

# Scalable Model Exploration for Model-Driven Engineering

Antonio Jiménez-Pastor<sup>a</sup>, Antonio Garmendia<sup>a</sup>, Juan de Lara<sup>a</sup>

<sup>a</sup>*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 combine model fragmentation strategies – to split models into more manageable chunks – with model abstraction and visualisation mechanisms, able to provide simpler views of the models. In this paper, we describe the underlying methods and techniques, as well as the supporting tools. The feasibility and benefits of our approach are confirmed based on evaluations in the embedded systems, and the reverse engineering domains, where large benefits in terms of visualisation time (speeds up of up to 55×), and reduction in memory consumption (reduction of 97%) are obtained.

*Keywords:* Model-Driven Engineering, Model Scalability, Model Fragmentation, Model Visualisation, Model Abstraction

---

## 1. Introduction

Model Driven Engineering (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 [39]. While models abstract details of the real system they represent, they may become large and unwieldy and therefore difficult to understand

---

*Email addresses:* antonio.jimenezp@estudiante.uam.es (Antonio Jiménez-Pastor), antonio.garmendia@uam.es (Antonio Garmendia), Juan.deLara@uam.es (Juan de Lara)

and process. This way, large models are problematic from a human point of view (difficult to explore and comprehend), and from a machine point of view (large models may not fit in memory, and may become a bottleneck for model management operations). Hence, methods to cope with large models are necessary for a wider adoption of MDE in industrial practice and its use in more complex scenarios meeting today’s need of increasing scale [25].

In this paper we present techniques, backed up by tools, for the scalable exploration and processing of large MDE models. First, we show a method to specify strategies for fragmenting models. Most current tools handle models as monolithic entities, and hence suffer from scalability problems when the model becomes large. For example, they are unable to load the whole model in memory. Instead, taking inspiration from the way programming languages organise projects, our strategies organise 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 [19].

Second, we present a method for the visual exploration of models. The method represents models in a graph-based way, using a generic concrete graphical syntax. It 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 to navigate through abstract nodes to the submodels they contain. The tool is able to use the defined fragmentation strategies in order to efficiently navigate through the model structure without the need to show all the model content. Compared to fully representing a model on the screen, our approach allows higher space scalability (as fewer nodes are represented), but requires algorithms to compute and navigate the abstractions. While the main focus of the tool is on models built using the Eclipse Modelling Framework (EMF) [40], which is the de-facto standard modelling technology nowadays, the tool accepts other model formats as well (e.g., GraphML [8]) and is open and extensible in this sense.

As a running example, we use the Knowledge Discovery Meta-model (KDM) [30]. KDM is an OMG standard in the reverse engineering domain, used as a common intermediate representation for existing software systems and their operating environments. For experimental evaluation, we use two other meta-models. The first one is based on a synthetic generation

of models, but based on a real case study of an EU project<sup>1</sup> in the embedded systems domain. The second experiment is also in the reverse engineering domain. It is based on the large models (up to 5 million objects) provided by the GraBaTs'09 competition case study<sup>2</sup>, which consists in representing Java programs as models.

As a lesson learnt from these experiments, we conclude that our visual exploration gives reasonable abstraction times ( $\sim 2$  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. This shows the power of combining model fragmentation and abstraction. We have also observed that large benefits in terms of visualisation time (speed ups of up to  $55\times$ ), and number of objects that need to be loaded (reduction of about 97%) are obtained by using fragmentation in comparison to using the EMF default tree editor over the original monolithic models. We also discuss the benefits that model fragmentation brings to large-scale graph visualisation tools like Gephi [4], and to model repositories and persistence backends like Connected Data Objects (CDO) [12].

This paper is an extended version of [20], where we provide a detailed description of the abstraction and exploration algorithms, we have applied our approach to KDM models, and show additional experiments. The approach has also been improved, providing a tighter integration of both tools, so that it is possible to use the fragmentation strategies as a help in the navigation; supporting staged exploration of fragmented models via proxies; making the exploration independent of the underlying modelling technology, and improving extensibility.

The rest of the paper is organised as follows. Section 2 motivates our approach and provides an overview of its realization. Section 3 describes our method to define model fragmentation strategies. Section 4 introduces some techniques for model visualisation and exploration. Section 5 describes tool support. Section 6 evaluates the approach. Section 7 compares with related research and Section 8 concludes the paper.

---

<sup>1</sup><http://mondo-project.org>

<sup>2</sup>[http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs\\_2009\\_Case\\_Study](http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study)

## 2. Motivation and overview of the approach

### 2.1. Motivation

Several scenarios require exploration and visualisation of large MDE models. Generally, these scenarios are a form of so-called graph *sensemaking* [34], where large graphs are analysed, in order to gain insights of the underlying data, learn about the problem domain or detect anomalies [47]. Large models can arise from the extraction of graph-based data from many sources like social networks, network traffic, intelligence analysis or on-line auctions [34].

In MDE, large models are common in model-based reverse engineering [9], where the source code of a legacy system is parsed and transformed into a model conformant to a meta-model. Even for small programs, the resulting model may become extremely large. This is because the abstract syntax tree of the source code (and possibly other artefacts, like binaries or configuration files) needs to be represented as a model, while many times the inherent modularization mechanisms of the programming language (e.g., division in files) are not reflected at the model level, yielding a large monolithic model. Some studies [9] report a multiplier of  $4\times$  or  $5\times$  between LOCs and number of model elements, and a factor of about  $400\times$  in storage size [37]. These studies also describe the difficulties of loading monolithic models of such size into memory.

Meta-models for complex domains may also become large. For example, the UML meta-model [31] has about 240 classes and more than 580 properties. Hence, approaches especially targeted to meta-model visualisation and comprehension have been proposed [7]. Visualisation has also been employed to help understanding and maintaining model transformations [35], or chains of transformation executions [46]. Large model visualisation and exploration has been used to detect anomalies in models in different domains [17]. However, even though some tools have been proposed for specific tasks like transformation or meta-model visualisation, there is a lack of approaches and tools directed to generic visualisation of large models, independent of their meta-models.

### 2.2. Overview

Figure 1 provides an overview of our approach for scalable model exploration. Given a model, the first step is to decide whether this model is too large to be explored as it is, or if it is more efficient to partition the model into more manageable chunks. If that is the case, the second step is



Once the strategy is specified, the model is split (label 3), resulting in a set of model fragments, related to each other via cross-references. A fragmented model is more efficient to explore, since the whole model does not need to be loaded and processed for its visualisation, but only one fragment at a time. Fragmenting models is performed using the EMF-Splitter tool [19], and will be explained in Section 3. The experiment described in Section 6.2 provides some guidelines on when a model would be too large to be manageable. Fragmentation strategies are not only used to split existing models, but also to enforce such organization on newly created models.

5

first case, we lose the possibility of using file-based storage for version control, which allows uniformity of the version control system with the project code-base. In addition, by using fragmented models we reduce possible conflicts due to concurrent modifications. Model indexers are useful to speed up model searches, but would not solve the problem of physically loading a very large model in memory.

Given a monolithic model, or a fragmented one, our approach provides different abstraction strategies and filters for its efficient visualisation (label 4 in the figure). The idea is not to show the whole model (or fragment) on the screen, but just to display a small region of interest, while the rest of the model is shown in an abstracted way, or is completely eliminated through a filter. An abstracted model can be navigated (label 5) using a hyperlink metaphor. This way, an abstract node representing a submodel can be expanded in the same view as the current model, or in a different one. Similarly, cross references between different model fragments can be navigated through proxy nodes (i.e., a virtual node representing a real node in a different fragment, which is currently not loaded in memory). Additionally, the structure in the file system (i.e., the folders and files) created by the fragmentation strategy on the model can be used for its efficient navigation. The abstraction and navigation strategies are explained in Section 4. Tool support for fragmentation and exploration is described in Section 5.

### 3. Fragmenting models

Instead of working with large, monolithic models, we propose fragmenting them, following modular principles adopted by many programming languages and IDEs [19]. For example, a Java program is normally not built within a single file, but organised into projects, (nested) packages and different compilation units (files) within them. These modularity concepts are provided by the programming language itself. Therefore, our proposal is to adapt these principles to the construction of models. This way, models will no longer be monolithic, but organised as a *Project*. The model can then be fragmented into *Packages* (which are mapped to folders in the file system), which may hold *Units* (which are mapped into files of the file system). Alternatively, *Units* can be placed directly inside a project.

This kind of hierarchical organization allows 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*,

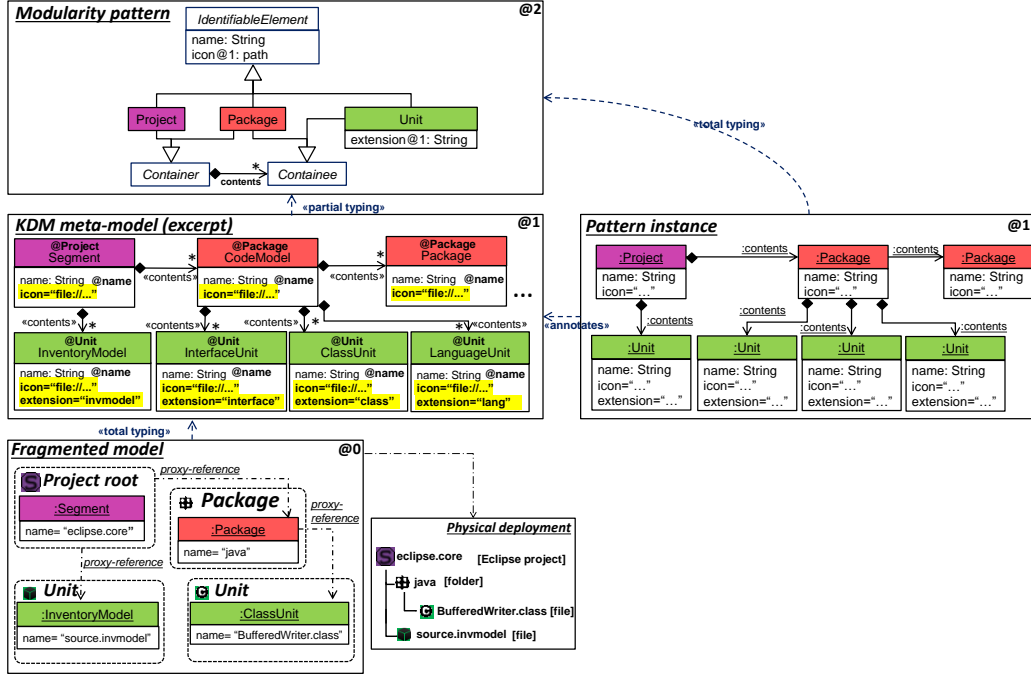


Figure 2: Pattern to describe the modular structure of a meta-model (top). Instantiation of the pattern and application to the KDM meta-model (middle). A structured model and its physical deployment (bottom).

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 2 (labelled “modularity pattern”), and then mapping such instantiation to the meta-model to which we want to apply the fragmentation strategy.

The figure shows one instance of the pattern meta-model (labelled “pattern instance”), made of a Project instance, two nested Package instances, and four types of Units. While the first type of unit can be placed directly inside the project, the other three are inside the first package. Actually, this is the fragmentation strategy that we have created for KDM [30]. KDM is a standard specification of the Object Management Group (OMG) that is widely used in software modernization projects [10]. 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 soft-

ware 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.

We have tagged the elements in the KDM meta-model with the role names of the modularity meta-model they have been assigned. It can be observed that the root class of the KDM meta-model (class `Segment`) is mapped to `Project`. In EMF [40], meta-models normally contain a root class. This class is instantiated once in every model, and every object in such model is contained directly or indirectly in such object. Therefore, when such class is instantiated, we will create a `Project` as a side effect. Then, classes `CodeModel` and `Package` are marked as `Package`. This means that, when such classes are instantiated, we will create a folder in the file system. Finally, the classes `InventoryModel`, `InterfaceUnit`, `ClassUnit` and `LanguageUnit` are marked as `Unit`. Therefore, when we instantiate such classes, we will create a file, containing all objects of classes included (i.e., reachable by composition references) directly or indirectly in them.

Conceptually, the instantiation of the fragmentation strategy over the particular meta-model (KDM in this case), can be explained using multi-level modelling [2]. This is an approach enabling modelling at an arbitrary number of meta-levels, not necessarily two. Multi-level modelling allows characterizing features of objects at lower meta-levels, not only at the next one, by the use of potency. The potency is a natural number (or zero) attached to all model elements, which indicates at how many meta-levels the element can be instantiated (if not specified, the element takes the potency of its enclosing container).

The modularity pattern can be instantiated twice, at two subsequent meta-levels, and hence has potency 2 (indicated using the notation @2 in the figure). However, the `icon` and `extension` fields have potency 1 (shown using the notation @1), so that they can be instantiated just once, and hence they receive a value at the next meta-level. The `extension` field holds the extension name of the associated fragment file, while the `icon` contains a path to the file icon. For example, the `InventoryModel` has as extension `invmodel` and has been given an icon, as shown in the box labelled “physical deployment”. Please note that while these fields with potency 1 produce extensions to the KDM meta-model (slots marked in yellow), the elements with potency 2 (like `name` or `Project`) simply require a mapping from some compatible element in the KDM meta-model. This can be seen as a partial a-posteriori typing of the KDM meta-model with respect to the modularity pattern [15, 33]. It is a



partial typing, because not every element of the KDM meta-model receives a typing with respect to the modularity pattern. For example, the elements inside a `Unit` do not receive a type. In practice, to represent this orthogonal typing, we instantiate the pattern meta-model and annotate the KDM meta-model with this instance. The annotation is realised as a separate model (an annotation model), with cross-references to the meta-model, as shown in Figure 2.

In our implementation, when creating a project we create a (hidden) file, which contains the root object, and similar for packages. All such model fragments are related to each other via proxy nodes and cross-references. These are normal model references, but which span different files. A schema of the obtained modular structure of a model is shown at the bottom of Figure 2 (boxes “Fragmented model” and “Physical deployment”).

Details on tool support for this fragmentation approach will be given in Section 5.1.

#### 4. Exploring models

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 completely 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, the current situation in EMF is that many times models are visualised using the tree editor, which difficults their comprehensibility. This is so 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, we have developed a new framework that offers 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.

The first step is to simplify a large graph<sup>3</sup>. If this graph has extra information about its allowed structure (meta-data, e.g., via a meta-model), it will be useful for making such simplification. Our proposal is based on the composition of small operations (called *Simplification Operations*) that, after a finite number of steps, obtain the desired simpler graph representation. Hence, after each step in the simplification, we will obtain a new graph having fewer vertices than the previous graph, but preserving the information contained in the original graph. These operations can be linearly combined into *Nested Simplification Algorithms*. Once we have used one of those algorithms, the resulting graph is displayed using some graph layout algorithm. Though our framework allows creating and using different layout algorithms in an extensible way, layouting is not the main focus of this paper. Finally, it is possible to navigate through the graph, using what we call *Exploration Operations*.

The rest of this section is organised as follows. In Section 4.1 we detail the simplification operations currently available, while in Section 4.2 we describe the exploration operations.

#### 4.1. Simplification Operations

The simplification operations are the key elements of our approach to explore graphs. When composed, they form a *nested simplification algorithm*, which applied to a complex graph produces a simpler and more readable graph. This simpler graph is easier to explore using the two exploration operations described in Section 4.2.

We classify the possible simplification operations in three types:

- **Filters.** These operations are based on deleting vertices. A filter selects a collection of vertices and removes them from the graph. To keep the information of those vertices, the filter creates new edges. Hence, as shown in Figure 3, if the filter deletes a node **b**, it creates an edge between each node **a** that has an edge reaching **b**, and each node **c** receiving an edge from **b**. We call them “abstract edges”, because they do not belong to the original graph, but they represent several original

---

<sup>3</sup>In the following we use graph and model interchangeably.

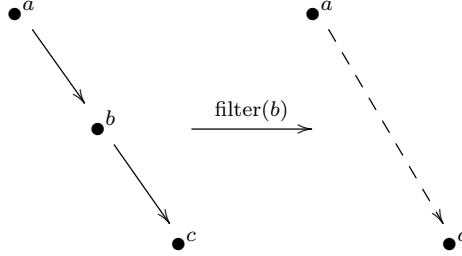


Figure 3: Filter operation over a graph.

edges. In practice, we depict such edges differently, to avoid confusion with the original graph edges. Several filters can be defined over the same graph. Indeed, any algorithm that selects and removes vertices should be considered as a filter. For instance, we can filter vertices by their type, by their name (as in case of Figure 3), or by the number of neighbours, among many other criteria.

- **Global abstractions.** This strategy aggregates together vertices that have some relation. Hence, they detect a *pattern* in the graph and, in the new graph, they create just a vertex to represent such pattern. For instance, if we have a tree, we may not want to represent all leaves, but just convey that a vertex has some children, and hence the children are aggregated in one node. Another example is applying a clustering algorithm (like  $k$ -means) [36] over the graph and collect together the vertices in the same cluster. In such case, the graph would be represented as a collection of (connected) abstract vertices. These abstract vertices can have abstract edges between them, one abstract edge representing one or more edges between concrete nodes inside each abstract node.
- **Local abstractions.** This kind of abstractions follows the same idea as the *global abstractions*, adding in the new graph “abstract”<sup>4</sup> vertices that represent a collection of vertices of the previous graph. The difference with global abstractions is that local abstractions search the pattern around a specific vertex of the graph. A possible local strat-

---

<sup>4</sup>We use *abstract vertex* and *aggregated vertex* to refer to vertices abstracting a part of the model

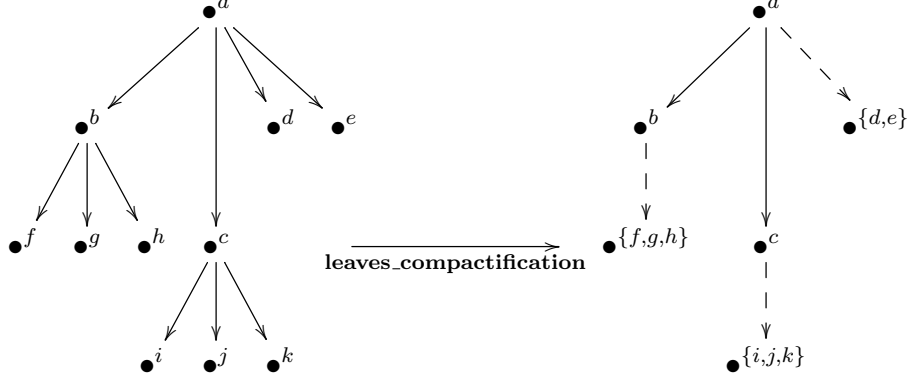


Figure 4: Leaves Compactification over a graph.

egy is to focus on a vertex, i.e., create a new graph with a vertex and its neighbours, while all the other vertices are aggregated in a single vertex.

Next, we give some examples of simplification operations.

- **Leaves Compactification:** this is a *global abstraction* that considers the graph as a tree and aggregates together the leaves of the tree, differentiating the leaves according to their parents. If the graph to be represented is an EMF model, then the tree structure can be extracted from the containment relations between objects. Figure 4 shows an example of this operation, where the children of nodes  $b$  and  $c$  are added together in two nodes, and the leaf nodes of  $a$  ( $d$  and  $e$ ) are aggregated in another node. Concrete edges with source or target an abstract node are aggregated into an abstract edge (e.g., edge between  $b$  and  $\{f, g, h\}$ ). In practice, the aggregated nodes are not assigned a label made of the concatenation of the inner nodes. Instead, they are unlabelled, and their contents can be inspected via exploration operations.
- **Bi-Clustering:** this is a *global abstraction* that applies the Cheeger Cut of the graph to separate it in two balanced pieces [11]. Essentially, this algorithm divides the vertices in two sets, minimizing the number of edges to/from nodes belonging to different sets. Hence, the end result is a graph with two abstract vertices. An example of this operation is

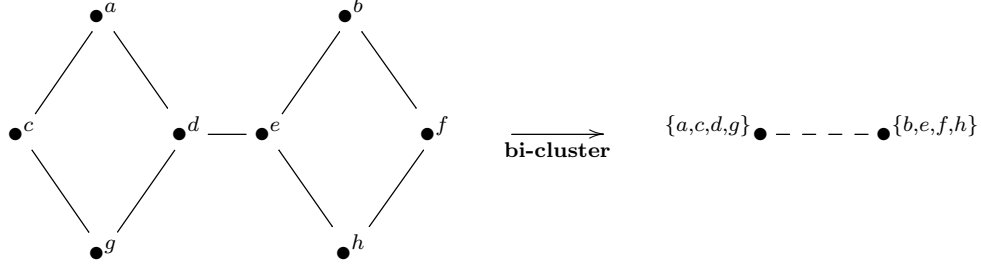


Figure 5: Cheeger Cut over a graph.

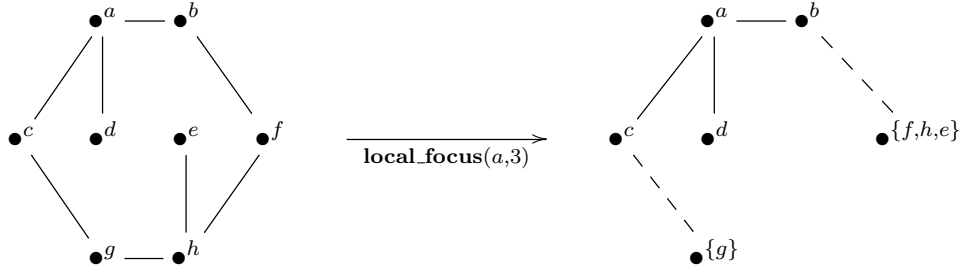


Figure 6: Local Focus on a graph.

shown in Figure 5. Applying this abstraction allows us to know if the model we are visualizing is strongly connected [11].

- **Local Focus:** this is a *local abstraction*. It receives two parameters: a vertex of the graph and a positive integer  $n$ , which is the number of original vertices that will remain in the abstraction. This operation uses the vertex given as root of the graph and makes a breadth first search until it finds  $n$  vertices. These  $n + 1$  vertices are added to the new graph and the rest are put together in a single node according to their parents in the breadth search. An example of the operation is shown in Figure 6, where the operations leaves three neighbours of a, aggregating the other nodes.

#### 4.2. Exploring models

As in our approach the model is not presented entirely to the user, but some parts of the graph will be filtered or aggregated into other nodes, there

is the need for *exploration operations*. There are two kinds of exploration operations:

- **Expanding a vertex.** Our graphs have some vertices that represent a collection of the original graph vertices. Hence, if we want to explore the graph, we need to access the content of those aggregated nodes. The expansion can be performed within the same graph, or a new view with the expanded graph can be created.
- **Search.** When we are analysing a graph, we usually want to look and focus on a vertex of the graph. This exploration operation typically requires defining some kind of filtering criteria and then select the desired vertex from a reduced list of results.

In addition, we can use the information given by the model fragmentation for exploration in two ways:

- **Staged exploration.** When a fragment of a model is to be visualised, only such fragment needs to be loaded in memory and visualised. In model fragments, some special nodes called proxies [40] represent nodes that actually belong to other fragments. Proxy nodes represent the *boundaries* of a fragment. They can be navigated, and so the system will load the fragment where the real node pointed by the proxy resides.
- **Hierarchical exploration.** This is the exploration via the folders and files produced by the fragmentation strategy. This way, folders and units are represented as special nodes. They can be expanded, and so the system will load the content of the given unit (i.e., its model elements) or the files and folders within the given folder (which are represented as nodes).

## 5. Tool support

We have provided tool support for the previous concepts as a set of Eclipse plugins [13]. The architecture of the tool is shown in Figure 7.

In a first step, the engineer designing a meta-model may decide to apply a fragmentation strategy to a meta-model. This is supported by EMF-Splitter, which generates a customised, modular modelling environment, so

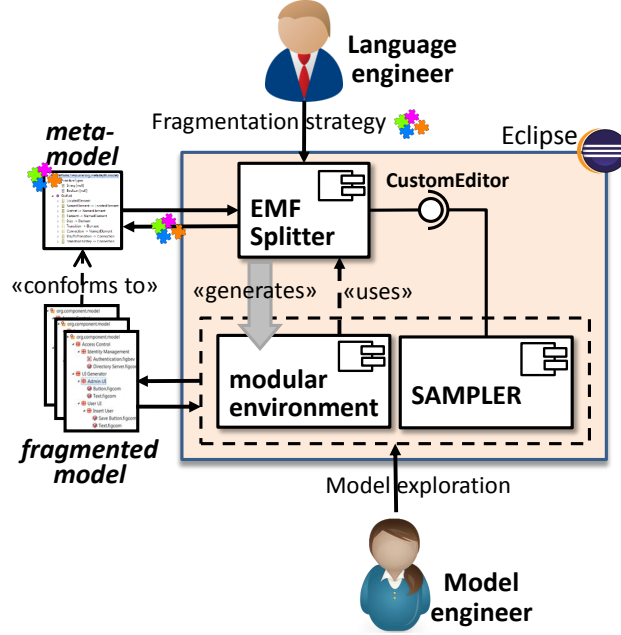


Figure 7: Architecture of the combined tools.

that models can be created according to the specified fragmentation strategy. EMF-Splitter will be described in Section 5.1. Then, the models (fragmented or not) can be visualised using the described techniques in Section 4. Such visualisation is supported by SAMPLER and will be described in Section 5.2. Both plugins are independent, but can be used in coordination to explore large models. In particular, EMF-Splitter offers an extension point (*CustomEditor* in the figure) to visualise models using customised editors. By default, visualisation is done with the EMF tree editor, but SAMPLER implements such extension point, so that models can also be visualised with it. In particular, this combination of tools allows using the exploration strategies for fragmented models described in Section 4.2. We describe this combined tool support in Section 5.3.

The choice of realizing our approach as Eclipse plugins is justified by the use of EMF as underlying modelling framework for EMF-Splitter (while SAMPLER is still independent of the modelling platform). Moreover, this choice permits interoperability with the rich Eclipse ecosystem of EMF-based MDE tools. Finally, Eclipse extension points facilitate the extensibility of both SAMPLER and EMF-Splitter.

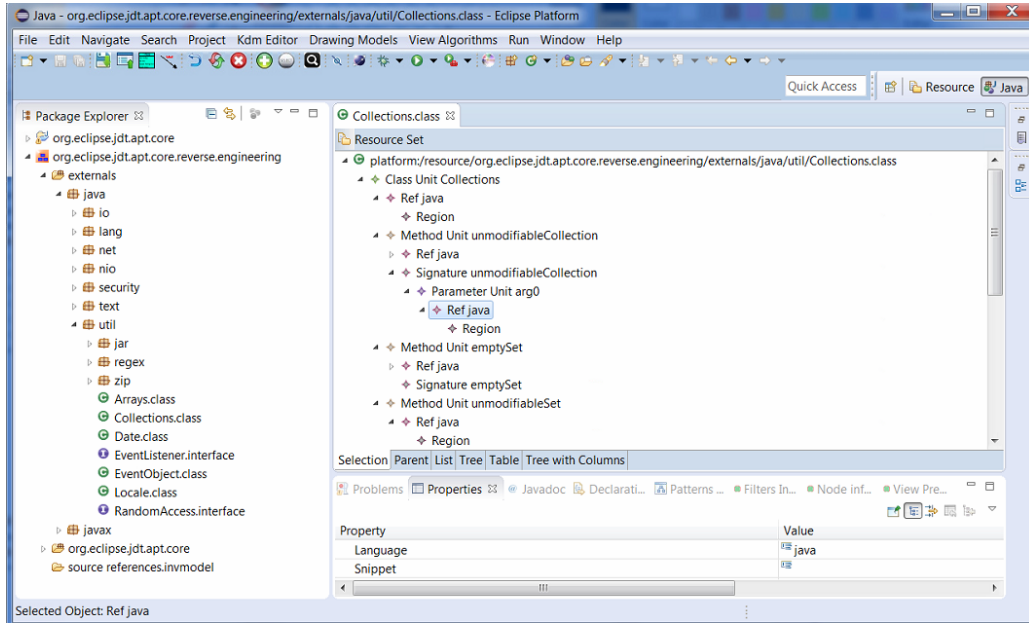


Figure 8: Environment generated by EMF-Splitter for the KDM meta-model.

### 5.1. EMF-Splitter

We have built tool support to apply fragmentation strategies to a meta-model. The tool produces a customised modelling environment that splits monolithic instances of the meta-model according to the fragmentation strategy and also supports the creation of models according to such organization. Our tool is called EMF-Splitter, it is built atop of Eclipse and is freely available at <http://www.miso.es/tools/EMFSplitter.html>.

The tool allows applying fragmentation strategies to a meta-model, and generates a customised modelling environment for it (called “modular environment” in Figure 7). This environment allows creating models, which are no longer monolithic, but are fragmented according to the defined strategy. In addition, it allows fragmenting an existing monolithic model according to the strategy, and also to compose a fragmented model into a unique file.

Figure 8 presents the generated modelling environment for KDM. The environment shows an Eclipse project, named `org.eclipse.jdt.apt.core.reverse.engineering`, created from the KDM Code Model of the project `org.eclipse.jdt.apt.core`. This project is one of the plug-ins that are part of Eclipse Java Development Tools Core (JDT Core). The project explorer shows the structure of folders and



files generated from the model, which follows the specified fragmentation strategy. The project mimics the structure that a Java project would have. In particular, we have chosen icons for folders and units that resemble the ones used by the JDT browser to represent packages and Java classes, but it actually is a model conformant to the KDM meta-model, split across the file system. The project explorer to the left makes evident the hierarchical structure of the model and facilitates model navigation. To the right, a tree editor shows the content of one of the fragments. This approach promotes scalability, as the original model has about 104.591 model elements, while the fragmentation strategy fragments it into 358 files. These files are much reduced in size, which allows faster loading of each fragment, and a better navigation of the model. We will discuss the benefits of fragmentation through a set of experiments in Section 6.

## 5.2. *SAMPLER*

To solve the visualisation and exploring problems, we have developed a tool called *SAMPLER* (ScAlable Model exPLorER). This is a collection of Eclipse plug-ins that implement the visualisation and exploration facilities described in Section 4. The tool is highly extensible, enabling the addition of new filters and simplification operations through Eclipse extension points. It is also agnostic with respect to the modelling technology, currently supporting EMF models and graphs stored in the GraphML format [8], but support for other formats can be added through extension points as well. The tool is freely available at <http://miso.es/tools/sampler.html>.

### 5.2.1. *Architecture*

Some of the *SAMPLER* strengths are its versatility and its extensibility. This has been achieved using the plug-in and extension-point mechanisms of Eclipse. An overview of its main components is shown in Figure 9. The *GraphAbstraction* plug-in provides the basic interfaces for abstracting a graph. It provides some extension-points so that developers can extend the tool. In particular, *Search* can be used to implement new search criteria. *Step* is used to declare simplification operations (see Section 4.1). They can be *filters*, *global* or *local* operations. The steps declared using this extension point can be used by the user to create custom nested simplification algorithms, using the *Algorithm* extension. Finally, it also offers an extension to configure the simplification algorithm (extension *Configuration*).

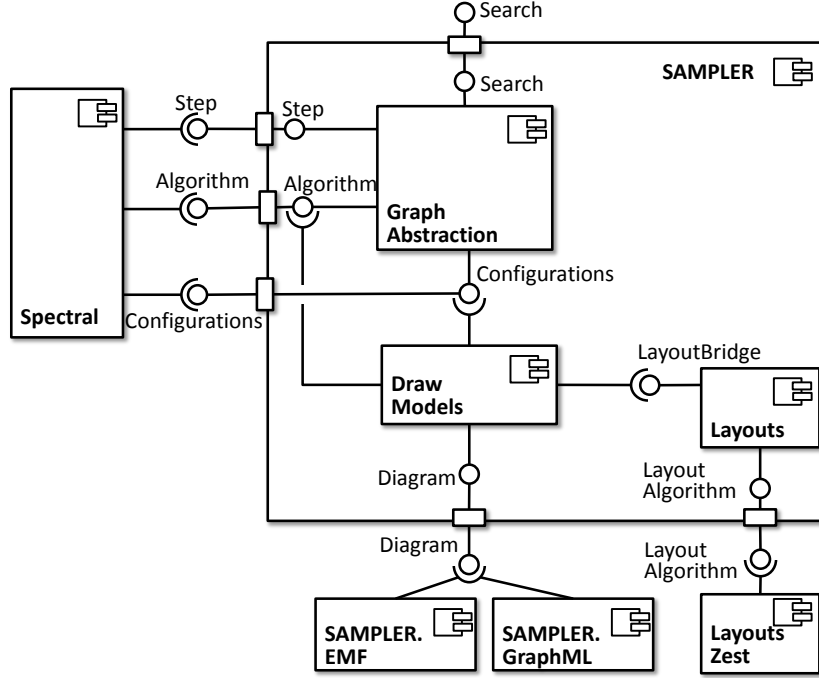


Figure 9: Architecture of SAMPLER

SAMPLER provides by default some implementation of these extension points, for example offering simplification operations like **Connected Components** and **Leaves Compactification**, nested algorithms like **Show All** (shows all nodes in the graph), **Total Compactification** (aggregates all nodes in the graph in a single node), **Connected Components** (aggregates the graph connected components as single nodes), and **Leaves** (aggregates the leaves of the graph). It provides some default configurations for these algorithms, like the maximum number of nodes to show, among others.

The figure also shows a component named **Spectral**, which provides several *global abstractions* based on the analysis of the Laplacian spectrum of the graph (a matrix combining vertex degree and vertex adjacency) [11]. The use of this spectrum has been proven its potential while analysing unstructured graphs. As SAMPLER is intended to deal with any kind of model, these general algorithms can be useful. The **Spectral** component implements some simplification operations (bi-clustering, iterative-clustering and hierarchical clustering), and a nested simplification algorithm (bi-cluster algorithm). In addition, the **Spectral** component configures the graph abstraction with the

maximum number of clusters.

The `Layouts` component provides interfaces to add layout algorithms that SAMPLER can apply to the models. The figure shows component `Layouts.Zest`, which is a wrapper for the layout algorithms provided by the Zest Eclipse library<sup>5</sup>. These include grid, radial, spring and tree layouts, among others. The `DrawModels` components adds an Eclipse-based user interface to SAMPLER. It also provides an extension-point (*diagram*) to handle different types of graphs, so that SAMPLER can deal with different technology such as EMF and GraphML.

### 5.2.2. Usage

When a model is opened with SAMPLER, the user may choose between different nested simplification algorithms to apply. In addition, it is also possible to create a custom algorithm on the spot using the simple operations (filters, global and local abstractions) that are available in the tool. The operations that are implemented in SAMPLER by default are:

- *Filter by EClass*: this is a filter based on EMF models that remove from the graph the vertices that instantiate a certain EClass.
- *Compactification of leaves*: this is a global abstraction that explores the containment tree of an EMF model and for each set of leaves of the same parent creates an aggregated node.
- *Connected components*: this is a global abstraction that aggregates all the vertices of the model in the same connected component. It is possible to configure this abstraction to use strong connectivity.
- *Clustering*: these include several clustering algorithms that put together all the vertices of the model in the same cluster. The possible clustering algorithms includes the bi-clustering method mentioned in Section 4.1. It also includes an iterative extension of this algorithm, which creates as many clusters as desired by just iterating the process in each of the obtained partial clusters. These algorithms have been chosen as examples of clustering, but any new clustering criteria can be implemented and added to SAMPLER by the user.

---

<sup>5</sup><https://www.eclipse.org/gef/zest/>

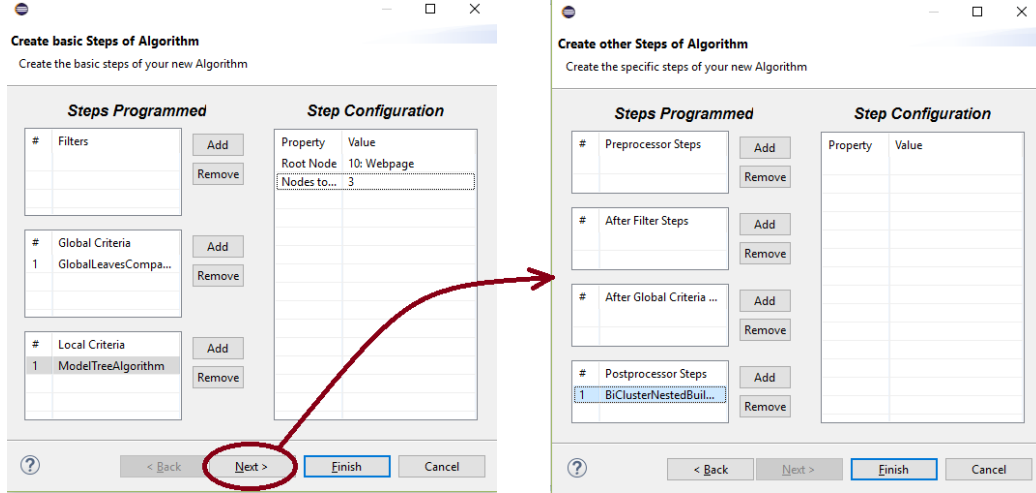


Figure 10: Creating a custom nested simplification algorithm.

- *Focus on a vertex*: this is a local abstraction that, given a vertex of the models, makes a breadth first search on the graph until it finds a number of vertices. Those vertices are represented without changes and the others are aggregated in a new vertex.

While the first two operations are specific to EMF technology, the three last ones are technology independent. Figure 10 shows an example of a custom nested simplification algorithm. The algorithm created works as follows. First, it takes the original graph and joins all the leaves of the containment tree together (which we called *Compactification of leaves*). Second, it *focuses on a vertex* that is sought in the graph. Finally, a *clustering* algorithm is applied to split the result into two nodes. The resulting graph is then drawn on the screen. Figure 11 presents the result of applying such algorithm to a simple graph after selecting the vertex number 10 for the second step. As expected, the result has only two vertices and one of them is marked because it contains the original vertex that we sought at step 2.

It is also possible select a layout algorithm to be used to draw the simplified graph obtained after the execution of the nested simplification algorithm selected by the user.

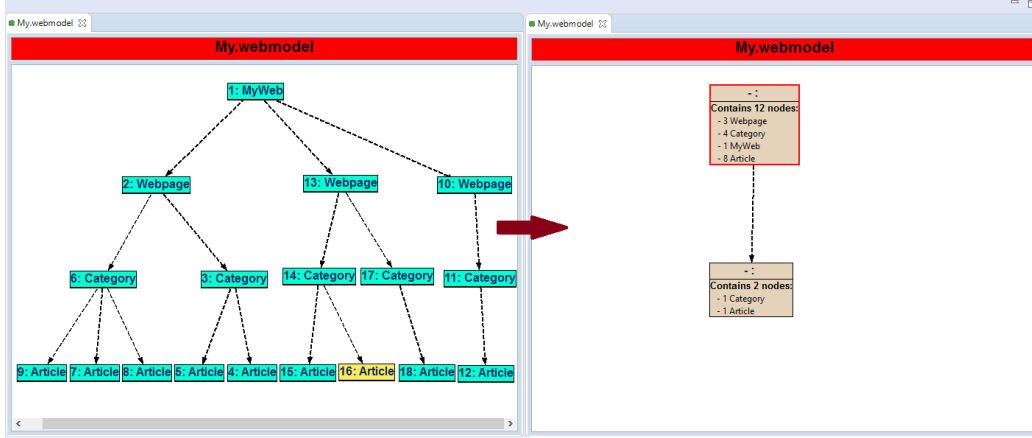


Figure 11: Result of applying the custom algorithm.

Once the user has configured the initial algorithms to be applied to the model, SAMPLER represents the model in an Eclipse Editor. It shows in different colours the vertices that have been inherited from the original graph (called *original vertices*) and the vertices that aggregate a collection of model nodes (we call them *compacted vertices*). Figure 12 shows an example of how SAMPLER works. We can see an abstracted KDM model where we have applied the *Focus on a vertex* operation over the vertex `Package` (with label 6, whose border appears in red) and show the 4 nearest vertices (in blue, with labels 345, 332, 7 and 19). The other vertices are compacted according to their parents in the containment tree, which has resulted in five compacted nodes shown in yellow and brown. The currently selected node is the aggregated vertex labelled as `:-Value` (shown in yellow). By default SAMPLER draws differently containment and non-containment references (dashed and dotted edges respectively).

In addition, the names of the references and the attributes of objects can be shown or omitted (they have been omitted in the figure). The *Node Information* view (at the bottom) shows the information contained in a vertex of the graph that the user has currently selected (see the bottom of the figure). In case an aggregated vertex is selected, it shows the different vertices

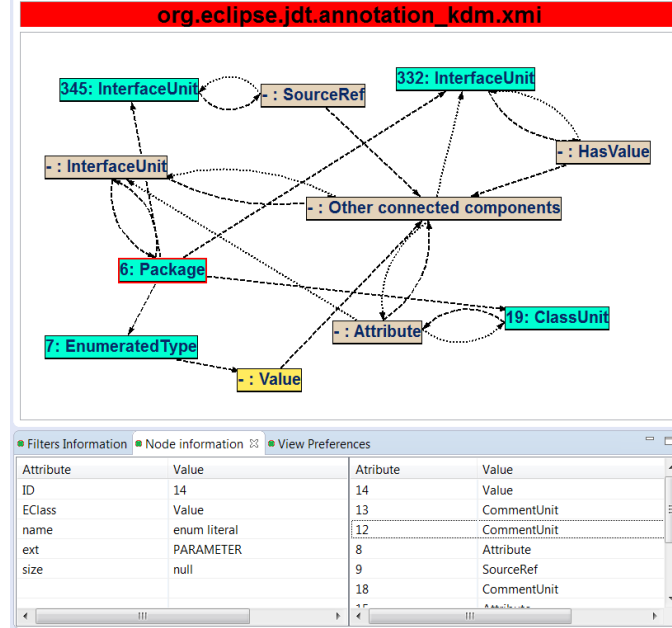


Figure 12: Exploring a model with SAMPLER.

it contains, and allows visualising their features.

At this point, it is possible to explore the model in three different ways. The first one is by expanding a compacted vertex. After removing the compacted vertex, SAMPLER takes the first vertex  $v$  in the collection it contains, and adds it to the graph. Then, it adds new compacted vertices containing the vertices  $v$  is connected to (and were inside the compacted vertex). Second, it is possible to search an original vertex. Using different search criteria it is possible to select and inspect the information of an original vertex of the model and, if a local abstraction is being applied, use it as the main vertex for this abstraction. Third, one can apply extra filter operations after the nested simplification algorithm. We consider this an exploration feature because it is independent of the current used algorithm, and allows removing some vertices that are not important to the user. Finally, as obtaining an interesting visualisation for a model can sometimes be tedious and time consuming, SAMPLER offers the option to save the status of the exploration of the model. This way, the user can load it to recover the abstracted graph exactly in the same state it was saved, so she can continue exploring the model from this point.

### 5.3. Combining EMF Splitter and SAMPLER

While EMF-Splitter and SAMPLER can be run separately, as Figure 7 shows, they have been integrated to work in a coordinated way. In particular, SAMPLER provides the possibility of opening and visualising fragmented models with it, and SAMPLER is able to explore the fragmented models in stages, and also using the structure in the file system created by EMF-Splitter (as explained in Section 4).

Figure 13 shows a snapshot of the exploration of the KDM fragmented model named `org.eclipse.jdt.apt.core.reverse.engineering` (a part of it was shown in Figure 8). 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 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 a leaves-local abstraction, which shows a maximum of five nodes and aggregates together several nodes within two abstract nodes. Overall, compared to Figure 8, SAMPLER provides an alternative graph-based visualisation to the EMF tree editor, as well as abstraction and exploration mechanisms.

## 6. Evaluation

Next, we evaluate the performance of our tools to deal with large models. For this purpose, we present five experiments. In the first one (Section 6.1), our intention is to analyse to what extent large models can be explored with SAMPLER. When models become difficult to be visualised with the tool, they are fragmented first using a fragmentation strategy, so that the smaller chunks can be visualised individually. Hence, in a second experiment (Section 6.2) we analyse the incurred cost of fragmentation. Finally, we make

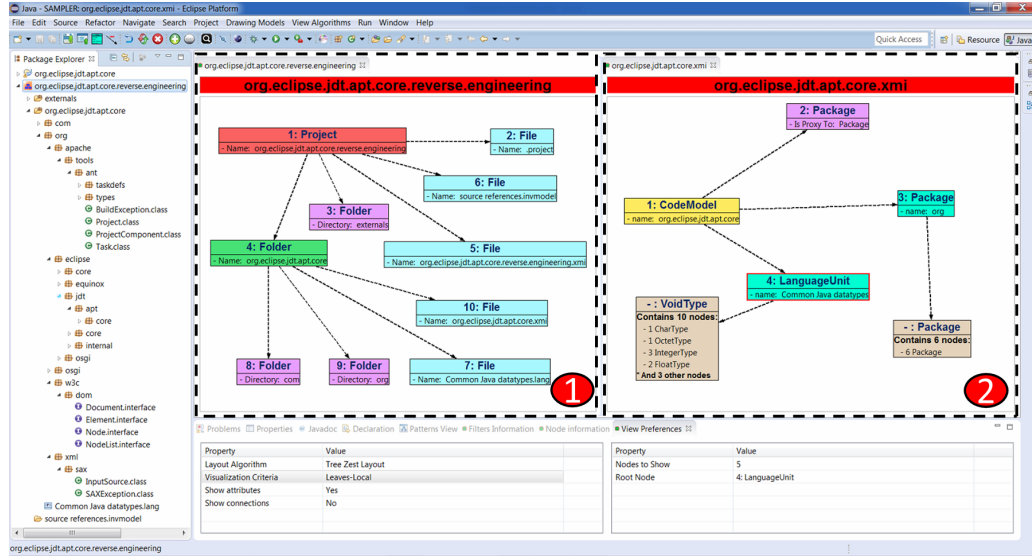


Figure 13: Visualisation of a KDM model using SAMPLER.

three experiments (described in Section 6.3) assessing the advantages of our approach with respect to de-facto standard tools like the EMF's default tree editor, CDO (a widely used model repository and persistence backend) and Gephi (a popular graph visualisation tool).

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

In the experiments we use both synthetically generated models and models developed by third parties. For this purpose, we use two meta-models.



One is based on a case study of the EU project MONDO in the domain of component-based embedded systems. The other one is a meta-model for Java (JDTAST), taken from the reverse engineering domain. This is a meta-model used by the Modisco reverse engineering tool to represent Java programs in the form of models (i.e., similar to the abstract syntax tree, but allowing references between elements, forming a graph) [9]. For the first case we use a synthetic model generator, while for the second we use models resulting from reverse engineering Java projects.

### 6.1. Exploration performance

In this section the goal is to check the performance of some of the SAMPLER abstraction strategies for large models. We present two experiments, one using synthetic models, and the other one using real models created by third parties.

#### 6.1.1. Synthetic models

For the first experiment, we generate models using an EMF random instantiator from the ATLANMOD team<sup>6</sup>. We use a meta-model taken from a case study of the EU project MONDO in the domain of component-based embedded systems. A small excerpt of the meta-model is shown in Figure 14 (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.

In each test, we take four measures: the time taken to read the model, and the execution time of three simplification algorithms:

- 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).
- A global algorithm that explores the whole model detecting the leaves of the containment tree and compact them (global algorithm).

---

<sup>6</sup><http://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/>

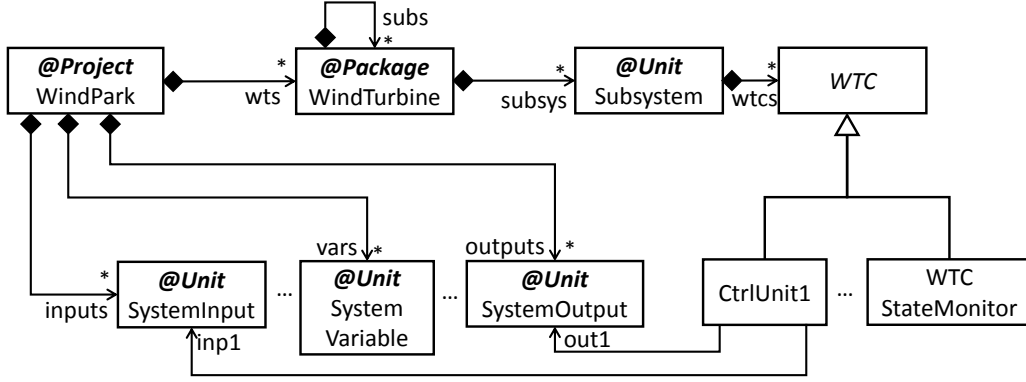


Figure 14: Excerpt of the meta-model of one of the MONDO case studies, with fragmentation strategy annotations.

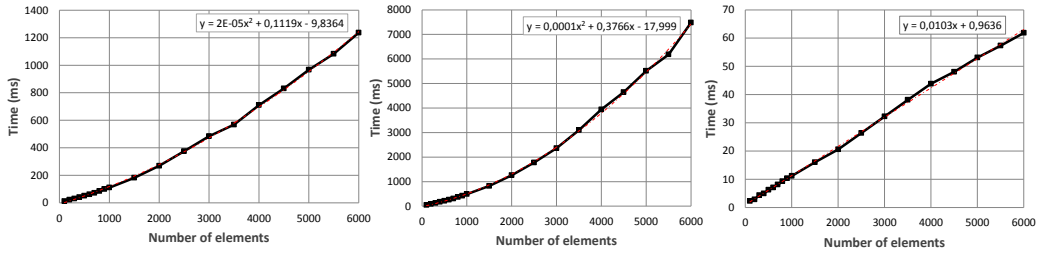


Figure 15: Performance (ms) of the different algorithms of abstraction. From left to right: local, global and compactification.

- 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).

For each test model and for each algorithm, we initiate a new Java Virtual Machine with the configuration mentioned above and run the methods directly on it without doing any previous warmups. Thus, our measures give information about the time that will take using the tool for the first time.

The graphics in Figure 15 show that every algorithm takes no more than 8 seconds for 6.000 elements in the model (which can be considered reasonable for an exploration application) and that the local and global algorithms take a quadratic polynomial time to execute. As expected, the total compactification algorithm is the most efficient one, as it does not need to perform any computation, while the most demanding method is the global algorithm,

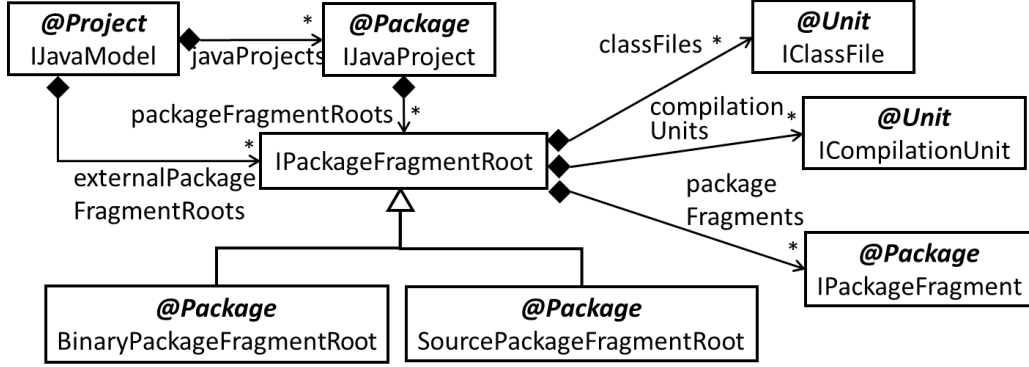


Figure 16: JDAST meta-model, with fragmentation strategy annotations.

because it needs to identify the leaves of the model containment tree. The times for both the local and global algorithms can be adjusted by quadratic polynomial curves, while the compactification can be adjusted by a linear polynomial.

#### 6.1.2. Realistic large models

In the second experiment, we execute the same algorithms in the same conditions over the two first sets of JDAST models of the GraBaTs competition, which have a larger size. A small excerpt of the JDAST meta-model is shown in Figure 16 (the meta-model contains more than 120 classes). A Java model is made of projects, which are divided in binary and source packages, each having class files and compilation units. The four models of the GraBaTs competition contain a model-based representation of several large Java projects. The smallest model (set0) has about 71.000 model elements, while the largest one (set4) has about 5.000.000 model elements.

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 24 minutes with the *set0* model and more than 5 hours with the *set1* model (which has about 204.000 elements). Those times are not acceptable for an interactive application, and hence we resort to model fragmentation. The next subsection analyses its performance.

Model	Local Algorithm		Global Algorithm		Compactification	
	<i>Measure</i>	Estimation	<i>Measure</i>	Estimation	<i>Measure</i>	Estimation
<i>set0</i> (71.458 els)	24min27s	1min22s	20min24s	13min6s	0,78s	0,75s
<i>set1</i> (203.938 els)	5h43min16s	10min11s	3h48min9s	1h42min6s	2,8s	2,97s

Table 1: Performance of SAMPLER over some JDTAST models.

## 6.2. Fragmentation performance

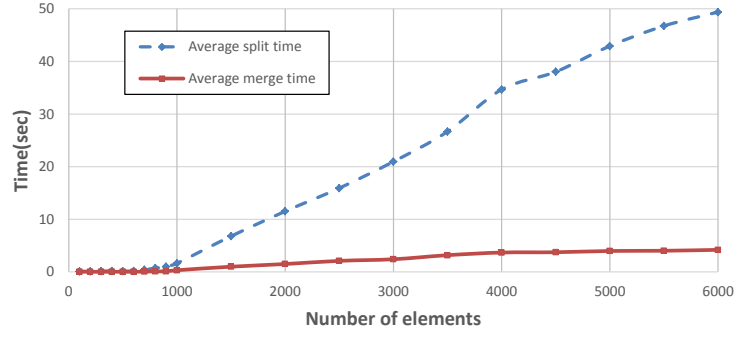
In this subsection we evaluate the performance of the fragmentation strategy, using the same two case studies as in the previous section.

### 6.2.1. Synthetic models

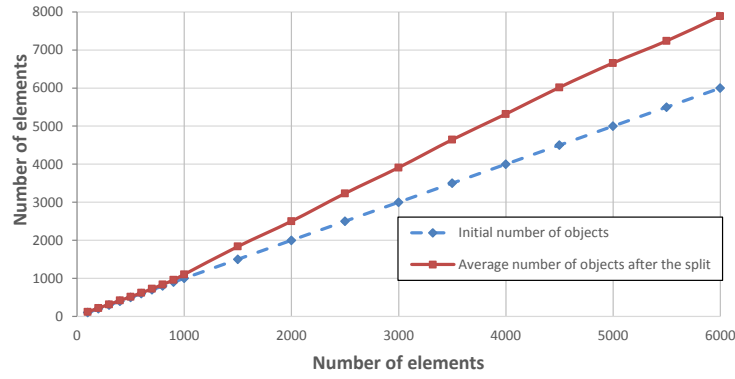
First, we evaluate the fragmentation performance using the same synthetic models used with SAMPLER. Figure 14 shows an excerpt of the meta-model for wind turbines, with the annotations of the desired fragmentation strategy. This way, class `WindPark` is the root class, and has been tagged as `Project`. `WindTurbines` are tagged as `Package`, so that folders 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.

Figure 17 shows the results of the fragmentation of the synthetic models. As can be seen, EMF-Splitter is able to split a model that contains 6.000 objects in less than a minute and merge back all fragments into a monolithic model in less than 10 seconds, as shown in Figure 17(a). For splitting, the times include loading the monolithic model, perform the fragmentation in memory, and serialize 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.

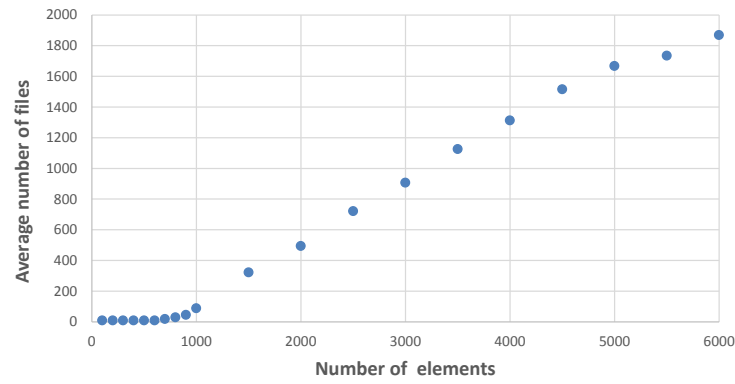
In Figure 17(b), we compare the number of initial and final objects, before and after the fragmentation. It can be observed that more objects (proxy references) need to be created in order to maintain the cross-references between the different fragments. The amount of proxies created depends on the number of files created by the fragmentation strategy. Figure 17(c) shows the average number of files created for each model size. In average around



(a) Merge and split time.



(b) Comparison of initial and final objects.



(c) Average number of files.

Figure 17: Results of splitting/merging IKERLAN's synthetic models.

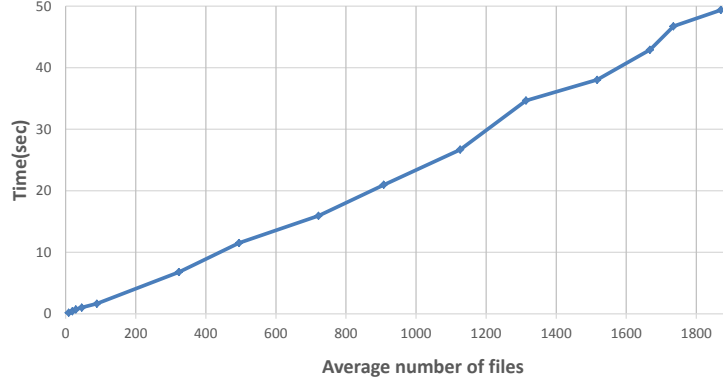


Figure 18: Effect of the number of files created in split time. Average times of all models, with sizes in the range 100 - 6.000 elements.

1.800 files are created for models that contain 6.000 objects. We also observe that in average one proxy object or two as a maximum is 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 17(a) shows an increase of splitting time at around 1.000 objects. This can be attributed to a combined effect of model size and number of generated fragment files (see Figure 17(c), where the number of files also increases abruptly at around such number of objects). Figure 18 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. A linear increase can be observed. To better understand the effect of the number of fragments, Figure 19 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.

### 6.2.2. Realistic large models

In the second experiment, we used the JDTAST models of the GraBaTs competition. Figure 16 shows the fragmentation strategy applied to the JD-TAST meta-model. In this case, the `IJavaModel` class is mapped to Project

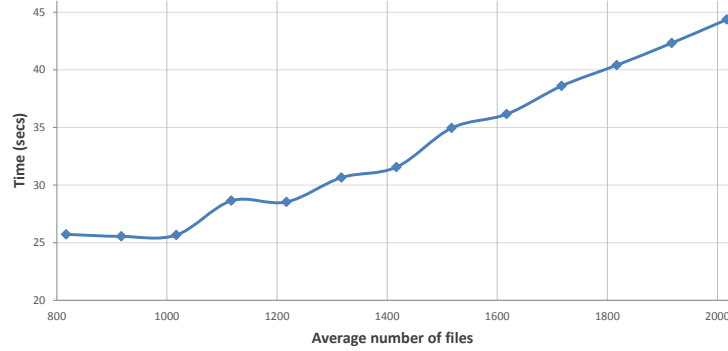


Figure 19: Effect of the number of files in split time. Average times of a set of models of size 6.000.

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 patterns demands by means of relation `javaProjects`. Another composition relation between `IJavaProject` and `IPackageFragmentRoot` allows classes which 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 modularity pattern, we split all the models of the GraBaTs case study, turning each one of them into an Eclipse project. As an example, Figure 20 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 fragmentation strategy fragments 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.

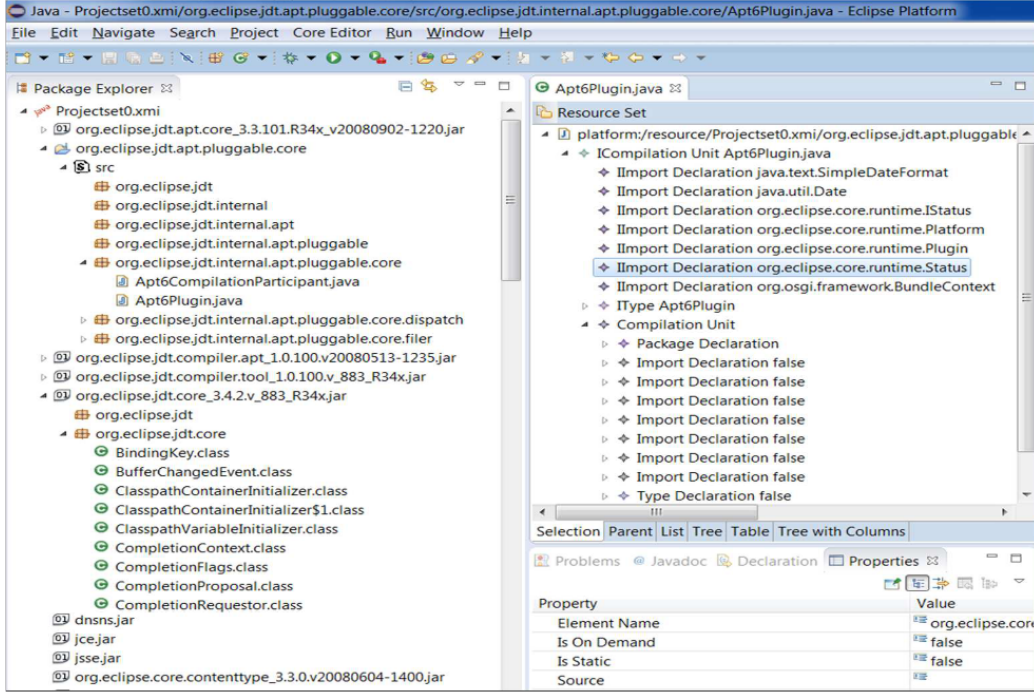


Figure 20: Environment generated by EMF-Splitter for the JDTAST meta-model.

Table 2 and Figure 21 present the results of the experiment. The graphic shows a comparison between split and merge times. As in the experiment of Section 6.2.1, 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 the 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.

The results show that with the fragmentation, the exploration of the files with SAMPLER would become easier, because the largest average fragment



Model	Split time	Merge time	#files	Avg	Max	# model elements
<i>set0</i>	<i>1min34s</i>	<i>8s</i>	1.779	40,17	1.322	71.458
<i>set1</i>	<i>3min51s</i>	<i>38s</i>	6.240	32,68	4.549	203.938
<i>set2</i>	<i>9min5s</i>	<i>1min24s</i>	6.050	345,27	50.718	2.088.890
<i>set3</i>	<i>12min28s</i>	<i>3min20s</i>	4.460	1.031,24	50.718	4.599.358
<i>set4</i>	<i>13min28s</i>	<i>8min32s</i>	5.068	980,04	50.718	4.966.846

Table 2: Results of splitting and merging the JDTAST models.

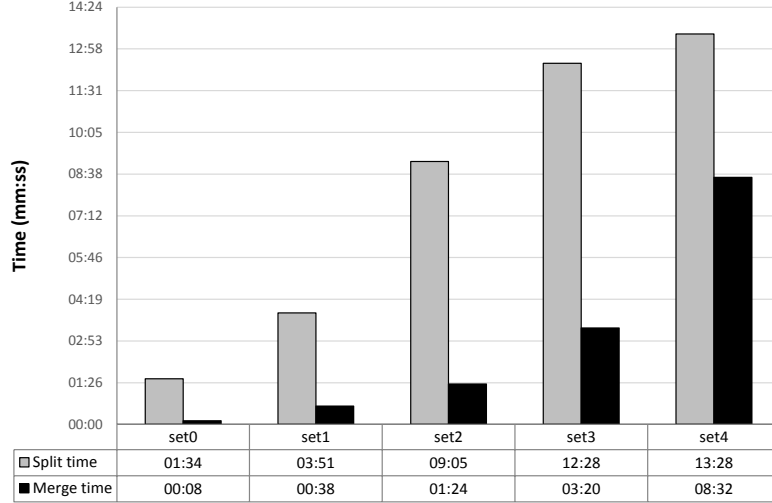


Figure 21: Split/merge times over the JDTAST models.

size is about 1.000 elements (which are easily explorable with SAMPLER), while the maximum number of elements in a file is 50.718, which would take about a minute and a half to be visualised.

### 6.3. Comparing with third party tools

In this section, the goal is to assess the benefits that our 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 also compare with Gephi, one of the most widely used tools for graph visualisation.

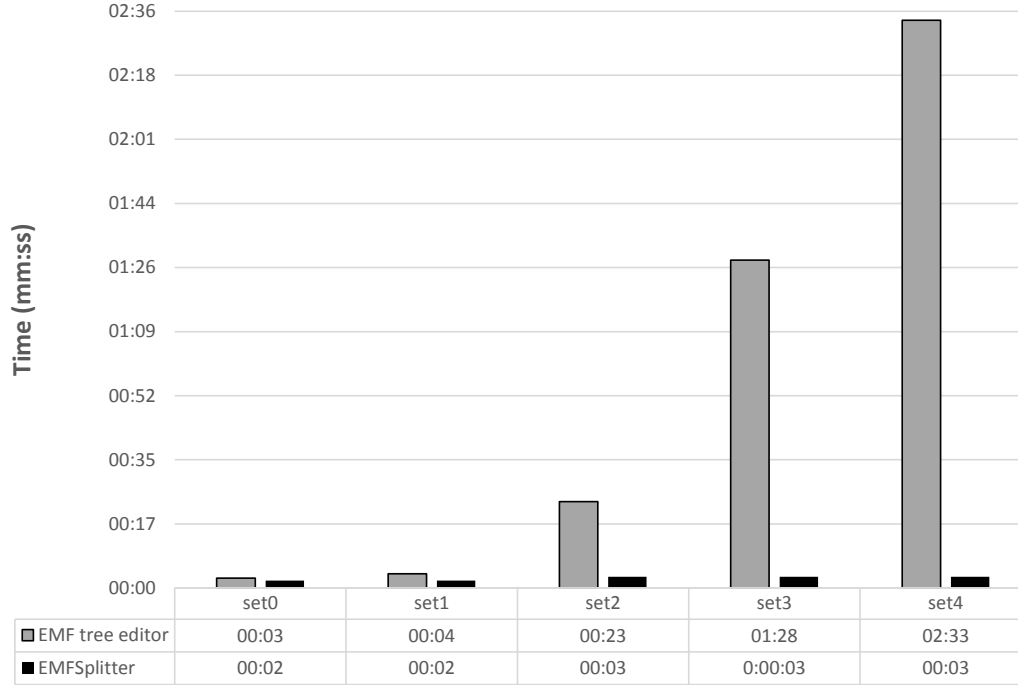


Figure 22: Time required to opening the model with EMF’s reflective tree editor (grey columns to the left) and EMF-Splitter (black columns to the right).

### 6.3.1. 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 editor (instead of SAMPLER). This way, we compare the time needed to open the monolithic models 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 22 shows the time needed by the tree editor to open the complete models (grey columns to the left of each series). It can be observed that, for the largest model, it needs two and a half minutes to open.

The figure also shows a comparison with the time needed to open the largest fragment in each model (black columns to the right of each series). As this time is much smaller, the times are detailed in Table 3. 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	<code>ProblemReporter.class</code>	1.322	1,67s	1,57x
set1	<code>BaseConfigurationBlock.java</code>	4.549	1,76s	2,19x
set2	<code>CompletionEngine.java</code>	50.718	2,65s	8,77x
set3	<code>CompletionEngine.java</code>	50.718	2,65s	33,36x
set4	<code>CompletionEngine.java</code>	50.718	2,65s	55,76x

Table 3: Time required (ms) to open the biggest fragment model of each project with the tree editor.

Next, we assess the benefits in terms of memory consumption with respect to the tree editor and monolithic models. For this purpose we evaluate the combined use of EMF-Splitter and SAMPLER, which may use the information provided by the fragmentation strategy, as discussed in Section 5.3. For this kind of exploration, it is not necessary to load the entire model as we evaluated in Section 6.1. Instead, only the resources associated with the root class need to be loaded, and then the nodes corresponding to folders and files can 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 4 shows the results of the evaluation. The columns depict the average number of loaded objects, the minimum and maximum. It can be noted that 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.

As we have seen, 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

Model	Average	Min	Max	Reduction w.r.t. tree editor (worst case)
<i>set0</i>	502,26	250	1.613	97,74%
<i>set1</i>	719,26	34	4.800	97,65%
<i>set2</i>	11.590,62	61	54.141	97,4%
<i>set3</i>	7.859,59	83	54.163	98,82%
<i>set4</i>	7.550,56	89	54.169	98,9%

Table 4: Necessary number of loaded objects to explore the fragmented models.

the amount of resources (files and folders) that it contains directly or indirectly. Table 5 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 suggest a high scalability of the combined exploration.

Model	Directly Contained Resources		Directly/Indirectly Contained Resources	
	Average Amount of Resources	Max	Average Amount of Resources	Max
<i>set0</i>	18,27	231	53,61	1.882
<i>set1</i>	28,98	270	85,88	6.463
<i>set2</i>	21,37	270	68,30	6.347
<i>set3</i>	12,20	157	46,86	4.858
<i>set4</i>	11,97	157	45,99	5.531

Table 5: Resources contained by the models at each hierarchical stage.

### 6.3.2. Fragmentation vs. Database Persistence Layer

An alternative to fragmentation is to use a more performing backend, not based on file storage. A widely used option is CDO, 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 backend. 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 backend, 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

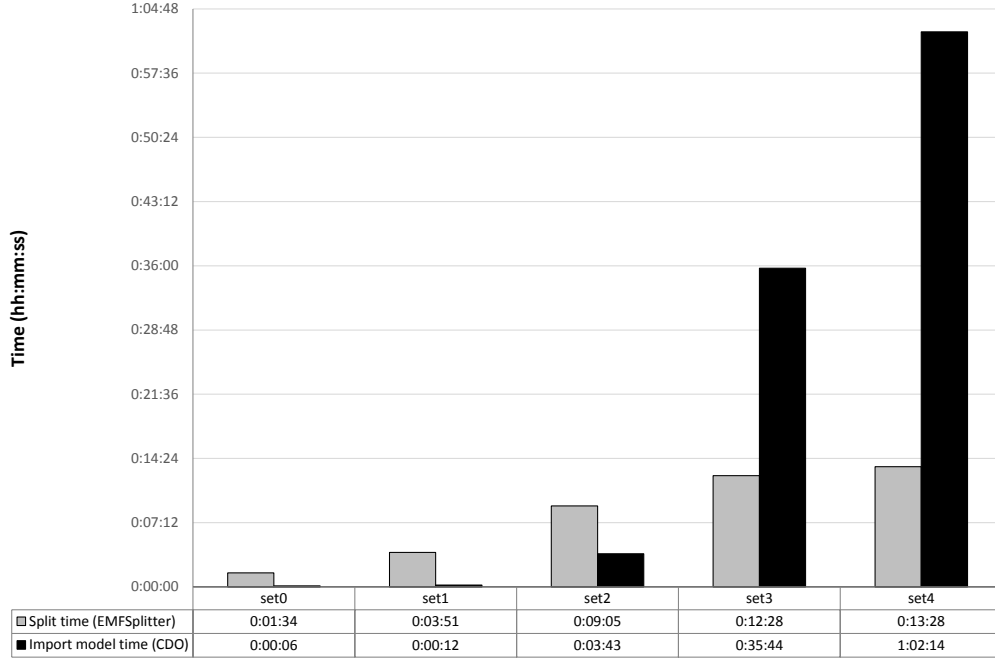


Figure 23: Time required to split the models with EMF-Splitter (grey columns to the left) and import the models into CDO (black columns to the right).

the result average is shown in Figure 23. The figure also shows the splitting model time (see also Figure 21). 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 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 Identifier (UUIDs), the scheme provided by EMF.

As it can be observed, 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

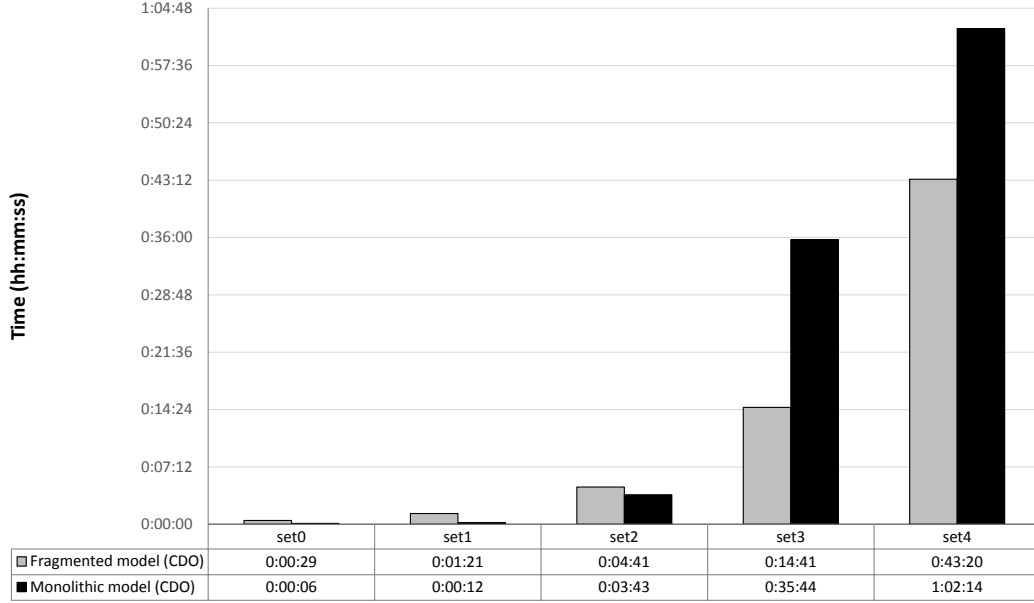


Figure 24: Time required by CDO to import fragmented (grey columns to the left), and monolithic models (black columns to the right).

use the models of JDTAST, fragmented in Section 6.2.2 using EMF-Splitter. The operation of importing the fragmented models was repeated three times and the average time is shown in Figure 24. It can be observed 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.

### 6.3.3. 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*<sup>7</sup> [4], a popular, open-source tool for graph exploration and analysis. In order to make a reasonable comparison, we converted each JDTAST model (*set0*, *set1*,

<sup>7</sup><https://gephi.org/>

*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 already 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 6 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 the model 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 6 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	Load Biggest Fragment
<i>set0</i>	3s	1min34s	1,67s
<i>set1</i>	6s	3min51s	1,76s
<i>set2</i>	3min40s	9min4s	2,65s
<i>set3</i>	9min36s	12min27s	2,65s
<i>set4</i>	14min10s	13min28s	2,65s

Table 6: Comparison between opening a model with *Gephi*, splitting the model with EMF-Splitter, and loading the biggest fragment produced.

#### 6.4. Discussion and Threats to Validity

In these experiments, we have evaluated the scalability of our model visualisation mechanisms, obtaining good results for large models when the fragmentation strategies are used. In terms of size, we see large reduction of size of the biggest fragment with respect to the total model size (see Table 2), which implies a drastic reduction in loading time using the default tree editor (see Figure 22 and Table 4). Moreover, fragmentation allows the handling of large models using SAMPLER (as seen in Section 6.2), which offers more sophisticated visualisation mechanisms than the tree editor. We have seen that fragmentation 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 backend (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.

The performance of the abstraction-based visualisations provided by SAMPLER is limited by two factors: model loading and computation of the abstraction. The first problem is aggravated by the fact that EMF loads all elements of the XMI files in memory. Partial loading mechanisms, like those developed by Wei et al. [48] could be of help. Second, some graph abstraction algorithms have quadratic growth (see Figure 15) on the number of elements. If the model is large, then simply these times are too large. XML databases, like BaseX are another persistence alternative that we plan to investigate in future work [21].

Fragmenting an existing, large monolithic model can be time consuming, as shown in Table 2. However, this is a one-time operation, and it benefits if 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.

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 other ones used in software modernization (KDM) have such hierarchical structure. However, one may also find “flat” meta-models with few compositions, for which no option but to include all objects in the model 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 the reverse engineering domain, 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. A study of how common flat meta-models are (e.g., by analysing meta-model repositories) is left for future work.

Finally, these results investigate the efficiency of the approach, 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 a future contribution.

## 7. Related work

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

Due to the need to process large models, some authors have proposed to split them for solving different tasks. For instance, Scheidgen and Zubow [38] 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. The approach has been used for the analysis of large code repositories [37], 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. This is similar to our approach, but our modularity pattern creates a richer structure, including projects and packages. Moreover, EMF-Splitter generates a modelling environment, which permits creating such fragments while modelling with the tool. Another difference is the persistence mechanism. EMF fragments stores fragmented models in memory, and for persistency 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. Instead, EMF-Splitter relies on XMI files, and fragmentation

is reflected in the created folder and file structure. While EMF-Splitter is limited by the capabilities of current XMI parsers, by being file-based it can benefit from traditional version control systems (like SVN or git).

Other works [24, 41] 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 [24]. 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 [41]. 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 [42]. The techniques are based on several clustering algorithms operation on a distance matrix. Such matrix is obtained by weighting different meta-model relations (generalization, composition, association) according to their relevance. While our approach is applicable to existing large models, a distinctive feature of our work 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 [27, 18] guided by quality criteria. Moody and Flitman [27] use genetic algorithms to cluster a data model into a multi-level structure, using principles of human information processing. More recently, inspired by our modularity pattern, and our previous work on generic transformations [14], Fleck and collaborators present an approach for model modularization applicable to arbitrary modelling languages [18]. 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, according to quality criteria, like cohesion and coupling. The transformation rules are applied according to multi-criteria optimisation algorithms. Our modularization is richer, as we support both projects and packages, and nested packages, while Fleck’s approach lacks hierarchical decomposition (modules within modules). However, the idea of using quality criteria to drive the splitting phase is quite interesting, and we will consider it in future work.

In the previous work of one of the authors [1] a theory of model fragmentation was developed, upon which we have based our practical solution. The theory is based on graphs and morphisms, and was developed using the Z for-

mal 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.

Other works directed to define model composition mechanisms [22, 23, 43] are intrusive. These papers [22, 43] present techniques for model composition and realise the importance of modularity in models as a research topic to minimise the effort. Strüber et. al [23] present a structured process for model-driven distributed software development which is based on split, edit and merge models for code generation.

Landesberger and collaborators present a survey of graph representation, exploration and analysis techniques used for the visual analysis of large graphs [47]. Among other aspects, the authors stress the need for supporting user interaction techniques for graph exploration. These include *lenses* (to focus on an interesting part of a graph), semantic zooming (increasing the level of detail by drilling down to lower levels of aggregation) and filtering techniques. Some of the operations supported by SAMPLER, like *local focus* can be interpreted as a lense, while the tool provides different ways to explore aggregated nodes, and filtering criteria. Additional visualisation techniques for large graphs, like combining node-link and matrix-based representations are left for future work. Similarly, a more recent survey [34] distinguishes techniques to obtain both global views of a large graph (like filtering, sampling, partitioning or clustering), and to handle local views (exploration, navigation). While SAMPLER offers filtering, clustering, exploration and navigation techniques, we also profit from pre-processing the graph using model fragmentation. This is a novel pre-processing technique according to [47, 34].

Regarding concrete tools, MoVi [28] is a visualisation tool that provides an interactive environment to process models. This tool has some main functions for a manageable exploration, like details on demand, partitioning, zoom and filter. However, it does not propose scalability techniques for working with big models. The ELVIZ framework for model visualisation [32] is based on the transformation of input models to appropriate output formats. For example, given a class diagram, ELVIZ can extract the number of methods per class, and visualise such numbers as a bar chart. ELVIZ facilitates the generation of input models to different visualisation outputs relying on mappings. However, this tool does not target large models. The incQuery

tool [45] allows querying a model using a pattern-based language. While it focuses on achieving good performance upon model changes (by an efficient indexing scheme), our focus is on scalability through fragmentation.

Explan [6, 7] is a tool that uses slicing techniques in order to visualise 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). Explan includes a flattening filter, which presents a hierarchy in the form of a unique class. SAMPLER supports the visualisation of models and meta-models, and the abstractions/slice criteria are extensible. Moreover, we support different navigation strategies from abstracted models. In previous work of one of the authors [16] some reusable abstractions for modelling languages were presented. Those abstractions produce a simpler view of an existing model, and are described in a generic way, being applicable “as is” to different modelling languages. The extensible architecture of SAMPLER would allow incorporating these abstractions into our approach.

The analysis of large graphs arising in e.g., social networks have produced some summarization techniques, which try to encode in smaller graphs [26] or as a variety of statistics [29] the main features of the large graph. For this purpose, some approaches are based on finding the most often occurring subtype graphs (cliques, stars, chains, etc) in those bigger 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 [44], and nodes with similar values are summarised in a single node. This would be similar to SAMPLER’s global abstractions.

Regarding tools for graph visualisation, perhaps the most well-known is Gephi [4]. The tool includes a force-based layout, enables the calculation of metrics, and dynamic graph analysis. While it is targeted to large graphs, we have seen that large models arising in the reverse engineering domain – like *set2*, *set3* and *set4* in the JDAST case study – are beyond Gephi’s capabilities (according to its web page, it can handle up to 100.000 nodes and 1.000.000 edges). Our experiments have shown that model fragmentation is a useful pre-processing technique for this kind of tools to handle these graphs.

Instead of using fragmentation, other persistence options to handle large models have been proposed, like CDO, or NeoEMF [5]. The latter allows storing the models in graph-based backends. In Section 6.3.2, we have seen the advantages of fragmentation to insert large models into these backends. Our approach keeps a file-based approach for storing models, which allows

their versioning in traditional version control systems like git or SVN, in a uniform way with the project code base. Moreover, model fragmentation permits reducing the possible conflicts that concurrent modifications may produce. In any case, the open architecture of SAMPLER (see Figure 9) enables using different modelling technologies, and hence NeoEMF or CDO could be used by our approach instead of using fragmentation. Similar to database indexes, model indexes have been proposed [3] to efficiently perform queries on large models. While this does not fully solve the problem of loading a very large model into memory, our plan is to combine indexing with fragmentation.

Altogether, to the best of our knowledge, our approach to combine model fragmentation and model visualisation techniques for scalable model visualisation is novel. Moreover, our report on this combination can be useful to existing graph visualisation tools.

## 8. Conclusions and future work

In this work, 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. 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 case study, a direct visualisation is limited, but our proposal is fragmenting them first. 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  $55\times$  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. We have also seen that tools for large scale graph visualisation, like Gephi, or for model persistence, like CDO, could benefit from fragmentation as a pre-processing step.

In the future, we would like to improve the visualisation capabilities of SAMPLER, by enabling combined node-link and matrix representations of graphs, and supporting hierarchical abstractions. We also plan to integrate

Hawk [3], a model indexer, within our tools to achieve better performance when operating with multiple model fragments. Finally, we will perform empirical studies about the tools usability, as well as a field study on the applicability of the fragmentation strategies. We also plan to integrate other techniques for fragmentation, like those proposed in [41] within our approach.

## Acknowledgements

Work supported by the Spanish Ministry of Economy and Competitiveness (TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125). We thank the reviewers for their excellent comments on a previous version of this paper. We also thank Markus Scheidgen for his help in experimenting with EMF Fragments.

## References

- [1] 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*, pages 106–115. IEEE, 2015.
- [2] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
- [3] K. Barmpis and D. S. Kolovos. Hawk: towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary*, page 6. ACM, 2013.
- [4] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009*. The AAAI Press, 2009. See also <https://gephi.org>.
- [5] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4emf, A scalable persistence layer for EMF models. In *ECMFA*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2014.

- [6] 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.
- [7] A. Blouin, N. Moha, B. Baudry, H. A. Sahraoui, and J. Jézéquel. Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information & Software Technology*, 62:124–142, 2015.
- [8] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. Graphml progress report. In *Graph Drawing*, pages 501–512, 2001. See also <http://graphml.graphdrawing.org/>.
- [9] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information & Software Technology*, 56(8):1012–1032, 2014.
- [10] H. Brunelière, J. Cabot, J. L. C. Izquierdo, L. O. Arrieta, O. Strauß, and M. Wimmer. Software modernization revisited: Challenges and prospects. *IEEE Computer*, 48(8):76–80, 2015.
- [11] T. Bühler and M. Hein. Spectral clustering based on the graph  $p$ -laplacian. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 81–88. ACM, 2009.
- [12] Connected Data Objects (CDO). <https://eclipse.org/cdo/>.
- [13] E. Clayberg and D. Rubel. *Eclipse plugins, 3<sup>rd</sup> Edition*. Addison-Wesley Professional, 2008. See also <http://www.eclipse.org/>.
- [14] J. S. Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE Trans. Software Eng.*, 40(11):1042–1060, 2014.
- [15] J. de Lara and E. Guerra. *A Posteriori* typing for model-driven engineering: Concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.*, 25(4):31:1–31:60, 2017.

- [16] J. de Lara, E. Guerra, and J. S. Cuadrado. Reusable abstractions for modeling languages. *Inf. Syst.*, 38(8):1128–1149, 2013.
- [17] K. Dhambri, H. A. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *12th European Conference on Software Maintenance and Reengineering, CSMR*, pages 279–283. IEEE Computer Society, 2008.
- [18] M. Fleck, J. Troya, and M. Wimmer. Towards generic modularization transformations. In *15th International Conference on Modularity*, pages 190–195. ACM, 2016.
- [19] 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.
- [20] A. Garmendia, A. Jiménez-Pastor, and J. de Lara. Scalable model exploration through abstraction and fragmentation strategies. In *Big-MDE’2015*, volume 1406 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2015.
- [21] C. Grün. *Storing and querying large XML instances*. PhD thesis, University of Konstanz, 2010. See also <http://basex.org>.
- [22] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *T. Asp.-Oriented Soft. Dev.* VI, 6:39–82, 2009.
- [23] P. Kelsen and Q. Ma. A modular model composition technique. In *Proceedings of FASE’10*, volume 6013 of *LNCS*, pages 173–187. Springer, 2010.
- [24] 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.
- [25] 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.



- [26] 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.
- [27] D. L. Moody and A. Flitman. A methodology for clustering entity relationship models - A human information processing approach. In *Conceptual Modeling - ER'99*, volume 1728 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 1999.
- [28] M.-K. Mufida, G. Calvary, S. Dupuy-Chessa, and Y. Laurillau. MoVi: Models visualization for mastering complexity in model driven engineering. In *Proceedings of the 2015 British HCI Conference*, British HCI '15, pages 281–282, New York, NY, USA, 2015. ACM.
- [29] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [30] OMG. Knowledge Discovery Meta-model specification. <http://www.omg.org/spec/KDM/>.
- [31] OMG. UML 2.5 specification. <http://www.omg.org/spec/UML/>.
- [32] M. Ostendorp, J. Jelschen, and A. Winter. ELVIZ: A query-based approach to model visualization. In *Modellierung 2014*, pages 105–120, 2014.
- [33] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara. Pattern-based development of domain-specific modelling languages. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015*, pages 166–175. IEEE, 2015.
- [34] R. Pienta, J. Abello, M. Kahng, and D. H. Chau. Scalable graph exploration and visualization: Sensemaking challenges and opportunities. In *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015*, pages 271–278. IEEE, 2015.
- [35] A. Rentschler, Q. Noorshams, L. Happe, and R. H. Reussner. Interactive visual analytics for efficient maintenance of model transformations. In *Theory and Practice of Model Transformations - 6th International*

- Conference, ICMT 2013*, volume 7909 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.
- [36] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.
  - [37] 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*, volume 8769 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2014.
  - [38] 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.
  - [39] D. C. Schmidt. Guest editor’s introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb. 2006.
  - [40] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2<sup>nd</sup> Edition*. Addison-Wesley Professional, 2008. See also <http://www.eclipse.org/modeling/emf/>.
  - [41] 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.
  - [42] D. Strüber, M. Selter, and G. Taentzer. Tool support for clustering large meta-models. In *BigMDE 2013*, page 7. ACM, 2013.
  - [43] 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.
  - [44] 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.
  - [45] 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.*, 98:80–99, 2015.

- [46] M. F. van Amstel, M. G. J. van den Brand, and A. Serebrenik. Traceability visualization in model transformations with tracevis. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations*, ICMT'12, pages 152–159, Berlin, Heidelberg, 2012. Springer-Verlag.
- [47] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. Fekete, and D. W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, 30(6):1719–1749, 2011.
- [48] 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*, MODELS '16, pages 329–339, New York, NY, USA, 2016. ACM.