can be defined using the OCLLabel or EAttributeLabel classes. The first permits the definition of labels using an OCL expression and the latter specifies the list of node attributes that will be displayed as the node label.

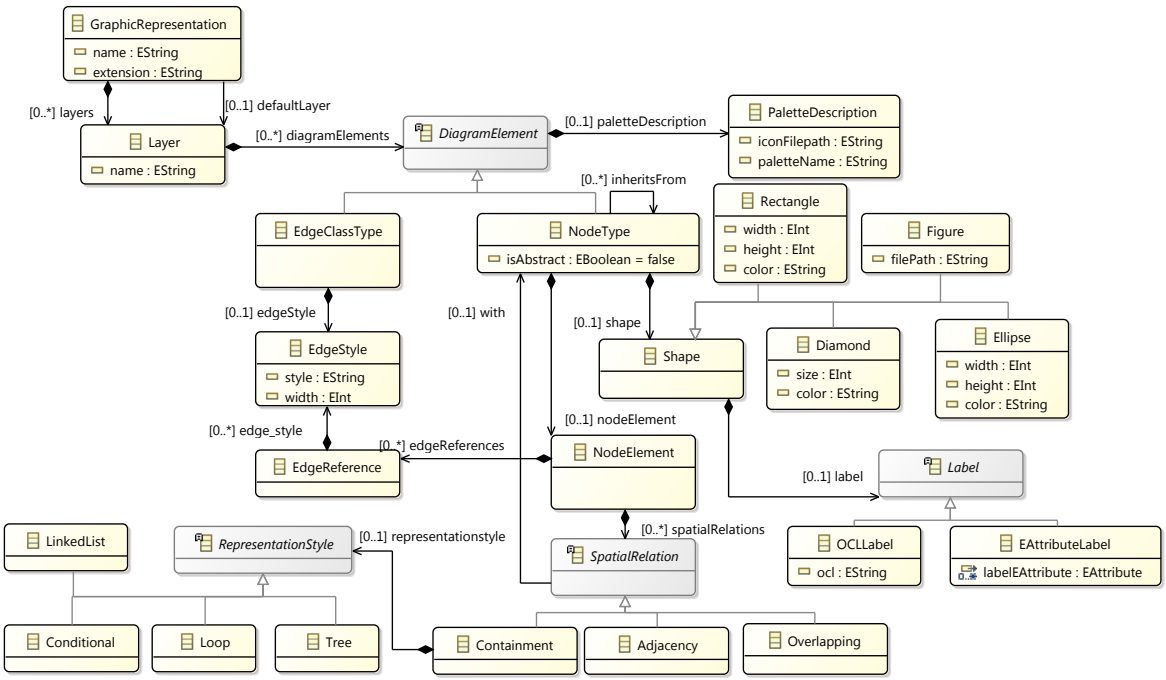


Fig. 5.1 *GraphicRepresentation* meta-model.

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. Currently, the implementation supports the adjacency and containment spatial relationships. Edges can specify a line style like solid, dash, dot or dash-dot (class EdgeStyle). In addition, the containment relationship may have a representation (class RepresentationStyle), which means that the displayed nodes within a container will have a predefined representation style. The meta-model currently supports 4 representation styles: LinkedList, Conditional, Tree and Loop. These styles will be explained in detail in Section 5.2.2. Finally, we enable the reuse of graphical properties by means of relation inheritsFrom and attribute isAbstract in class Node, so that graphical properties defined for a node are inherited by its children nodes.

The definition of the concrete syntax requires establishing a correspondence between the abstract syntax meta-model of the DSML and the concrete syntax meta-model in Figure 5.1. We consider abstract syntax meta-models defined with Ecore, for which mappings can be established according to Figure 5.2. In particular, classes in the domain meta-model (class EClass) can be represented either as nodes (class NodeType) or as edges (class EdgeClassType), and are referred to through the reference DiagramElementType.eClass. In case the class is represented as an edge, it is possible to configure the references of the class acting as source and target of the edge. Attributes in the domain meta-model

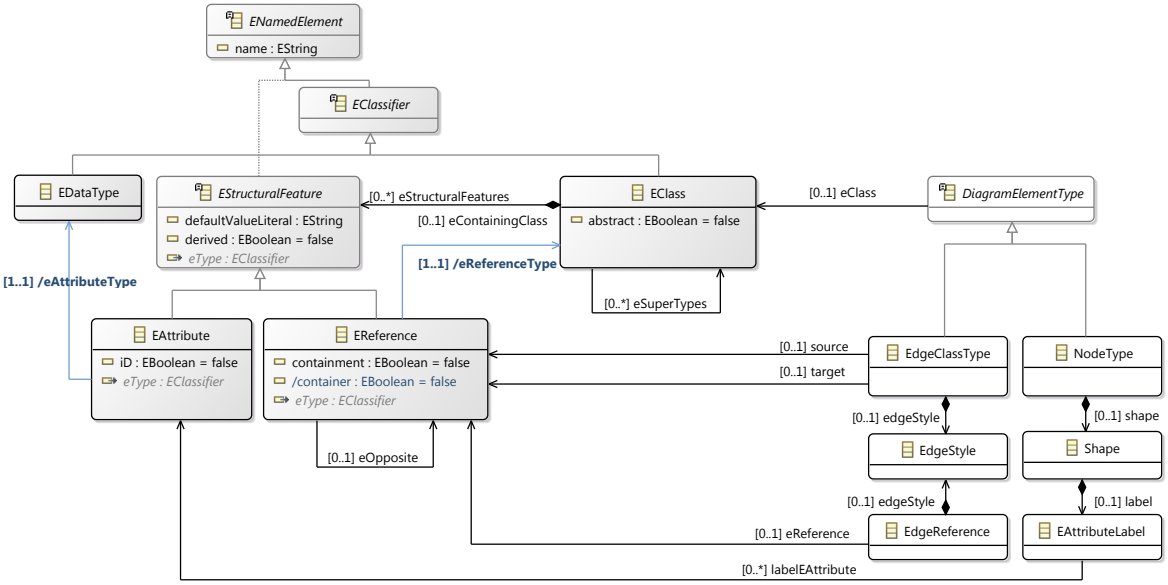


Fig. 5.2 Mapping between *GraphicRepresentation* meta-model and Ecore meta-model (classes of Ecore are shaded).

(class EAttribute) can be mapped into a ELabelAttribute. References in the domain meta-model (class EReference) can be mapped into EdgeReferences, and their concrete syntax annotations are mapped into an EdgeStyle. In addition, the references can be assigned as Containment or Adjacency object, respectively (not shown in Figure 5.2). All created graphical elements are included in the default layer and receive a palette description.

The goal of the *GraphicRepresentation* meta-model is to allow the definition of concrete syntaxes. For this purpose, instances of its elements should be created and mapped into classes, attributes and references in the domain meta-model. Figure 5.3 shows an example of how this meta-model can be instantiated. In this case, the figure shows the instantiation of two nodes and one edge. In the event that the number of elements to be mapped is large, the mapping process may become tedious. For that reason, we provide heuristics in the next section, to assist in this process.

5.2.1 Heuristics

The concrete syntax of a DSML is defined by mapping the meta-model elements to the elements of a *GraphicRepresentation* model. Since the definition of concrete syntaxes has many specificities and may be tedious, sometimes inexperienced developers need assistance to build the editor. To alleviate this problem, we have designed heuristics that recommend a specific graphical representation for classes and features.

The heuristics analyse the structure of the meta-model and suggest a default visualization for its elements. We have devised four groups of heuristics to identify different occurrences of structures in meta-models and one for assignment of styles:

- *Root class selection heuristics*: In order to define the graphical concrete syntax, one of the classes defined in the meta-model needs to be chosen as the root element, as this will represent the diagram canvas. The class selected as the root element, is typically the one that contains all meta-model elements directly or indirectly, that is, all other meta-model classes can be reachable from the root through containment references. To select the root class in an automatic way, two strategies have been designed based on the containment tree defined in the meta-model. These two strategies are:

  1. *Contains more classes* counts how many classes contain each class, and selects the one that contains more.
  2. *Class with no parents* suggests classes that are not contained in other classes.

- *Node & edge heuristics*: We select as edge-like classes those that define two non-containment references with lower bound `0` or `1`, and upper bound `1`. These two references will be mapped to the source and target of the edge representation for the class. To determine the source and target role of the two references in the edge, two strategies have been defined:

  1. *Simple direction arc strategy* selects the source and target reference randomly.
  2. *Parameter direction arc strategy* takes into account possible naming conventions (e.g., `source` or `src` for the source reference).

  Otherwise, if the class is not assigned an edge-like representation, then it will be assigned a node-like representation. In order to define the node style (e.g., ellipse or diamond), we use additional heuristics for the graphical style definition, which will be described later.

  Figure 5.3 shows an automatic mapping example applying the heuristic explained above. Specifically Figure 5.3(a) shows the subset of the WT meta-model defining state machines. On one hand, classes `InitialState` and `SimpleState` are assigned a node-like representation (Figure 5.3(b)), because they do not declare two non-containment mono-valued references. On the other hand, class `Edge` defines two non-containment references with upper bound 1. For this reason, this class gets assigned an edge-like representation.

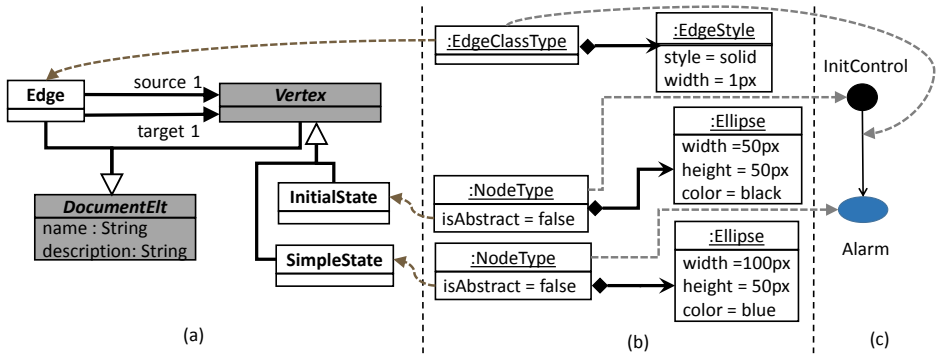


Fig. 5.3 (a) Excerpt of WT meta-model. (b) Inferred graphical representation for classes. (c) Visualization of a graphical representation.

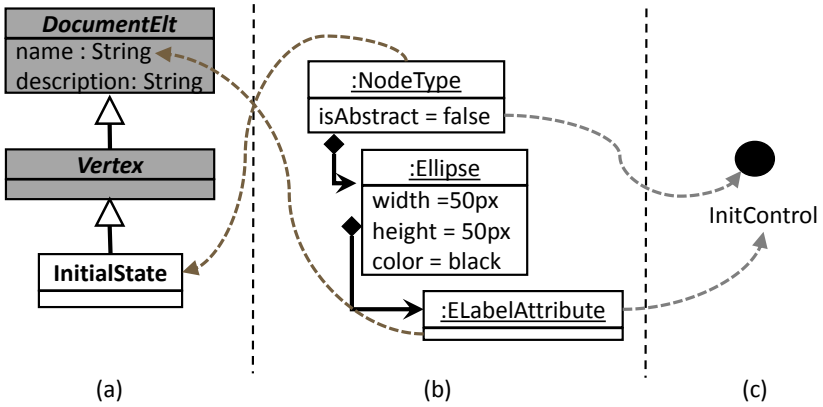


Fig. 5.4 (a) Excerpt of WT meta-model. (b) Inferred labels. (c) Visualization of graphical representation.

- *Label selection heuristics*: These heuristics are used to decide the attributes that node-like or edge-like classes will display close to their representations. We identify the following three strategies:

  1. *First string attribute* uses the first string attribute of the class as its label.
  2. *Identifier of the class* uses its identifier.
  3. *Parameter string attribute* is parametrized with several input strings, and selects the attribute whose name contains some of them.

  An example of a label selection heuristic is shown in Figure 5.4. To illustrate this heuristic we use the third strategy configured with the parameter "name". Figure 5.4(a) shows and excerpt of the WT meta-model. Figure 5.4(b) maps the name attribute as the node label, because it matches the parameter. Figure 5.4(c) illustrates the graphical representation that would be obtained.

- *Default representation of containment references*: These strategies identify references that will be displayed graphically as edges, compartments or affixes. The defined heuristics are the following:

  1. *Containment references as links*, all containment references will be represented as links.
  2. *Containment references as compartments*, containment references will be displayed as containers for the objects conformant to the type of the reference.
  3. *Containment references as affixed*, the nodes will be placed on the border of another element.

  Figure 5.5 shows an example of graphical representation using the link strategy. We show the classes of the WT meta-model to represent the relationship between components and ports (Figure 5.5(a)). The ports containment reference is assigned an edge-like representation (Figure 5.5(b)) which is visualized in Figure 5.5(c) as edges from the objects of type component to the ports.

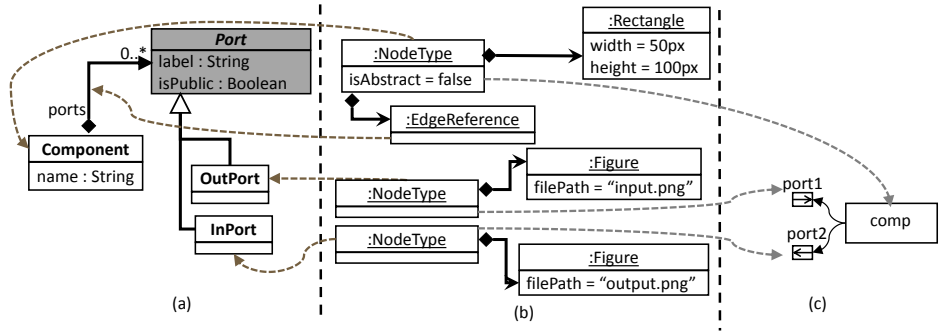


Fig. 5.5 (a) Excerpt of WT meta-model. (b) Graphical representation of compositions using the link strategy. (c) Visualization of graphical representation.

  The strategy that maps elements using the containment heuristic is shown in Figure 5.6. In this example, we use the same classes used to illustrate the link strategy. Hence, ports are created within component objects (Figure 5.6(c)).

  Finally, the affixed strategy is shown in Figure 5.7. In this case, the reference ports is mapped as affixed. Due to this, objects of type InPort and OutPort are depicted overlapped to the component object (Figure 5.7(c)).

- *Graphical style heuristic*: This heuristic assigns a style to the node and edge representations created by the previous strategies. For example, we can depict an identified node with a shape or an icon. In the case of links, they could be

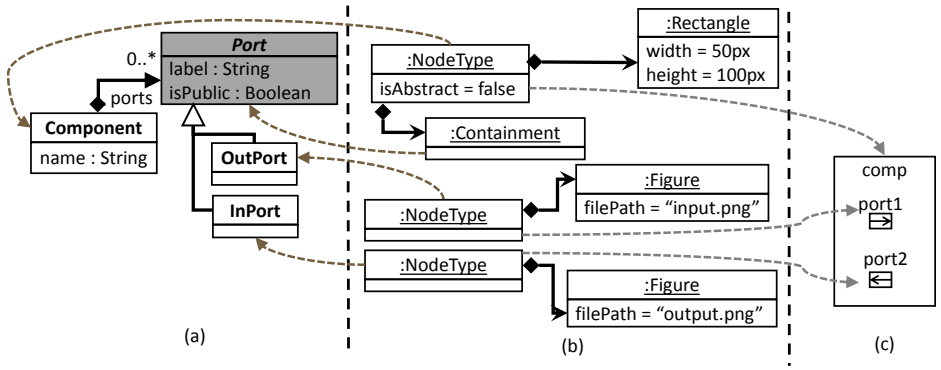


Fig. 5.6 (a) Excerpt of WT meta-model. (b) Graphical representation of compositions using the containment strategy. (c) Visualization of graphical representation.

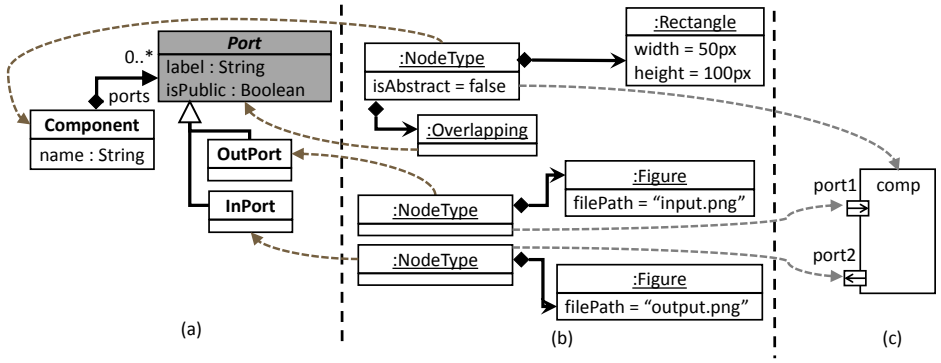


Fig. 5.7 (a) Excerpt of WT meta-model. (b) Graphical representation of compositions using the affixed strategy. (c) Visualization of graphical representation.

depicted with available decorators and line styles. By default, the source arrow of an edge is represented with no decorator and the target arrow with an input fill closed arrow decorator. We have defined two strategies for assigning a style to a diagram element:

1. *Semiotic clarity* which is one of the principles defined by Moody [92]. According to this principle, there should be a 1:1 correspondence between classes and graphical symbols. These symbols are different as long as any of the 8 visual variables such as colour, size and texture are different. In this case, the generated representation must verify that each graphical style is not equal to the others defined.
2. *Random* strategy, which does not verify the singularity of each symbol.

As a complement of the two strategies previously explained, another possibility is assigning icons (instead of shapes) to node-like classes. In such a case, for

each node-like class in the domain meta-model, we perform a search in an image repository using the class name as parameter. Then, we assign the first icon found as the representation of the node, but giving the user the possibility to select another icon among the ones found.

The combination of all heuristics results in the creation of a default graphical representation by making use of the meta-model information. This representation can be later adjusted by the user if so desired.

5.2.2 Read-only Representation Style for Collections

This section proposes a set of representation styles that can be found when designing a DSML. These representation styles are default visualizations that are attached to a certain structure of the abstract syntax. The goal is providing a representation for recurrent non-trivial visualizations of collections which resemble statements of general-purpose languages, and simplifying their definition by a simple mapping between the meta-model structure and the representation style. In total, we have identified four styles that are: Loop, Conditional, Tree and LinkedList. These representations are defined in the *GraphicRepresentation* meta-model (Figure 5.1). We will describe these styles through examples of how the abstract syntax is mapped to obtain the desired representation using synthetic meta-models.

An example of the LinkedList representation is shown in Figure 5.8. The synthetic meta-model shown in Figure 5.8(a) contains a class A, which has a reference with cardinality * to a class B. Classes of the synthetic meta-model are mapped to the *GraphicRepresentation* model. The visualization result is shown in Figure 5.8(c), where a list of B objects referenced by an A object is displayed as a list. We add to this visualization an initial and final shape, which are not mapped to any class of the meta-model. However, we can customize the visualization so that the initial and final shapes do not appear, or only one of them is displayed.

Figure 5.9 shows the Tree representation. The synthetic meta-model shown in Figure 5.9(a) it is the same one used to illustrate the previous style. In this case, the resulting visualization shown in Figure 5.9(c) displays the B objects linked to the root, which is a shape not mapped to any meta-model class. This root is depicted as a diamond, but it can be customized to suit the user's needs. This representation style can be configured so that the initial and final shapes do not appear in the visualization.

The third proposed representation is Loop of which an example is shown in Figure 5.10. In this case, the synthetic meta-model contains a class A with a reference with

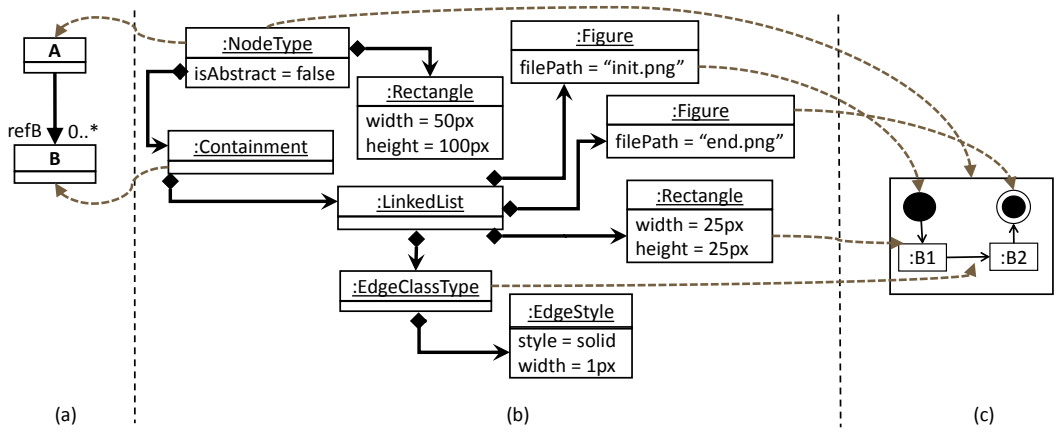


Fig. 5.8 (a) Synthetic meta-model. (b) Mapping to create a LinkedList visualization. (c) Visualization using the LinkedList representation.

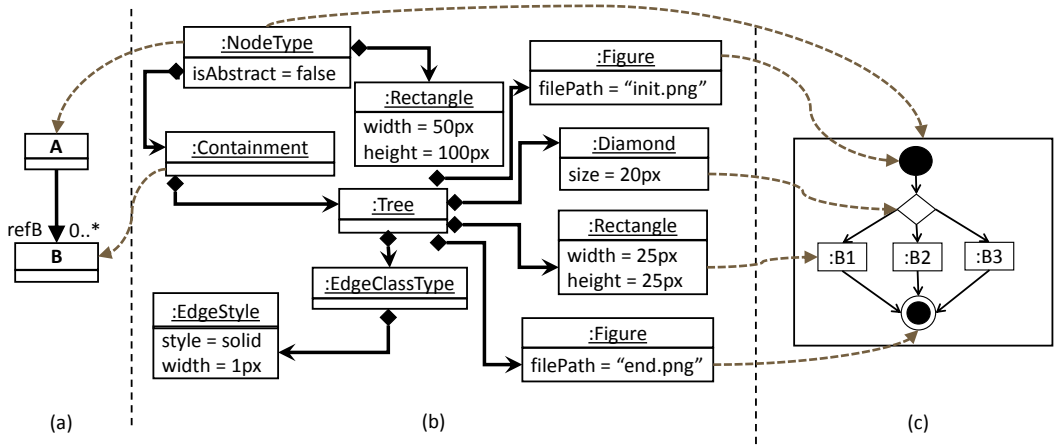


Fig. 5.9 (a) Synthetic meta-model. (b) Mapping to create a Tree visualization. (c) Visualization using the Tree representation.

cardinality [0..1] to a class B. Figure 5.10(c) shows the visualization result obtained after the mapping shown in Figure 5.10(b). In this representation, we also add customizable initial and final shapes.

Finally, Figure 5.11 shows the Conditional representation style. The synthetic meta-model (Figure 5.11(a)) contains a class A with three references, each of these references points to the same class B, and represent the if, then and else branches of a conditional statement. Figure 5.11(c) shows the visualization result. As in the previous representation styles, it has customizable initial and final shapes.

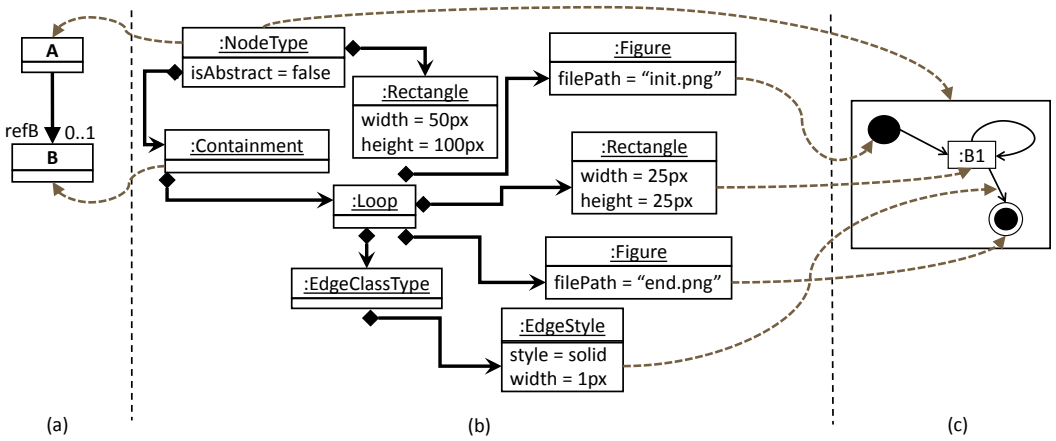


Fig. 5.10 (a) Synthetic meta-model. (b) Mapping to create a Loop visualization. (c) Visualization using the Loop representation.

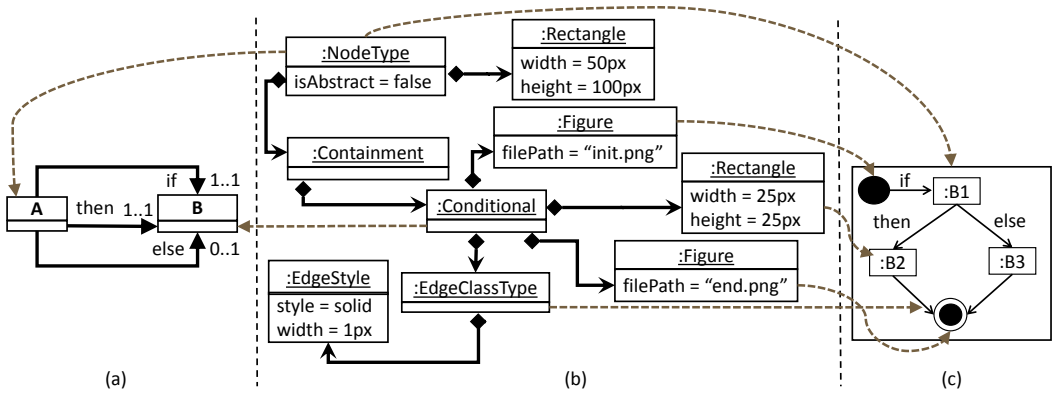


Fig. 5.11 (a) Synthetic meta-model. (b) Mapping to create a Conditional visualization. (c) Visualization using the Conditional representation.

5.3 Tabular Concrete Syntax

With respect to the tabular representation, our approach to instantiate the concrete syntax is to create an intermediate meta-model that stores the corresponding data. This same approach was used to represent graphical concrete syntaxes (though through a different meta-model) in Section 5.2.

Figure 5.12 shows the platform independent meta-model to specify tabular concrete syntaxes. The specification of a tabular representation consists of a selection of the attributes that will be shown in the columns of the table as well as the references that will be displayed in the rows. As an example, Figure 5.13 shows a mapping to create a table of state machines objects defined within a Component. Figure 5.13(b) shows the tabular representation model to define a table with all StateMachine objects referenced

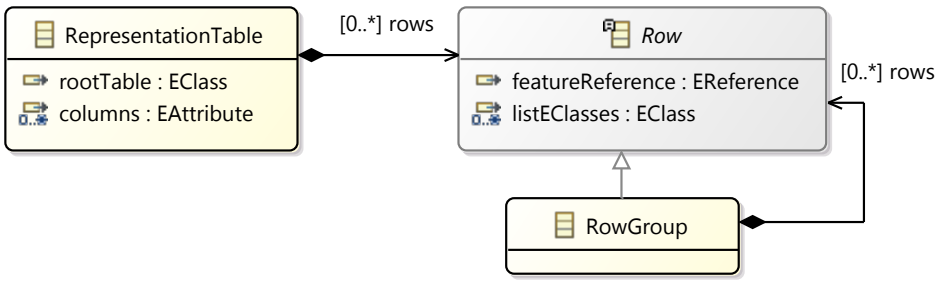


Fig. 5.12 Tabular representation meta-model.

by a component Figure 5.13(d) shows an example of such tabular representation for the model shown in Figure 5.13(c).

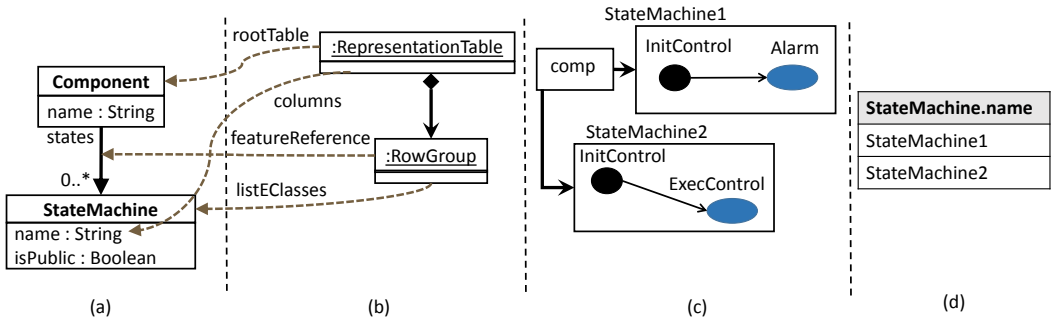


Fig. 5.13 (a) Excerpt of WT meta-model. (b) Mapping to create a tabular visualization of state machines. (c) Example of a model. (d) Visualization of state machines in tabular form.

5.4 Graphical Representation of a Fragmented Model

Section 4.3.1 described a fragmentation pattern for the creation of models. By applying this pattern, different files are created in a hierarchical structure that is mapped to the file system, which together conform the entire model. The fragmentation mechanism relies on the use of cross-references by which objects can refer to other objects that are in other files. Cross-references can be containment or not. A containment reference induces a tree, which means that the reference can contain objects (tree nodes). Otherwise, cross-references are links between the objects in different files.

We propose transferring the hierarchical organization of models by the fragmentation pattern, to the concrete syntax. Using this mechanism the diagrams can also be constructed in a fragmented way. Since the objects annotated as Unit are those that group elements in the same file, we allow creating a diagram for each of them.

Figure 5.14 shows an example of how fragmentation would be applied to graphical editors at the level of the concrete syntax. Our running example allows creating two types of units: State Machine or Architecture. The data of each diagram will be saved separately (one diagram per fragment), and, the editors must allow objects to be referenced by other object in another document. Such is the case of the reference that is shown from the component object to HUnitControl which is a non-containment reference.

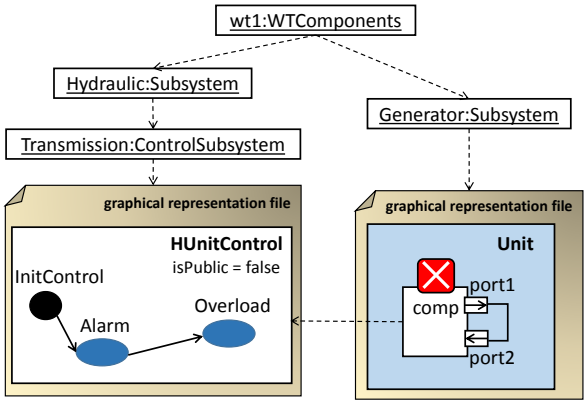


Fig. 5.14 Graphical representation of a fragmented model.

5.5 Summary and Conclusions

In this chapter we have described our approach to speed up the creation of graphical and tabular representations of models. The approach is based on the creation of a model of the concrete syntax, and its mapping to the abstract syntax meta-model. In the case of the graphical representation, a set of heuristics assist developers in its implementation.

In the following chapter, we describe the research tools developed atop Eclipse to support the ideas presented in this and the previous chapters. The tools are called EMF-Splitter and EMF-Stencil. EMF-Splitter provides dedicated wizards to instantiate the catalogue of patterns discussed in Chapter 4 and also generates the corresponding modelling editors. EMF-Stencil offers wizards to instantiate concrete syntaxes to generate graphical editors and tabular representations.

Chapter 6

Tool Support

This chapter provides a description of the two Eclipse plug-ins proposed in this thesis, called EMF-Splitter and EMF-Stencil. The first section shows an overview of DSL-tao since both plug-ins use this tool as a front-end to instantiate the fragmentation and graphical patterns at the meta-model level as explained in Chapters 4 and 5. On one hand, EMF-Splitter (Section 6.2) implements the catalogue of modularity patterns, providing dedicated wizards to instantiate the patterns, and being able to generate modelling environments that integrate the associated modularity services. On the other hand, EMF-Stencil (Section 6.3) implements our approach to specify the graphical and concrete syntaxes presented in Chapter 5 and generates the corresponding graphical editor for a specific language framework currently in Sirius.

6.1 DSL-tao

DSL-tao is a tool that permits the pattern-based creation of DSMLs, as explained in Chapter 4. DSL-tao proposes a catalogue of patterns divided into five categories: domain, design, concrete syntax, dynamic semantics and infrastructure patterns [105]. These patterns may include services which can contribute to the functionality of the generated environment. This tool is available at http://www.miso.es/tools/DSLtao.html.

DSL-tao uses Eclipse/EMF [118, 128] as implementation platform to profit from its plugin-based architecture, which allows extending the platform to incorporate new functionality through extension points. This simplifies the task of adding new patterns to the system, and it is a natural deployment infrastructure for our generated DSMLs environments, as we will show in Section 6.2.

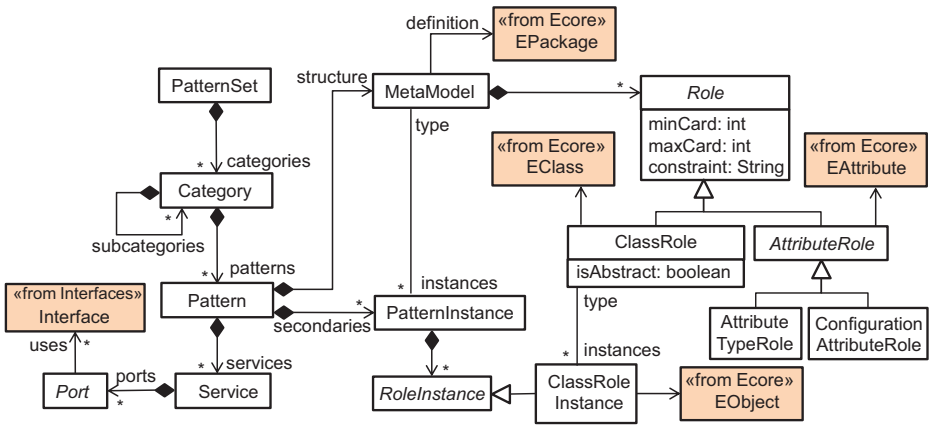


Fig. 6.1 Excerpt of DSL-tao's pattern meta-model.

The tool includes an extensible catalogue of patterns. New patterns can be defined by providing its meta-model and roles. Figure 6.1 shows an excerpt of the underlying meta-model for pattern definition. A Pattern declares Roles, which annotate the elements of the ecore MetaModel with the structure of the pattern. Roles can be ClassRoles if they annotate meta-model classes, AttributeRoles if they annotate attributes, or ReferenceRoles if hey annotate references. As explained in Section 3.2, for attributes and references, we allow both field roles and configuration roles. Roles may define a constraint to validate the pattern instantiation, using OCL. Patterns may have any number of PatternInstances, which contain RoleInstances of the roles declared by the pattern. Whereas roles in patterns point to meta-elements like EClass or EAttribute, role instances point to EObjects in the structure of the pattern instance. In addition to their own instances, patterns may have associated secondary instances from other patterns. For example, the user can define a secondary pattern instance with the information on the concrete syntax of the main pattern. Finally, patterns may define Services, which need to declare their Ports and Interfaces.

By means of extension points, the pattern developer is allowed to extend the base definition of a pattern to include (i.e., the meta-model and roles) pattern-specific validations, heuristics or services (if needed):

- *pattern-specific validations*: The pattern developer may include extra validations, e.g., expressed in OCL, to check whether the binding of a pattern instance to a domain meta-model is correct. It is used only for pattern-specific validations, as DSL-tao already performs generic correctness checkings (e.g., that if a meta-model attribute is bound to a pattern attribute, their owner classes are bound as well). An example of pattern-specific validation for the modularity pattern is checking

that the class bound to role Project is root (i.e., it contains all other classes through containment relationships, directly or indirectly).

- *pattern-specific heuristics*: These are heuristics that facilitate the correct instantiation of a pattern. For example, the model fragmentation pattern identifies the root class of a meta-model as the optimal binding for the Project role.

- *services*: The pattern semantics can be realized via services, typically through code generation using Acceleo or any other code generation language, and it is encapsulated as an Eclipse plugin. Code generation is needed because the generated plugin for the modelling environment needs to be customized with information of the pattern instance. For example, the plugin generated for the HierarchicalOrganization service needs to enforce the desired project/package/unit structure. Service dependencies are realized as plugin dependencies and appropriate instances of extension points.

Figure 6.2 shows a screenshot of DSL-tao. The tool enables the construction of DSML meta-models by dragging elements from a palette into the canvas (label 3). A *Patterns View* (omitted in the figure) lists the patterns of the different categories. When a pattern is selected, a wizard (labels 1 and 2) facilitates its instantiation as follows: first, the pattern variant is selected, together with its secondary patterns (if any). In the figure (label 1), the StateMachine pattern is to be applied, and the designer may choose among three default visualizations (three secondary patterns). Then, the designer can bind meta-model elements to the pattern roles by dragging the former into the latter (label 2), and instantiate the pattern roles according to their cardinality.

Label 3 in Figure 6.2 shows the resulting meta-model. To the right (label 5), the *Applied Patterns View* displays a tree containing each pattern instance and instantiated role. Selecting a role highlights the bound meta-model element in the canvas (InitialState in the figure). Each meta-model element shows its roles in patterns as annotations, and the canvas itself shows the list of applied patterns in the upper-right corner.

The tool includes a *Pattern Services View* (label 4 in the figure), where each row indicates a service instance, the pattern that provides the service, and if it is activable or not. If a service is optional, the designer can activate it. A pattern is activable if all its port dependencies are resolvable. In case a service is not activable, the view informs of which ports are unconnected, and which patterns in the repository could resolve the given dependency. The figure shows the Hierarchical Organization service deactivated, which makes the Filtering and the two Graph-based editing services unavailable. Note that the DSML designer is unaware of the different port types (slot, plug, injector,

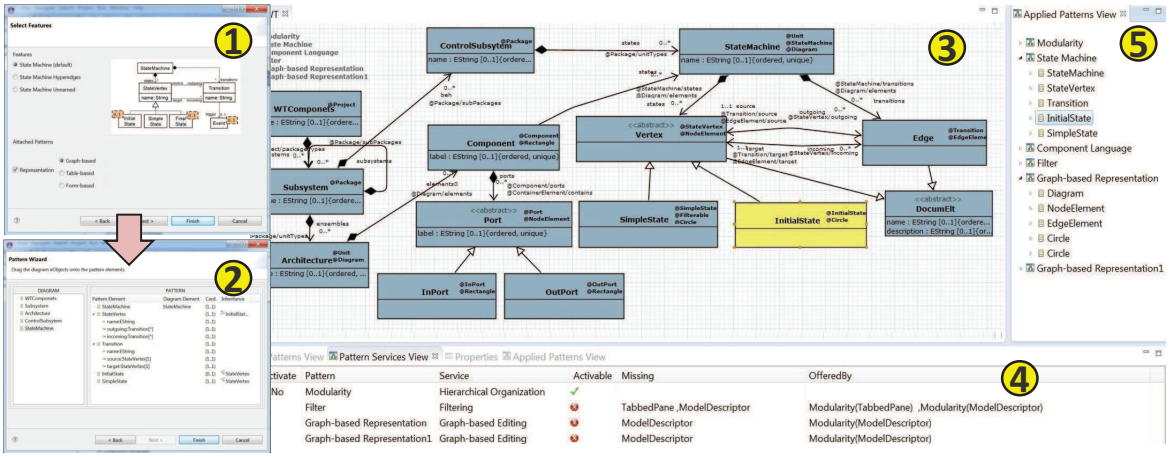


Fig. 6.2 Using DSL-tao. (1, 2) Applying the StateMachine pattern. (3) Resulting meta-model. (4) Services. (5) Applied patterns.

consumer) and the composition of services, as this is automatically performed by the tool.

6.2 EMF-Splitter

To give support to our pattern-based approach to modularity in DSMLs, we have implemented an Eclipse-based solution called EMF-Splitter [48, 49] freely available at http://www.miso.es/tools/EMFSplitter.html.

Figure 6.3 shows its architecture. It relies on EMF to represent the domain meta-models and their instances. It contains the modelling front-end DSL-tao described in the previous section. We have extended the pattern repository provided by DSL-tao with all modularity patterns presented in Section 4.3, and EMF-Splitter implements the extension points in charge of generating the modelling environment once the patterns have been applied. While EMF-Splitter complements DSL-tao, it can also be used stand-alone.

This way, with our solution, the modelling language designer creates the domain meta-model and applies to it the modularity patterns using DSL-tao. The pattern applications are persisted as annotation models over the domain meta-model, which EMF-Splitter uses to synthesize a modelling environment. This environment provides the functionality defined by the patterns, like model creation using the defined fragmentation strategy and incremental validation of scoped constraints. The environment also provides support for working with legacy monolithic models, as it can automatically

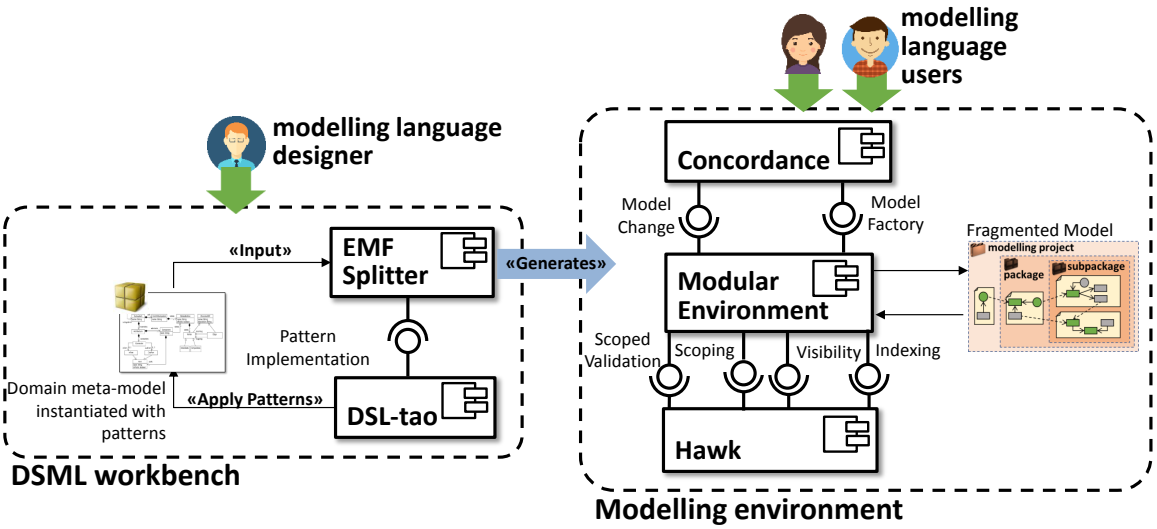


Fig. 6.3 Architecture of EMF-Splitter.

fragment an existing model according to the fragmentation strategy, and merge a partitioned model into a monolithic one.

The models created or fragmented by the generated environment are distributed in files and folders in the file system. In consequence, the integrity of a model can be affected if a file path changes (e.g., if a fragment is moved from a folder to another one). In order to maintain the model integrity, the generated environment integrates an indexer for cross-references called Concordance [108] (see Figure 6.3). By default, Concordance only indexes cross-references. In addition, the generated environment implements its extension point ModelFactory to index containment references as well, and its extension point ModelChange to receive notifications of model events in the workspace.

The generated environment can be extended through Eclipse extension points, for example, to integrate it with other tools. Based on this mechanism, our modularity patterns for scoped validation, reference scoping, visibility and indexing generate code that integrates Hawk [46], a scalable model indexer that improves the performance of queries over large models. This results in more scalable environments, as we will demonstrate in Section 7.4.4.

Figure 6.4 shows DSL-tao being used to define a modelling environment for the running example. The domain meta-model is built in the main canvas (label 1), and the patterns in the repository can be applied over it (Patterns View, with label 2). The repository includes patterns with services for filtering, graphical visualization and modularity, among others. Once a pattern is applied, the meta-model elements get tagged with the

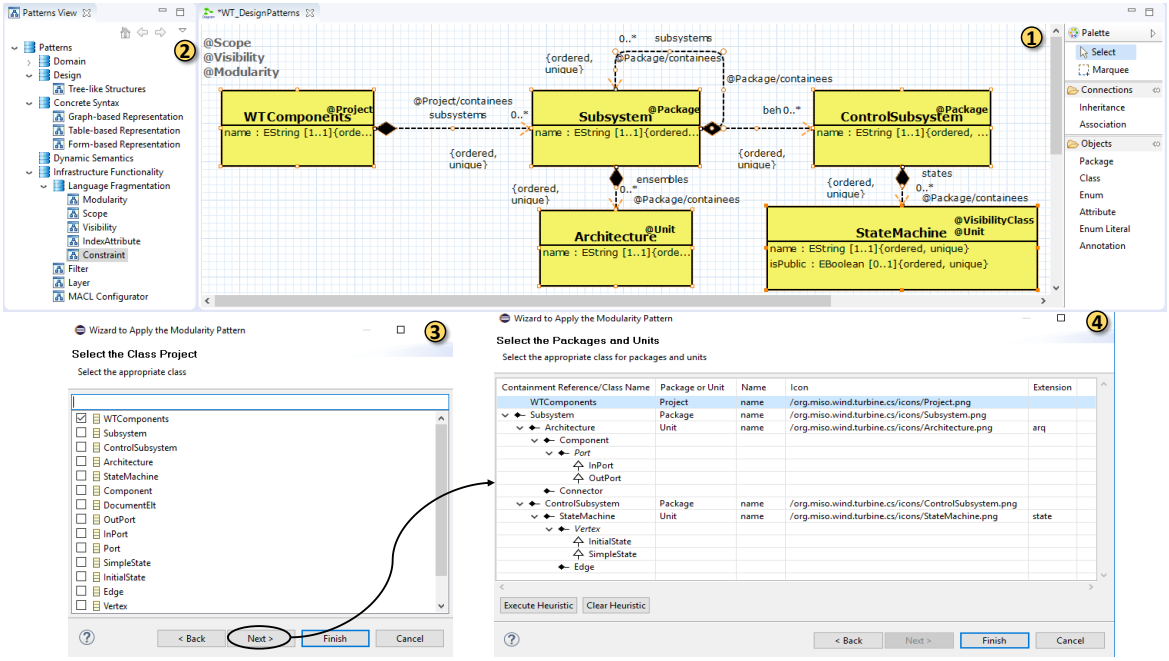


Fig. 6.4 Application of the fragmentation pattern using DSL-tao.

pattern role names (see for instance the tag @Project on class WTComponents). The environment provides a list of applied patterns, and when one is selected, the affected meta-model elements are highlighted in the canvas. The figure shows highlighted an application of the fragmentation pattern, while the upper-left corner of the canvas indicates that the Scope and Visibility patterns have been applied as well. Patterns can be instantiated using either a generic wizard or a specific wizard contributed by the pattern. The figure shows the wizard to instantiate the fragmentation pattern (label 3). This wizard permits selecting the project class in the first page (label 3). The second page (label 4) shows a table containing in the first column the meta-model classes organized following the containment tree, taking as root class the one defined as project. The remaining columns permit configuring the classes instantiated as package, unit or a mix of them; the attribute that will be used as the class name or identifier; an icon to represent the instances of the class in the generated; and the file extension for unit classes.

With respect to the reference scoping pattern, Figure 6.5 shows the dedicated wizard to apply this pattern and define scope rules related to the pattern as discussed in Section 4.3.2. The wizard permits selecting the class and reference with a scope, which can be one among the following: same unit, same package, same root package, same project or same workspace.

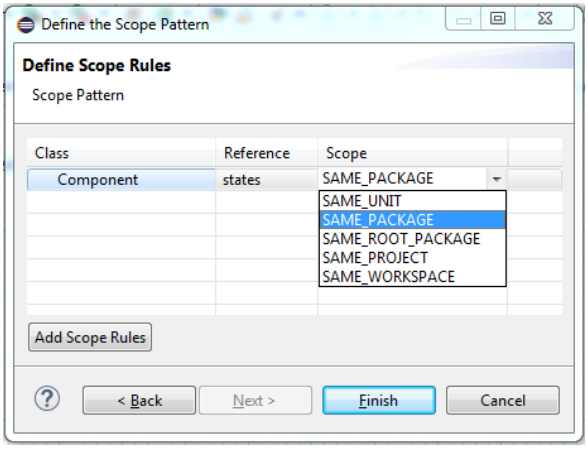


Fig. 6.5 Dedicated wizard for the application of reference scoping pattern.

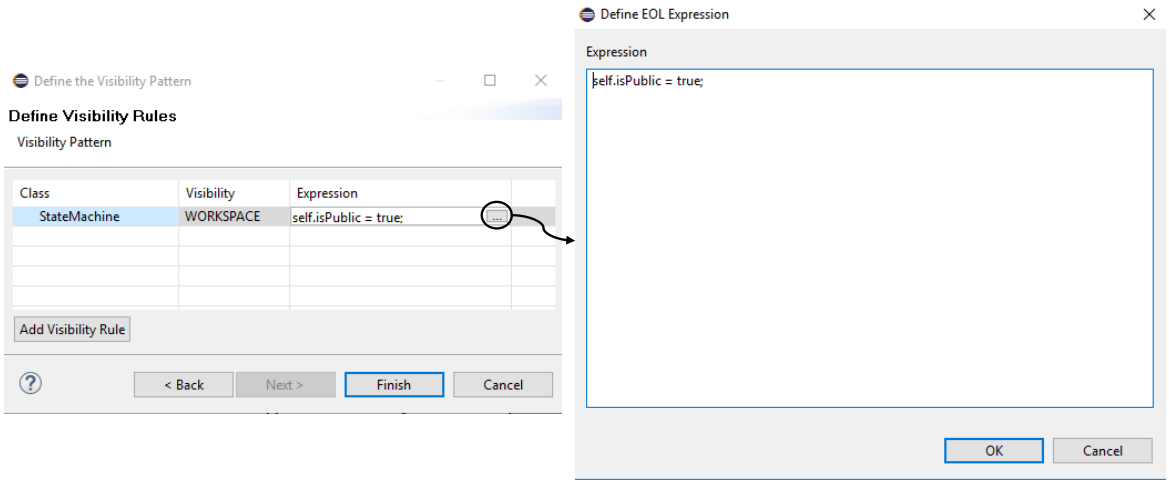


Fig. 6.6 Dedicated wizard for the visibility pattern application.

As we discussed in Section 4.3.3, the visibility pattern is based on the fragmentation one to define whether an object is accessible or not from other model fragments. To facilitate the application of this pattern, EMF-Splitter provides the wizard shown in Figure 6.6. This wizard permits selecting the class, visibility scope, and filter expression so that only the objects that fulfil the filter become visible within the provided scope. We have built a dedicated editor with syntax validation for the filter expression, which can be specified using the Epsilon Object Language (EOL) [73], a variant of OCL.

The indexing pattern described in Section 4.3.4 can be used to speed up the execution of queries over models through the definition of attribute indexes. Figure 6.7 shows the wizard developed. This wizard has a table to define different indexed attributes, the first column is used to select the class and the second the attribute.

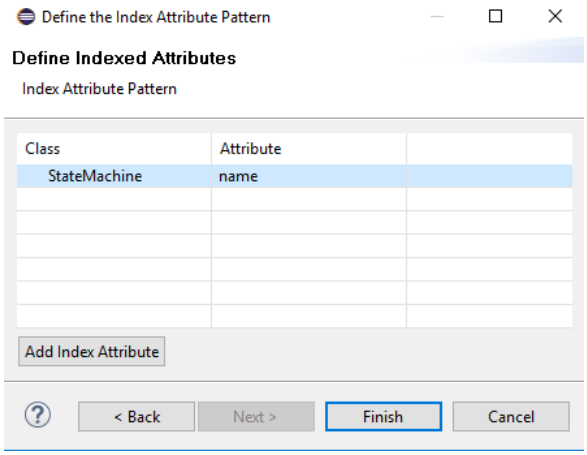


Fig. 6.7 Dedicated wizard for the indexing pattern application.

Figure 6.8 shows the implemented wizard for the scoped validation pattern. In the first page of the wizard (label 1) allows selecting the name of the constraint, context class, validation scope, and constraint statement using a dedicated editor with syntax checking for EOL (label 2).

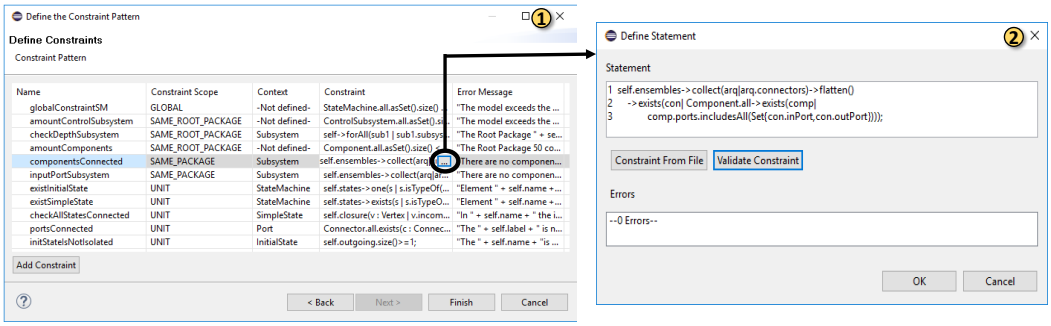


Fig. 6.8 Dedicated wizard for the scoped validation pattern application.

From the domain meta-model annotated with pattern instances, EMF-Splitter can synthesize a modelling environment. DSL-tao invokes the code generators of the services associated to the applied patterns. Figure 6.9 shows a snapshot of the synthesized environment for the running example. With the application of the fragmentation pattern, each model is represented as an Eclipse project (see Package Explorer view, with label 1), where packages are mapped to folders, and units to files. For instance, VariableSubsystem is a folder, which contains files like the VTUnit1.state, which represents a state machine object. The model fragments in files can be edited using a tree editor (label 2). In the next section, we will show how to improve this

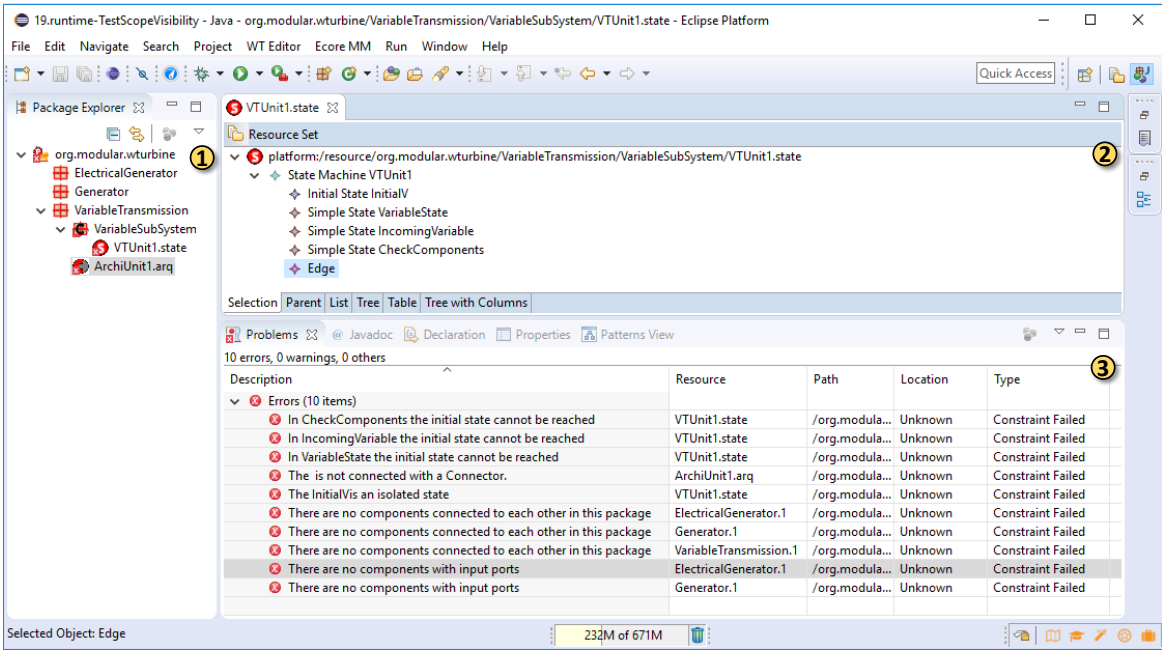


Fig. 6.9 Modelling environment synthesized for the running example.

environment, so that these editors can be graphical using the EMF-Stencil tool. The Problems view (label 3) shows the violations of scoped constraints.

6.3 EMF-Stencil

The approach to define graphical and tabular syntaxes explained in Chapter 5 is implemented as an Eclipse plug-in called EMF-Stencil available for download at http://www.miso.es/tools/EMFSplitter.html. Figure 6.10 shows the architecture of this tool, which provides one wizard for each concrete syntax to facilitate the instantiation at the meta-model level. As in the case of EMF-Splitter, this plug-in uses the DSL-tao tool as a front-end, implementing its interfaces to provide the dedicated wizards, although these wizards may be used stand-alone. EMF-Stencil is an Eclipse plug-in solution based on EMF which implements an extension point to generate a graphical editor realised using an existing language framework. Currently, we support Sirius as a technology specific editor and to do this, an ATL a transformation is implemented to obtain the *.odesign* model from the *GraphicRepresentation* one. Section 2.2.4 explained the *.odesign* model is used in Sirius to represent the concrete syntax.

Figure 6.11 shows the first page of the wizard to customise the graphical editor. In this first step, the wizard allows the modelling language designer to choose a

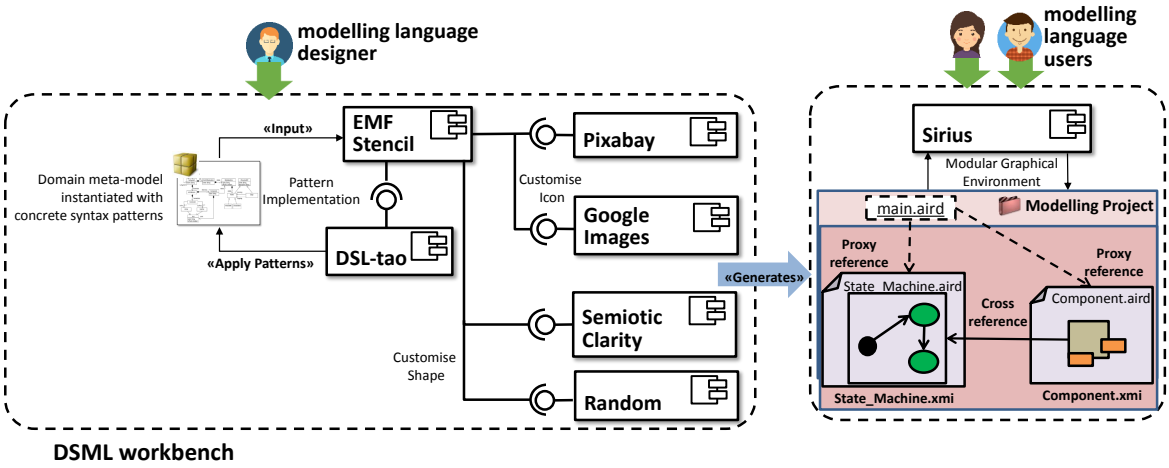


Fig. 6.10 Architecture of EMF-Stencil.

combination of strategies to generate a default editor. These strategies were described in Section 5.2.1. This wizard can produce a graphical editor to create monolithic models or it can be combined with the model fragmentation pattern. In the latter case, a different editor will be created for each class annotated with unit, allowing the editing of the models with an instance of the unit class as their root. EMF-Stencil provides the extension point CustomiseShape to define different strategies to automatically assign a default shape to classes, which can be later refined by the language designer. We have implemented two such strategies, called *Semiotic clarity* and *Random*. On one hand, the *Semiotic clarity* is a principle within Moody's criteria [92], which says that the figures need to have different visual variables (e.g., colour, shape, etc.) to be distinguishable. Hence, our strategy in this case consist in assigning to each class a different shape, and do not allow that two classes related by a reference have the same colour or shape. On the other hand, the *Random* strategy assigns the visual variables randomly without checking if they are the same as others previously chosen.

Regarding the assignment of icons to node elements, EMF-Stencil defines the extension point CustomiseIcon, which allows configuring the source repository from which the generated environment will retrieve the icons. This feature to obtain icons can be used with semiotic clarity. In this way, the nodes will be represented as icons, instead of using shapes. The addition of an icon data source requires the URL and the search parameters to obtain the icons list. In addition, the implementation should download and save the images within an Eclipse project, specifically the one that contains the meta-model. In this case, we implemented two search options:

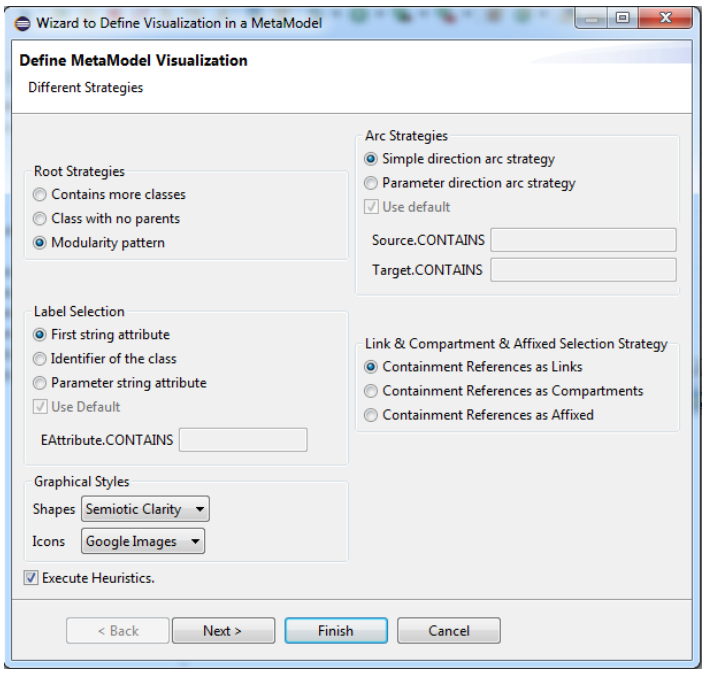


Fig. 6.11 Dedicated wizard for assigning a graphical concrete syntax. Step 1: Customize heuristics.

Pixabay [2]and Google Custom Search [3]. Pixabay is a repository of free images which provides a Rest API [4] for retrieving them. The Google Custom Search is a JSON API with RESTFul requests that allows getting images.

As a second step, EMF-Stencil infers a concrete syntax according to the selected heuristics and strategies, which the DSML designer can modify if desired. Figure 6.12 shows the inferred representation taking *Architecture* as root class. Within the Default Layer we can see the classes that are directly or indirectly contained by the root class (Architecture in this example). In addition, this wizard permits adding other classes belonging to the meta-model. For example, we may add the class *Subsystem* (label 1) to the Default Layer. In that case, we show a warning if the class is not contained directly or indirectly in the root class (label 2).

[2]https://pixabay.com/
[3]https://developers.google.com/custom-search/v1/overview
[4]https://pixabay.com/api/docs/

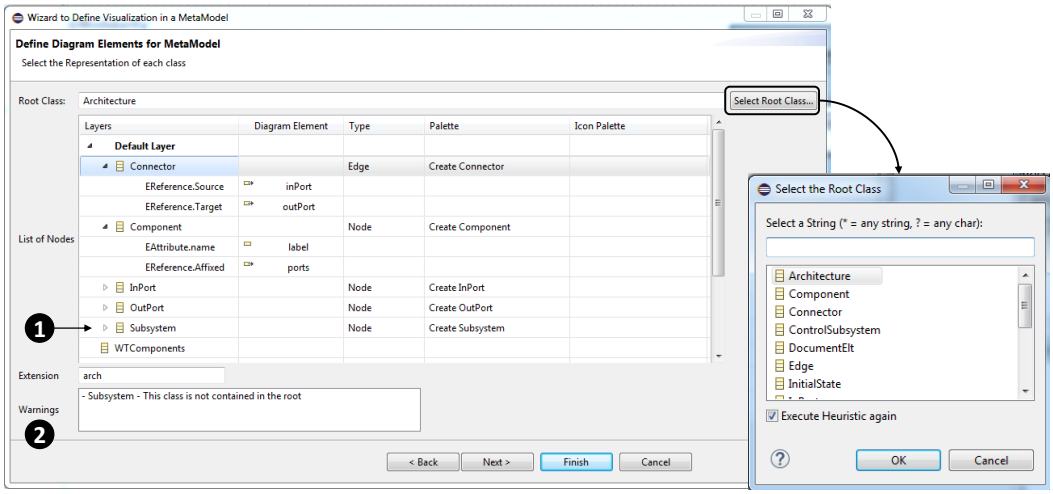


Fig. 6.12 Dedicated wizard for assigning a graphical concrete syntax. Step 2: customization of inferred concrete syntax.

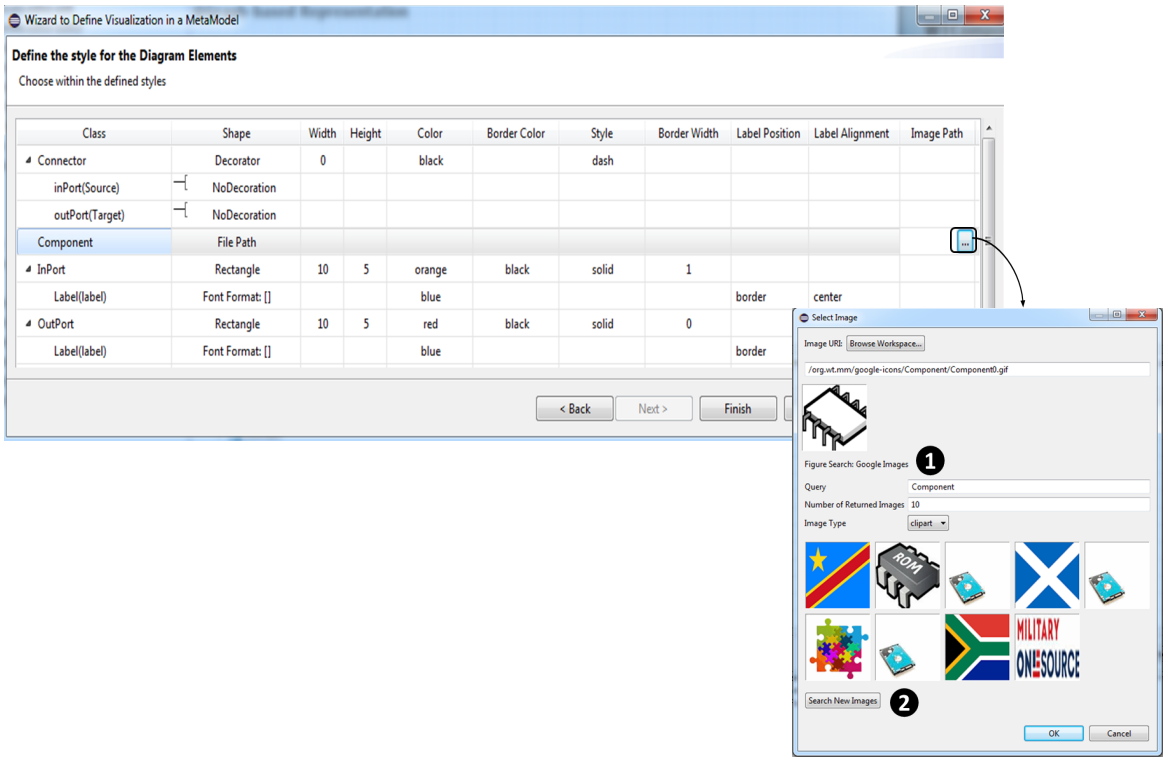


Fig. 6.13 Dedicated wizard for assigning a graphical concrete syntax. Step 3: customization of appearance of nodes and edges.

Finally, the last step is shown in Figure 6.13. This wizard page permits to update the generated styles. For example, in the case of nodes, the DSML designer is allowed to customise the shape, the colour and the border style. In the case of edges, there is

a list of available decorators and line styles. In addition, we implemented the dialog (right bottom corner of Figure 6.13) to select the icons for the class Component. In this example, we retrieve the images using the Google JSON API and implement two search parameters, the image type (e.g., clipart, face or lineart) and the number of images to return (label 1). The user can change the search parameters and download new icons (label 2).

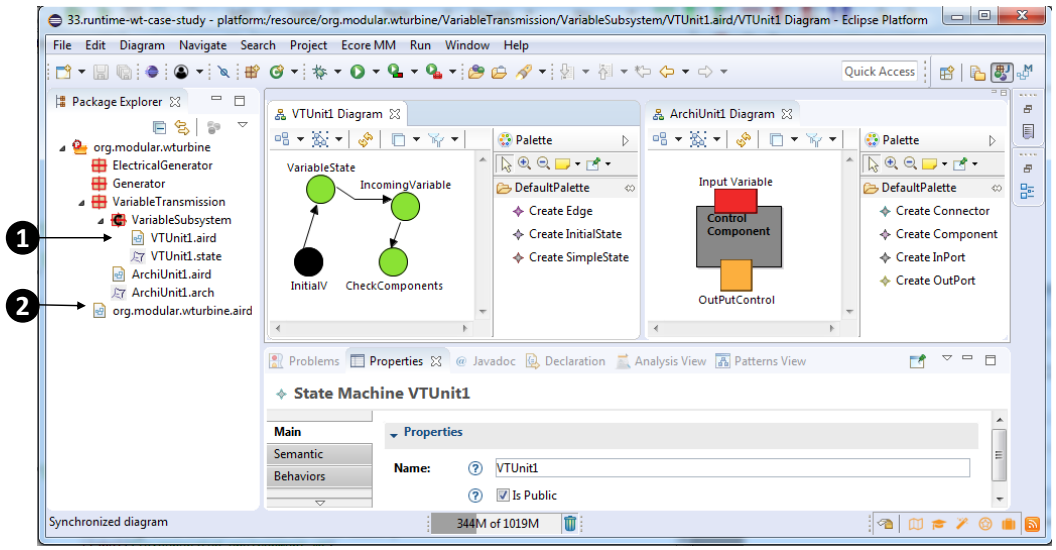


Fig. 6.14 Generated modelling environment for the WT meta-model.

A scalable modelling environment is generated from the definition of the concrete syntax using this wizard. Figure 6.14 shows the environment in Sirius, with two editors, one for state machines files and another for Component objects. For each sub-model file, the graphical information is saved in a representation file defined by Sirius (with extension .aird). Label 1 points to the VTUnit1.aird file, which saves the colours and positions of elements, defined in file VTUnit1.xmi. Furthermore, a representation file is created for each project which will contain cross references to all created diagrams (label 2). These graphical editors for model edition can replace the tree-editor shown in Figure 6.9.

6.4 Summary and Conclusions

This chapter has introduced the functionalities of our two developed Eclipse plug-ins. On one hand, EMF-Splitter implements the language modularity catalogue providing a set of dedicated wizards and code generators. On the other hand, EMF-Stencil implements

the heuristics defined in Chapter 5 and has a wizard to facilitate the mapping between the meta-model elements and their graphical representation. Altogether, these two tools facilitate the development of graphical modelling editors for DSMLs supporting modularity services.

In the next chapter, we will evaluate our approach and show its applicability using different case studies.

Chapter 7

Evaluation

This chapter describes the evaluation of our approach in its application to different contexts. Section 7.1 analyses some meta-model repositories to prove the applicability of our fragmentation pattern. Section 7.2 describes the fragmentation performance evaluation, carried out with synthetic and realistic models. Section 7.3 compares our fragmentation approach with third party tools. Section 7.4 evaluates the scoped validation pattern with synthetic models and measures the performance speed-up when we combine this pattern with a model indexer. Section 7.5 describes the case studies in which we apply our catalogue of patterns and graphical heuristics to obtain scalable environments. Finally, Section 7.6 shows some applications of our approach, in combination with other research work.

7.1 Applicability of the Fragmentation Pattern

In this section, our goal is to answer the following research question:

RQ1: Is the proposed model fragmentation approach into projects, packages and units applicable in practice?

Our approach exploits the containment relations in meta-models to customize a fragmentation strategy. The rationale is that EMF meta-models make heavy use of containment relations, so that models have a tree structure where each object is contained under one parent, except the root object. Hence, a common idiom for EMF meta-models is to have a root class containing directly or indirectly all other classes of the meta-model. This root class plays the role of Project in our fragmentation pattern, while Package and Unit classes require subsequent containment relations.

To have an intuition of the practical applicability of our fragmentation approach, we have analysed the following two meta-model repositories to assess to which extent they make use of containment relations:

- The ATL meta-model zoo[5]. This is a repository hosted by the AtlanMod research team, consisting of 301 EMF meta-models created by developers with mixed experience. Hence, the repository includes meta-models used in academia and appearing in research papers, but also meta-models of large standards like BPMN or BPEL.
- Meta-models of OMG standards[6]. The Object Management Group (OMG) is a standardization body that produces meta-model-based standards for technologies like UML, OCL, QVT or BPMN, among others. These meta-models were created by professional engineers with high expertise. For our analysis, we have considered 224 meta-models of OMG standards, corresponding to those meta-models which are in a format we can parse, and considering that some standards provide several meta-models.

The repositories contain some very large meta-models. In the ATL zoo, the Industry Foundation Classes (IFC) meta-model contains 699 classes, and the OMG standard with the highest number of classes is the Robotic Interaction Service Specification (RoIS) with 657. On the other hand, both repositories have meta-models with a small number of classes. In the ATL zoo, 75% of meta-models have less than 38 classes, while in the OMG, 75% have less than 63 classes.

For each meta-model, we computed its *containment depth*. This is the length of the longest path of containment relations starting from the root class. We detected the root class of each meta-model automatically as follows. For each class, we calculated the number of classes that it contained directly or indirectly. Then, the class containing more classes was selected as the root, and in case of tie, the class not contained by any other was selected. If several classes had those characteristics, then the first one in the list of possible roots was selected. The goal was to leave out as few classes as possible.

Within the analysed repositories, our algorithm did not detect a root in 7% of meta-models because they lacked containment relations, while 58% of meta-models had one root class. In the rest of cases, the algorithm found several roots in the same meta-model, which justifies our decision to support multiple roots in the fragmentation pattern.

[5]http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore
[6]https://www.omg.org/spec/

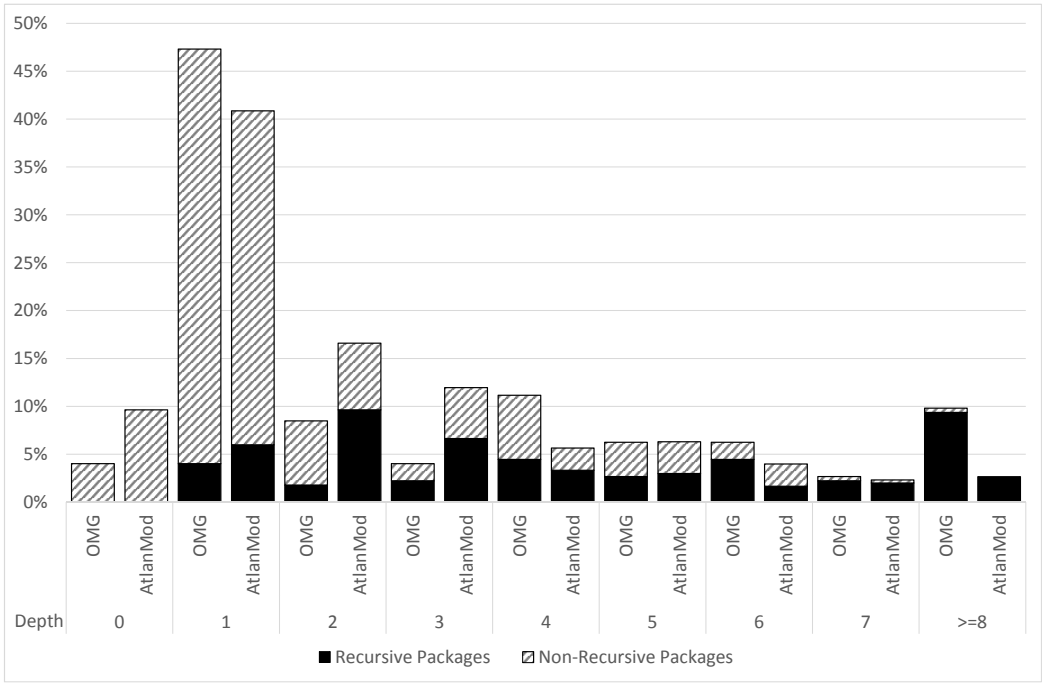


Fig. 7.1 Containment depth across the repositories, and distribution of packages and recursive packages according to the depth.

Next, we heuristically assigned a fragmentation strategy to each meta-model. The heuristic annotated the root classes as Projects; the classes with recursive containment, or with containment depth greater than 1, as Packages; and the classes with containment depth equal to 0 or 1 as Units. By recursive containment we mean containment relations that can store instances of the class defining the relation, like Subsystem.subsystems in Figure 2.3.

Figure 7.1 shows the containment depth of the meta-models in both repositories (x axis) and the percentage of meta-models with that depth (y axis). We separate the ATL and OMG meta-models to understand whether there are differences between them, given the different background of their developers and the different scope of the meta-models.

The figure shows that less than 10% of the ATL meta-models and less than 5% of the OMG meta-models have no containment relations. In the ATL repository, 29 meta-models have no root, and 75% of them have less than 16 classes. In the OMG repository, 9 meta-models have no root, and 75% of them have less than 29 classes. Our fragmentation pattern cannot be applied to these meta-models as they lack a root class.

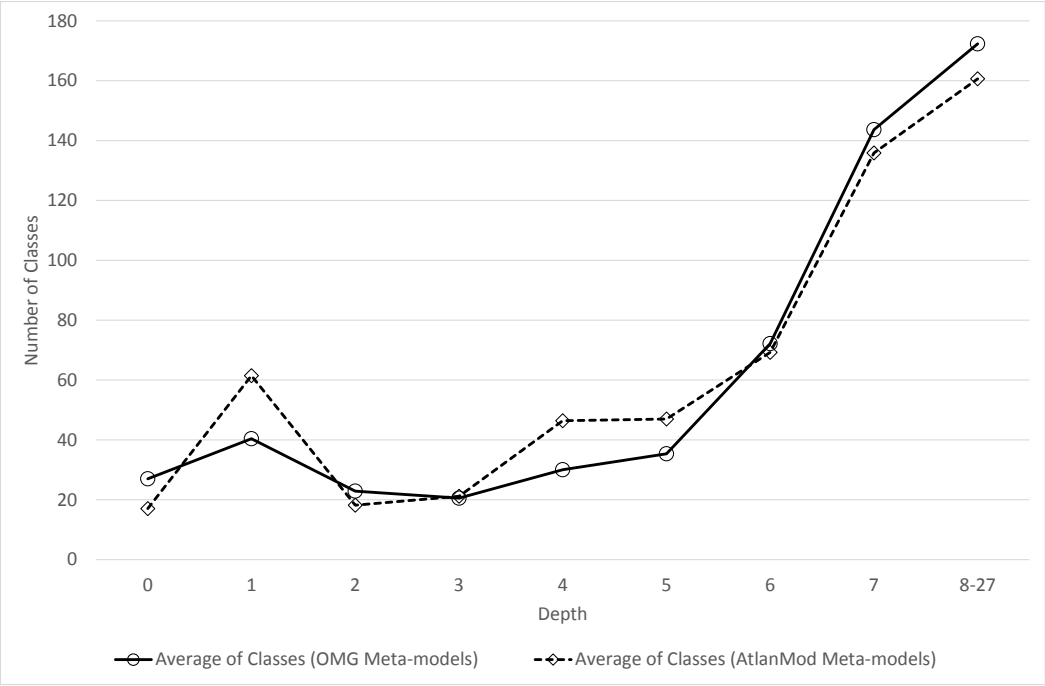


Fig. 7.2 Meta-model size (in classes) vs containment depth.

On the contrary, the depth is greater than zero in more than 90% of the meta-models, and our approach can be used on them. Most meta-models in both repositories have depth 1. The ratio of recursive containment relations (leading to recursive packages) vs. non-recursive packages increases as the containment depth increases. Interestingly, excluding the meta-models with depth zero, the containment depth follows a power law distribution, found in many natural and man-made phenomena [94].

Figure 7.2 depicts the correlation of containment depth (x axis) with meta-model size expressed as the number of classes (y axis). We can see that there is a tendency for longer containment depths in bigger meta-models in both repositories.

From this experiment, we conclude that our fragmentation approach can be applied to most meta-models in both repositories (potentially 90% of ATL meta-models and 95% of OMG meta-models). With regard to the fragmentation strategies that we computed heuristically, we found that near 50% of the meta-models in both repositories contain some Package class. Packages permit grouping model fragments and provide a modular structure, which is one of the main objectives of our pattern. The analysis also shows that there are two features that make a meta-model more amenable to fragmentation using our approach: first, deep containment trees permit creating different types of packages; second, recursive containment relations permit nested packages. Hence, our

fragmentation is more useful on larger meta-models, as their containment depth and ratio of recursive containment relations are higher.

Altogether, we can answer the research question **RQ1** affirmatively: our fragmentation pattern is applicable in practice, being more beneficial for large meta-models. The evaluation materials (e.g., meta-models, models) are available at https://github.com/antoniogarmendia/emfsplitter-materials.

7.1.1 Threats to Validity

The analysis considers a large number of meta-models (more than 500) including standards, which ensures the robustness of the findings. Moreover, it takes into account two different repositories to foster diversity of meta-models. To strengthen our results, we plan to repeat the analysis on meta-models hosted in public code versioning systems like Github. On the other hand, our experiment applied fragmentation strategies automatically computed according to a set of heuristics, but these strategies may be different from the ones that a human may have manually defined. It is future work to perform another evaluation using fragmentation strategies manually defined by developers.

7.2 Fragmentation Scalability

In this section, our goal is to answer the following research question:

RQ2: Which is the incurred cost of fragmentation?

We evaluate the scalability of our tools to deal with large models. For this purpose, we present two experiments, one using synthetic models (Section 7.2.1), and the other one using real models created by third parties (Section 7.2.2).

In all our tests, we use the following environment:

- Execution environment:
  - Operating System: Windows 7 Professional Service Pack 1.
  - Processor: Intel(R) Core(TM) i7-2600, 3.40GHz
  - RAM: 12 GB
- Java Virtual Machine Configuration:

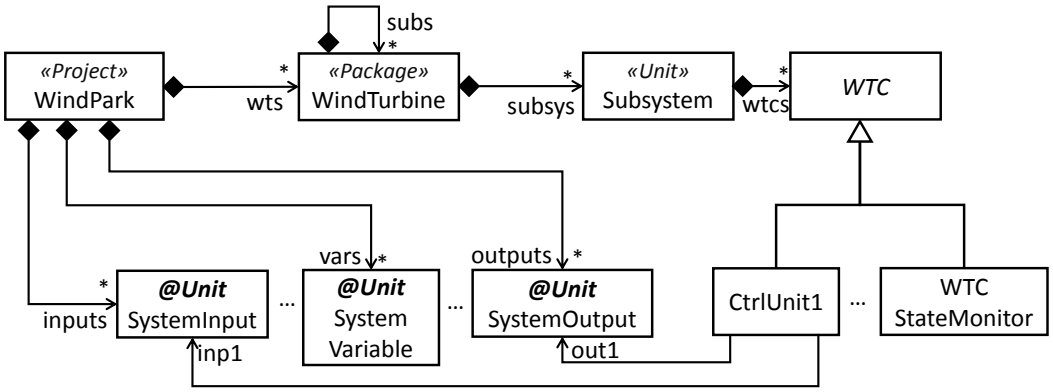


Fig. 7.3 Excerpt of the meta-model of one of the MONDO case studies, with fragmentation strategy annotations.

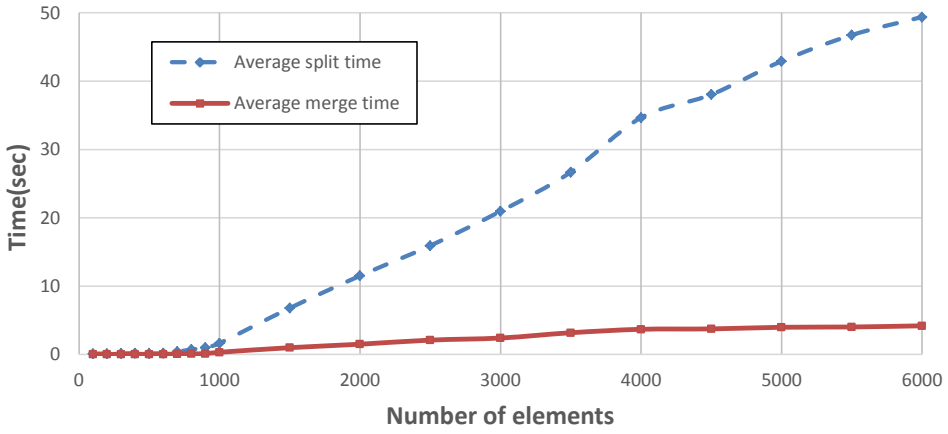
- Execution environment: Java SE 1.8 (*jre1.8.0_40*)
- Initial memory: 512 MB
- Maximum memory: 8 GB

7.2.1 Synthetic Models

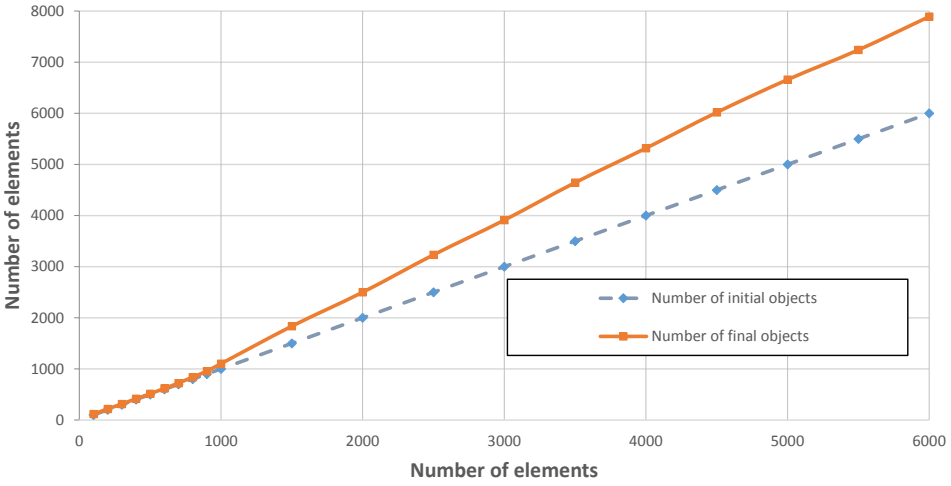
For this experiment, we generate models using an EMF random model instantiator from the AtlanMod team[7]. We use a meta-model taken from a case study of the EU project MONDO in the domain of component-based embedded systems (a variation of the running example using throughout the thesis). Figure 7.3 shows a small excerpt of the meta-model (the complete one has about 150 classes). A WindPark, has a set of control parameters (inputs, outputs, variables, etc.), and organises the controllers for the WindTurbines hierarchically. There is a large number of predefined controllers (subclasses of WTC, just two classes are shown for illustration), each with its own set of control parameters. For the experiment, we consider models with sizes ranging from 100 to 6 000 model elements. For each size, we generate 500 different models.

Figure 7.3 shows the roles of the desired fragmentation strategy for the wind turbine meta-model. This way, class WindPark is the root class, and has been tagged as Project. WindTurbines are tagged as Package, so that a folder will be created for each (nested) wind turbine system. All control parameters (inputs, outputs, variables, etc.) are stored in a separate file depending on their type, while the set of components of each subsystem controller (class Subsystem) is stored in a file as well. For clarity purposes, we did not show in the figure the roles of the references.

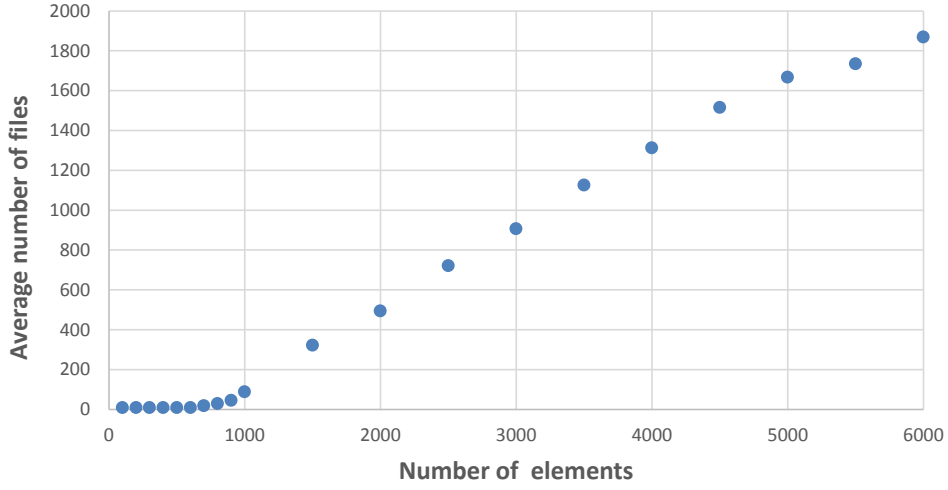
[7]http://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/



(a) Merge and split time.



(b) Comparison of initial and final objects.



(c) Average number of files.

Fig. 7.4 Results of splitting/merging IKERLAN's synthetic models.

Figure 7.4 shows the results of the fragmentation of the synthetic models. Figure 7.4a shows that EMF-Splitter is able to split a model that contains 6 000 objects in less than one minute and merge back all fragments into a monolithic model in less than 10 seconds. For splitting, the times include loading the monolithic model, performing the fragmentation in memory, and serializing all the fragments in disk using XMI. For merging, the times include loading all fragments from disk, performing the merging in memory, and serializing the monolithic model back in disk in XMI format. We can see that splitting is more time-consuming than merging, due to the computation that needs to be performed to locate the right folder or unit each element belongs to.

In Figure 7.4b, we compare the number of initial and final objects, before and after the fragmentation. It can be observed that there are more objects (proxy references) after the fragmentation in order to maintain the cross-references between the different fragments. The amount of proxies created depends on the number of files created by the fragmentation strategy. Figure 7.4c shows the average number of files created for each model size. In average, around 1 800 files are created for models that contain 6 000 objects. We also observe that, in average, a maximum of one or two proxy objects are created for each model fragment. Therefore, we can conclude that the overload caused by the splitting (in terms of increased size per fragment) is low. Moreover, it should be stressed that fragmenting a monolithic model is a one-time operation, which needs to be performed just once, as the fragmented model can then be used in place of the monolithic one.

Figure 7.4a shows an increase of splitting time for models with more than 1 000 objects. This can be attributed to a combined effect of model size and number of generated fragment files (see Figure 7.4c, where the number of files also increases abruptly at around the same number of objects). Figure 7.5 shows the effect of the generated number of fragment files in the average split time of all models in the experiment, with sizes ranging from 100 to 6 000 elements. We observe a linear increase. To better understand the effect of the number of fragments, Figure 7.6 shows the variation in split time for a set of 13 models with 6 000 elements each, but leading to a number of fragments ranging from 800 to 2 000 files. It can be observed that fragmenting models leading to around 800 fragments is 20 seconds faster than fragmenting models (of the same size) leading to around 2 000 fragment files.

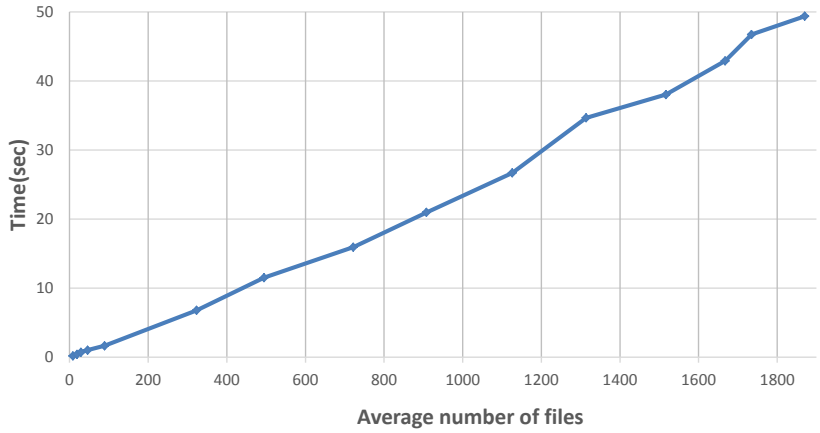


Fig. 7.5 Effect of the number of files created in split time. Average times of all models, with sizes in the range 100 - 6 000 elements.

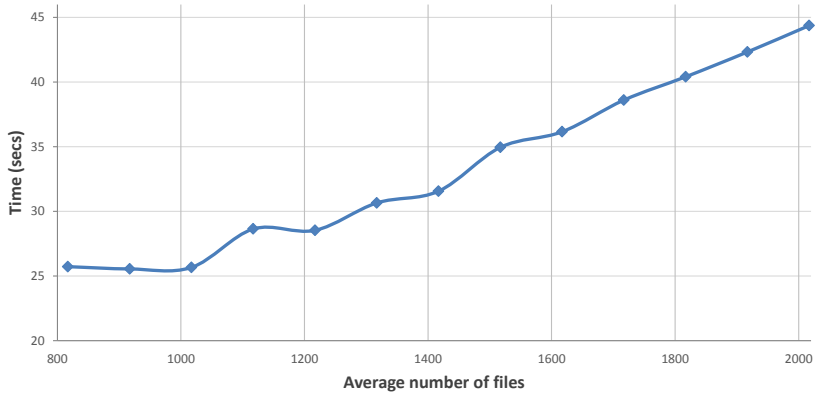


Fig. 7.6 Effect of the number of files in split time. Average times of a set of models of size 6 000.

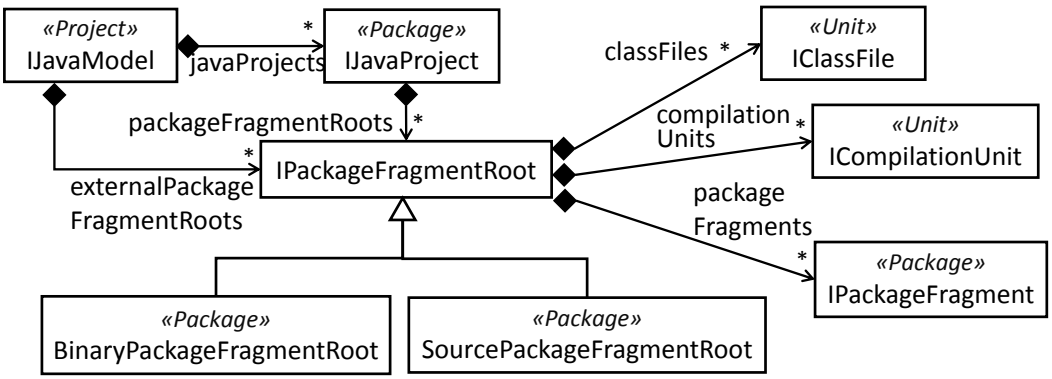


Fig. 7.7 JDTAST meta-model, with fragmentation strategy annotations.

7.2.2 Realistic Large Models

In this experiment we use realistic models from the JDTAST models of the GraBaTs'09 competition. This contest provides a case study[8], which consists in representing Java programs as models. Figure 7.7 shows the fragmentation strategy applied to the JDTAST meta-model. In this case, the IJavaModel class is mapped to Project and the IJavaProject class is tagged as Package. This is possible because there is a composition relation from IJavaModel (the project) to IJavaProject, as the pattern demands by means of relation javaProjects. Another composition relation between IJavaProject and IPackageFragmentRoot allows classes that inherit from the latter (BinaryPackageFragmentRoot and SourcePackageFragmentRoot) to be tagged as Package. Additionally, the relation packageFragments enables IPackageFragment to be tagged as Package. Finally, both IClassFile and ICompilationUnit are annotated as Unit.

After the application of the fragmentation pattern, we split all the models of the GraBaTs case study, turning each one of them into an Eclipse project. As an example, Figure 7.8 shows the generated modelling environment with an Eclipse project, named Projectset0, created from the model set0.xmi. The project explorer shows the structure of folders and files generated from the model, which follows the specified fragmentation strategy. The project created from the model has similar structure to a Java project. As can be seen, we have chosen icons for folders and units that resemble the ones used by the Java plugin to represent packages and Java classes. However, the project is a model conformant to the JDTAST meta-model, fragmented across the file system. The project explorer to the left shows the hierarchical model structure, and can be used to navigate through it. To the right, a tree editor shows the content of one of the fragments. While the original model has about 70 000 model elements, the

[8]http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study.

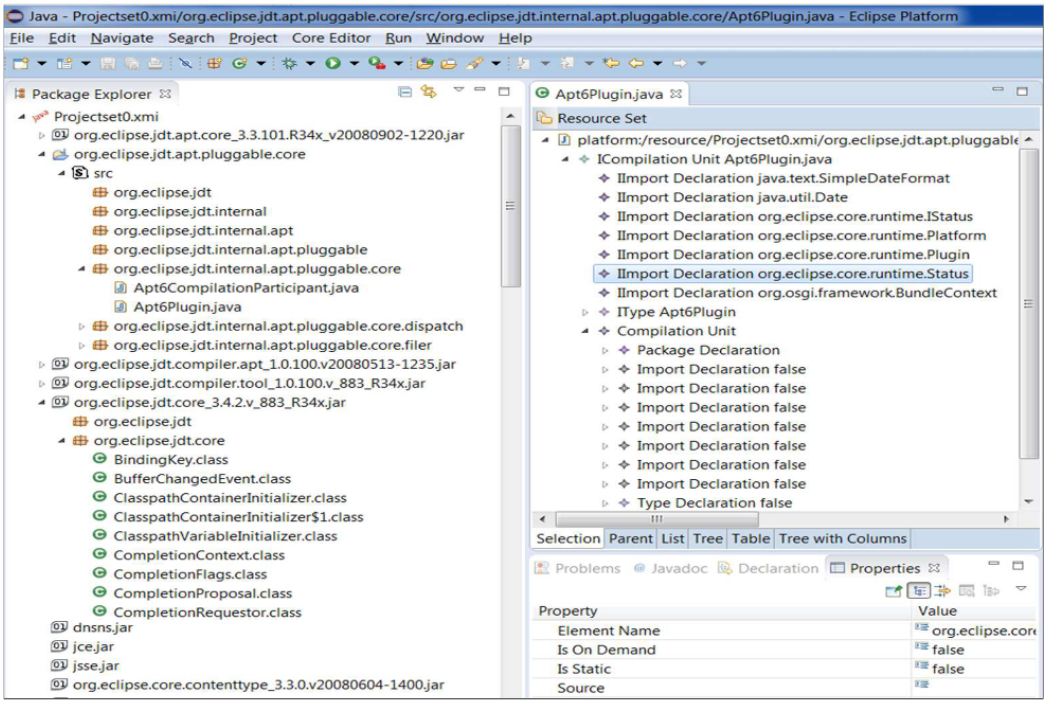


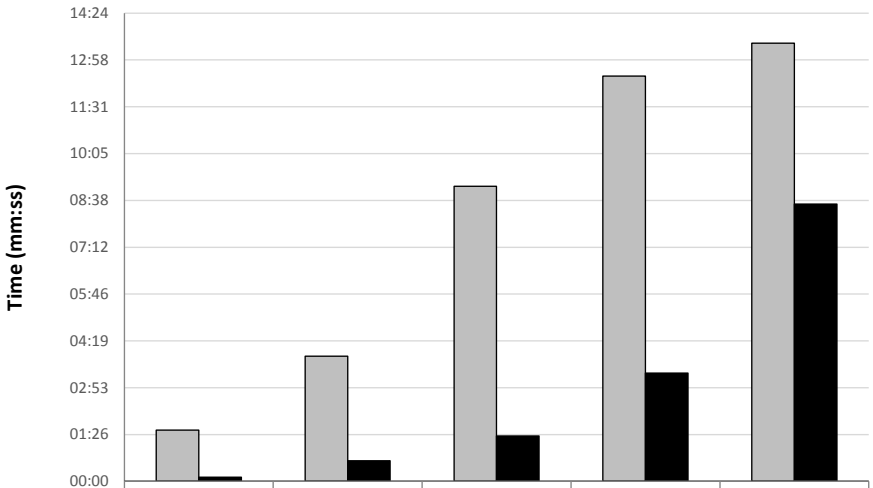
Fig. 7.8 Environment generated by EMF-Splitter for the JDTAST meta-model.

fragmentation strategy splits it into 1 800 files of much lower size (with an average of 40 elements). This allows faster loading of each fragment, and a better navigation of the model.

Model	Split time	Merge time	#files	Avg	Max	# model elements
set0	$1min34s$	$8s$	1 779	40.17	1 322	71 458
set1	$3min51s$	$38s$	6 240	32.68	4 549	203 938
set2	$9min5s$	$1min24s$	6 050	345.27	50 718	2 088 890
set3	$12min28s$	$3min20s$	4 460	1 031.24	50 718	4 599 358
set4	$13min28s$	$8min32s$	5 068	980.04	50 718	4 966 846

Table 7.1 Results of splitting and merging the JDTAST models.

Table 7.1 and Figure 7.9 present the results of the experiment. The graphic shows a comparison between split and merge times. As in the experiment of Section 7.2.1, splitting is slower than merging. The table contains more detailed information, including columns for the split time, merge time (merging all files of a fragmented model into one file), generated number of files, mean and maximum number of elements of each fragment, and total number of elements in the whole model. We can observe that the maximum number of elements in a file is repeated for models *set2*, *set3* and *set4*. The reason is that this group of models was built by adding Java classes incrementally. For example, *set2* is formed by *set1* and the addition of some Java packages.



	set0	set1	set2	set3	set4
Split time	01:34	03:51	09:05	12:28	13:28
Merge time	00:08	00:38	01:24	03:20	08:32

Fig. 7.9 Split/merge times over the JDTAST models.

These experiments over synthetic and realistic models answer the **RQ2**. Fragmentation is a time-consuming task, for example, models up to 2 000 000 elements, the fragmentation time is around 9 minutes. However, model splitting is a one-time operation, which lead us to analyse, if it brings any profit when we handle large models, in terms of visualization, indexing or edition. Section 7.3 analyses the benefits of fragmenting models and its compatibility with de-facto standard tools.

7.2.3 Threats to Validity

In these experiments, we have evaluated the scalability, obtaining good results for large models when the fragmentation strategies are used. In terms of size, we see a large reduction of size of the biggest fragment with respect to the total model size (see Table 7.1).

The question arises whether these results are generalizable to other arbitrary meta-models (i.e., the external validity of the experiments). Fragmentation gives good results with meta-models that have a strong hierarchical structure, reflected by the existence of composition references between classes. EMF meta-models tend to heavily use composition references and it is usual to have a root class in every meta-model, which contains directly or indirectly all other classes. For the experiments and the running example, we used three meta-models (developed by third parties), for which the fragmentation strategies worked well. We can also observe that meta-models for programming languages (e.g., JDTAST) or the WT meta-model have a hierarchical

structure. However, one may also find "flat" meta-models with few compositions, for which no option but to include all model objects in the same fragment would be available. Therefore, we cannot generalize the fragmentation results to arbitrary meta-models, but we can see that this technique fits especially well in meta-models for programming languages, or domains in which models are hierarchical. For example large modelling languages, like the UML, tend to have hierarchical elements (e.g., models are divided into diagrams and packages), which would be suitable for fragmentation strategies. As shown in the experiment in Section 7.1, the number of flat meta-models is low, around 7% of 525, for the analysed repositories.

7.3 Comparison with Third Party Tools

In this section, our goal is to answer the following research question:

RQ3: Is fragmentation profitable and compatible with the use of other de-facto standard tools?

The goal is to assess the benefits that our fragmentation approach has over de-facto standard tools. Firstly, we compare (both in terms of time to open a model and in reduction of memory size) with the standard modelling choice within the Eclipse ecosystem: using monolithic models with the default XMI serialization and EMF's default tree editor. Second, we compare with CDO, a common alternative to XMI for model persistence. Finally, we compare with *Gephi*, one of the most widely used tools for graph visualisation.

7.3.1 Fragmentation vs. Monolithic Models and EMF Tree Editor

In this experiment, the objective is to assess the gain obtained by using a fragmentation strategy in case that it is used in combination with the default tree model editor. This way, we compare the time needed to open the monolithic models from the JDTAST use case with the default model visualisation of EMF (the reflective tree editor), with respect to open the largest model fragment produced by the fragmentation strategy. Figure 7.10 shows the time needed by the tree editor to open the complete models (grey columns to the left of each series). For the largest model, it needs two and a half minutes to open.

The figure also shows the time needed to open the largest fragment in each model (black columns to the right of each series). We detail these times also in Table 7.2

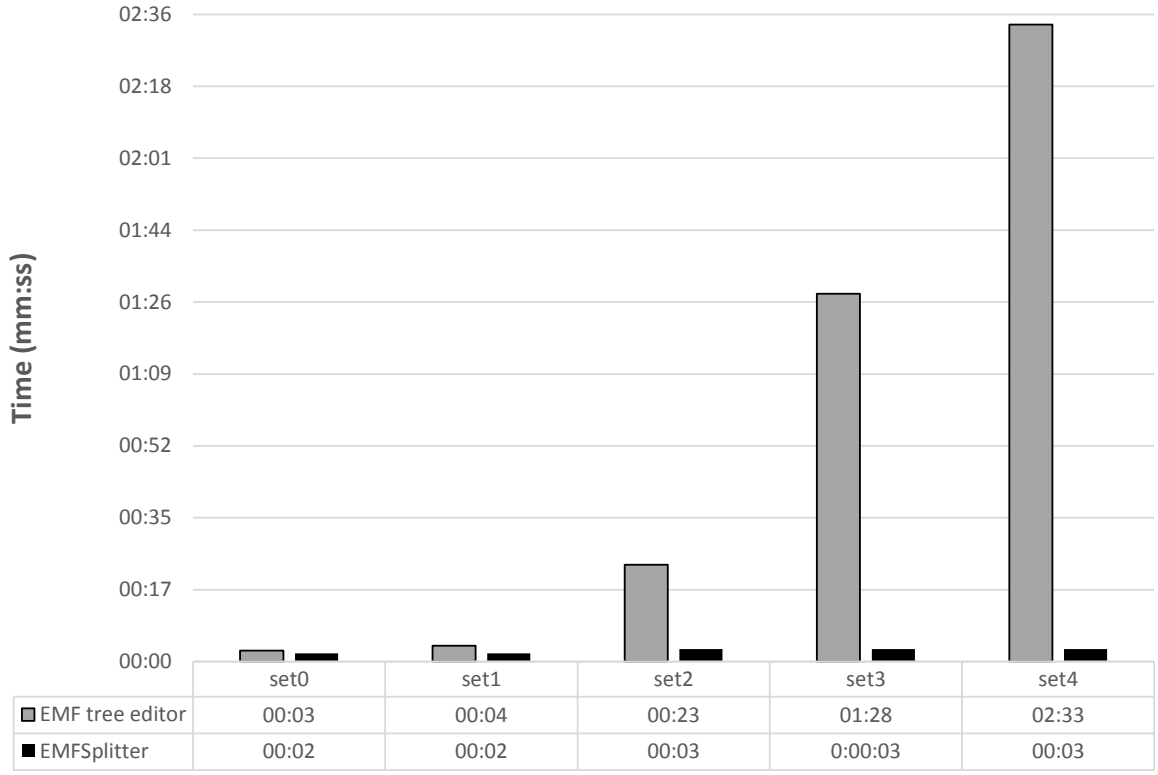
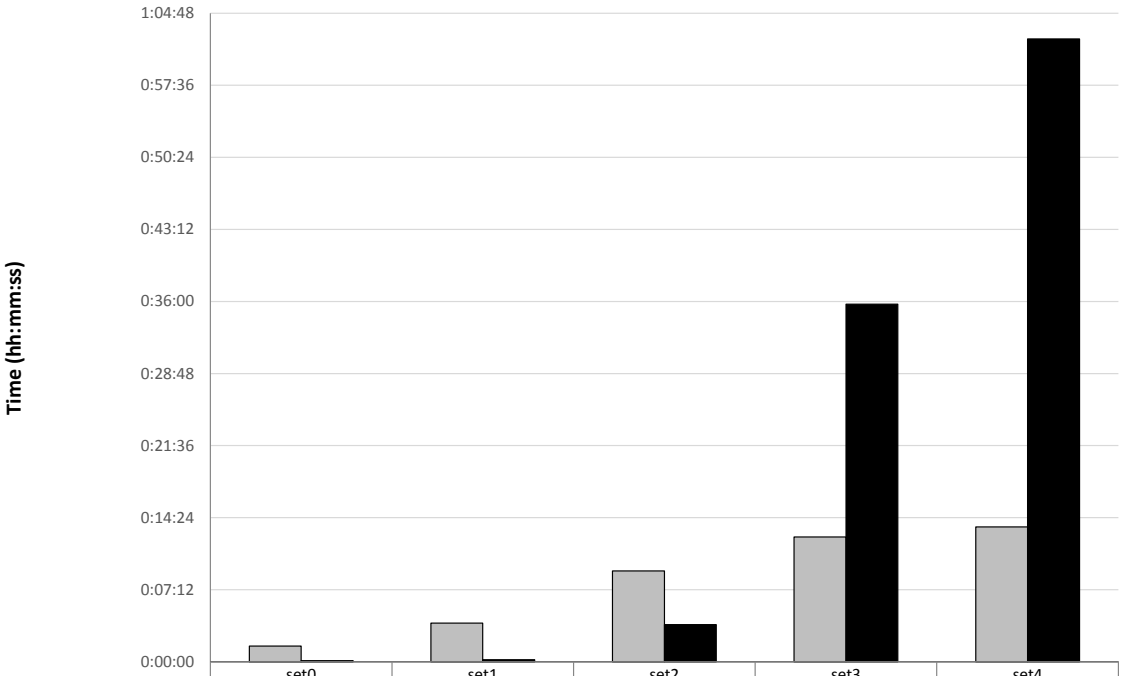


Fig. 7.10 Time required to open the model with EMF's reflective tree editor (grey columns to the left) and EMF-Splitter (black columns to the right).

as they are much smaller than the time needed to open the complete models. It can be observed that for the largest model it takes less than 3 seconds, which is more than 55 times less than the time used for the monolithic model. The largest fragment (CompletionEngine.java) is the same for *set2*, *set3* and *set4* because (as previously mentioned) the latter two models are extensions of *set2*. Hence, we can conclude that the fragmentation strategy really makes more scalable the model visualisation using the default tree model editor.

Model	Fragment	# Model elements	Time	Gain w.r.t. monolithic
set0	ProblemReporter.class	1 322	1.67s	1.57x
set1	BaseConfigurationBlock.java	4 549	1.76s	2.19x
set2	CompletionEngine.java	50 718	2.65s	8.77x
set3	CompletionEngine.java	50 718	2.65s	33.36x
set4	CompletionEngine.java	50 718	2.65s	55.76x

Table 7.2 Time required (s) to open the biggest fragment model of each project with the tree editor.



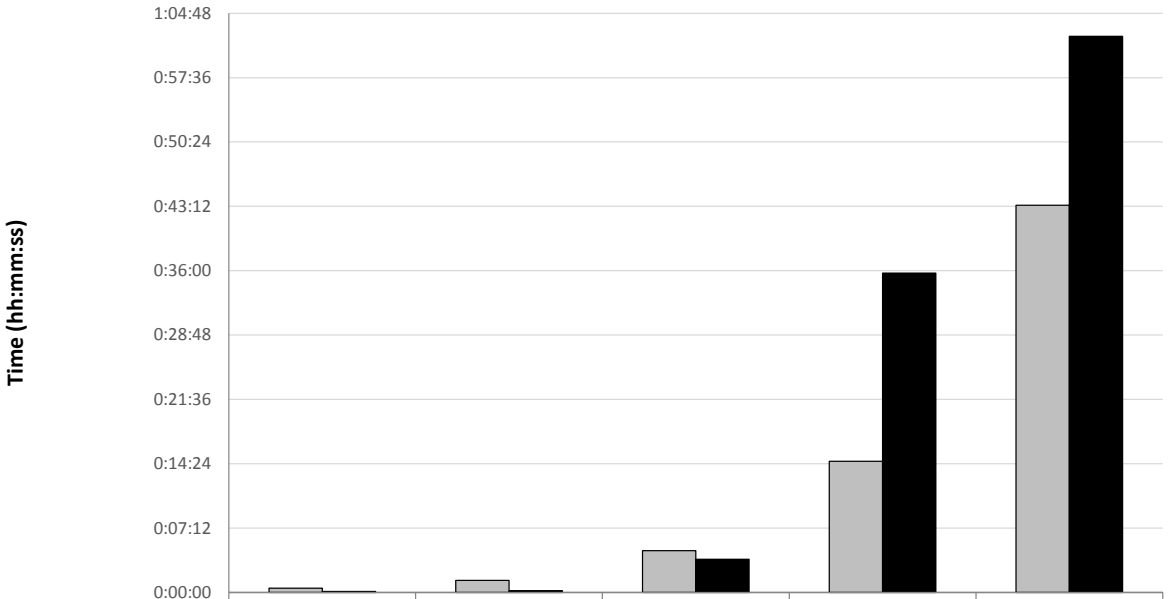
	set0	set1	set2	set3	set4
Split time (EMFSplitter)	0:01:34	0:03:51	0:09:05	0:12:28	0:13:28
Import model time (CDO)	0:00:06	0:00:12	0:03:43	0:35:44	1:02:14

Fig. 7.11 Time required to split the models with EMF-Splitter (grey columns to the left) and import the models into CDO (black columns to the right).

7.3.2 Fragmentation vs. Database Persistence Layer

An alternative to fragmentation is to use a more performing back-end, not based on file storage. A widely used option is CDO [25], which is both a model repository and a persistence framework. CDO has many drivers for persistence, usually connected to relational databases. In order to use CDO, the meta-models need to be migrated, and the models inserted in the back-end. Similar to fragmentation, model insertion is a pre-requisite to use the CDO technology. Therefore, we compare model insertion time in CDO with model fragmentation time in EMF-Splitter. As a concrete back-end, we use *DB Store*, because it is the default choice for CDO. Furthermore, it has been maintained throughout the different versions of CDO and supports all its features. For model versioning, we use the default choice, which is *branching*.

In the experiment, we measure the time required for CDO to import the JDTAST models. Importing each model was repeated three times and the result average is shown in Figure 7.11. The figure also shows the splitting model time (see also Figure 7.9). While model insertion in CDO is faster than fragmentation for the first three models, fragmentation is faster for the last two models of bigger size. This suggests a better



	set0	set1	set2	set3	set4
Fragmented model (CDO)	0:00:29	0:01:21	0:04:41	0:14:41	0:43:20
Monolithic model (CDO)	0:00:06	0:00:12	0:03:43	0:35:44	1:02:14

Fig. 7.12 Time required by CDO to import fragmented (grey columns to the left), and monolithic models (black columns to the right).

scalability of fragmentation to handle large monolithic models. It must be stressed that CDO optimizes the persistence of some EMF features, like object identifiers. CDO uses counters as identifiers for newly inserted objects, which is more efficient than using Universally Unique Identifiers (UUIDs), the scheme provided by EMF.

The figure shows that inserting large monolithic models into CDO is very costly, and takes more than one hour for *set4*. Hence, a natural question is whether a pre-processing phase of fragmentation enhances this time. That is, we may wonder whether fragmented models are faster to insert than monolithic ones. To answer this question, we performed another experiment comparing the performance of CDO for importing large monolithic models and for importing the same model divided into files. For this experiment, we use the models of JDTAST, fragmented in Section 7.2.2 using EMF-Splitter. The operation of importing the fragmented models was repeated three times and the average time is shown in Figure 7.12. We can see that fragmentation reduces the import time both for the set3 and set4 models, while the overhead of file access results in slightly higher insertion times for the three smaller models. Hence, this experiment shows that large monolithic models take longer to process by CDO than fragmented ones. Even though model insertion in CDO is an operation that needs to be performed once, for set3 and set4 it pays off to fragment them first.

7.3.3 Fragmentation vs. *Gephi*

The goal of this experiment is to compare with a graph visualisation tool outside the Eclipse ecosystem. A typical choice would be *Gephi* [9, 50], a popular, open-source tool for graph exploration and analysis. In order to make a reasonable comparison, we converted each JDTAST model (*set0*, *set1*, *set2*, *set3* and *set4*) from XMI into the GraphML format. The experiment measures the time needed by *Gephi* to open those models. In particular, we measure the time spanning since the windows of *Gephi* open (i.e., since the Java Virtual Machine is started) until the graph is completely drawn in the screen. We did such tests 10 times to obtain an average of the time we want to measure.

We obtained the following qualitative results:

- *set0* and *set1* are properly opened.
- *set2*, *set3* and *set4* required more memory than the default *Gephi* memory for the virtual machine. Once opened, it was impossible to work with those graphs because *Gephi* got frozen very often doing internal computations. Actually, these graphs are beyond *Gephi*'s announced capabilities in its web page (100 000 nodes and 1 000 000 edges).

Table 7.3 summarises the time required to load the models in *Gephi*, the time needed to split the models using EMF-Splitter, and the time needed to load the biggest fragment using the default tree editor. Generally, splitting the models is slower than loading them in *Gephi*. However, model splitting is a one-time operation, which only needs to be performed once. Opening the model afterwards takes – in the worst case – the time of opening the biggest fragment. Those numbers are much lower than loading the whole model, as Table 7.3 shows. It has the additional advantage that the amount of objects in memory is considerably lower, and hence the visualisation techniques can be done faster. Therefore, we can conclude that fragmentation is a very useful pre-drawing technique for tools aiming at large scale graph exploration.

Model	Gephi	Split time	Time to load biggest fragment
set0	$3s$	$1min34s$	$1.67s$
set1	$6s$	$3min51s$	$1.76s$
set2	$3min40s$	$9min4s$	$2.65s$
set3	$9min36s$	$12min27s$	$2.65s$
set4	$14min10s$	$13min28s$	$2.65s$

Table 7.3 Comparison between opening a model with *Gephi*, splitting the model with EMF-Splitter, and loading the biggest fragment produced.

7.3.4 Threats to validity

Splitting the model to obtain small chunks, resulted in a drastic reduction in loading time using the default tree editor (see Figure 7.10). Moreover, fragmentation allows the handling of large models, which is generally a useful pre-processing technique when the model is to be visualized (as we have seen in the *Gephi* experiment) or inserted into model persistence back-end (as we have seen in the CDO experiment). Overall, we can conclude that for the experiments performed, the proposal provides good scalability, and that fragmentation can be a good complement to other tools to handle very large models.

Fragmenting an existing, large monolithic model can be time-consuming, as shown in Table 7.1. However, this is a one-time operation, so the benefits increase as the fragmented model is visualised more repeatedly. Also, please note that the fragmented models are not read-only, and can be modified. Hence, once a monolithic model is fragmented, there is no need to merge it back anymore.

Finally, these results investigate the efficiency of the approach, and replying positively to **RQ3**, but leave out the usability of the tools. An empirical study with users would be needed to assess the usability of the approach, which is also left for future work.

7.4 Performance of Scoped Constraints

In this section, our goal is to answer the following research question:

RQ4: Is the evaluation of scoped constraints on fragmented models more efficient than the evaluation of standard constraints on monolithic models?

To evaluate the performance gains of our approach, next we report on four experiments analysing the effects of fragmentation and scope in the execution time of constraint validation:

1. First, we compare the validation time of standard constraints on monolithic models (the baseline) with respect to the full validation of equivalent scoped constraints on fragmented models.

2. Second, we investigate whether the number of fragments affects the validation performance.

3. Next, we compare the full validation of scoped constraints (i.e., on all units and packages of a fragmented model) vs. their incremental validation (i.e., only on the elements affected by a model change).

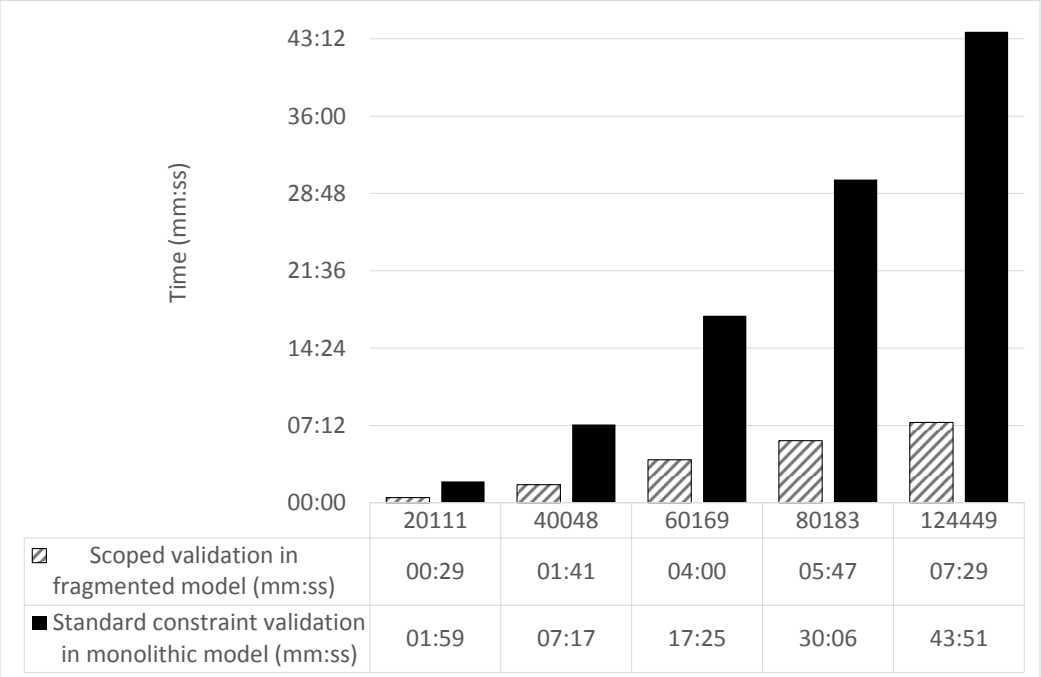
4. Finally, we analyse the efficiency gains when integrating the Hawk model indexer with scoped constraint validation.

Experiment setting In all four experiments, we used the meta-model of the running example (see Figure 2.3) and the fragmentation strategy defined in Figure 4.2. We considered a suite of eleven EOL constraints in both standard and scoped versions. The constraints are available in Appendix A, and Table 7.4 summarizes their characteristics. We considered constraints with all kinds of scope (one with scope sameProject, three with scope sameRootPkg, two with samePkg, and five with sameUnit). As a measure of their complexity, the table includes the number of nodes in the abstract syntax tree of each constraint expression. For example, the expression StateMachine.allInstances()→size() <= 10 corresponding to constraint numberStateMachines has three nodes. The average number of nodes in the constraints is 6.3, ranging from three to fifteen.

Constraint	Scope	Complexity (#nodes)
numberStateMachines	sameProject	3
numberControlSubsystems	sameRootPkg	3
numberComponents	sameRootPkg	3
depthSubsystem	sameRootPkg	13
connectedComponents	samePkg	11
inputPortSubsystem	samePkg	8
oneInitialState	sameUnit	3
existsSimpleState	sameUnit	3
reachableState	sameUnit	5
connectedPorts	sameUnit	15
initStateIsNotIsolated	sameUnit	3

Table 7.4 Characteristics of scoped constraints used in the evaluation of performance.

The experiments were executed four times in a computer with Windows 10 Education version, processor Intel(R) Core(TM) i7-3770, 3.40GHz, and Java SE 1.8 with 8GB as initial and maximum memory. The constraints were validated on synthetic models of increasing size (from around 20 000 objects to around 125 000 objects) created using the EMF random instantiator from the AtlanMod team that we also used in the previous experiments.



	20111	40048	60169	80183	124449
Scoped validation in fragmented model (mm:ss)	00:29	01:41	04:00	05:47	07:29
Standard constraint validation in monolithic model (mm:ss)	01:59	07:17	17:25	30:06	43:51

Fig. 7.13 Constraint validation times in monolithic and fragmented models.

7.4.1 Full Constraint Validation in Monolithic and Fragmented Models

The first experiment compares the validation of standard constraints in a monolithic model. With the validation of equivalent scoped constraints in the fragmented version of the same model. We consider the full validation of scoped constraints, i.e., their validation on all units and packages of the fragmented model. The objective is to assess whether reducing the number of objects in the validation scope also reduces the validation time.

Figure 7.13 shows the experiment results. The vertical axis shows the validation time, and the horizontal one the size of the model in number of objects. The graphic shows that the validation of scoped constraints is faster even for the smallest model of 20 111 objects. For the largest model, scoped validation is six times faster than standard validation. As explained in Section 4.3.5, this happens because scoped constraints are validated within a limited scope, and hence, less objects need to be loaded/queried. Moreover, scoped validation is only performed on those packages/units that may contain instances of the context class on which the scoped constraint is defined.

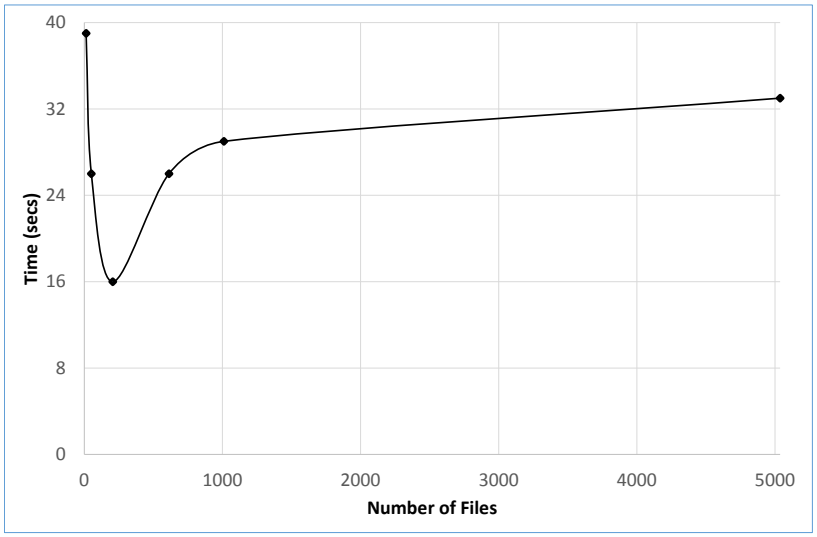
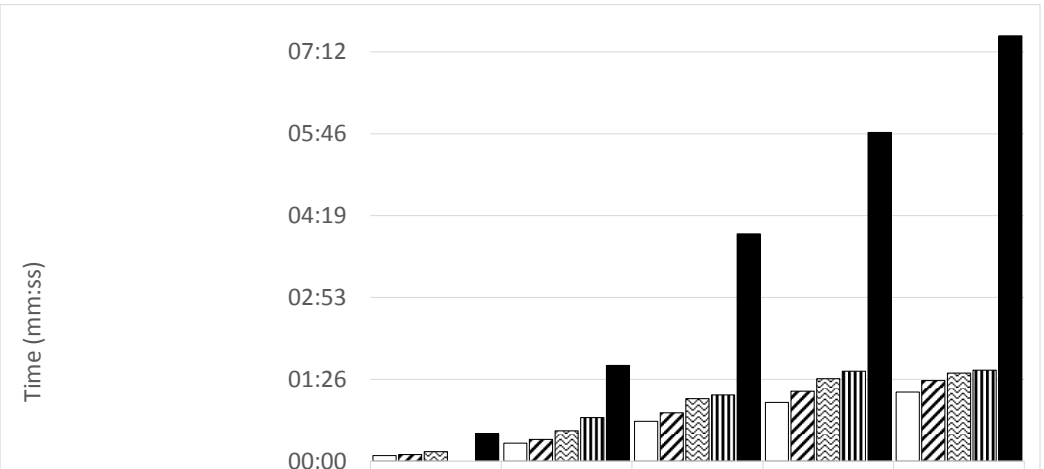


Fig. 7.14 Effect of the number of files on the scoped validation performance.

7.4.2 Effect of Number of Fragments on Scoped Validation Performance

The previous experiment shows that scoped validation pays off even for medium-sized models. However, fragmentation incurs an overhead, as each fragment requires an access to disk to load the fragment in memory. Therefore, this second experiment analyses the impact of the number of model fragments on the scoped validation performance. For this experiment, we created six synthetic models of 20 000 objects, each one of them fragmented in a different number of files ranging from 1 to 5 000. Then, we measured the validation time of the eleven constraints used in the first experiment.

Figure 7.14 shows the results. If the number of files is low (horizontal axis), the cost to iterate through the objects in the fragments increases (vertical axis). In the limit, if the model is in one file, the scoped validation time is similar to the time of evaluating the constraints in a monolithic model. Conversely, if the model is fragmented in many files, the overhead of loading them becomes apparent and the efficiency decreases. In this experiment, the best validation time was obtained when fragmenting the model of 20 000 objects in 200 files. This gives a ratio of 1 file for each 100 objects. However, this ratio cannot be taken as a general guideline, as the optimal ratio may depend on the structure and scope of the involved constraints. It is up to future work to investigate methods to obtain optimal fragmentation sizes given a set of constraints.



	20111	40048	60169	80183	124449
Incremental validation: level 2	00:06	00:19	00:42	01:02	01:13
Incremental validation: level 3	00:07	00:23	00:51	01:14	01:25
Incremental validation: level 4	00:10	00:32	01:06	01:27	01:33
Incremental validation: level 5		00:46	01:10	01:35	01:36
Full validation	00:29	01:41	04:00	05:47	07:29

Fig. 7.15 Comparison of incremental and full scoped validation times.

7.4.3 Comparison of Full Validation and Incremental Validation

Section 4.3.5 presented an algorithm for incremental scoped validation that is applicable when a localized model change occurs, optimizing the re-evaluation of constraints to the scope of the change. Hence, in this experiment, we emulate a model change, and then compare the incremental validation time (i.e., the re-evaluation of constraints on the affected model elements) and the full validation time (i.e., the re-evaluation of constraints on all model elements). The experiment considers changes on units that are located at different containment depths, from 2 to 5. We distinguish the depth of the changed resource because changes in resources that are deeper in the containment tree need to re-evaluate more constraints, incurring longer validation times.

Figure 7.15 presents the results. The figure does not show data for the model with 20 111 objects at level 5, because this model has no units or packages at this level. We can observe that the incremental validation scales better than the full one. In average, the incremental validation time is just 1 second slower for models of 124 449 objects than for models of 80 183 objects, while the full validation is more than 90 seconds slower. For the biggest model, the incremental validation yields a speed-up of around 4.5x.

7.4.4 Effect of a Model Indexer on Scoped Validation Performance

Next, we evaluate the use of the Hawk model indexer to execute scoped constraints. Although Hawk is integrated with several database technologies, we used Neo4j[9] for this experiment.

This case study led to a number of optimisations and additions to Hawk. The most notable change was the addition of two new query scoping modes for Hawk. Hawk already had the capability to limit the scope of a query so that Type.allInstances() would only return the instances within a certain subset of the indexed locations and/or files. Previously, Hawk only had one implementation for this: it would go to the type node, and then visit each instance node while checking if it belonged to a file node within the desired scope. For sufficiently common types, Hawk would end up visiting more instances than necessary. Two alternative implementations of Type.allInstances() were implemented to reduce the number of misses:

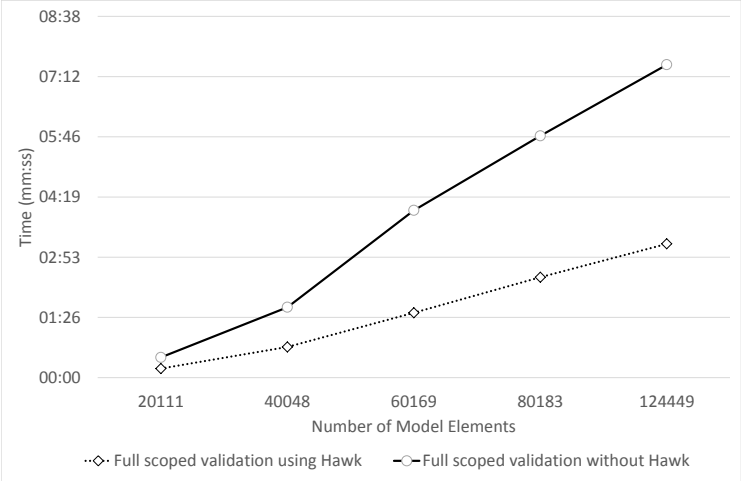
- The new *file-first* mode was used for validation rules spanning single files. It iterates over the contents of the file, filtering objects by type. This is useful when we have more instances of the type than objects in a typical fragment, which may be the case for sufficiently large fragmented models.
- The new *subtree scoping* mode was used for validation rules spanning projects or packages. The mode uses an advanced feature of Hawk called *derived edges*. Derived edges are references that are precomputed according to a *derivation logic* specified by the user. Derived edges are updated incrementally as new versions of the fragments are detected.

  Upon a request for Type.allInstances(), the subtree scoping mode will ensure that Hawk precomputes "allof_Type" derived edges from each instance to all its containers. Once this is done, answering Type.allInstances() scoped to the subtree rooted at the x project/package only requires following the "allof_Type" edges from x in reverse.

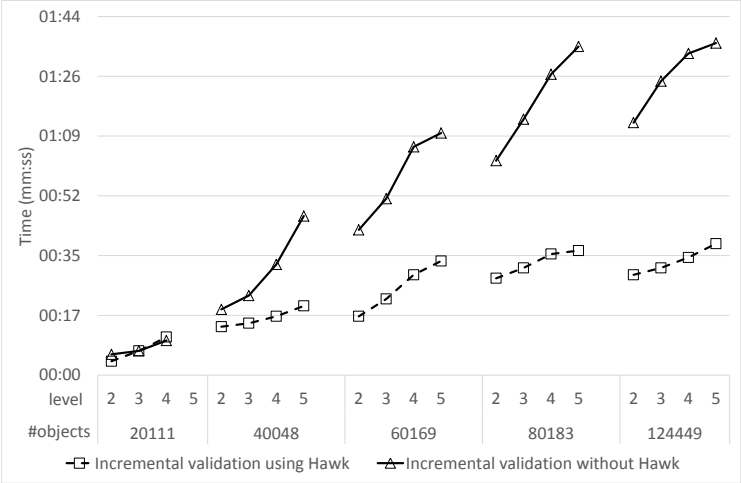
Figure 7.16a compares the full validation time with and without Hawk. For the biggest model, Hawk speeds up constraint evaluation more than 2x.

Figure 7.16b makes the same comparison but for incremental validation, and distinguishing the depth of the changed resource. As before, we generally obtained

[9]https://neo4j.com/



(a) Full validation times.



(b) Incremental validation times.

shorter validation times using Hawk, peaking a speed up of around 3x for the biggest model. However, the validation time for some of the smallest models (the deeper ones) was slightly faster without indexing.

In view of the results obtained in the four experiments presented in this section, we can answer **RQ4** affirmatively: scoped constraint validation is more efficient than the evaluation of equivalent non-scoped constraints on monolithic models. Model fragmentation and scoped constraints permit the incremental validation of constraints, which leads to speed ups of up to 4.5x. Incremental validation can be enhanced with model indexers, obtaining an additional increase of efficiency up to 3x. The number of files in which a model is fragmented has an effect in performance as well; for this particular experiment, having 1 file for each 100 objects yielded optimal validation times, though this ratio may vary for other cases.

7.4.5 Discussion and Threats to Validity

This evaluation has been performed using a meta-model that comes from an industrial partner in a European project[10]. However, we slightly adapted the meta-model to better illustrate the concepts introduced in this thesis, removing some of its complexity. Moreover, we used synthetic models and a random instantiator in our experiments. In consequence, the models used in the evaluation may not be fully representative of realistic models. To mitigate the threats to the validity of our results due to this risk, the Section 7.5.1 present a case study built over a meta-model and a set of constraints built by a third-party.

7.5 Case Studies

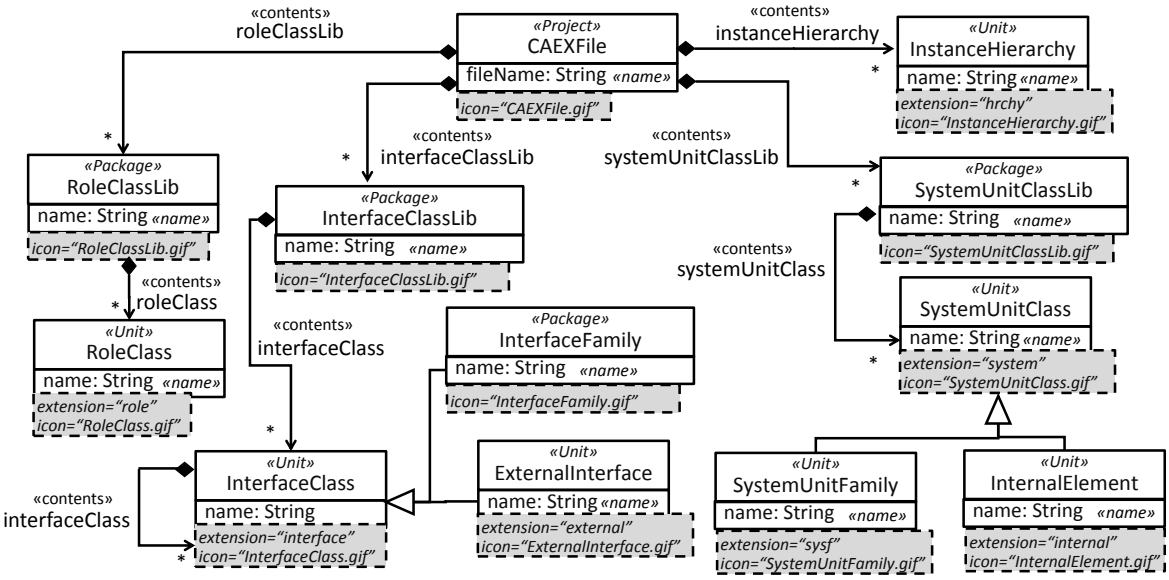
This section describes a gallery of scalable environments built using the approach proposed in this thesis. These environments were generated applying the modularity patterns. The dedicated wizard to develop graphical editors were used for the case studies presented in Section 7.5.2 and 7.5.3.

7.5.1 CAEX

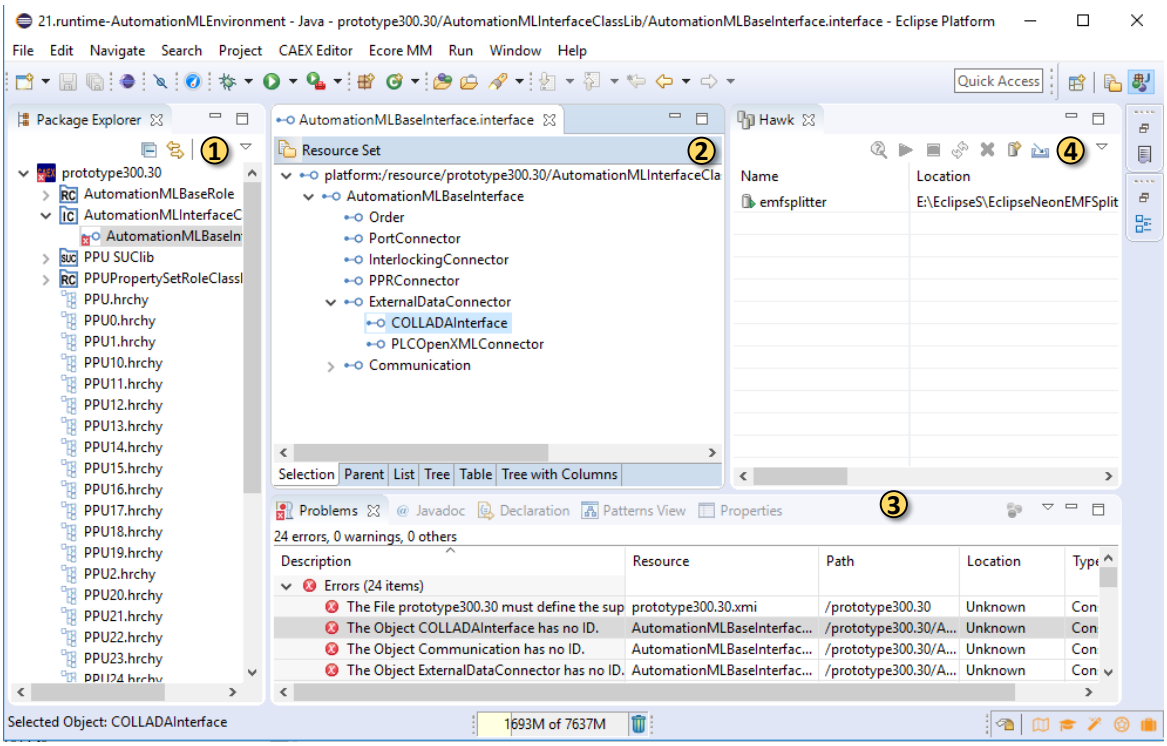
In this section, we compare an existing meta-model-based modelling environment developed by a third-party, with another one created by us using our approach. We use as a case study the environment developed for Computer Aided Engineering Exchange (CAEX), which is described in [87]. CAEX is a neutral data format, used as a standard by the International Electrotechnical Commission (IEC) to represent the hierarchical structure of production systems. Other IEC standards use CAEX. For example, AutomationML uses CAEX to model plant components, including devices and communication structures.

To create an environment for CAEX, we first applied the fragmentation pattern to its meta-model, using as guiding principles the editor and models in the Github repository https://github.com/amlModeling/. Figure 7.17a shows an excerpt of the meta-model annotated with the instantiated fragmentation pattern. A CAEXFile (project) stores the engineering data in instanceHierarchies and contains several libraries of elements (RoleClassLib, InterfaceClassLib and SystemUnitClassLib, which are packages). Altogether, we identified one project class, four package classes, and seven unit classes in the meta-model.

[10]http://mondo-project.org



(a) Excerpt of the CAEX meta-model annotated with the fragmentation strategy.



(b) Generated modelling environment.

Fig. 7.17 Building a modelling environment for CAEX.

Then, we converted the EVL [75] constraints available in the same Github repository into EOL scoped constraints in our meta-model. The Appendix B contains the list of scoped constraints, and Table 7.5 shows their characteristics. There is one constraint with scope sameProject, one with scopes samePkg and sameUnit at the same time, and seven with scope sameUnit. The average number of nodes in the constraints is 10.3, ranging from two to seventeen.

Constraint	Scope	Complexity (#nodes)
superiorStandardVersionIsMandatory	samePkg	3
CAEXObject	samePkg,sameUnit	2
inheritanceMustPointToSUC	sameUnit	5
strongConformanceSUC2IE	sameUnit	15
strongConformanceIE2SUC	sameUnit	15
noInheritanceForIEs	sameUnit	2
processContainsProcesses	sameUnit	17
resourceContainsResources	sameUnit	17
productContainsProducts	sameUnit	17

Table 7.5 Characteristics of scoped constraints used in the case study (CAEX).

Finally, we used EMF-Splitter to generate an environment for CAEX from the meta-model and instantiated patterns. Figure 7.18 shows the environment. The Package Explorer view contains a project that represents a fragmented model (label 1). One model fragment is being edited (label 2). The Problems view lists the violated constraints (label 3). The Hawk view shows a running instance of Hawk for EMF-Splitter (label 4).

As a next step, we compared the constraint validation performance in our environment and the original one. For this purpose, we generated models of size 210 162, 310 222 and 410 282 objects, and then measured the time to validate the defined constraints on them. To generate the models, we used a model generator available in the same Github repository, slightly modified to enable the generation or larger, more balanced models, compatible with CAEX version 3.0.

Figure 7.19 shows the time of validating the scoped constraints on the fragmented models, considering both full validation and incremental validation with Hawk upon an emulated model change. It also shows the validation time of the standard non-scoped constraints on the equivalent monolithic models. Using incremental validation with Hawk is the most efficient, as it improves in one third the validation time of standard constraints. The incremental validation on a model with 410 282 objects takes less than one second.

Overall, this case study shows that we could improve an existing industrial modelling environment with model fragmentation (strengthening our positive answer to **RQ1** in Section 7.1), and improved performance of constraint evaluation (strengthening our positive results regarding **RQ4** also in Section 7.4).

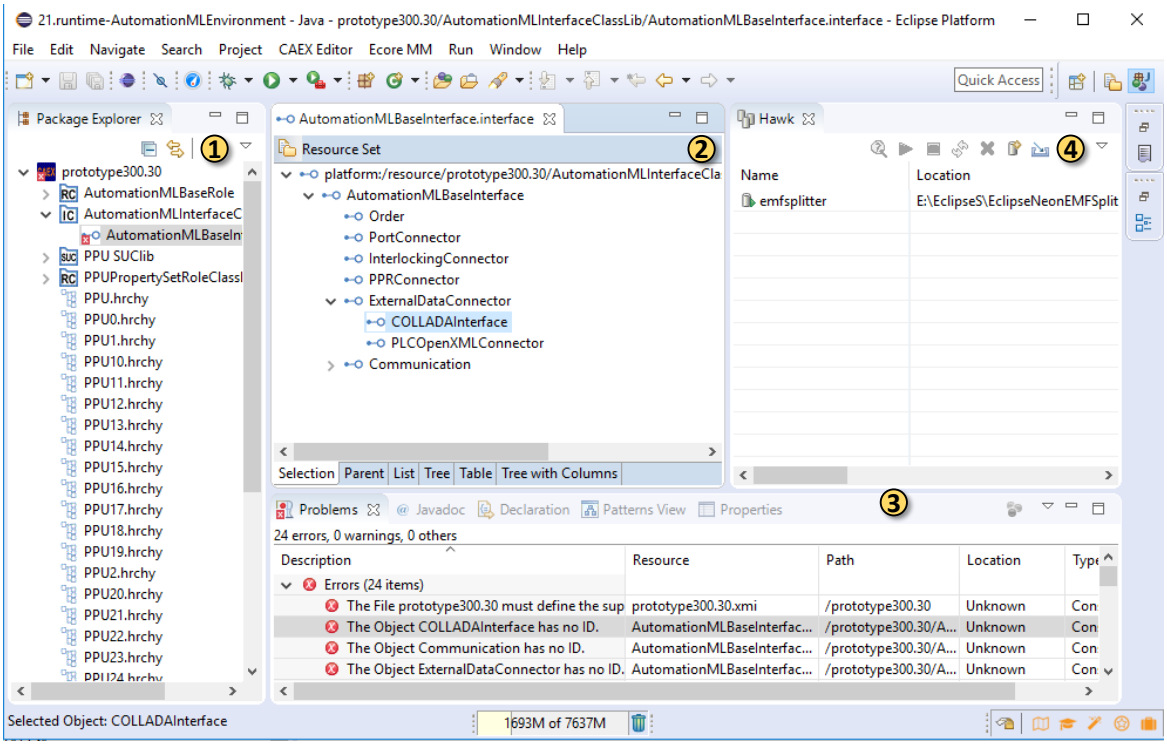


Fig. 7.18 Environment generated for CAEX.

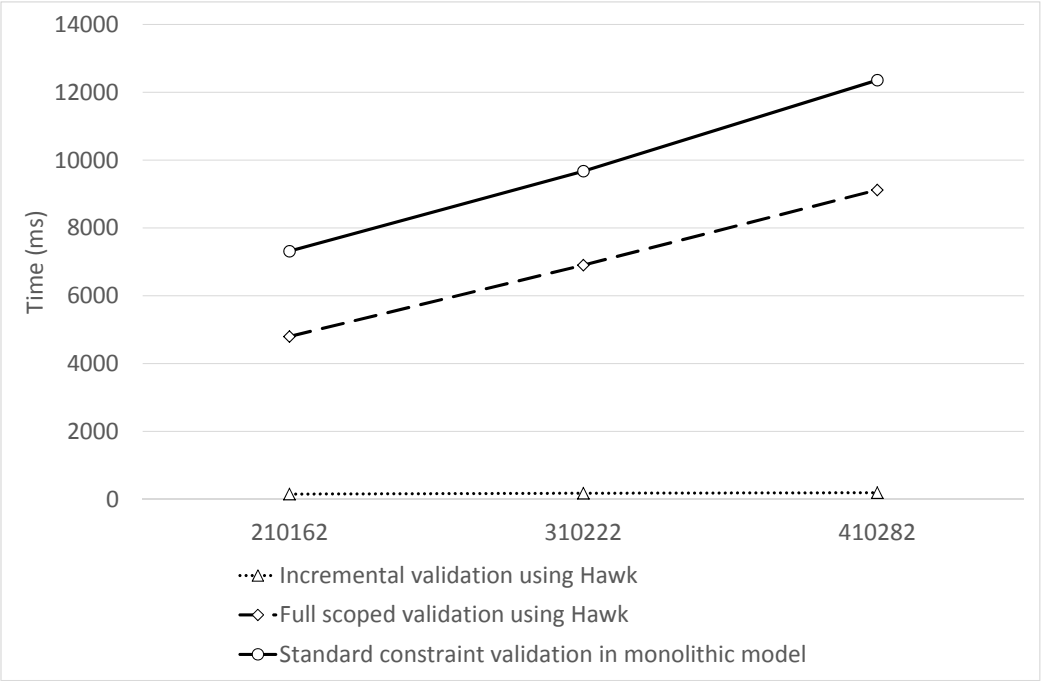


Fig. 7.19 Comparison of full validation, incremental validation and baseline for CAEX.

7.5.2 Henshin

This section describes the development of a modelling environment for the *Henshin* tool [4]. *Henshin* is a tool developed in EMF to create in-place transformations. This tool defines a transformation language with a set of rules based on the definition of attributed graph transformations [38]. We choose *Henshin* as a case study, with the aim of improving its graphical editor and provide scalability to the modelling environment. At the moment, the Henshin editor is based on GMF and their models are created in a monolithic way. For this purpose, we apply the fragmentation pattern to the *Henshin* meta-model, and also, we use the GraphicRepresentation meta-model to define their graphical concrete syntax.

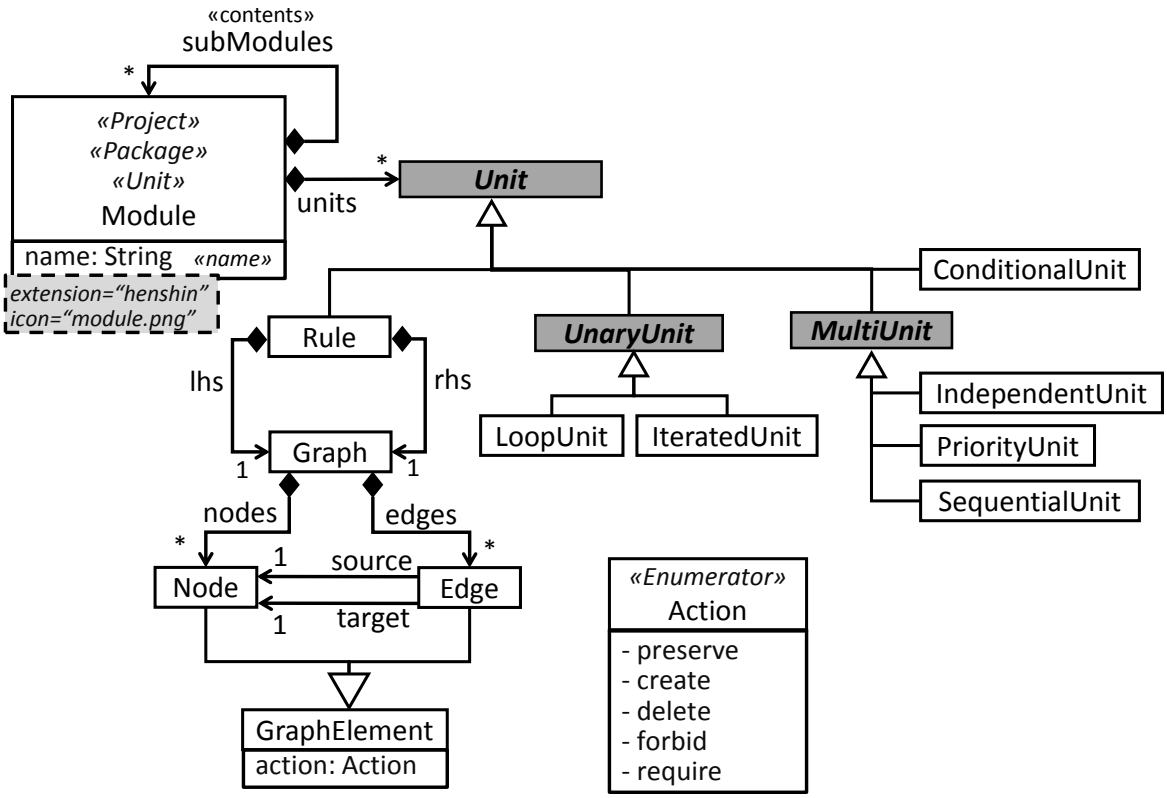


Fig. 7.20 Application of the fragmentation to the Henshin meta-model.

Figure 7.20 shows an excerpt of the abstract syntax of *Henshin*, with the fragmentation pattern application. Module is the root class, which plays the roles of Project, Package and Unit. Modules created as Project or Package are used to group other Module objects. If the module is created as a file, it will contains different types of units (e.g., LoopUnit or SequentialUnit). The rules for graph transformations are defined using the Rule class, which defines a left (lhs) and right (rhs) hand side graphs.

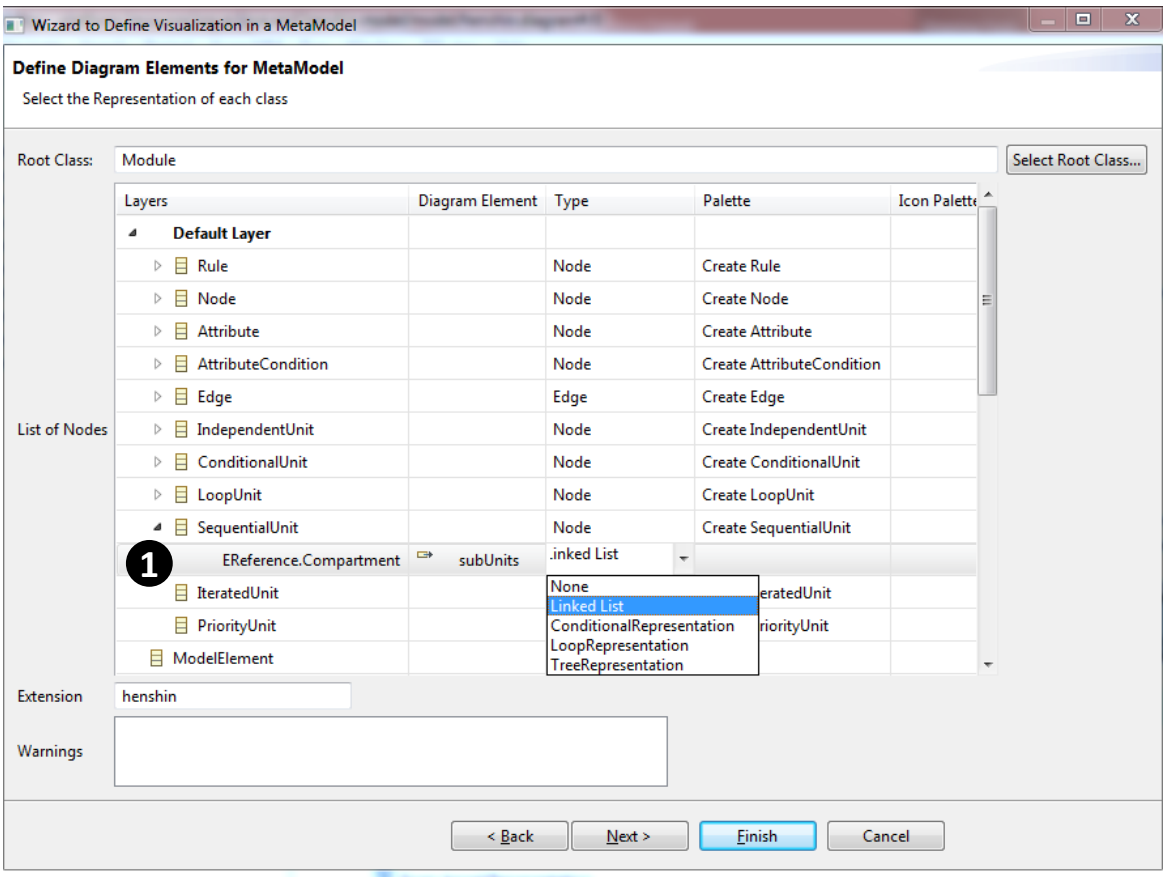


Fig. 7.21 Mapping Henshin meta-model elements to diagram elements.

As a further step, after the fragmentation pattern application, we use the dedicated wizard to define the graphical concrete syntax. Figure 7.21 shows the first wizard page. Label 1 shows the definition of a LinkedList style for the subUnits reference. This wizard supports the definition of all read-only collections representation style described in Section 5.2.2. For the representation of *Henshin* units, we use the read-only representation style for collections. Table 7.6 shows the mapping between the styles and *Henshin* units.

Henshin Unit type	Representation Style
IndependentUnit	Tree
ConditionalUnit	Conditional
PriorityUnit & SequentialUnit	LinkedList
IteratedUnit & LoopUnit	Loop

Table 7.6 Mapping between Henshin Unit types and representation styles

The second page of the wizard is shown in Figure 7.22. This wizard page is used to customize the style of the diagram elements. Label 1 shows how the diagram elements are automatically created to represent the sequence of the referenced units. This wizard page permits specifying the style configuration of these elements and variations thereof. For example, the initial and final node of the representation could be removed.

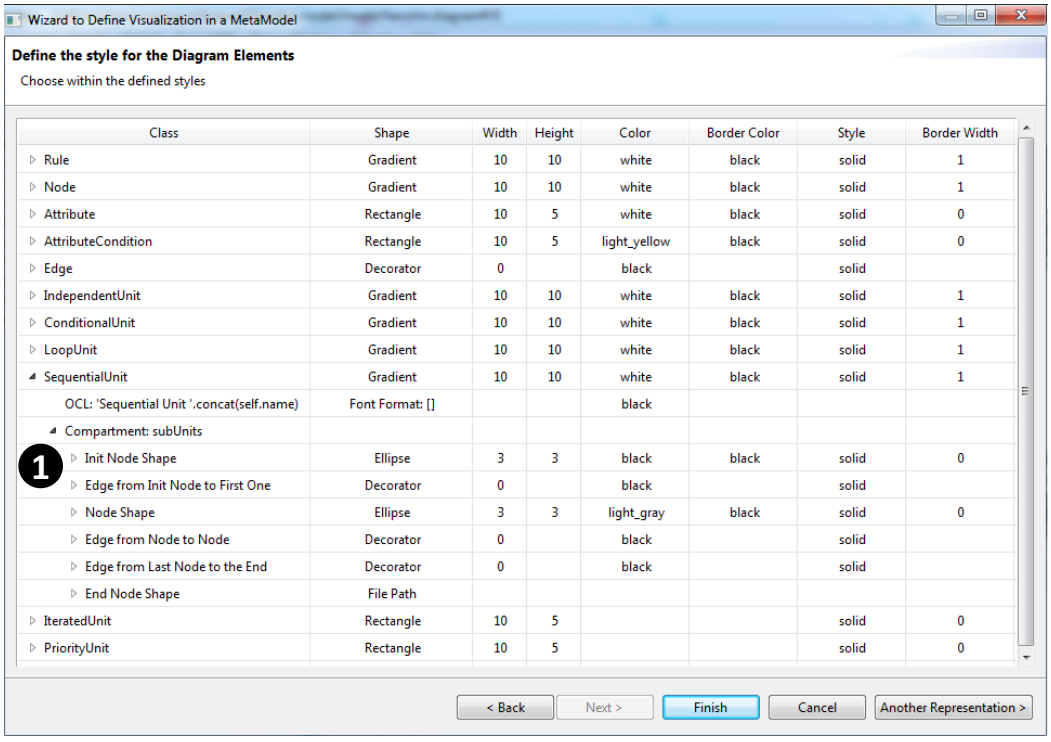


Fig. 7.22 Customization of the style diagram elements.

With the fragmentation pattern application and the concrete syntax definition over the *Henshin* meta-model, we can generate the scalable modelling environment shown in Figure 7.23. However, this final graphical editor could not be generate in its

entirety. For this reason, some functionalities were manually added to the generated *.odesign. Specifically, nodes with action equals to preserve are represented in the model by creating an element in the rhs and a mirror element in the lhs. Both elements must have the same information and for this case, the synchronization has been implemented programmatically. With respect to edges, we added also a programmatic functionality to infer the action type, which is usually equal to the action of the target element.

Since, the dedicated wizard does not support all Sirius features, in case of changing the generated *.odesign, modifying the concrete syntax using EMF-Stencil, it overwrite the manual changes. As future work, we will implement a mechanism to save automatically in a file the changes made manually to the generated *.odesign. By doing this, the *.odesign will be generated as usual by EMF-Stencil and after that, it is transform with saved changes.

For the example shown in Figure 7.23, we use the WT meta-model and implemented an a sample In-place model transformation for illustration. Label 1 shows the createInitialState rule, which checks that all state machines have a defined initial state, creating it if it does not exist. This rule finds a StateMachine object (node «preserve»:StateMachine), verifies that it does not contain an object of type InitialState (node «forbid»:InitialState) and if it does not contain it, then it creates it with the name 'Init' (node «create»:InitialState). Label 2 points to the other rule called createSimpleState, which creates an Edge object that connects the initial state element with a SimpleState object, in case it does not exist. The createSimpleState rule should find a InitialState object (node «preserve»:InitialState) within a StateMachine object (node «preserve»:StateMachine) that does not contain an Edge object from an InitialState (node «preserve»:InitialState) object to a SimpleState object (node «forbid»:SimpleState). If this condition is fulfilled, then the transformation creates a simple state object (node «create»:SimpleState) and an edge(node «create»CreateEdge:Edge) from the initial state object to the created simple state.

The transformation includes a LoopUnit to apply the rules to all model objects. Figure 7.23 (label 3) shows the defined loops for both rules above-mentioned. Finally, we created a SequentialUnit (label 4) to apply first the rules that create the missing initial objects, and after that, create the corresponding edges.

In this case study, we combine the fragmentation pattern and the graphical representation definition to provide to *Henshin* a scalable environment with a new editor developed in Sirius. Using this approach, the creation of interconnected modules in different files is allowed, which is not possible with the current environment implemented in GMF.

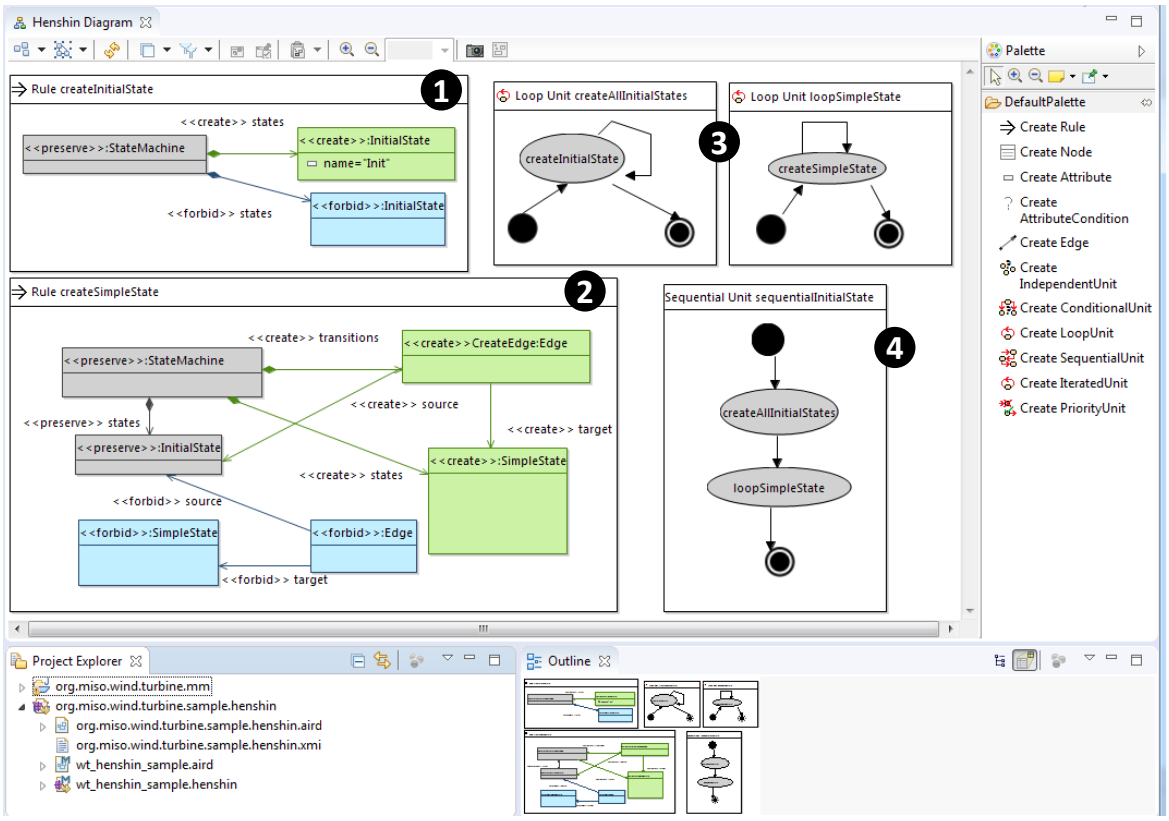


Fig. 7.23 Generated *Henshin* modelling environment.

7.5.3 Cloud Robotics System

This section illustrates the combined use of EMF-Splitter and DSL-tao through a case study based on the DSML presented in [134]. The DSML, called CRALA, is a language for architecture-centric cloud robotics systems. Figure 7.24 shows DSL-tao with the CRALA meta-model. The class CloudSystem is the root containing directly the Configuration and the ArchitectureSpecification classes. On one hand, the Configuration element represents the VirtualMachines with the connections of components that may be a WebService or a ComponentClass type of object. On the other hand, the Architecture-Specification is used for defining the robots, their sensors and the connection between them. For this case study, we have applied the fragmentation patter contributed by EMF-Splitter and, also we employs the Graphical Representation implemented in EMF-Stencil. For these, the combination of both plug-ins provides the pattern structure, the application wizard and the code generation service.

We applied a fragmentation strategy to the language, using the wizard shown in Figure 7.25. The class CloudSubsystem is tagged as a Project and can contain a unit

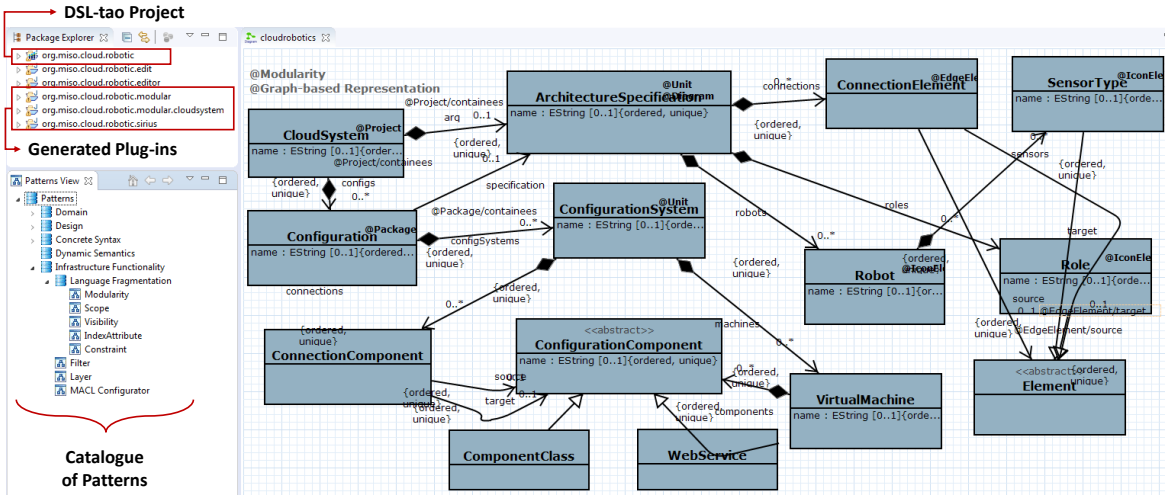


Fig. 7.24 CRALA meta-model in the DSL-tao environment.

of type ArchitectureSpecification. This specification is possible due to the containment relationship that exists between these two classes. The class Configuration is tagged as Package and can contain several units of type ConfigurationSystem. Figure 7.24 shows these elements mapped to Project, Package and Unit.

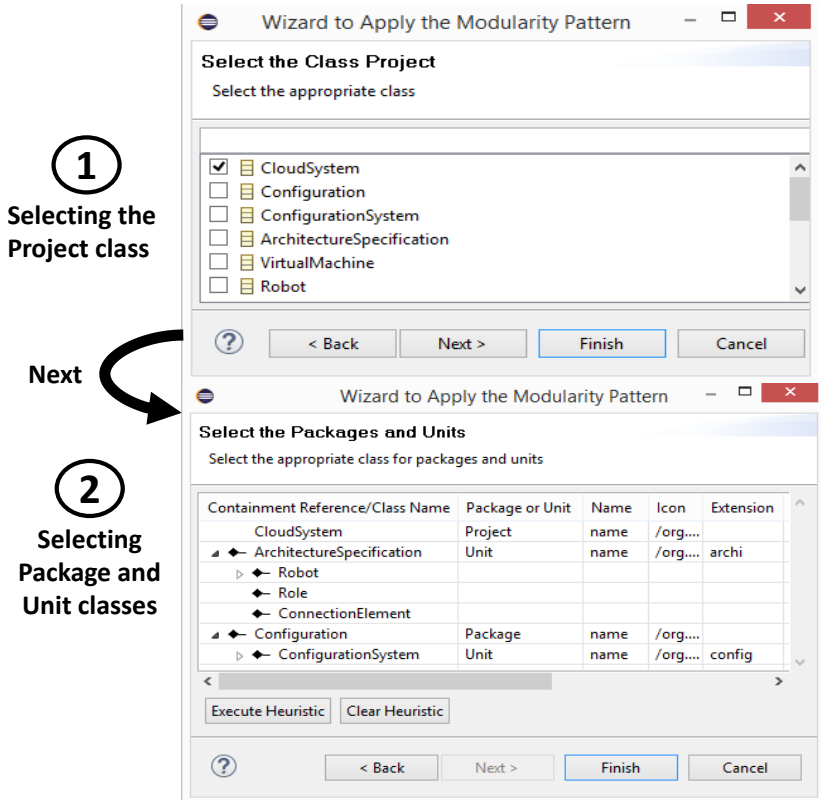


Fig. 7.25 Fragmentation Pattern wizard contributed by EMF-Splitter.

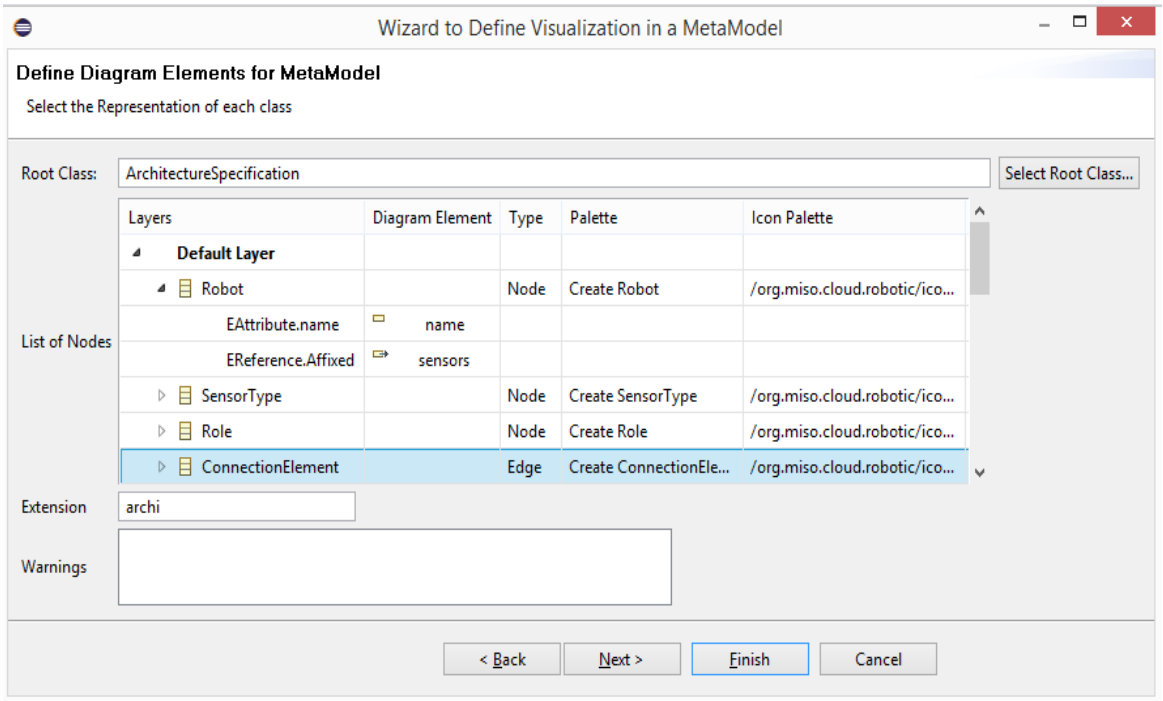


Fig. 7.26 Wizard page 1 contributed by EMF-Stencil.

The instantiation of the graphical representation is made through a dedicated wizard. Figure 7.26 shows the wizard for specifying the graphical syntax for unit ArchitectureSpecification. The first page of the wizard allows selecting the elements to be visualized, and heuristically proposes their shape (node/edge). The second page, shown in Figure 7.27 permits customizing the appearance of elements. For this example, we retrieve the web images using the Google JSON API, and this figure shows the ones obtained when using: 'Robot' as parameter, selecting a maximum of 10 images and the retrieval of the clip-art ones.

Once the meta-model is annotated with patterns, the code generation services provided by EMF-Splitter and EMF-Stencil can be invoked to generate an environment where the models are fragmented and visualized according to the defined patterns. Figure 7.28 shows the Sirius-based generated environment.

7.6 Applications

The aim of this section is to demonstrate the usefulness of our approach within others applications. Section 7.6.1 describes the integration of EMF-Splitter with a tool that supports visualization mechanisms for the fragmented models. Next section, shows a third-party tool using the functionality of EMF-Stencil to generate modelling

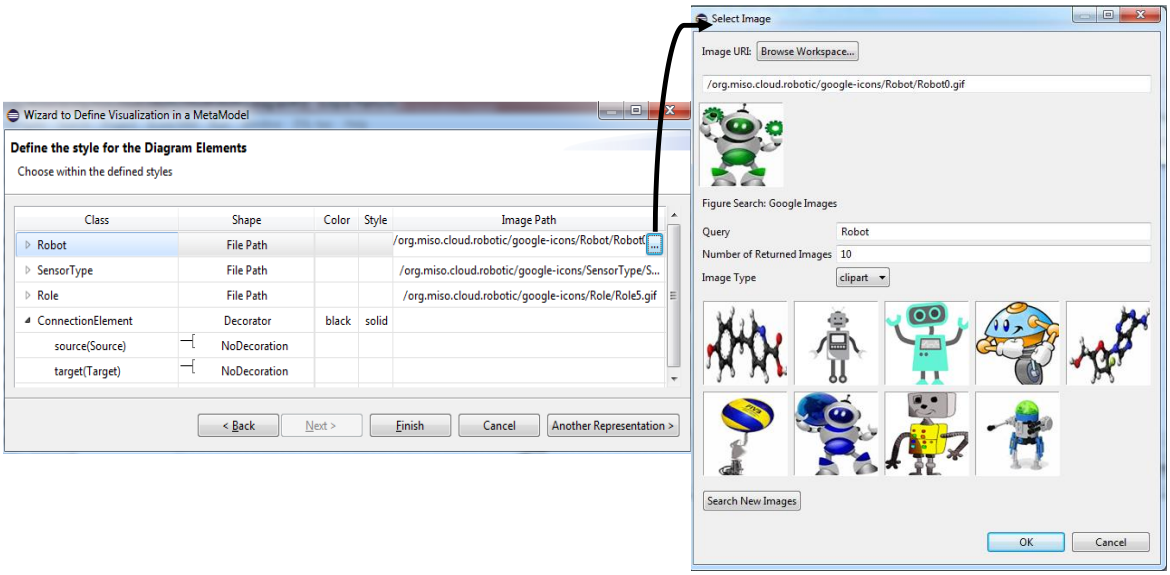


Fig. 7.27 Wizard page 2 contributed by EMF-Stencil.

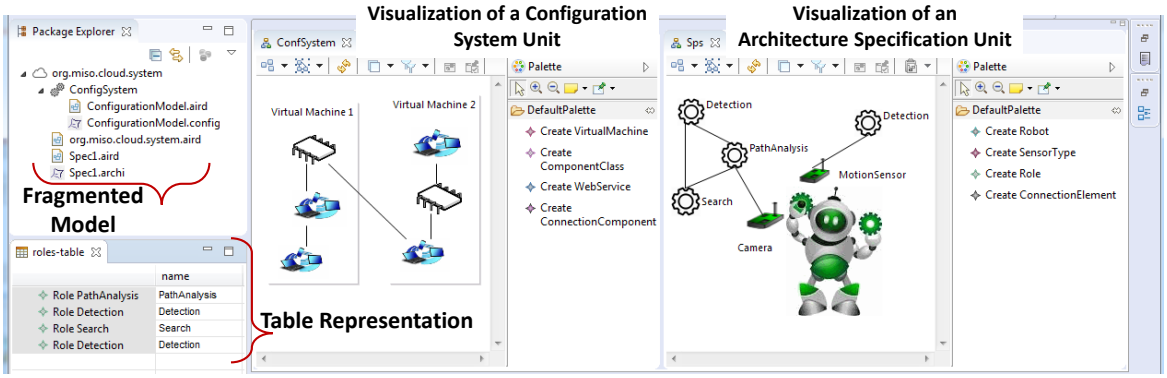


Fig. 7.28 Graphical modelling environment generated by EMF-Splitter and EMF-Stencil.

environments. Finally, Section 7.6.3 shows the integration of EMF-Stencil with a mobile modelling tool.

7.6.1 Scalable Model Exploration

When working with models, it is very useful to explore them to obtain some insight using our intuition, to analyse their different parts, or to find unusual or interesting features. However, big models are impossible to be understandably represented in a computer screen at once. When using the EMF framework, it is common that models lack a dedicated graphical editor providing visualisation and exploration services. One reason is that frequently, only a meta-model is developed, but no further effort is spent

to create a graphical concrete syntax. Hence, EMF models are frequently visualised using the tree editor, which difficults their comprehensibility as this editor does not provide facilities to visualise, search and navigate in a graph-based way. For example, it may display two related elements at very distant places, just because they belong to two different container objects. Instead, showing those elements closer, using a graph-based representation, may be more intuitive in some cases. Even if a specialised editor exists, these editors usually do not offer support for scalable exploration.

As a generic solution to this issue, SAMPLER (ScAlable Model exPLorER) [63] is a collection of Eclipse plug-ins that implement a graph-based exploration for arbitrary models. The framework is targeted to large models, and its key idea is not showing all model elements at once, but displaying only a region of interest, and abstract or filter the other elements. Then, different navigation strategies can be used to walk through the model.

While EMF-Splitter and SAMPLER can be run separately, we integrated both tools in a coordinated way. In particular, SAMPLER provides the possibility of opening and visualising fragmented models, exploring the fragmented models in stages, and using the file system structure created by EMF-Splitter (as explained in Section 6.2). We have proposed the combination of model fragmentation and model visualisation techniques to explore large models. Model fragmentation is performed by applying fragmentation strategies at the meta-model level. Model exploration is done by applying different abstraction strategies to the model, and with the availability of model exploration techniques.

As illustration, we visualize Knowledge Discovery Meta-model (KDM) models. KDM is a standard specification of the OMG that is widely used in software modernization projects [21]. KDM is used to represent existing software artefacts (legacy code) in different languages (COBOL, C, Fortran, Java) in a platform independent way. KDM models can become extremely large, as normally the whole code and additional artefacts of a software application are included in a monolithic KDM model. Therefore, we define a fragmentation strategy over the meta-model, so that KDM models can be split. Figure 7.30 shows the fragmentation strategy we have designed for KDM [100].

Figure 7.30 shows a snapshot of the exploration of the KDM fragmented model named org.eclipse.jdt.apt.core.reverse.engineering. This project was created from the KDM Code Model of the project org.eclipse.jdt.apt.core, which is one of the plug-ins that are part of Eclipse Java Development Tools Core (JDT Core). The left of the figure shows the project explorer, containing the fragmented model across the file system, as created by EMF-Splitter. The right part (labels 1 and 2) are model browsers contributed by

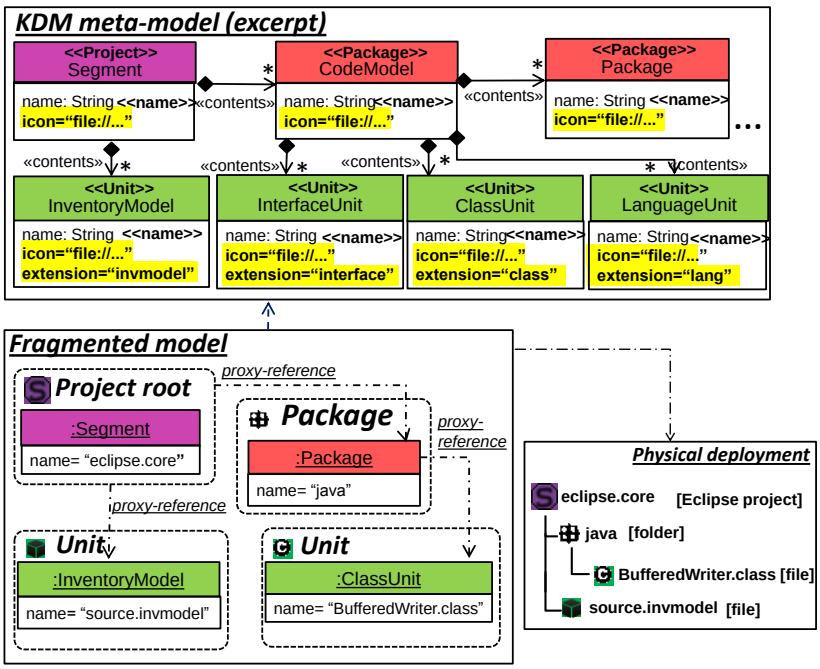


Fig. 7.29 Instantiation of the pattern and application to the KDM meta-model (top). A structured model and its physical deployment (bottom).

SAMPLER. The tab with label 1 visualises the root node of the model (node with label 1, coloured in red) and depicts a part of its structure. For example, node 6 indicates the existence of a file with name references.invmodel, node with label 3 (name external) and 4 (name org.eclipse.jdt.apt.core) indicate two folders. Node with label 4 is expanded, showing its content. Please note that, because SAMPLER reads the content of the file system, it shows in the form of nodes some hidden files created by EMF-Splitter (node with labels 5 and 10, with name ending in xmi). These nodes can be omitted using a filter created for this purpose.

The tab on the right (with label 2) shows the expansion of the node org.eclipse.jdt.apt.core (a folder in the left tab). The expanded model is shown aggregated using an abstraction called "leaves-local", which shows a maximum of five nodes and aggregates together several nodes within two abstract nodes. Overall, compared to Figure 7.8, SAMPLER provides an alternative graph-based visualisation to the EMF tree editor, as well as abstraction and exploration mechanisms.

Section 7.3.1 described the gain in time when a fragmented model is opened using the EMF reflective tree editor. Likewise, we evaluate the combined use of EMF-Splitter and SAMPLER using the JDTAST models. For this kind of exploration, it is not necessary to load the entire model. Instead, only the resources associated with the root class need to be loaded, and then, the nodes corresponding to folders and files can

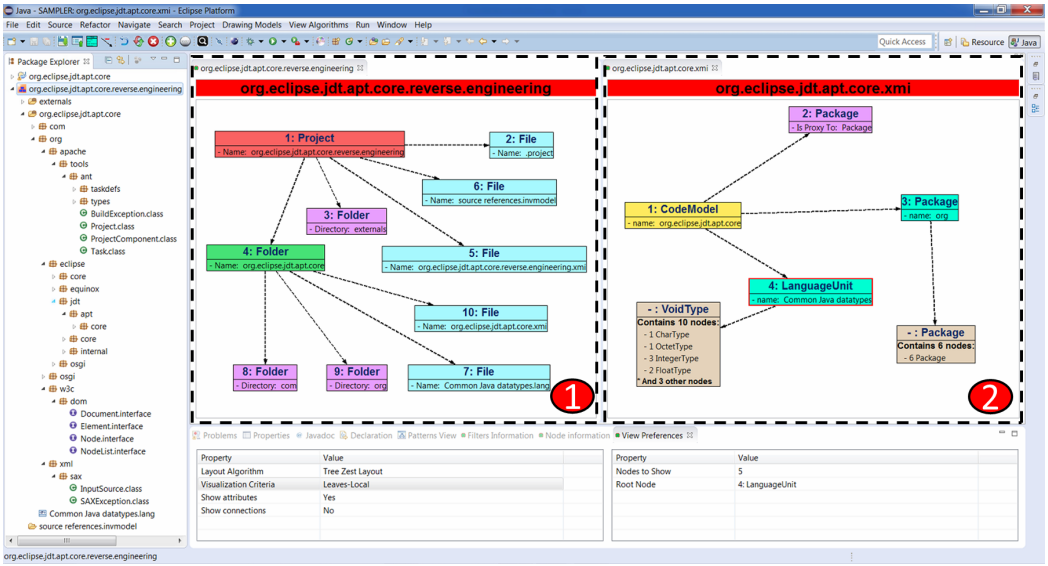


Fig. 7.30 Visualisation of a KDM model using SAMPLER.

be expanded on demand, providing a drill down hierarchical visualisation. Therefore, we calculated the minimum number of objects that need to be loaded to display each object in the model. This number of objects is the sum of all objects in all fragments existing in the path from the given object to the root.

Table 7.7 shows the results of the evaluation. The columns depict the average, minimum and maximum number of loaded objects. Generally, the total number of elements to load is much lower than the actual model size. For example, for *set4* with almost 4 000 000 model elements, the number of elements to load in the worst case is 54 160, which amounts to a reduction of 98.9% of the objects that need to be held in memory. This great reduction shows the power of fragmentation for scalability.

Model	Average	Min	Max	Reduction w.r.t. tree editor (worst case)
set0	502.26	250	1 613	97.74%
set1	719.26	34	4 800	97.65%
set2	11 590.62	61	54 141	97.4%
set3	7 859.59	83	54 163	98.82%
set4	7 550.56	89	54 169	98.9%

Table 7.7 Necessary number of loaded objects to explore the fragmented models.

The hierarchical exploration of SAMPLER uses the fragmentation produced by EMF-Splitter, so that the content of nodes representing packages (folders) can be displayed by double-clicking on it. For this purpose, SAMPLER needs to read such information from the file system. Therefore, we made an assessment consisting in computing for each folder the amount of resources (files and folders) that it contains

directly or indirectly. Table 7.8 shows the results, with the average and maximum number of resources to show, which allows the exploration of the fragmented model. Again, the low numbers even for the biggest model suggests a high scalability of the combined model exploration using SAMPLER and EMF-Splitter.

	Directly Contained Resources		Directly/Indirectly Contained Resources	
Model	**Average Amount of Resources**	**Max**	**Average Amount of Resources**	**Max**
set0	18.27	231	53.61	1 882
set1	28.98	270	85.88	6 463
set2	21.37	270	68.30	6 347
set3	12.20	157	46.86	4 858
set4	11.97	157	45.99	5 531

Table 7.8 Resources contained by the models at each hierarchical stage.

Overall, fragmentation according to a strategy may be costly (if many files need to be produced), but it is a one-time operation. In this case, fragments become of manageable size, and then can be visually explored. The experiments show big gains obtained by fragmentation for the visualisation of large models (a speed up of 55 ×in terms of time to load and a reduction of up to 98% percent in size) with respect to using the standard EMF tree editor over monolithic models.

7.6.2 Creating Graphical Environments by Example

The construction of DSMLs is costly and highly technical. This relegates domain experts to a rather passive role in their development and hinders a wider adoption of DSMLs. In order to fill this gap, the thesis of Jesús J. López-Fernández [84] proposed an example-based approach to DSML development, which permits the automatic generation of graphical modelling environments from drawings made with informal diagramming tools. This includes the automatic derivation of the abstract and concrete syntax of the targeted DSML. The architecture of this solution encompasses drawing tools like yED or Dia to draw the graphical fragments that provide usage examples of the DSML, and two Eclipse plug-ins: metaBup [83] and EMF-Stencil (Section 6.3).

The metaBUP tool extracts the graphical information encoded in the fragments provided by the domain experts using diagramming tools like yED. Then, metaBUP uses the extracted graphical information to derive a concrete syntax close to the domain expert's conception. For this purpose, metaBUP proceeds in two steps. First, it converts the information gathered from the fragments into a technology-neutral representation (based on EMF-Stencil's meta-model), and then, this representation is translated into a technology-specific editor specification.

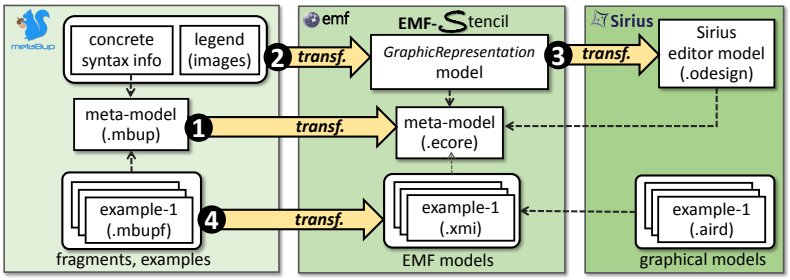


Fig. 7.31 Generating a graphical modelling editor from examples using metaBup and EMF-Stencil.

Figure 7.31 outlines the metaBUP process to create modelling environments, where three transformations take place: one generates the meta-model with the abstract syntax of the DSML (label 1); another takes care of the concrete syntax by constructing a technology-neutral model with the graphical information (label 2) from which a modelling environment for a specific technology is synthesized (label 3); the last transformation converts the provided fragments into models conformant to the derived meta-model (label 4). The target meta-model in the second transformation is the one described in Section 5.2. From the model with the graphical representation that metaBUP generates, EMF-Stencil synthesizes automatically a modelling environment.

7.6.3 Enabling Mobile Domain-Specific Modelling

Modelling environments have been traditionally supported by desktop computers. However, there are scenarios where devices like smartphones or tablets might be better suited to perform a certain task. In [127] a working architecture and a prototype tool, called *DSL-comet* were proposed, which enable collaborative mobile modelling and integrate seamlessly desktop and mobile graphical modelling environments.

One of the scenarios described in that work combined a desktop client with *DSL-comet*. The desktop client defines the abstract syntax of the DSML using DSL-tao (Section 6.1). As a running example, the authors defined a meta-model of the home networking domain, an excerpt of which is shown in the back of Figure 7.32.

The concrete syntax of the meta-model can be defined using EMF-Stencil (see window at the front of Figure 7.32). The concrete syntax specification is described through a model that annotates the meta-model. Both the meta-model and the concrete syntax description can be uploaded to a server, from where they can be downloaded by mobile and desktop clients to produce a modelling environment.

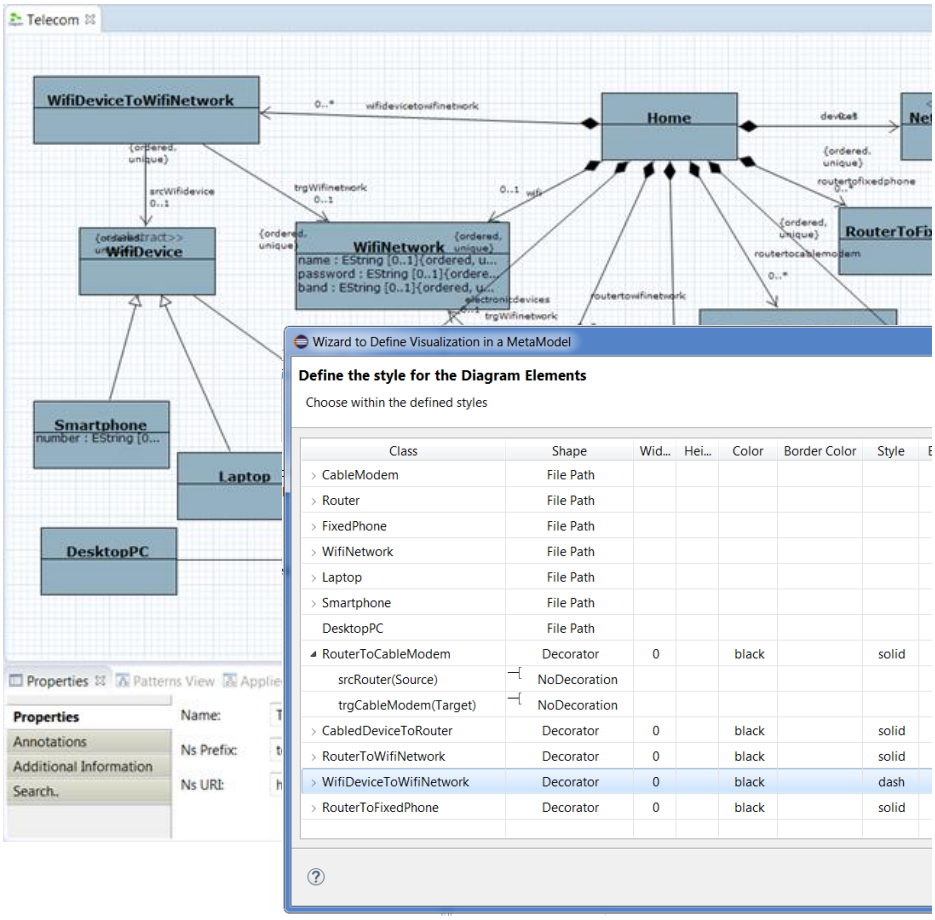


Fig. 7.32 Meta-model for the home networking domain (back). Wizard to define the graphical representation (front).

The desktop client is based on EMF-Stencil (Section 6.3) which generates a desktop environment in Sirius. Figure 7.33 shows a screenshot of the resulting editor for the home networking meta-model. The view "Mobile Server View" permits downloading models and DSL modelling environments from the server, like the one shown in the figure.

7.7 Summary and Conclusions

In this chapter, we have evaluated different aspects of our proposal. The analysis of meta-model repositories has confirmed the applicability of our fragmentation pattern. Our fragmentation pattern was applied in contexts, showing good performance for large models. We showed that the validation of scoped constraints is faster than the evaluation of standard constraints, especially when only the affected objects are

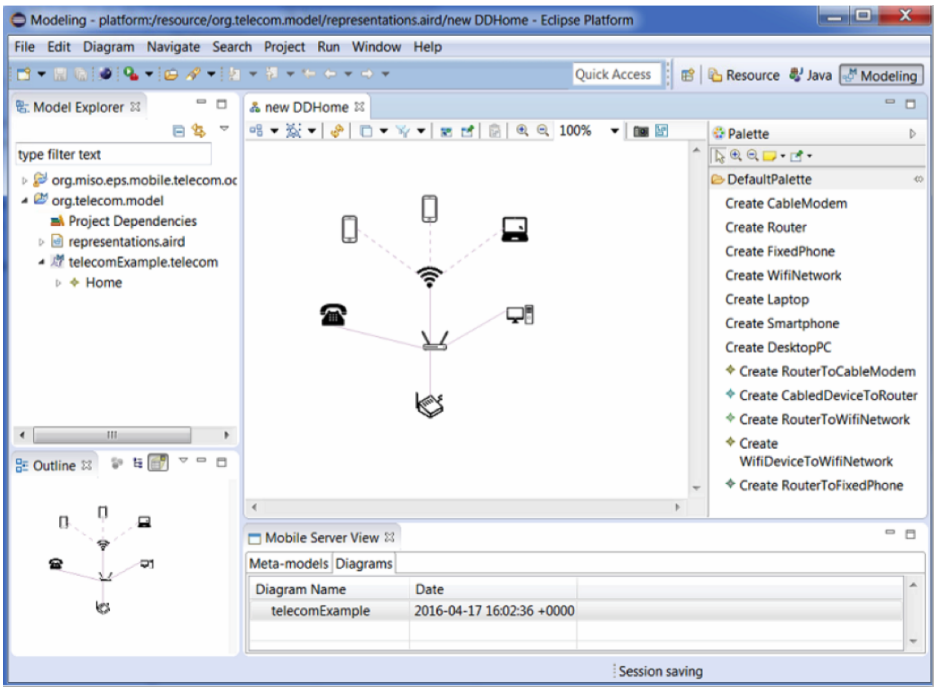


Fig. 7.33 Screenshot of the Sirius desktop client.

re-evaluated, and when combined with attribute indexes. In addition, we used our approach to generate modelling environments for CAEX, Henshin and CRALA. The first case study described how we improve an existing industrial modelling environment with our approach, particularly in terms of constraint evaluation, in which we obtained good results. The second case study illustrated the read-only collections representation styles. The final case study showed the functionalities of EMF-Stencil to generate a default graphical syntax, using services for image search. Then, such syntax can be modified according to the DSML requirements. Finally, we showed further utility of the approach as a complement to other tools, like SAMPLER, DSL-Comet and metaBup. This integration proved the versatility of our approach in terms of visualization and generation of modelling environments.

Chapter 8

Conclusions and Future Work

In this thesis, we have proposed solutions to some challenges in the construction of environments for DSMLs when adopting the MDE paradigm. This chapter, in its first section, discusses the conclusions of this work summarizing the given solutions for each detected challenge. The second section analyses the future research lines in order to continue improving and applying our approach.

8.1 Conclusions

As fundamental contributions, in this thesis we propose a systematic approach to develop scalable modelling environments for DSMLs. In Section 2.1.1, we identified some drawbacks in the classical application of MDE that hampers its adoption in some contexts. The models created using the MDE paradigm are usually monolithic, and when they become large they are difficult to handle by tools. For this reason, our goal was to provide developers with tools that help in defining ways to structure models, facilitating distributed development through division into fragments, which improves comprehension and enhances automated processing. In this thesis, we proposed a fragmentation pattern which allows splitting a model into files and folders.

We introduced four more patterns (reference scoping, object visibility, field indexing, and scoped validation) in order to enforce the separation of concerns, improve usability, manage model complexity through information hiding, and speed up the evaluation of constraints and model queries. These five patterns (including the fragmentation one) contribute with modularity services to the developed modelling environments. In this sense, we propose a systematic approach based in this catalogue, which can be used in the creation of new environments or the migration of existing ones.

Another issue is the high effort required for the development of graphical editors. To ease their creation, we implemented a set of heuristics to help in the automatic assignment of a graphical concrete syntax to meta-models. In this way, we speed up the generation of graphical editors, because by applying the heuristics we can automatically generate the first version of the graphical editor. Using this technique, we reduce the need for specialized knowledge by developers. However, an empirical user study to confirm this intuition will be the subject of future work.

These contributions have been realised in two Eclipse plug-ins, called EMF-Splitter and EMF-Stencil to support the approach and facilitate the process. We implemented the catalogue of patterns in EMF-Splitter. This tool provides dedicated wizards for each pattern, to assist developers in their instantiation at the meta-model level. Using the information provided by the instantiated patterns, this tool generates Eclipse plug-ins that realise a scalable modelling environment. EMF-Stencil provides the wizards for graphical and tabular concrete syntax specification. Specifically, the wizard for graphical syntax definition has implemented heuristics and the read-only collections representation style.

The tools have been evaluated under different angles. Specifically, we evaluated the applicability of the fragmentation pattern by analysing meta-model repositories built by third parties. We noted in this experiment that our fragmentation pattern can be applicable in practice, because containment references in meta-models are frequent. In our evaluation of model fragmentation, we found that for large models it is generally useful to fragment them prior to their visualization or indexing with third party tools. Regarding the visualization, either with *Gephi* or with the EMF tree editor, the fragmentation provides better handling in large models and in the case of the CDO experiment it is worthwhile to fragment first, especially big models, in order to perform the indexing. The fragmentation is a time-consuming pre-processing technique, but it is done only once. The gain in time is obtained in the future, since the sub models are smaller and can be processed separately. In this way, the approach provides good scalability to models and it is also compatible with others tools.

We introduced scoped validation in Section 4.3.5, providing an approach to execute the constraints in a subset of model objects instead of the complete model. The results of our evaluation demonstrated the efficiency gain with respect to the execution of standard non-scoped constraints. Additionally, we evaluated this approach in combination with Hawk, resulting in another improvement in the execution time of constraints. Moreover, we performed a comparison with an existing modelling environment for CAEX, showing that the approach could improve existing industrial

modelling environments. Altogether, the application of model fragmentation could be applicable and useful in the industrial context.

With respect of graphical editors, we choose existing modelling tools like Henshin and CRALA, migrating their environments with the combination of EMF-Splitter and EMF-Stencil. The new developed environments provide the creation of modularized models, in contrast with the currently developed environments. In this way, we provided a scalable graphical environment to visualize the different parts of a model separately.

Moreover, the developed tools were integrated with exiting ones, like SAMPLER, DSL-comet and metaBup. This demonstrates the versatility and potential of the developed Eclipse plug-ins.

Altogether, the presented approach proposed a systematic way of producing scalable modelling environments for DSML. This approach is based on the use of a modularity patterns catalogue which showed their effectiveness in different contexts.

8.2 Future Work

This thesis has opened a number of research lines, which we plan to investigate in the future. At the technical level, we are currently developing an API to facilitate the programmatic use of EMF-Splitter. Moreover, we plan to improve the fragmentation service with the possibility to fragment models across non-containment references, e.g., based on strategies like those described in [120].

Another line of future research is developing heuristics to recommend a scope and attribute indices for a given constraint, based on the static analysis of the constraint. In particular, we plan to recommend a scope for an OCL constraint based on the scope and visibility of the objects and references appearing in it. The idea is traversing the constraint expression and, for each reference access, look at the applied patterns to extract the scope assigned to the reference, and the visibility assigned to the reference type, annotating the reference with the narrowest of both. When the scope of all reference accesses has been computed in this way, the widest one would be heuristically suggested as the scope of the constraint. We will investigate methods to recommend optimal fragmentation granularities given a set of constraints. Finally, fragmented models are more amenable to collaborative use – for example via version control systems – than monolithic models, as they tend to produce fewer conflicts. Hence, we plan to integrate our approach with version and access control systems for collaborative modelling [31].

Regarding patterns, one direction of future work is supporting the definition of model management operations (e.g., transformations, code generators, constraints) defined over patterns. Such operations would be transferred to the domain metamodel when the binding is performed. This would be a way to reuse and combine the operations contributed to the different patterns used in [20].

EMF-Stencil is able to generate graphical editors providing an automatically inferred model representation. In fact, this tool can generate a set of representations for the same abstract syntax. Based on this functionality, we would like to evaluate with users the quality of these representations.

Regarding the engineering to implement modelling environments, we are planning to apply this approach to other scenarios. We will find other contexts, especially industrial ones, to prove even more the applicability of our approach. By doing this, we will compare existing environments with the ones generated using our plug-ins, in terms of scalability and the provided functionalities.

Chapter 9

Conclusiones y Trabajo Futuro

En esta tesis, se han propuesto soluciones a algunos desafíos en la construcción de entornos para DSMLs, cuando se quiere adoptar el paradigma MDE. En este capítulo, en la primera sección, se analizan las conclusiones de este trabajo, el cual resume las soluciones para cada desafío detectado. La segunda sección analiza las líneas de investigación futuras para continuar en la mejora y aplicación de nuestro enfoque.

9.1 Conclusiones

Como aportaciones fundamentales, en esta tesis proponemos un enfoque sistemático para desarrollar entornos de modelado escalable para DSMLs. En la Sección 2.1.1, identificamos algunas desventajas en la aplicación clásica de MDE que dificulta su adopción en algunos contextos. Los modelos creados utilizando el paradigma MDE son generalmente monolíticos, y cuando estos alcanzan un gran tamaño, se hace difícil su manejo por las herramientas. Por este motivo, nuestro objetivo es proporcionar a los desarrolladores herramientas que ayuden a definir modelos de manera estructurada, facilitando el desarrollo distribuido a través de la división en fragmentos, que mejore la comprensión y el procesamiento automatizado. En esta tesis, hemos propuesto un patrón de fragmentación que permite dividir un modelo en ficheros y carpetas.

Además del patrón de fragmentación, hemos incorporado cuatro patrones más (alcance de la referencia, visibilidad de objetos, indexación de campos, y validación jerárquica) para imponer un bajo acomplamiento, mejorar la usabilidad, gestionar la complejidad del modelo mediante la ocultación de información y acelerar la evaluación de restricciones y consultas a modelos. Estos cinco patrones (incluido el de fragmentación) contribuyen con servicios de modularidad a los entornos de modelado desarrollados.

En este sentido, proponemos un enfoque sistemático basado en este catálogo, que se puede utilizar en la creación de nuevos entornos o en la migración de los existentes.

Otro problema abordado, fue el gran esfuerzo que se requiere para el desarrollo de editores gráficos. Para facilitar su creación, implementamos un conjunto de heurísticas para ayudar en la asignación automática de una sintaxis gráfica concreta a los meta-modelos. De esta manera, aceleramos la generación de editores gráficos, ya que, aplicando las heurísticas podemos generar automáticamente la primera versión del editor gráfico. Usando este enfoque, reducimos la necesidad de conocimiento especializado por parte de los desarrolladores. Sin embargo, un estudio empírico de usuarios para confirmar esta intuición será objeto de un trabajo futuro.

Estas contribuciones se han implementado en dos plug-ins de Eclipse, llamados EMF-Splitter y EMF-Stencil, que dan soporte al enfoque y facilitan su proceso. Implementamos el catálogo de patrones en EMF-Splitter. Esta herramienta proporciona asistentes dedicados para la instanciación de cada patrón, para ayudar a los desarrolladores en su definición a nivel de metamodelo. Utilizando la información proporcionada por los patrones instanciados, esta herramienta genera plug-ins de Eclipse, que conforman un entorno de modelado escalable. EMF-Stencil proporciona los asistentes para la especificación de la sintaxis gráfica concreta y tabular. Específicamente, el asistente para la definición de la sintaxis gráfica ha implementado las heurísticas y los estilos de representación de colecciones de solo lectura.

Estas herramientas han sido evaluadas teniendo en cuenta diferentes aspectos. Específicamente, evaluamos la aplicabilidad del patrón de fragmentación mediante el análisis de repositorios de meta-modelos construidos por terceros. Notamos en este experimento, que nuestro patrón de fragmentación puede ser aplicable en la práctica, porque las referencias de contención en los metamodelos son frecuentes. En nuestra evaluación de la fragmentación de modelos, encontramos que para modelos grandes, generalmente es útil fragmentarlos antes de su visualización o indexación con herramientas de terceros. Con respecto a la visualización, ya sea con *Gephi* o con el editor de EMF en forma de árbol, la fragmentación proporciona un mejor manejo de modelos grandes y, en el caso del experimento con CDO, vale la pena fragmentar primero, especialmente modelos grandes, para realizar la indexación. La fragmentación es una técnica de pre-procesamiento que consume mucho tiempo, pero se realiza sólo una vez. La ganancia de tiempo se obtiene en el futuro, ya que los submodelos son más pequeños y se pueden procesar por separado. De esta manera, el enfoque proporciona una buena escalabilidad a los modelos, y también es compatible con otras herramientas.

Presentamos en la Sección 4.3.5 un enfoque para ejecutar las restricciones en un subconjunto de objetos del modelo en vez del modelo completo. Los resultados de nuestra evaluación demostraron la ganancia en tiempo con respecto a la ejecución de restricciones de manera estándar. Además, evaluamos este enfoque en combinación con Hawk, lo que resultó en otra mejora en el tiempo de ejecución de las restricciones. Adicionalmente, realizamos una comparación con un entorno de modelado existente para CAEX, que muestra que el enfoque podría mejorar los entornos de modelado industriales existentes. En general, la aplicación de la fragmentación del modelo podría ser aplicable y útil en el contexto industrial.

Con respecto a los editores gráficos, elegimos las herramientas de modelado existentes como Henshin y CRALA, migrando sus entornos con la combinación de EMF-Splitter y EMF-Stencil. Los nuevos entornos desarrollados proporcionan la creación de modelos modularizados, en contraste con los entornos desarrollados actualmente. De esta manera, proporcionamos un entorno gráfico escalable para visualizar las diferentes partes de un modelo por separado.

Por otra parte, los plug-ins desarrollados se integraron con herramientas existentes, como SAMPLER, DSL-comet y metaBup. Esto demuestra la versatilidad y el potencial de los plug-ins implementados para Eclipse.

En conjunto, el enfoque presentado ha propuesto un proceso sistemático para producir entornos de modelado escalables para DSMLs. Este enfoque se basa en el uso de un catálogo de patrones de modularidad que mostró su efectividad en diferentes contextos.

9.2 Trabajo Futuro

Esta tesis ha abierto una serie de líneas de investigación, que planeamos investigar en el futuro. A nivel técnico, actualmente estamos desarrollando una API para facilitar el uso programático de EMF-Splitter. Además, planeamos mejorar el servicio de fragmentación con la posibilidad de fragmentar modelos, pero que no sea a través del uso de referencias contenedoras, sino basarnos en estrategias como las descritas en [120].

Otra futura línea de investigación es el desarrollo de heurísticas para recomendar el subconjunto de objetos y la indexación de atributos para una restricción dada, basado en el análisis estático de la restricción. En particular, planeamos recomendar un alcance para una restricción OCL basándonos en las clases y las referencias que conforman la consulta y la visibilidad de los objetos. La idea es recorrer la expresión de restricción y,

para cada acceso de referencia, identificar los patrones aplicados para extraer el alcance asignado a la referencia y la visibilidad asignada al tipo de referencia, anotando la referencia con el más restrictivo de ambos. Cuando el alcance de todos los accesos de referencia se ha calculado de esta manera, el más amplio se sugiere heurísticamente como el alcance de la restricción. Investigaremos métodos para recomendar granularidades de fragmentación óptimas dado un conjunto de restricciones. Finalmente, los modelos fragmentados son más susceptibles para su uso de forma colaborativa, por ejemplo a través de sistemas de control de versiones, que los modelos monolíticos, ya que tienden a producir menos conflictos. Por lo tanto, planeamos integrar nuestro enfoque con un sistema de control de versiones para el modelado colaborativo [31].

Con respecto a los patrones, una línea de trabajo futuro es soportar la definición de operaciones de gestión de modelos (por ejemplo, transformaciones, generadores de código, restricciones) definidas sobre patrones. Dichas operaciones se transferirían al meta-modelo de dominio cuando se realice el mapeo. Esta sería una forma de re-utilizar y combinar las operaciones que contribuyeron con los diferentes patrones utilizados en [20].

EMF-Stencil puede generar editores gráficos que proporcionan una representación de modelo deducida automáticamente. De hecho, esta herramienta puede generar un conjunto de representaciones para la misma sintaxis abstracta. Sobre la base de esta funcionalidad, nos gustaría evaluar con los usuarios la calidad de estas representaciones.

Con respecto a la ingeniería para implementar entornos de modelado, estamos planeando aplicar este enfoque a otros escenarios. Buscaremos otros contextos, especialmente industriales, para demostrar aún más la aplicabilidad de nuestro enfoque. Al hacer esto, compararemos los entornos existentes con los generados utilizando nuestros plug-ins, en términos de escalabilidad y funcionalidades proporcionadas.

References

[1] *Acceleo*, https://www.eclipse.org/acceleo/, (last accessed in 2019).

[2] *ADM*, https://www.omg.org/adm/, (last accessed in 2019).

[3] N. Amálio, J. de Lara, and E. Guerra, "Fragmenta: A theory of fragmentation for MDE," in *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015*, IEEE, 2015, pp. 106–115.

[4] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations," in *Model Driven Engineering Languages and Systems*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–135.

[5] *ArgoUML*, http://argouml.tigris.org/, (last accessed in 2019).

[6] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Trans. Model. Comput. Simul.*, vol. 12, no. 4, pp. 290–321, 2002.

[7] *ATL*, https://www.eclipse.org/atl/, (last accessed in 2019).

[8] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context — motorola case study," in *Model Driven Engineering Languages and Systems*, L. Briand and C. Williams, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 476–491.

[9] M. Bastian, S. Heymann, and M. Jacomy, *Gephi: An open source software for exploring and manipulating networks*, 2009.

[10] J. Baton and R. Van Bruggen, *Learning Neo4j 3. X - Second Edition*. Packt Publishing, 2017.

[11] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, "Neo4emf, a scalable persistence layer for EMF models," in *Modelling Foundations and Applications*, J. Cabot and J. Rubin, Eds., Cham: Springer International Publishing, 2014, pp. 230–241.

[12] S. Berger, G. Grossmann, M. Stumptner, and M. Schrefl, "Metamodel-based information integration at industrial scale," in *Model Driven Engineering Languages and Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 153–167.

[13] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd. Packt Publishing, 2016.

[14] E. Biermann, K. Ehrig, C. Ermel, and G. Taentzer, "Generating eclipse editor plug-ins using tiger," in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, M. Nagl, and A. Zündorf, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 583–584.

[15] M. Blaha, *Patterns of data modeling*. CRC Press, 2010.

[16] P. Bottoni and A. Grau, "A suite of metamodels as a basis for a classification of visual languages," in *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Washington, DC, USA: IEEE Computer Society, 2004, pp. 83–90.

[17] P. Bottoni, E. Guerra, and J. de Lara, "A language-independent and formal approach to pattern-based modelling with support for composition and analysis," *Information & Software Technology*, vol. 52, no. 8, pp. 821–844, 2010.

[18] P. Bottoni, E. Guerra, and J. de Lara, "Enforced generative patterns for the specification of the syntax and semantics of visual languages," *J. Vis. Lang. Comput.*, vol. 19, no. 4, pp. 429–455, 2008.

[19] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd. Morgan & Claypool Publishers, 2017.

[20] J.-M. Bruel, B. Combemale, E. Guerra, J.-M. Jézéquel, J. Kienzle, J. De Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe, "Model transformation reuse across metamodels," in *International Conference on Theory and Practice of Model Transformations*, Springer, 2018, pp. 92–109.

[21] H. Brunelière, J. Cabot, J. L. C. Izquierdo, L. O. Arrieta, O. Strauß, and M. Wimmer, "Software modernization revisited: Challenges and prospects," *IEEE Computer*, vol. 48, no. 8, pp. 76–80, 2015.

[22] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: A generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ACM, 2010, pp. 173–174.

[23] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90.

[24] J. Cabot and E. Teniente, "Incremental integrity checking of UML/OCL conceptual schemas," *Journal of Systems and Software*, vol. 82, no. 9, pp. 1459–1478, 2009.

[25] *CDO*, https://www.eclipse.org/cdo/, (last accessed in 2019).

[26] H. Cho and J. Gray, "Design patterns for metamodels," in *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops, Portland, Oregon, USA: ACM, 2011, pp. 25–32.

[27] K. Chodorow, *MongoDB: The Definitive Guide*, 2nd. O'Reilly Media, Inc., 2013.

[28] J. S. Cuadrado, E. Guerra, and J. de Lara, "A component model for model transformations," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1042–1060, 2014.

[29] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, pp. 621–646, 2006.

[30] *DB Store*, https://wiki.eclipse.org/CDO/DB_Store, (last accessed in 2019).

[31] C. Debreceni, G. Bergmann, M. Búr, I. Ráth, and D. Varró, "The MONDO collaboration framework: Secure collaborative modeling over existing version control systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, ACM, 2017, pp. 984–988.

[32] B. Demuth and C. Wilke, "Model and object verification by using dresden ocl," in *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*, Citeseer, 2009, pp. 687–690.

[33] E. Dijkstra, *On the role of scientific thought*, EWD447, 1974.

[34] J. Dingel, Z. Diskin, and A. Zito, "Understanding and improving uml package merge," *Software & Systems Modeling*, vol. 7, no. 4, pp. 443–467, 2008.

[35] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach, "Graph based modeling and implementation with EER/GRAL," in *15th International Conference on Conceptual Modeling — ER '96*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 163–178.

[36] *Eclipse Modeling Project (EMF)*, https://www.eclipse.org/modeling/, (last accessed in 2019).

[37] A. Egyed, K. Zeman, P. Hehenberger, and A. Demuth, "Maintaining consistency across engineering artifacts," *IEEE Computer*, vol. 51, no. 2, pp. 28–35, 2018.

[38] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[39] *Enterprise Architect*, http://sparxsystems.com/products/ea/, (last accessed in 2019).

[40] J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, pp. 77–92.

[41] M. Fleck, J. Troya, and M. Wimmer, "Towards generic modularization transformations," in *15th International Conference on Modularity*, ACM, 2016, pp. 190–195.

[42] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[43] W. Frakes, R. Prieto, C. Fox, *et al.*, "Dare: Domain analysis and reuse environment," *Annals of software engineering*, vol. 5, no. 1, pp. 125–141, 1998.

[44] R. B. France, D. Kim, S. Ghosh, and E. Song, "A UML-based pattern specification technique," *IEEE Trans. Software Eng.*, vol. 30, no. 3, pp. 193–206, 2004.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[46] A. García-Domínguez, K. Barmpis, D. S. Kolovos, R. Wei, and R. F. Paige, "Stress-testing centralised model stores," in *Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, 2016, pp. 48–63.

[47] J. García Molina, F. O. García Rubio, V. Pelechano, A. Vallecillo, J. M. Vara, and C. Vicente-Chicote, *Desarrollo de Software Dirigido por Modelos Conceptos, Métodos y Herramientas*. España: Ra-Ma, 2012.

[48] A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara, "EMF splitter: A structured approach to EMF modularity," in *XM@MoDELS*, ser. CEUR, vol. 1239, CEUR-WS.org, 2014, pp. 22–31. [Online]. Available: http://ceur-ws.org/Vol-1239.

[49] A. Garmendia, E. Guerra, J. de Lara, A. García-Domínguez, and D. Kolovos, "Scaling-up domain-specific modelling languages through modularity services," *Information and Software Technology*, 2019.

[50] *Gephi*, https://gephi.org/, (last accessed in 2019).

[51] M. Gerhart and M. Boger, "Concepts for the model-driven generation of graphical editors in eclipse by using the graphiti framework," *International Journal of Computer Techniques*, vol. 3, no. 4, 2016.

[52] *GMF*, https://www.eclipse.org/gmf-tooling/, (last accessed in 2019).

[53] A. Gómez, X. Mendialdua, G. Bergmann, J. Cabot, C. Debreceni, A. Garmendia, D. S. Kolovos, J. de Lara, and S. Trujillo, "On the opportunities of scalable modeling technologies: An experience report on wind turbines control applications development," in *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, 2017, pp. 300–315.

[54] D. Granada, J. M. Vara, V. A. Bollati, and E. Marcos, "Enabling the development of cognitive effective visual DSLs"," in *Model-Driven Engineering Languages and Systems*, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds., Cham: Springer International Publishing, 2014, pp. 535–551.

[55] *Graphiti*, https://www.eclipse.org/graphiti/, (last accessed in 2019).

[56] T. J. Grose, G. C. Doney, and S. A. Brodsky, *Mastering XMI: Java Programming with XMI, XML and UML*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[57] C. Guychard, S. Guerin, A. Koudri, A. Beugnard, and F. Dagnat, "Conceptual interoperability through models federation," in *Semantic Information Federation Community Workshop*, 2013.

[58] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler, "On language-independent model modularisation," *T. Asp.-Oriented Soft. Dev. VI*, vol. 6, pp. 39–82, 2009.

[59] *Hibernate*, https://wiki.eclipse.org/CDO/Hibernate_Store, (last accessed 2019).

[60] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 633–642.

[61] *IBM Rational*, https://www-01.ibm.com/software/rational/uml/, (last accessed in 2019).

[62] *JDT, Java Development Tools*, https://www.eclipse.org/jdt/, (last accessed in 2019).

[63] A. Jiménez-Pastor, A. Garmendia, and J. de Lara, "Scalable model exploration for model-driven engineering," *Journal of Systems and Software*, vol. 132, pp. 204–225, 2017.

[64] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis feasibility study," CMU-SEI, Tech. Rep., 90.

[65] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages," *arXiv*, 2014.

[66] L. C. Kats and E. Visser, "The spoofax language workbench: Rules for declarative specification of languages and ides," *SIGPLAN Not.*, vol. 45, no. 10, pp. 444–463, Oct. 2010, ISSN: 0362-1340.

[67] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling*. USA: John Wiley & Sons, Inc., 2007.

[68] P. Kelsen and Q. Ma, "A modular model composition technique," in *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, 2010, pp. 173–187.

[69] P. Kelsen, Q. Ma, and C. Glodt, "Models within models: Taming model complexity using the sub-model lattice," in *International Conference on Fundamental Approaches to Software Engineering, FASE*, ser. LNCS, vol. 6603, Springer, 2011, pp. 171–185.

[70] S. Kent, "Model driven engineering," in *Proceedings of the Third International Conference on Integrated Formal Methods*, ser. IFM, London, UK: Springer-Verlag, 2002, pp. 286–298.

[71] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[72] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige, "Assessing the use of eclipse MDE technologies in open-source software projects," in *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), Ottawa, Canada, September 29*, 2015, pp. 20–29.

[73] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon Object Language (EOL)," in *European Conference on Model Driven Architecture-Foundations and Applications, ECMFA*, Springer Berlin Heidelberg, 2006, pp. 128–142.

[74] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon Transformation Language," in *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2008, pp. 46–60.

[75] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Eclipse development tools for Epsilon," in *Eclipse Summit Europe, Eclipse Modeling Symposium*, vol. 20062, 2006, p. 200.

[76] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck, "Taming EMF and GMF using model transformation," in *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2010, pp. 211–225.

[77] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*, 2013, p. 2.

[78] T. Kühn, S. Böhme, S. Götz, and U. Aßmann, "A combined formal model for relational context-dependent roles," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, 2015, pp. 113–124.

[79] J. de Lara, E. Guerra, and J. S. Cuadrado, "Reusable abstractions for modeling languages," *Inf. Syst.*, vol. 38, no. 8, pp. 1128–1149, 2013.

[80] J. de Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, 12:1–12:46, 2014.

[81] J. de Lara and H. Vangheluwe, "AToM3: A Tool for Multi-formalism and Metamodelling," in *Fundamental Approaches to Software Engineering*, R.-D. Kutsche and H. Weber, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 174–188.

[82] A. Ledeczi, M. Maroti, G. Karsai, and G. Nordstrom, "Metaprogrammable toolkit for model-integrated computing," in *Proceedings of the IEEE Conference on Engineering of Computer-based Systems*, ser. ECBS, Nashville, Tennessee: IEEE Computer Society, 1999, pp. 311–317.

[83] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, "Example-driven meta-model development," *Software and Systems Modeling (Springer)*, vol. 14, no. 4, pp. 1323–1347, 2015.

[84] J. J. López-Fernández, A. Garmendia, E. Guerra, and J. de Lara, "An example is worth a thousand words: Creating graphical modelling environments by example," *Software and Systems Modeling (Springer)*, vol. 18, no. 2, pp. 961–993, 2019.

[85] *Magic Draw*, https://www.nomagic.com/products/magicdraw, (last accessed in 2019).

[86] R. C. Martin, D. Riehle, and F. Buschmann, *Pattern languages of program design 3*. Addison-Wesley, 1997.

[87] T. Mayerhofer, M. Wimmer, L. Berardinelli, and R. Drath, "A model-driven engineering workbench for CAEX supporting language customization and evolution," *IEEE Trans. Industrial Informatics*, vol. 14, no. 6, pp. 2770–2779, 2018.

[88] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, pp. 316–, Dec. 2005.

[89] M. Minas, "Concepts and realization of a diagram editor generator based on hypergraph transformation," *Sci. Comput. Program.*, vol. 44, no. 2, pp. 157–180, Aug. 2002.

[90] *Modelio*, https://www.modelio.org/, (last accessed 2019).

[91] *MONDO*, https://http://www.mondo-project.org/, (last accessed in 2019).

[92] D. L. Moody, "The "physics" of notations: A scientific approach to designing visual notations in software engineering," *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 485–486, 2010.

[93] D. L. Moody and A. Flitman, "A methodology for clustering entity relationship models - A human information processing approach," in *Conceptual Modeling - ER'99*, ser. Lecture Notes in Computer Science, vol. 1728, Springer, 1999, pp. 114–130.

[94] M. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005.

[95] Object Management Group (OMG), https://www.omg.org/, (last accessed in 2019).

[96] Object Management Group (OMG), *Meta-Object Facility (MOF) Specification, Version 2.5.1*, OMG Document Number formal/16-11-01 (https://www.omg.org/spec/MOF/2.5.1/), 2016.

[97] Object Management Group (OMG), *XML Metadata Interchange (XMI) Specification, Version 2.5.1*, OMG Document Number formal/15-06-07 (https://www.omg.org/spec/XMI/2.5.1/), 2015.

[98] *Objectivity/DB*, https://wiki.eclipse.org/CDO/Objectivity_Store, (last accessed in 2019).

[99] *OCLinEcore*, https://wiki.eclipse.org/OCL/OCLinEcore, (last accessed 2019).

[100] OMG, *Knowledge Discovery Meta-model specification*, `http://www.omg.org/spec/KDM/`, (last accessed in 2019).

[101] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack, "The design of a conceptual framework and technical infrastructure for model management language engineering," in *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, Potsdam, Germany, 2-4 June*, IEEE, 2009, pp. 162–171.

[102] *Papyrus*, https://www.eclipse.org/papyrus/, (last accessed in 2019).

[103] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[104] L. Pedro, V. Amaral, and D. Buchs, "Foundations for a domain specific modeling language prototyping environment: A compositional approach," in *Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*, University of Jyväskylän, Oct. 2008.

[105] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara, "Pattern-based development of domain-specific modelling languages," in *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE Computer Society, 2015, pp. 166–175.

[106] A. Pescador and J. de Lara, "Dsl-maps: From requirements to design of domain-specific languages," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 438–443.

[107] *QVT*, https://www.omg.org/spec/QVT/, (last accessed 2019).

[108] L. M. Rose, D. S. Kolovos, N. Drivalos, J. R. Williams, R. F. Paige, F. A. C. Polack, and K. J. Fernandes, "Concordance: A framework for managing model integrity," in *European Conference on Modelling Foundations and Applications, (ECMFA)*, vol. 6138, Springer, 2010, pp. 245–260.

[109] D. Rubel, J. Wren, and E. Clayberg, *The Eclipse Graphical Editing Framework (GEF)*, 1st. Addison-Wesley Professional, 2011.

[110] A. L. Santos and E. Gomes, "Xdiagram: A declarative textual DSL for describing diagram editors (tool demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016, Amsterdam, Netherlands: ACM, 2016, pp. 253–257.

[111] C. Schäfer, T. Kuhn, and M. Trapp, "A pattern-based approach to dsl development," in *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops, Portland, Oregon, USA: ACM, 2011, pp. 39–46.

[112] M. Scheidgen and J. Fischer, "Model-based mining of source code repositories," in *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014*, ser. Lecture Notes in Computer Science, vol. 8769, Springer, 2014, pp. 239–254.

[113] M. Scheidgen, A. Zubow, J. Fischer, and T. H. Kolbe, "Automated and transparent model fragmentation for persisting large models," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, 2012, pp. 102–118.

[114] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML Classroom: An Introduction to Object-Oriented Modeling*. Springer, Heidelberg, 2012.

[115] *Sirius*, https://www.eclipse.org/sirius/, (last accessed in 2019).

[116] D. Spinellis, "Notable design patterns for domain-specific languages," *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, Feb. 2001.

[117] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, "Model-driven software development: Technology, engineering, management," John Wiley & Sons, Inc.

[118] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008, See also http://www.eclipse.org/modeling/emf/.

[119] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Softw. Pract. Exper.*, vol. 39, no. 15, pp. 1253–1292, Oct. 2009.

[120] D. Strüber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting models using information retrieval and model crawling techniques," in *Fundamental Approaches to Software Engineering - 17th International Conference, FASE, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Grenoble, France, April 5-13*, Springer, 2014, pp. 47–62.

[121] D. Strüber, M. Selter, and G. Taentzer, "Tool support for clustering large meta-models," in *BigMDE 2013*, ACM, 2013, p. 7.

[122] D. Strüber, G. Taentzer, S. Jurack, and T. Schäfer, "Towards a distributed modeling process based on composite models," in *Fundamental Approaches to Software Engineering - 16th International Conference, FASE, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Rome, Italy, March 16-24*, Springer, 2013, pp. 6–20.

[123] R. N. Taylor, W. Tracz, and L. Coglianese, "Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 5, pp. 27–38, 1995.

[124] J.-P. Tolvanen and S. Kelly, "Metaedit+: Defining and using integrated domain-specific modeling languages," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, Orlando, Florida, USA: ACM, 2009, pp. 819–820.

[125] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "Emf-incquery: An integrated development environment for live model queries," *Sci. Comput. Program.*, vol. 98, pp. 80–99, 2015.

[126] *UML*, https://www.omg.org/spec/UML/, (last accessed in 2019).

[127] D. Vaquero-Melchor, A. Garmendia, E. Guerra, and J. de Lara, "Towards enabling mobile domain-specific modelling," in *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.*, 2016, pp. 117–122.

[128] L. Vogel, *Eclipse IDE.* vogella.com; Third Edition (April 22), 2013.

[129] R. Wei, D. S. Kolovos, A. Garcia-Dominguez, K. Barmpis, and R. F. Paige, "Partial loading of xmi models," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16, Saint-malo, France: ACM, 2016, pp. 329–339.

[130] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[131] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[132] D. Wüest, N. Seyff, and M. Glinz, "Flexisketch: A lightweight sketching and metamodeling approach for end-users," *Software & Systems Modeling*, vol. 18, no. 2, pp. 1513–1541, 2019.

[133] *Xtend*, https://www.eclipse.org/xtend/, (last accessed in 2019).

[134] L. Zhang, Huaxi, Zhang, Z. Fang, X. Xiang, M. Huchard, and R. Zapata, "Towards an architecture-centric approach to manage variability of cloud robotics," in *DSLRob: Domain-Specific Languages and models for ROBotic systems, Hamburg, German*, 2015.

Appendix A

Scoped constraints for Wind Turbines meta-model

This appendix contains the scoped constraints used in the experiment of Section 7.4. For completeness, we consider constraints with all kinds of scopes: one with scope sameProject, three with scope sameRootPkg, two with scope samePkg, and five with scope sameUnit.

Listing 6 shows the constraint with scope sameProject, which controls the number of state machines in the whole model.

```
1 context StateMachine inv numberStateMachines:
2   StateMachine.allInstances()→size() <= 10
```

Listing 6 Scoped constraint with scope sameProject.

Listing 7 shows the invariants with scope sameRootPkg. The first two constraints control the maximum number of instances of ControlSubsystem and Component. The third one validates that there are no more than 5 nested Subsystems (i.e., nested packages of type Subsystem).

```
1  context ControlSubsystem inv numberControlSubsystems:
2      ControlSubsystem.allInstances()→size() <= 10
3
4  context Component inv numberComponents:
5      Component.allInstances()→size() <= 50
6
7  context Subsystem inv depthSubsystem:
8      self.subsystems→forAll(sub1 |
9          sub1.subsystems→forAll(sub2 |
10             sub2.subsystems→forAll(sub3 |
11                 sub3.subsystems→forAll(sub4 |
12                     sub4.subsystems→forAll(sub5 |
13                         sub5.subsystems→size() = 0)))))
```

Listing 7 Scoped constraints with scope sameRootPkg.

Listing 8 shows the invariants with scope samePkg. The first one checks that every subsystem contains a component connected with itself through references inPort and outPort. The last one validates that each Subsystem has at least one component with an input port.

```
1  context Subsystem inv connectedComponents:
2      self.ensembles→collect(connectors)→flatten()→exists(con |
3          Component.allInstances()→exists(comp |
4              comp.ports→includesAll(Set{con.inPort, con.outPort})))
5
6  context Subsystem inv inputPortSubsystem:
7      self.ensembles→collect(elements)→flatten()→exists(c |
8          c.ports→exists(p | p.oclIsTypeOf(InPort)))
```

Listing 8 Scoped constraints with scope samePkg.

Finally, Listing 9 shows the constraints with scope sameUnit. The first two constraints ensure that every StateMachine has exactly one InitialState and at least one SimpleState. The third constraint checks that every SimpleState is reachable from the InitialState. The fourth constraint checks that every Port is connected to another one. The last constraint ensures that each InitialState is connected to some state.

```
1  context StateMachine inv oneInitialState:
2      self.states→one(s | s.oclIsTypeOf(InitialState))
3
4  context StateMachine inv existsSimpleState:
5      self.states→exists(s | s.oclIsTypeOf(SimpleState))
6
7  context SimpleState inv reachableState:
8      self→closure(incoming.source)→exists(v | v.oclIsTypeOf(InitialState))
9
10 context Port inv connectedPorts:
11     Connector.allInstances()→exists(c |
12         (c.inPort = self and not c.outPort.oclIsUndefined()) or
13         (c.outPort = self and not c.inPort.oclIsUndefined()))
14
15 context InitialState inv initStateIsNotIsolated:
16     self.outgoing→size() >= 1
```

Listing 9 Scoped constraints with scope sameUnit.

Appendix B

Scoped constraints for CAEX

This appendix contains the nine scoped constraints used in the case study of Section 7.5.1. One constraint has scope sameProject, another has scopes samePkg and sameUnit simultaneously, and seven constraints have scope sameUnit.

Listing 10 shows the constraint with scope sameProject, which validates the version of the AutomationML model.

```
1 context CAEXFile inv superiorStandardVersionIsMandatory:
2     self.superiorStandardVersion→exists(v | v = 'AutomationML 3.0')
```

Listing 10 Scoped constraint with scope sameProject.

Listing 11 shows the constraint with two scopes: samePkg and sameUnit. This happens because CAEXObject is a base class from which many other classes inherit, and therefore, its instances can be found in packages and units. The constraint ensures non-empty object identifiers.

```
1 context CAEXObject inv idIsMandatory:
2     self.iD <> null
```

Listing 11 Scoped constraint with scope samePkg and sameUnit.

Finally, Listing 12 shows the constraints with scope sameUnit. The first one checks that the base class of a SystemUnitClass is a SystemUnitClass as well. The second and third constraints validate that InternalElements with a base system unit define, for every attribute in the base system unit, another attribute with the same name and value, and vice versa. The fourth constraint ensures that InternalElements have no base class. The last tree constraints ensure that if an InternalElement contains a requirement with role class name *Process*, *Resource* or *Product*, then, all its internal elements must also define a requirement with an equally named role class.

```
1  context SystemUnitClass inv inheritanceMustPointToSUC:
2      self.baseClass <> null implies self.baseClass.oclIsTypeOf(SystemUnitClass);
3
4  context InternalElement inv strongConformanceSUC2IE:
5      self.baseSystemUnit <> null implies
6          self.baseSystemUnit.attribute→forAll(aC |
7              self.attribute→one(cI |
8                  aC.name = cI.name and aC.value = cI.value));
9
10 context InternalElement inv strongConformanceIE2SUC:
11     self.baseSystemUnit <> null implies
12         self.attribute→forAll(aI |
13             self.baseSystemUnit.attribute→one(aC |
14                 aC.name = aI.name and aC.value = aI.value));
15
16 context InternalElement inv noInheritanceForIEs:
17     self.baseClass = null;
18
19 context InternalElement inv processContainsProcesses:
20     self.roleRequirements.roleClass.name→exists(r | r = 'Process') implies
21         self.internalElement→forAll(ie |
22             ie.roleRequirements.roleClass.name→exists(r | r = 'Process'));
23
24 context InternalElement inv resourceContainsResources:
25     self.roleRequirements.roleClass.name→exists(r | r = 'Resource') implies
26         self.internalElement→forAll(ie |
27             ie.roleRequirements.roleClass.name→exists(r | r = 'Resource'));
28
29 context InternalElement inv productContainsProducts:
30     self.roleRequirements.roleClass.name→exists(r | r = 'Product') implies
31         self.internalElement→forAll(ie |
32             ie.roleRequirements.roleClass.name→exists(r | r = 'Product'));
```

Listing 12 Scoped constraints with scope sameUnit.