Scalable model exploration through abstraction and fragmentation strategies

Antonio Garmendia**, Antonio Jiménez-Pastor, and Juan de Lara

Modelling and Software Engineering Research Group
htpp://www.miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract. Model-Driven Engineering (MDE) promotes the use of models to conduct all phases of software development in an automated way. However, for complex systems, these models may become large and unwieldy, and hence difficult to process and comprehend. In order to alleviate this situation, we explore the combination of model fragmentation strategies, to split models into more manageable chunks; and model abstraction and visualization mechanisms, able to provide simpler views of the models. The feasibility of this combination is confirmed based on an evaluation over a synthetic models, and the model sets of the GraBaTs'09 contest.

Keywords: Model-Driven Engineering, Model Scalability, Model Fragmentation, Model Visualization, Model Abstraction.

1 Introduction

Model Driven Engineering (MDE) promotes a model-centric approach for software development, where models are used to specify, design, test, and generate code for the final application. While models abstract details of the real system they represent, they may become large and unwieldy and therefore difficult to understand and process. Therefore, methods to cope with large models are key for a wider adoption of MDE in indutrial practice [6].

As a step in this direction, we present techniques, backed up by tools, for the scalable exploration and processing of large models. First, we show a method to specify strategies for fragmenting models. Taking inspiration from the way programming languages organize projects, our strategies organize a model as a project, which then can be divided into folders and files. Such strategies are specified over the meta-model, as "annotations" of the different classes [2].

Second, we present a method for the visual exploration of models. The method is based on filtering and abstracting models according to certain strategies, so that only a few nodes in the focus of interest are fully displayed, while others are aggregated into "abstract nodes". Then, different ways are provided

** Authors listed in alphabetical order.

to navigate through abstract nodes to the submodels they contain. Compared to fully representing a model on the screen, our approach permits higher space scalability (as fewer nodes are represented), but requires from algorithms to compute and navigate the abstractions.

We evaluate the approaches for large models and show how to combine them on the basis of two case studies. The first one is based on a synthetic generation of models, but based on a real case study of an EU project[1]. The second one is based on the large models (up to 5 million objects) provided by the GraBaTs'09 competition case study[2]. As a lesson from these experiments, we conclude that our visual exploration gives reasonable abstraction times (~ 2 secs.) for models up to roughly 10.000 objects. Beyond that point, even for a one-shot exploration, it is advisable to first fragment the model, and then apply the visual exploration.

The rest of the paper is organized as follows. Section 2 describes a method and tool support to define model fragmentation strategies. Section 3 introduces some techniques and support for model visualization and exploration. Section 4 evaluates the approaches with the two experiments. Section 5 compares with related research and Section 6 concludes.

2 Fragmenting models

We propose fragmenting models, following modular principles adopted by many programming languages and IDEs [2]. Therefore one model is organized as a Project. The model can then be fragmented into Packages (which are mapped to folders in the file system), which may hold Units (or these can be placed directly inside a project).

This kind of hierarchical organization permits structuring or defining different ways to fragment a model. Fragmentation strategies are specified at the meta-model level, where the different classes can be tagged as Project, Package and Unit, giving rise to different possible model organizations. Conceptually, the different model organizations are configured by instantiating the meta-model shown at the top of Figure 1, and then mapping such instantiation to the meta-model to which we want to apply the fragmentation strategy.

Figure 1 shows the application of the pattern to the Java JDTAST meta-model. In this case, the IJavaModel class is mapped to Project. The IJavaProject class is tagged as Package, this is possible because there is a composition relation from IJavaModel (the project) to IJavaProject, as the patterns demands by means of relation javaProjects. Another composition relation between IJavaProject and IPackageFragmentRoot allows classes which inherit from the latter (BinaryPackage-FragmentRoot and SourcePackageFragmentRoot) be tagged as Package. Finally, both IClassFile and ICompilationUnit are instantiated as Unit.

We have built tool support to apply such fragmentation strategies and to produce a modelling environment that splits monolithic instances of the meta-model according to the fragmentation strategy and supports the creation of mod-

[1] `http://mondo-project.org`

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

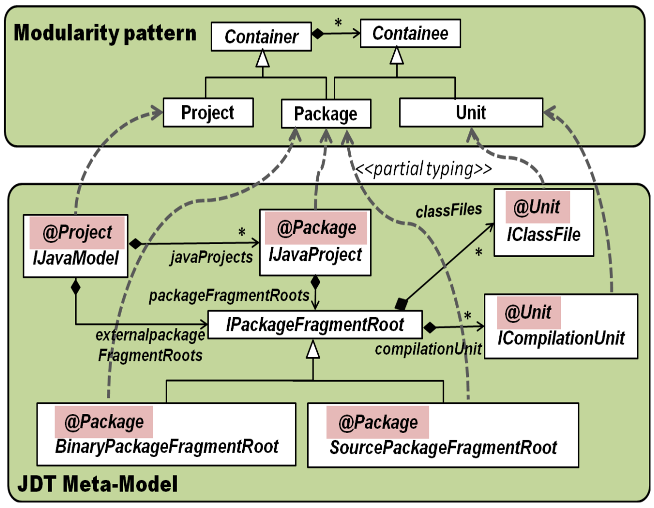


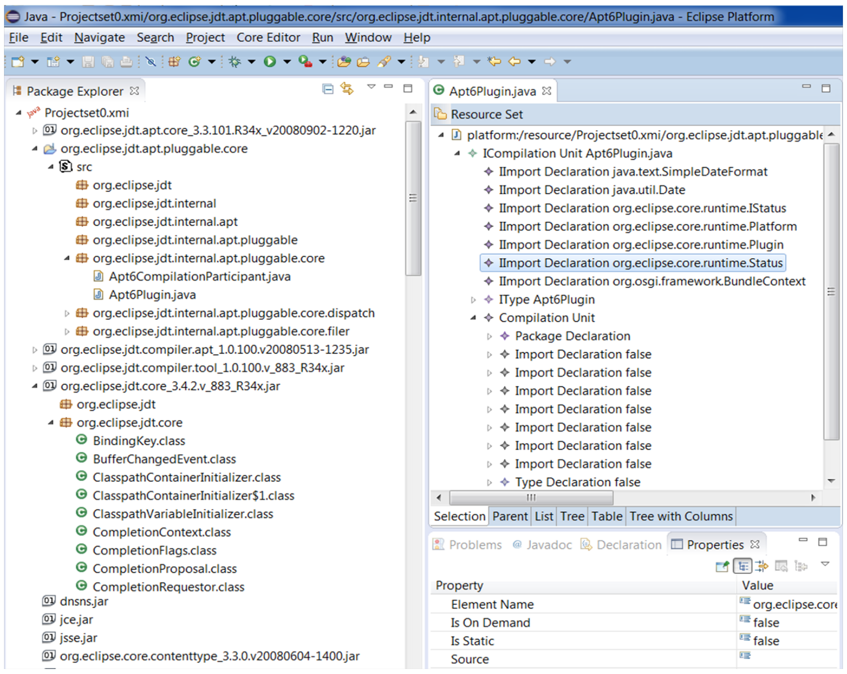
Fig. 1: Pattern to describe the modular structure of a meta-model (top). Application to the JDTAST meta-model (bottom).

els according to such organization. Our tool is called EMF-Splitter, it is built atop of Eclipse and freely available at `http://antoniogarmendia.github.io/EMF-Splitter/`. Figure 2 shows the generated modelling environment. The environment shows an Eclipse project, named Projectset0, created from the model set0.xmi of the GraBaTs'09 contest. The project explorer shows the structure of folders and files generated from the model, which follows the specified fragmentation strategy. To the right, a tree editor shows the content of one file. The original model has about 70.000 model elements, while the fragmentation strategy fragments it into 1.800 files.

3 Exploring models

When working with models it is very useful to explore them to get some insight using our intuition, to analyse its different parts, or to find unusual or interesting features. However, big models are impossible to be completely represented in a computer monitor. Exploring models through the default tree editor of EMF is also cumbersome, as it lacks facilities to visualize, search and navigate. Moreover, many times, models lack a dedicated graphical editor providing visualization and exploration services.

To solve these problems we have developed a tool called SAMPLER (ScAlable Model exPLorER). It offers several features to visualize big models in the form of graphs, such as focusing on a specific point of the graph or making some general abstractions over the model before painting it. It allows to navigate along the visualization and even search a node through the whole model. The tool permits exploring models for which no concrete syntax has been defined, as it uses a default graph-based representation.


```
Java - Projectset0.xmi/org.eclipse.jdt.apt.pluggable.core/src/org.eclipse.jdt.internal.apt.pluggable.core/Apt6Plugin.java - Eclipse Platform
File  Edit  Navigate  Search  Project  Core Editor  Run  Window  Help

Package Explorer
  Projectset0.xmi
    org.eclipse.jdt.apt.core_3.3.101.R34x_v20080902-1220.jar
    org.eclipse.jdt.apt.pluggable.core
      src
        org.eclipse.jdt
        org.eclipse.jdt.internal
        org.eclipse.jdt.internal.apt
        org.eclipse.jdt.internal.apt.pluggable
        org.eclipse.jdt.internal.apt.pluggable.core
          Apt6CompilationParticipant.java
          Apt6Plugin.java
        org.eclipse.jdt.internal.apt.pluggable.core.dispatch
        org.eclipse.jdt.internal.apt.pluggable.core.filer
    org.eclipse.jdt.compiler.apt_1.0.100.v20080513-1235.jar
    org.eclipse.jdt.compiler.tool_1.0.100.v_883_R34x.jar
    org.eclipse.jdt.core_3.4.2.v_883_R34x.jar
      org.eclipse.jdt
      org.eclipse.jdt.core
        BindingKey.class
        BufferChangedEvent.class
        ClasspathContainerInitializer.class
        ClasspathContainerInitializer$1.class
        ClasspathVariableInitializer.class
        CompletionContext.class
        CompletionFlags.class
        CompletionProposal.class
        CompletionRequestor.class
    dnsns.jar
    jce.jar
    jsse.jar
    org.eclipse.core.contenttype_3.3.0.v20080604-1400.jar

Apt6Plugin.java
Resource Set
  platform:/resource/Projectset0.xmi/org.eclipse.jdt.apt.pluggable
    ICompilation Unit Apt6Plugin.java
      IImport Declaration java.text.SimpleDateFormat
      IImport Declaration java.util.Date
      IImport Declaration org.eclipse.core.runtime.IStatus
      IImport Declaration org.eclipse.core.runtime.Platform
      IImport Declaration org.eclipse.core.runtime.Plugin
      IImport Declaration org.eclipse.core.runtime.Status
      IImport Declaration org.osgi.framework.BundleContext
      IType Apt6Plugin
      Compilation Unit
        Package Declaration
        Import Declaration false
        Import Declaration false
        Import Declaration false
        Import Declaration false
        Import Declaration false
        Import Declaration false
        Import Declaration false
        Type Declaration false
Selection | Parent | List | Tree | Table | Tree with Columns

Problems  Javadoc  Declaration  Properties
Property          Value
Element Name      org.eclipse.core
Is On Demand      false
Is Static         false
Source
```

Fig. 2: Generated environment.

The main goal of SAMPLER is to draw the model without painting every element on the screen. For this purpose we have developed a composition strategy where we combine different kinds of abstractions which, executed in sequence, give a fast and compact view of the model. There are two basic operations in SAMPLER to make a model more readable: removing elements from the view, and grouping some elements in a big composite element. When we decide to remove an element, we can not view it when exploring the model, but when we compact some elements into one bigger node, we can expand it and explore the smaller elements as we wish. These operations are applied in three different steps:

– **Filters**: the first step to make our visualization easier is to apply some kind of *filter*. In many EMF models there are many intermediate objects, which may not provide the user with meaningful information, but they are technically needed to make the model conform to the meta-model. Hence, SAMPLER provides mechanism to select and filter those undesired objects, removing them from the view. When an object is filtered out, its incoming references are composed with its outgoing ones, so that the connectivity of the model is preserved.
– **Global abstraction strategies**: after filtering, there are others groups of elements which may share properties of interest, and hence it makes sense to cluster them into abstract nodes. This kind of composite operation is what we will call *global abstraction*. There are many possibilities to create a global

abstraction strategy. For example, we can unify the leaves of the containment tree of the model or we can use some cluster algorithms (like k-means).
- **Local abstraction strategies**: the last step of the SAMPLER abstraction strategy is what we call *local strategies*. After applying the previous steps, we may still have thousands of elements to draw, so that it is impossible to read anything on the screen. In this case, our approach is to focus on a part of the model at once. The local strategies focus the visualization in some point of the model (a set of elements that the user can choose), fully displaying the elements around that point, and compacting the other elements into abstract nodes. These abstract nodes do not take much space in the screen, but are explorable.

These three kinds of steps are put together to create visualization algorithms, which create a drawing of the model. In SAMPLER, we offer some basic algorithms, like just performing a global abstraction, or just a local one. These basic algorithms can be composed, and new algorithms can be incorporated by implementing a dedicated extension point.

Further than the visualization algorithms, SAMPLER provides a navigation utility. It allows, once a model has been painted on the screen, to navigate through the model. There are three navigation options:

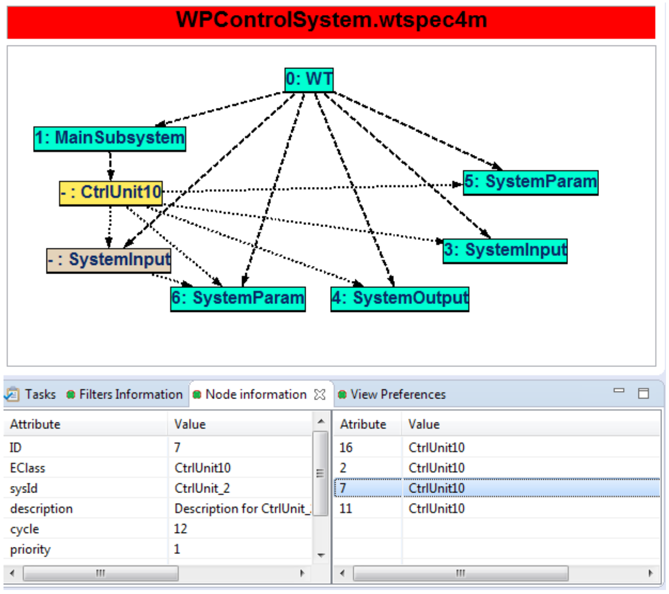
- It is possible to expand a compacted abstract node so that, in the same screen, the first elements of the abstract node are shown together with the others elements present in the view.
- It allows to open an abstract node in another window and apply a common visualization algorithm to view this part of the model.
- It is also possible to open a new window with the containment subtree of an element of the model. As in the last option, in this new window, a common algorithm can be used to view the subtree.

Finally, SAMPLER offers a search functionality. It uses the filters algorithms described before, and allows to dynamically define different criteria for searching.

All these functionalities and tools have been implemented in a Eclipse plug-in available at `http://rioukay.github.io/sampler/`. The main elements included in the plug-in are the different Eclipse views (see Figure 3):

- The *View Preferences* view allows to change the configuration of the visualization algorithm that is being using at that moment. It also gives the possibility to change between the existing visualization algorithms.
- The *Node Information* view shows the information of the elements that have been clicked on in the canvas. If the node clicked is an abstract node, then it shows the information of its contained elements.
- The *Filter Information* view allows to add and configure additional filter steps to the end of the algorithm.

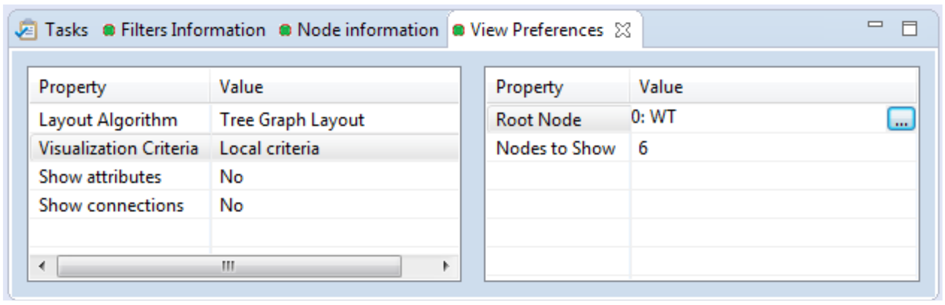
Figure 3 shows an example of how SAMPLER works. We can see the diagram visualization of the model where we have applied a local algorithm that shows the



Attribute	Value
ID	7
EClass	CtrlUnit10
sysId	CtrlUnit_2
description	Description for CtrlUnit_
cycle	12
priority	1

Atribute	Value
16	CtrlUnit10
2	CtrlUnit10
7	CtrlUnit10
11	CtrlUnit10

Fig. 3: Exploring a model with SAMPLER



Property	Value
Layout Algorithm	Tree Graph Layout
Visualization Criteria	Local criteria
Show attributes	No
Show connections	No

Property	Value
Root Node	0: WT
Nodes to Show	6

Fig. 4: Preview of the “Preferences View” of SAMPLER

root of the model (node *WT*) and the five nearest elements of the containment tree. The other elements are compacted according to their parents in that tree.

Each box in the diagram represent an element of the abstracted model. The blue boxes correspond to simple elements of the model and the brown ones are abstract nodes. There are two kinds of arrows connecting the nodes of the diagram: dot arrows represent references, and line arrows represent containment. Just below the canvas, we can see the views that we have described. Figure 3 shows the “Node Information” view. To the right we see the elements of the model contained in the selected compacted node (the yellow box in the diagram). To the left we show the attributes of the element selected on the right side of the view.

Figure 4 shows how the “View Preferences” view looks like. To the left we give the option to modify the generic options of the visualization and choosing the abstraction algorithm. To the right, we allow changing the configuration of the algorithm. In the example, with the local algorithm, we can choose how many elements to show near the root, and which element of the model is the root of the visualization.

4 Evaluation

Next, we evaluate the performance of our tools to deal with large models. Our intention is to analyse to what extent large models can be explored with SAMPLER. When models become difficult to be visualized with the tool, we will fragment them first, using a fragmentation strategy, so that the smaller chunks can be visualized individually. Hence, we also perform an experiment to give an account for the incurred cost of fragmentation.

In all our tests, we used the following environment:

- Execution environment:
  - Operative System: Windows 7 Professional Service Pack 1.
  - Processor: Intel(R) Core(TM) i7-2600, 3.40GHz
  - RAM: 12 GB
- Java Virtual Machine Configuration:
  - Execution environment: Java SE 1.8 (*jre1.8.0_40*)
  - Initial memory: 512 MB
  - Maximum memory: 8 GB

4.1 Exploration performance

In this experiment, the goal is to check the performance of some of the SAMPLER abstraction strategies for large models. We generated models using an EMF random instantiator from the ATLANMOD team[3]. We used a meta-model taken from a case study of the EU project MONDO[4] in the domain of component-based embedded systems. We created 500 test models of each size. The sizes we have tested go from 100 to 6.000 model elements.

In each test, we have taken four measures, the time taken to read the model, and the time of execution of three of SAMPLER basic algorithms. Those algorithms are:

- A global algorithm that creates only one composite node with all elements inside it. This is a measure of how much time SAMPLER takes to explore the whole model (compactification algorithm)
- A global algorithm that explores the whole model detecting the leaves of the containment tree and compact them (global algorithm)
- A local algorithm that, given an object of the model, shows this element and n of its neighbours while the others are compacted (local algorithm).

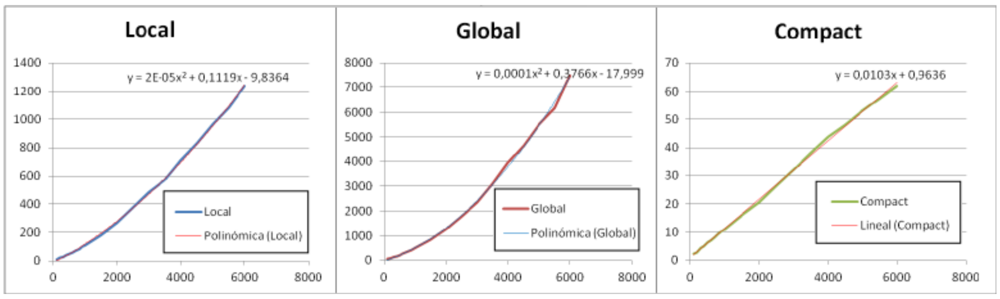


Fig. 5: Performance (ms) of the different algorithms of abstraction. From left to right: local algorithm, global algorithm and compactification algorithm.

The graphics in Figure 5 show that every algorithm takes a reasonable time to execute (no more than 10 seconds for 6.000 elements in the model) and that the local and global algorithm takes a quadratic polynomial time to execute.

After this synthetic tests, we have executed the same algorithms in the same conditions over the two first sets of JDTAST models of the GRaBaTs competitions, which have a larger size. Table 1 shows the results of the experiment for the three algorithms together with the estimation from the run of the smaller tests. As it can be noted, the time required to create the abstraction of the model is more than 25 minutes with the *set0* model and more than 5 hours with the *set1* model. Those times are not acceptable, and hence we resort to the application of another pre-drawing techniques, such as *fragmentation strategies*. The next subsection discusses on its performance.

Model	**Local Algorithm**		**Global Algorithm**		**Compactification**	
	Measure	Estimation	*Measure*	Estimation	*Measure*	Estimation
set0	1.527.543, 80	82.280, 29	1.224.024, 6,	786.525, 93	778, 8,	745, 04
set1	20.596.201	611.014, 12	13.689.961, 00	6.126.755, 18	2.080, 00	2.096, 68

Table 1: Performance (ms) of SAMPLER over some JDTAST models.

4.2 Fragmentation performance

Next, we evaluate the performance of model fragmentation. Figure 1 shows the fragmentation strategy that was applied to the JDTAST meta-model. After the application of the modularity pattern, we split all the models found in the GraBaTs'09 case study, turning each one of them into an Eclipse project.

Table 2 shows the results of our experiment. The columns depict 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,

[3] `http://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/`
[4] `http://www.mondo-project.org`

Model	Split time	Merge time	# files	Avg	Max	# model elements
set0	94.362, 04	7.808, 04	1.779	40, 17	1.322	71.458
set1	231.143, 78	37.847, 10	6.240	32, 68	4.549	203.938
set2	544.609, 02	83.905, 74	6.050	345, 27	50.718	2.088.890
set3	747.739, 30	199.811, 98	4.460	1.031, 24	50.718	4.599.358
set4	808.351, 00	511.554, 59	5.068	980, 04	50.718	4.966.846

Table 2: Performance (ms) of EMF-Splitter over the test-cases of GraBaTs'09.

and the total number of elements in the whole model. We can observe that the maximum number of elements in a file is repeated for *set2*, *set3* and *set4*. This happens because 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.

The results shows that, in average the exploration of the files with SAMPLER would become easier, because the largest average size is about 1.000 elements (which are easily explorable), while the maximum number of elements in a file is 50.718, which would take about a minute and a half to load.

5 Related work

In this section we focus on existing works dealing with model fragmentation, and model exploration and visualization of large graphs.

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 [5, 11] decompose models into submodels for enhancing their comprehensibility. For example, in [5], 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 [11]. Therefore, splitting models that belong to the same meta-model can produce different structures.

Other works directed to define model composition mechanisms [3, 4, 12] are intrusive. These papers [3, 12] present techniques for model composition and realize the importance of modularity in models as a research topic to minimise the effort. Strüber et. al [4] present a structured process for model-driven distributed software development which is based on split, edit and merge models for code generation.

Regarding model visualization, in [9], the authors propose a framework call ELVIZ for model visualization, based on the transformation of input models to appropriate output formats. For example, given a class diagram, they can extract the number of methods per class, and visualize such numbers as a bar chart. ELVIZ facilitates the generation of input models to different visualization outputs relying on mappings.

In [1], the authors present the tool Explen, which uses slicing techniques in order to visualize large meta-models. Similar to our approach, it is possible to focus on a given class, and select some slicing criteria (e.g., show the composition relations only, show only a certain radius of classes, or show the sub/super type hierarchy). They also include a flattening filter, which presents a hierarchy in the form of a unique class. SAMPLER supports the visualization of models and meta-models, and the abstractions/slice criteria are extensible. Moreover, we support different navigation strategies from abstracted models.

The analysis of large graphs arising in e.g., social networks have produced some summarization techniques, which try to encode in smaller graphs [7] or as a variety of statistics [8] the main features of the large graph. For this purpose, they find the most often occurring subtype graphs (cliques, starts, chains, etc) in graphs. In the context of MDE, this information is encoded in the meta-model. Other methods are more flexible, as they allow customization of the interesting attributes of nodes [13], and nodes with similar values are summarized in a single node. This would be similar to SAMPLERs global abstractions.

Altogether, to the best of our knowledge, our approach to combine model fragmentation and model visualization techniques is novel.

6 Conclusions and future work

In this work, we have proposed the combination of model fragmentation and model visualization 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. We have performed an evaluation of the approach for large models. We have seen that for models in the range of up to roughly six thousand elements, abstraction gives good results. For large models, such as those of the GraBaTs'09, our proposal is fragmenting them first. In this case, fragments become of manageable size, and then can be visually explored.

In the future, we aim at the tighter integration of SAMPLER with the information provided by the fragmentation strategies. In particular, when exploring a fragmented model, we currently need to use the package explorer to move between fragments. In the future, we would like SAMPLER to use the fragmentation information as a (global) abstraction algorithm. This way, fragments would be explored transparently from within the SAMPLER visual canvas.

Acknowledgements. Work supported by the Spanish Ministry of Economy and Competitivity (TIN2011-24139, TIN2014-52129-R), the EU commission (FP7-ICT-2013-10, #611125) and the Community of Madrid (S2013/ICE-3006)

References

1. A. Blouin, N. Moha, B. Baudry, and H. A. Sahraoui. Slicing-based techniques for visualizing large metamodels. In *Second IEEE Working Conference on Software Visualization, VISSOFT*, pages 25–29. IEEE Computer Society, 2014.
2. A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF splitter: A structured approach to EMF modularity. In *XM@MoDELS*, volume 1239 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org, 2014.
3. F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *T. Asp.-Oriented Soft. Dev. VI*, 6:39–82, 2009.
4. P. Kelsen and Q. Ma. A modular model composition technique. In *Proceedings of FASE'10*, volume 6013 of *LNCS*, pages 173–187. Springer, 2010.
5. 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 *LNCS*, pages 171–185. Springer, 2011.
6. D. S. Kolovos, L. M. Rose, N. 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 *Proc. BigMDE '13*, pages 2:1–2:10, New York, NY, USA, 2013. ACM.
7. D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. VOG: summarizing and understanding large graphs. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 91–99. SIAM, 2014.
8. M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
9. M. Ostendorp, J. Jelschen, and A. Winter. ELVIZ: A query-based approach to model visualization. In *Modellierung 2014*, pages 105–120, 2014.
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 *LNCS*, pages 102–118. Springer, 2012.
11. 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 *LNCS*, pages 47–62. Springer, 2014.
12. 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 *LNCS*, pages 6–20. Springer, 2013.
13. Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 567–580. ACM, 2008.