EMF Splitter: A Structured Approach to EMF Modularity

Antonio Garmendia[1], Esther Guerra[1], Dimitrios S. Kolovos[2], and Juan de Lara[1]

[1] Modelling and Software Engineering Group (miso, htpp://www.miso.es)
Computer Science Department
Universidad Autónoma de Madrid (Spain)
{Antonio.Garmendia, Esther.Guerra, Juan.deLara}@uam.es
[2] Enterprise Systems Group (http://www.enterprise-systems.org/)
Computer Science Department
University of York (United Kingdom)
dimitris.kolovos@york.ac.uk

Abstract. Model-Driven Engineering aims at reducing the cost of system development by raising the level of abstraction at which developers work. Thus, models become the main assets in this paradigm, guiding the development until code for the final application is obtained.
However, even though domain-specific, models may become large and complex, becoming cumbersome to edit and manipulate. In this scenario, mechanisms helping in the agile definition and management of models in the large are crucial. Modularity is one of such mechanisms.
In this paper, we describe a novel approach to the construction of EMF models in a structured way. It is based on the annotation of the Ecore meta-models with modularity concepts (like project, package and unit), from which we generate an Eclipse plug-in that enables the editing of models according to that structure (i.e., organized in projects and decomposed into folders and files). The paper presents our supporting tool and discusses benefits and future challenges.

Keywords: Meta-modelling, Modularity, Agile modelling, Eclipse Modeling Framework

1 Introduction

Nowadays, many organisations use Model-Driven Engineering (MDE) [11] to develop their systems or migrate legacy code. MDE proposes the use of models as primary artefacts to construct software, being supported by model management tools [2].

MDE-based solutions frequently involve the creation of Domain-Specific Languages (DSLs), which are defined by a meta-model that describes the set of models considered valid. When a new DSL is created, there is the need to build the corresponding modelling environment as well, in order to facilitate the construction of valid models of the DSL and provide basic functionalities like model persistence or model consistency checking.

In this respect, the Eclipse Modeling Framework (EMF) [12] is a well-known and widely-used framework that allows the definition of meta-models and models. Starting from the definition of a meta-model, the framework generates a basic tree-editor to edit the instance models. However, models defined with these editors are monolithic. As the requirements of a system grow over time, the models tend to become complex and large, which results in poor comprehensibility and maintainability, while their editing becomes a tedious task. Moreover, the manipulation of large models may also affect performance in terms of model persistence, loading, querying and transformation.

Software design and programming languages provide mechanisms to simplify the creation of complex systems. One of these mechanisms is modularity [8], which promotes a scalable approach to the construction of software systems through the composition of smaller subsystems which can be implemented separately in a simpler way. Other benefits of modularity include increased flexibility and reuse possibilities, facilitating distributed teamwork and version control.

In MDE, models are the main assets to create software. However, models frequently lack native modularization mechanisms, unless they are explicitly encoded in the modelling language and implemented in the supporting modelling environments. Thus, we propose to apply modularity mechanisms to the construction of models, allowing the definition of complex models from submodels which are easier to process and reuse. Our approach is based on the annotation of the meta-model elements that will play some role in the structuring and modularity of the models. We propose several structures, based on the concepts of project, package and unit. Starting from this definition, we automatically generate a modelling environment (an Eclipse plug-in) which permits editing models in modular way, following a similar philosophy to the Java Development Tools[3] (JDT). In this way, each model corresponds to an Eclipse project, and the model content can be organized in folders and files, with a direct mapping to the file system. Altogether, our plug-in enables:

- the editing of models according to a given modularity structure.
- the splitting of monolithic models according to a given modularity strategy.
- the composition of parts of a model to build a monolithic one.

The remaining of this paper is organized as follows. First, Section 2 describes our approach to incorporate modularity mechanisms to models, as well as the modular structures we support. Then, Section 3 presents the implementation of our approach for Eclipse and EMF. Next, Section 4 discusses related work. Finally, Section 5 concludes the paper with some conclusions and lines of future work.

2 Specifying the Modular Structure of Models

We propose a modular structure for models, based on the philosophy of the Eclipse JDT and how Eclipse organizes Java projects. Eclipse projects are orga-

[3] `http://www.eclipse.org/jdt/`

nized hierarchically, defining a root node which contains a tree view of the project content. In this way, the compilation units of the project (e.g., Java classes) are organized into different types of folders (e.g., source folders) and packages, which can be nested. This modular organization facilitates the structuring of projects, the provision of scoping mechanisms, and the use of indexes to enable load-on-demand, incremental builds and efficient resolution of cross-references, among other advantages.

Inspired by this framework, we propose a notion of modularity for models based on the concepts of project, package and (compilation) unit. The upper frame in Figure 1 shows a simplified version of the pattern that formalizes these concepts, as well as their relations. Thus, the main modularity concepts in our approach are `Project`, `Package` and `Unit`. `Project` is the root that contains the rest of the elements, which can be of any of the other types. Objects of type `Package` can contain units as well as other packages (i.e., it implements the *Composite* object-oriented design pattern [3]). Finally, objects of type `Unit` can be defined either inside of packages, or directly inside a project.

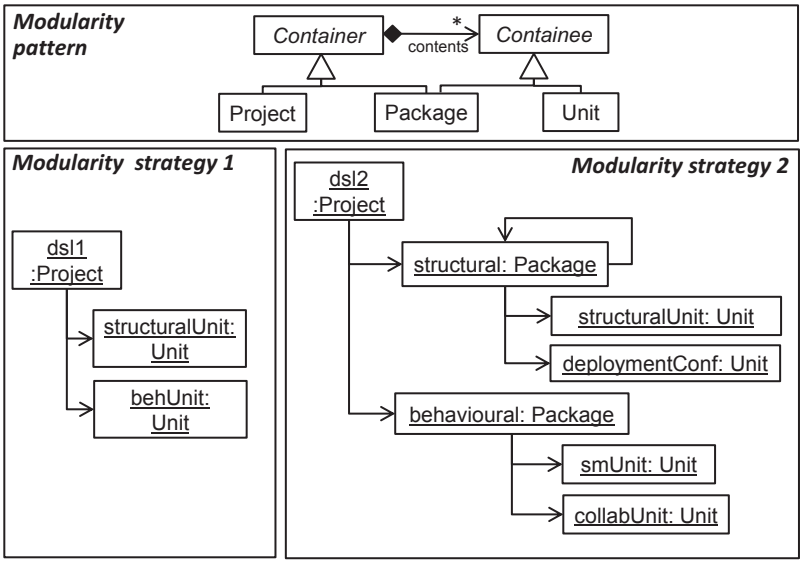


Fig. 1: Pattern to describe the modular structure of a meta-model (top). Two possible modularity strategies, as instances of the pattern (bottom).

This pattern allows for the configuration of different structures or ways of organizing a model. For instance, we can have projects that do not contain packages but units are directly placed in the project root node, projects that consider different types of packages containing different types of units and where package nesting might be allowed or not, or a mix of both. Conceptually, if we interpret the pattern in Figure 1 as a meta-model, then the possible structures

that can be applied to a particular DSL can be seen as instances of this meta-model, as shown to the bottom of the same figure.

While the modularity pattern allows configuring a particular modularity strategy, in addition, the strategy needs to be mapped to a particular meta-model. That is, we need to select the class playing the role of `Project`, and the classes for the different `Package`s and `Unit`s. Conceptually, it is natural to consider the application of the pattern to the meta-model of a DSL as a case of multi-level modelling [1], where the modularity pattern meta-model is interpreted as a partial type for the DSL meta-model, as Figure 2 shows. We refrain from introducing the full details of multi-level modelling here, which can be found in [1], and only present the information needed to understand how the pattern application works. The complete modularity pattern is shown in this figure. In particular, projects, packages and units have a descriptive `name`, as well as an `icon` which will be used in the modelling environment to identify them. On their side, units have an `extension` that will be used by the corresponding system files.

The particular structure to be used for the models of a DSL is determined by annotating which elements of the DSL meta-model will play the roles of project, package and root of a unit. Conceptually, this is equivalent to typing some elements of the DSL meta-model with the types offered by the modularity pattern. This typing is *partial*, because some elements of the DSL meta-model may be not typed by any pattern element. However, as shown in Figure 1, it is useful to consider the relation between the meta-models of the DSL and the pattern as a typing relation (and not just as annotations) because the typing rules ensure a correct annotation of the DSL meta-model.

The middle frame in Figure 2 shows the annotation (represented with dotted arrows) of an example meta-model with our modularity concepts. The annotation must respect the modular structure of the pattern, given by instantiation rules. In this example, the `ComponentSystem` class is annotated as `Project`, and the attribute `sysID` is bound to the project `name`. The `SubSystem` class is annotated as `Package`; this is possible because there is a composition relation from `ComponentSystem` (the project) to `SubSystem`, as the pattern demands by means of the relation `contents`. While this annotation permits creating `SubSystem` packages inside projects, it does not allow the nesting of subsystems inside subsystems, for which the meta-model would need to define an appropriate containment relation. Finally, both classes `Component` and `Behaviour` are annotated as `Unit` (i.e., we instantiate `Unit` twice). This means that the instances of `Component` and `Behaviour`, as well as the objects they contain through containment relationships (e.g., `Port` in the case of `Components`) will be stored in the same unit (i.e., in the same file).

A remark is interesting here. Some attributes in the modularity pattern, like the `icon`s to be used in the modelling environment or the file `extension` for units, need to receive a value in the DSL meta-model. Using multi-level terminology, we say that such attributes have potency 1, as they receive a value in the meta-level right below the pattern. Other attributes, like `name`, need to be bound to

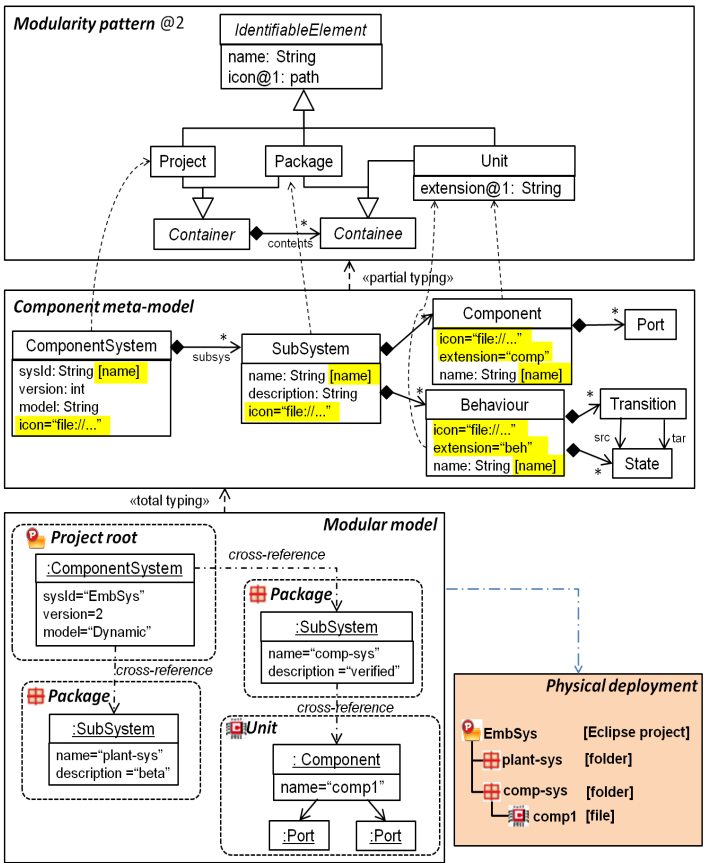


Fig. 2: Pattern to describe the modular structure of a meta-model (top). Application of pattern to a meta-model (middle). A structured model and its physical deployment (bottom).

some class attribute of the DSL meta-model, and only take a value at the model level (i.e., two levels below from the pattern point of view). Hence, we say that such attributes have potency 2. In the modularity pattern, every element has potency 2 (indicated by the `@2` of the meta-model), except attributes `icon` and `extension`, which have an explicit potency 1.

Once the DSL meta-model is annotated with the desired structure, an automated process generates the machinery to split existing monolithic models according to the chosen structure, and it also generates a modelling environment that permits building models according to that structure. The bottom of Figure 2 shows an illustration of a modular model, the corresponding Eclipse project structure, and how the modular structure is physically mapped to the file system. Hence, the model root (the instance of `ComponentSys`) is mapped to the project root, the two instances of `SubSystem` are mapped to two folders, and the

`Component` instance is mapped into a file. As discussed in the following section, hidden XMI (XML Metadata Interchange) files are also created for the project root and the packages, storing the properties of the corresponding objects, and allowing their manipulation by means of the *Property* view of Eclipse, so that the user has the impression of manipulating folders with different properties. Next, we explain the tool support implementing this modularity machinery.

3 Tool Support: EMF Splitter

This section describes our tool, called *EMF Splitter*[4], supporting the modular structure for EMF models proposed in the previous section. The main functionality of this tool is to, given an annotated Ecore model, generate an Eclipse plug-in that allows: (a) the creation of instance models according to the specified modularity strategy, (b) the decomposition of an existing monolithic model according to the modularity strategy, and (c) the composition of a single model out of a project consisting of folders and units. We have implemented this tool using Acceleo[5], a code generation language based on the Object Management Group (OMG) MOF Model to Text Language (MTL) standard.

Figure 3 shows the steps to create an Eclipse modular project for editing the instances of an Ecore meta-model. The first step is to annotate the meta-model classes with the concepts of project, package and unit. To facilitate this task, we have developed a graphical environment that allows building and annotating meta-models graphically according to different predefined patterns (in this case, according to our modularity pattern). In a second step, the annotated meta-model is fed into *EMF Splitter*. This tool automatically produces a *genmodel* configuration file, which is used to generate code implementing the chosen modularity strategy, and that is distributed as an Eclipse plug-in. Finally, tool users may use the generated plug-in to create models as dedicated Eclipse projects.

The generated plug-in provides functionalities to structure a model in several XMI files. This organization is transparent to the user, and can be useful for reusing parts of the models which can be nested. Each instance of a class annotated with project or package has an associated XMI file with the value of its attributes and cross-references to the objects it contains. These files are filtered and hidden to the user, who can edit the properties of such objects using the *Property* view of Eclipse when the corresponding folder or the root of the project is selected. Each instance of a class annotated as `unit` is stored in an XMI file, together with the objects it contains. These files can be modified through the generated standard tree editors.

Next, we present an example that makes use of the annotated meta-model from Section 2 and the corresponding plug-in generated using *EMF Splitter*. To show the benefits of the generated plug-in, we start from a synthetic, monolithic model which is shown in Figure 4(a). In practice, the model could be much larger, so that once decomposed into folders and smaller units, its parts would be easier

[4] See its web page at `http://www.miso.es/tools/EMFSplitter.html`

[5] `http://www.eclipse.org/acceleo/`

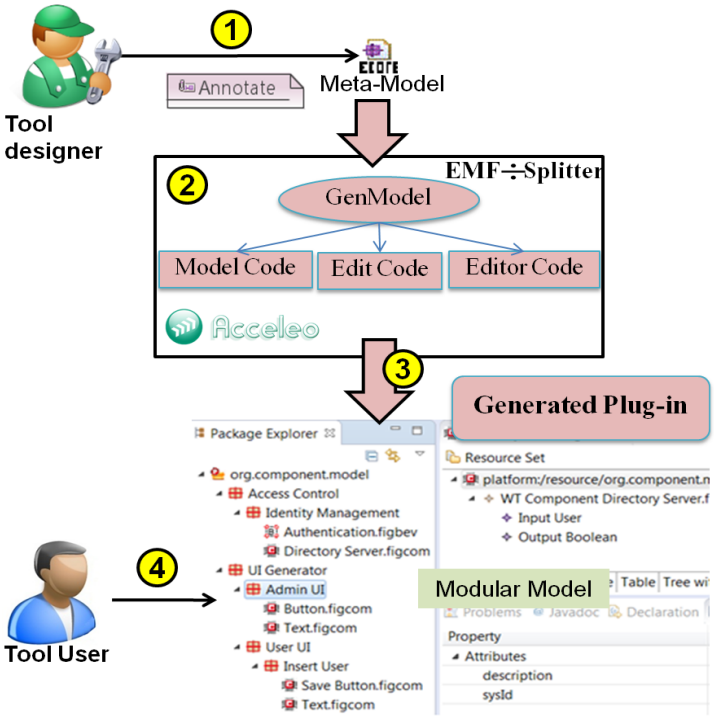
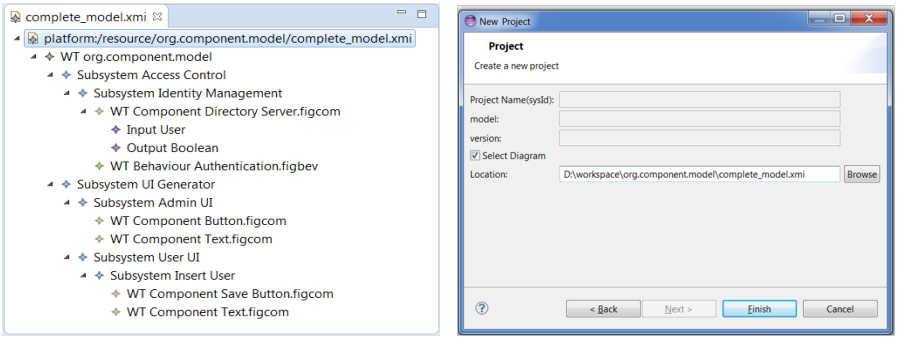


Fig. 3: Process overview of *EMF Splitter*.

to edit, navigate and comprehend. Thus, we first create a modular project using the plug-in. The plug-in offers a creation wizard with two alternatives, as shown in Figure 4(b). In the first one, we can create a model from scratch, giving values to the root class (`Project`) attributes. The wizard has input controls to introduce values for the attributes of the root object, which in this case is of type `ComponentSystem`. As the `sysId` attribute was mapped to the `name` of the project, the value introduced in this field is used as project name.

The second option of the wizard is to provide the path of an existing model. In that case, the wizard creates a new modular project, where the initial model is decomposed according to the selected modularization strategy. For the development of this example, we use the second option. As a result, we obtain the modelling environment shown in Figure 5. The *Package Explorer* view shows the structure of the containment hierarchy, made of all the objects that belong to the model, represented physically as folders and files within the project. In the top-right, a submodel (i.e., file) of the project is being edited using the tree-based editor. As we have explained previously, the root of the tree can be an instance of a class annotated with unit. At the bottom, the figure shows the edition of the attributes of a package using the *Property* view. While these attributes are actu-



(a) Example of model. (b) Wizard to create the Model Project.

Fig. 4: Generating a modular project from a model.

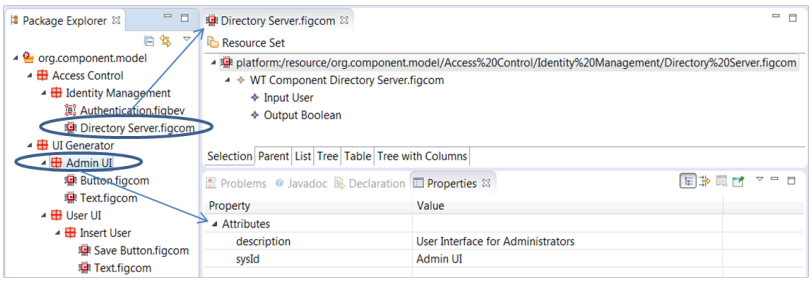


Fig. 5: Modelling environment in action.

ally stored in an XMI file, this is transparent to the user, who has the impression of editing properties of a folder.

The decomposition of a model into an Eclipse project offers many advantages. For example, we can include any kind of artefact (like documentation or source code) inside the folders, facilitating, e.g., the traceability from model elements to external artefacts. Moreover, we no longer have a monolithic model, but the division and organization into folders permits shorter loading times (of a fragment w.r.t. loading the complete monolithic model) and facilitates comprehensibility, reusability, distributed teamwork and version control.

4 Related Work

Our main goal is to provide a tool that allows building models in a structured way. For this purpose, we start by annotating the meta-model with the modularity concepts formalised in a pattern. Next, we compare with related works addressing model modularity, fragmentation and model slicing.

Although introducing modularity in ad-hoc ways into existing DSLs can deliver benefits, it is also costly [7]. Hence, it is desirable to have mechanisms to achieve modularity in a generic, automated way.

Due to the need to process large models, some authors have proposed to split models for solving different tasks. For instance, Scheidgen and Zubow [10] propose a persistence framework that allows automatic and transparent fragmentation to add, edit and update EMF models. This process is executed at runtime, with considerable performance gains. However, the user does not have a view of the different fragments as we have in *EMF Splitter*, which could help improving the comprehensibility of the fragments.

Other works [6, 13] decompose models into submodels for enhancing their comprehensibility. For example, in [6], the authors 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. 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 [13]. Therefore, splitting models that belong to the same meta-model can produce different structures.

Other works directed to define model composition mechanisms [4, 5, 14] are intrusive. These papers [4, 14] present techniques for model composition and realize the importance of modularity in models as a research topic to minimise the effort. Strüber et. al [5] present a structured process for model-driven distributed software development which is based on split, edit and merge models for code generation.

Hence, altogether, while techniques for model modularization have been proposed in the context of MDE, to the best of our knowledge, *EMF Splitter* is unique in its way to generate structured model editors from meta-models.

5 Conclusions and Future Work

The MDE paradigm is gradually being established for the production of software, giving rise to the problem that if the systems are complex, they may lead to large models, making their management more difficult and increasing the development costs. Our goal is to provide developers with tools that help in defining the way to structure models, facilitating distributed development through division into layers, which improve comprehension. To achieve these objectives, *EMF Splitter* permits defining a modularity strategy and generates an Eclipse plug-in that allows developers to build their models in a structured way, as well as split existing models according to the defined modularity strategy.

In the near future, we plan to assess the performance of the tool when handling large models, compared with using a monolithic model. We are working on the idea of using Concordance[6], an indexing mechanism to manage and reconcile EMF references when models are updated or deleted [9]. We are also working on heuristics to propose good modularization strategies for a meta-model and a set of (large) models. In the long term, we plan to generate more advanced editors, e.g., graphical ones, as an alternative to the tree-editors for the units. Moreover,

[6] `http://www.eclipse.org/epsilon/doc/concordance/`

we would like to enhance our modularity pattern, e.g., by including scoping and access rules, to allow enabling or disabling references between model elements in units that belong to different packages, or to define visibility rules for elements inside units.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).

References

1. C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
2. P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context - Motorola case study. In *Proceedings of MoDELS'05*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2005.
3. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
4. F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *T. Aspect-Oriented Software Development VI*, 6:39–82, 2009.
5. P. Kelsen and Q. Ma. A modular model composition technique. In *Proceedings of FASE'10*, volume 6013 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2010.
6. P. Kelsen, Q. Ma, and C. Glodt. Models within models: Taming model complexity using the sub-model lattice. In *Proceedings of FASE'11*, volume 6603 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2011.
7. J. L. Lawall, H. Duchesne, G. Muller, and A.-F. L. Meur. Bossa nova: Introducing modularity into the bossa domain-specific language. In *Proceedings of GPCE'05*, volume 3676 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2005.
8. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
9. 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 *Proceedings of ECMFA'10*, volume 6138 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2010.
10. M. Scheidgen, A. Zubow, J. Fischer, and T. H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Proceedings of MoDELS'12*, volume 7590 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 2012.
11. T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.
12. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition.* Addison-Wesley Professional, 2008. See also `http://www.eclipse.org/modeling/emf/`.
13. D. Strüber, J. Rubin, G. Taentzer, and M. Chechik. Splitting models using information retrieval and model crawling techniques. In *Proceedings of FASE'14*, volume 8411 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2014.
14. D. Strüber, G. Taentzer, S. Jurack, and T. Schäfer. Towards a distributed modeling process based on composite models. In *Proceedings of FASE'13*, volume 7793 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2013.