Intelligent run-time partitioning of low-code system models

Sorour Jahanbin, Dimitris Kolovos, Simos Gerasimou

Intelligent Run-Time Partitioning of Low-Code System Models

Sorour Jahanbin
University of York
York, United Kingdom
sorour.jahanbin@york.ac.uk

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Simos Gerasimou
University of York
York, United Kingdom
simos.gerasimou@york.ac.uk

ABSTRACT

Over the last 2 decades, several dedicated languages have been proposed to support model management activities such as model validation, transformation, and code generation. As software systems become more complex, underlying system models grow proportionally in both size and complexity. To keep up, model management languages and their execution engines need to provide increasingly more sophisticated mechanisms for making the most efficient use of the available system resources. Efficiency is particularly important when model-driven technologies are used in the context of low-code platforms where all model processing happens in pay-per-use cloud resources. In this paper, we present our vision for an approach that leverages sophisticated static program analysis of model management programs to identify, load, process and transparently discard relevant model partitions – instead of naively loading the entire models into memory and keeping them loaded for the duration of the execution of the program. In this way, model management programs will be able to process system models faster with a reduced memory footprint, and resources will be freed that will allow them to accommodate even larger models.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**.

KEYWORDS

Model Partitioning, Partial Loading, Memory Management, Model-Driven Software Engineering

ACM Reference Format:
Sorour Jahanbin, Dimitris Kolovos, and Simos Gerasimou. 2020. Intelligent Run-Time Partitioning of Low-Code System Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3420198

1 INTRODUCTION

Models and modelling have always been part of the software development process. While in traditional software development processes, models are used primarily for documentation and communication, in Model-Driven Engineering (MDE), models play a more central role and are considered as first-class artefacts that drive software development, thus enhancing productivity [10][5], maintainability, consistency, and traceability [11].

In MDE, models are manipulated using model management programs that carry out different tasks such as model transformation, model merging, model validation, etc [8]. These tasks can be achieved either by general-purpose programming languages such as Java and Python or using task-specific languages such as Acceleo [1], ATL[2] or the languages of the Epsilon platform [8].

Many industrial projects attempt to represent the system with models that minimise accidental complexity and use concepts which are close to the domain [4][10][18]. As projects become large, models grow large as well, and this pushes the current generation of model management tools and technologies to their limits. When model management runs on pay-as-you-go cloud-based resources, this inefficiency and reduced scalability incurs additional cost. Hence, there is vested interest from vendors of cloud-based low-code platforms efficient and scalable model management.

The primary reason for these limitations lies in the way that these technologies interact with models. For example, when a model management program needs to load and process (e.g., transform, validate) a model:

- If the model is a file-based model (e.g. XMI), as the execution engine does not know which parts of the model the program will access, the entire model needs to be loaded into memory.
- If the model is a repository-based model (e.g. CDO [15], Neo4EMF [3], Hawk [1]), the execution engine does not know which attributes/references of a model element the program will need to access. Thus, it either retrieves all or none of them when it retrieves the model element. If the execution engine retrieves none, then it needs to access the model repository every time the program needs an attribute/reference. If the execution engine retrieves all of them, some of them will occupy memory without ever being accessed by the program.
- In both formats, the execution engine keeps everything in memory until the end of the model management program, because it does not know when the program no longer needs certain parts of the model. This is inefficient in terms of memory use.

To summarise, the interaction of model management program execution engines with models (file-based or repository-based) is too "short-sighted" in the absence of static-analysis-based execution planning mechanisms. This limitation results in increased model loading time and increased memory consumption. This paper introduces an approach that can help execution engines for model

[1]https://www.eclipse.org/acceleo/
[2]https://www.eclipse.org/atl/

management programs to handle large models more efficiently. In our approach, by using in-advance knowledge about the program -provided by static analysis-, the execution engines will be able to identify, load and process model partitions which contain model elements of interest. This approach aims to enable model management programs to load only parts of models that are required for the execution of a program and eliminate the overhead of loading and keeping in memory unnecessary parts.

The remainder of this paper is structured as follows. Section 2 provides a motivating example and Section 3 discusses the proposed methodology. Section 4 provides an overview of the related work. Section 5 describes our current progress and outlines our next steps for the implementation of the proposed approach.

2 MOTIVATING EXAMPLE

Consider the domain-specific modelling language for implementing low-code form-based applications (from Figure 1). According to this modelling language, every *Application* consists of a number of *Entity* and *Form* elements. A *Form* has a reference to an *Entity*. Each *Entity* can be composed of *Properties* where every *Field* is assigned to at most one *Property*. For implementing the low-code form-based application, *Entity* and *Property* elements are used for generating the database schema, back-end CRUD code and web services. Furthermore, from *Form* and *Field* elements we can generate the front-end of the application (e.g. HTML/iOS/Android front-end code).

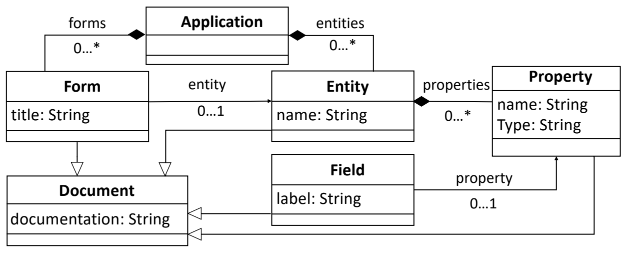


Figure 1: Form-based low-code application metamodel

Let us consider the back-end code generation scenario where a developer wishes to re-generate the back-end of an application (or of all applications hosted in the low-code platform) and leave the front-end intact. Assume this re-generation task involves an optimisation in the generated back-end code or the migration to a different database or web-services framework.

The back-end generator could be written in a model-to-text transformation (M2T) language such as the Epsilon Generation Language (EGL) [13] as shown in Listing 1. There is an *Entity2Class* rule which transforms every *Entity* to a Java class using the EGL code. EGL is a member of the Epsilon family language which is used for generating code from a model.

Listing 1: EGL transformation rule for generating part of the low-code application back-end

```
rule Entity2Class
transform e : Entity {
template : 'entity2class.egl'
target : "src-gen/" + e.name + ".java"}
```

In Listing 1, for every Entity (e is an instance of Entity) in the model, the program invokes the *entity2class.egl* template and stores the generated class in a .java file. The *entity2class.egl* template is shown in Listing 2. In order to generate a class from *Entity*, the name of class is assigned according to the name of *Entity* and from every *Property*, a new field of the class is declared. The *entity2class.egl* only accesses the name of the *Entity* and the names and types of its *properties* but not the *Form* or *Field* elements of the model.

Listing 2: The entity2class.egl template

```
public class [%=Entity.name%]{
    [% for (p in Entity.properties){ %]
        [%=p.type%] [%=p.name%] = new [%=p.type%]();
    [%}%]}
```

To load a model for executing this program, there are two possibilities.

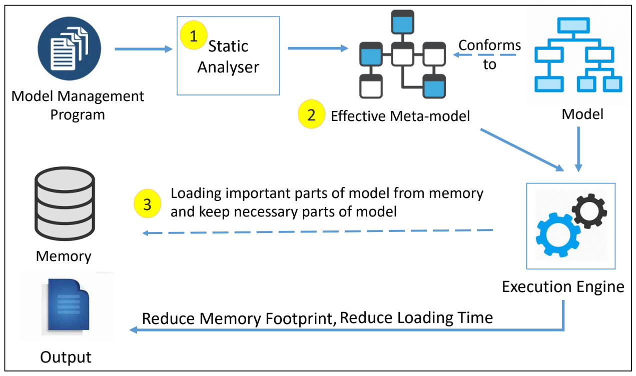
- If the models are file-based (e.g. XMI or XML-based), the EGL execution engine needs to decide in advance, which parts of them it will load in memory, as re-parsing the same model file several times can be expensive. In the absence of in-advance static analysis of the generator (M2T), the UI-related parts of the application model would be loaded as well, despite the fact that they will not be used by the back-end generator.
- If the models are not file-based (e.g., stored in a database-backed repository such as CDO or Hawk), the EGL execution engine can retrieve model elements on demand. Still, in the absence of static analysis, there is no way to tell which features of these model elements should be retrieved from the repository for each element. In this situation, there are two alternatives: either *greedily* fetch all features in advance or *lazily* fetch all features on demand. The former strategy favours execution time over memory consumption, while the second strategy requires less memory, but potentially multiple round-trips to the repository (detrimental to performance). Considering Listing 2 that *entity2class.egl* uses the name of the *Entity* and the names and types of its *properties* but not their documentation, the two strategies are sub-optimal:
  - Greedy: The documentation of the entity and its properties is fetched from the repository but is never used by the generator, thus wasting memory.
  - Lazy: Multiple round-trips to the repository are required to fetch the values of the name/type features of each accessed entity and property in the model, thus degrading performance.

  A static-analysis-based execution planner could determine which features are (not) accessed by the generator in advance and query the repository accordingly (e.g., populate the *name* and *type* of each *Field* in one go, but leave out the *documentation* which is not required).

As the *entity2class.egl* (Listing 2) only accesses features and properties of the *Entity* itself (i.e., it doesn't reach out to other *Entity* elements), after the execution of rule *Entity2Class* for a given entity, that entity is no longer needed and could be offloaded from memory to reduce the overall footprint of the generator. In the absence of in-advance static analysis, this cannot be determined by the generator. Therefore, all *Entity* elements will be kept in memory until the entire generation has concluded.

3 APPROACH

This paper introduces an approach which helps execution engines of model management programs to handle large models more efficiently. The general goal of this approach is reducing loading time and memory footprint, which is achieved by using static analysis for generating an execution plan. The proposed approach includes three main steps and illustrated in Figure 2.



Model Management Program
1 Static Analyser
2 Effective Meta-model
Conforms to
Model
3 Loading important parts of model from memory and keep necessary parts of model
Memory
Execution Engine
Reduce Memory Footprint, Reduce Loading Time
Output

Figure 2: The proposed approach

(1) *Static Analyser*
In the first step of our approach, a model management program is provided as input to a static analyser. The role of the static analyser is compiling the program and analysing it using the Abstract Syntax Graph. Static analysis of Epsilon started in [16] where the Abstract Syntax Tree of an EOL program is computed, then resolution algorithms including variable resolution (e.g., resolving identifiers to their definitions) and type resolution (e.g., primitive types and collection types) applied to derive an Abstract Syntax Graph. Thus, using the Abstract Syntax Graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program). Type inference is a pre-process activity which helps the static analyser to extract useful information for execution planning. The execution plan contains the information which helps the execution engine to load and process models efficiently (e.g., what part of the model should be loaded each time, which parts should be disposed of from memory, when each of these activities should occur). A part of execution plan is in the form of an effective metamodel which is considered as an output of the static analyser. The effective metamodel is a subset of the model's metamodel which consists of only types and properties of interest [16] (see below).

To illustrate how every step of the approach works, we use the motivating example of Section 2. Considering Listing 2, the model management program *entity2class.egl* only uses the name of the *Entity* and the names and types of its *properties*; the remaining information in the model is not required for executing this program (such as their documentation). In the motivating example, the static analyser detects the elements of the model that are necessary for executing the EGL program. Since EOL is the core language of the Epsilon platform, our approach applies to all model management languages of the platform. Also, the underlying principles, subject to suitable technical modifications, can be applied to the other modelling languages, e.g., ATL. Hence, instead of loading all model elements, we need to load only instances of *Entity* and *Property* and only the values of their *name/type* fields. This information is obtained by static analysis facilities of the EGL program.

It is worth noting that using static analysis is not only about extracting information to load model elements on demand. Using static analysis is useful to define the disposing strategy as well. By using the execution plan, the execution engine can detect which parts of the model are needed and how long this information should be kept in memory. In this way, the execution engine has a plan for executing the program to keep the parts of the model until the program needs them for execution. After that, if the program does not need elements anymore, the execution engine could unload them from memory (see memory management part).
(2) *Effective Metamodel*
The output of static analyser is in the form of an effective metamodel for each model involved in the program. The concept of effective metamodel was introduced in [17] in order to support partial loading of XMI files. As for partial loading, we need only the parts of the model which are accessed by the program. In our approach, we load each model according to its effective metamodel instead of the original one.

The structure of effective metamodel is presented in Figure 3. It consists of two classes: *EffectiveMetamodel* class with *name* and *nsuri* attributes and *EffectiveType* with *name*, *attributes* and *references*. The *EffectiveMetamodel* class connects to an *EffectiveType* by *allOfKind*, *allOfType* and *types* references. Figure 4 illustrates the effective metamodel of the EGL program from our motivating example. In comparison with the original metamodel, the effective metamodel does not include any additional information such as Field and Form classes and their properties. Thus, loading the model according to effective metamodel is expected to be more efficient. The attributes of EffectiveMetamodel class are filled by the original metamodel, which are the name and namespace URI (i.e., unique ID in terms of EMF terminology) of the metamodel. Two effective types refer to classes which are necessary for executing the EGL program; Entity and Property in this case. The attributes of classes are according to the attributes which are needed to access the code (e.g., name). Thus, in this step, the static analyser helps the execution engine in extracting elements of the model which are necessary for executing the program at compile time in the form of an effective metamodel.

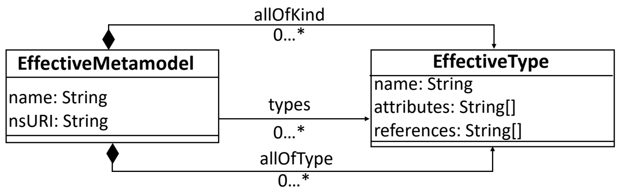


Figure 3: The structure of effective metamodel [17]

(3)*Memory management*
In the next step, the effective metamodel and the model that conforms to it are sent to the execution engine as inputs. Loading only relevant parts of a model into memory is an efficient way to reduce the time of loading but the way that the engine plans to load these parts of model is important.

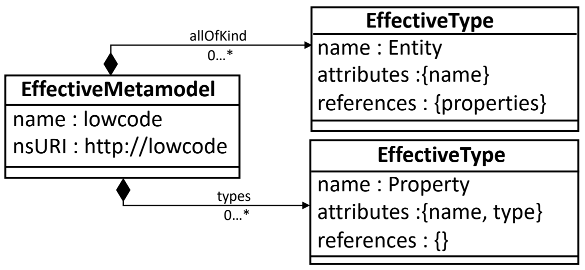


Figure 4: Effective metamodel of EGL code

Considering the motivating example (Listing 1), for applying the *Entity2Class* rule on every *Entity*, one way to partition the model is by *Entity*. Based on this partitioning plan, the execution engine could load every Entity, process it and dispose it every time the program finishes using that Entity. The engine could load each Entity in every network connection (when model is stored in repository) which is efficient in memory but an unsuitable way in terms of performance. On the other hand, the execution engine can load all Entity elements in one connection to the network. Loading all *Entity* elements of model in memory is not efficient as all elements should be kept in memory during the program execution, incurring additional memory consumption. Thus, the execution engine should consider a trade-off between performance and memory consumption for loading model partitions. Consequently, there is a need for intelligent partitioning of models underpinned by sophisticated strategies that use information extracted from code (model management programs) at compile time.

In addition to loading the necessary partitions of the model, another concern is memory consumption. When loaded model elements are no longer needed by the model management program, a sophisticated strategy is required for disposing them from memory. In Listing 1, if the execution engine executes the program for all entities, after the Java class of an entity is generated, there is no need to keep that entity in memory any more. Hence, the memory consumed by that entity can be freed or become available for loading more elements.

The last phase of this approach is producing an output after executing the program. As there are different types of model management programs, the output would be a model (executing a model transformation program) or code (executing a code generation program). Using this approach, we expect that the program output will be produced more efficiently with reduced loading time because of using intelligent loading strategies and with less memory consumption due to unloading unnecessary model parts.

4 RELATED WORK

Providing infrastructure for storing and indexing large models is an essential aspect of scalable MDE. The common format which is currently used for storing models is XML Metadata Interchange (XMI). While XMI is a suitable way of storing small models, it has some limitations when it deals with large models.

In the case of XMI alternatives, Jouault et al. [6] introduced an open binary format known as Binary Model Syntax (BMS), and they claimed that BMS files are three times smaller than corresponding XMI files. So, it can be a high-performance alternative to XMI. However, while BMS was introduced in 2009, there has not been any update or release of this format in the public domain until now.

To achieve partial loading, some of the related works propose database-backed persistence technologies. The mature ones are CDO [15], Morsa [12] and Neo4EMF [3].

The Connected Data Object (CDO) is a model repository for EMF models. Metamodels and models can be stored in all kinds of database backends like major relational databases or NoSQL databases. It is also a framework built on top of the EMF, which provides the persistence of large models. CDO supports scalability, which is achieved by object loading based on on-demand strategies and caching them in the application. Hence, it does not keep the objects which are no longer referenced by the application, and they are collected from the memory automatically. Although CDO claims to be able to load models up to 4GB, experimental evaluation with Intel CoreI5 760 PC at 2.80GHz with 8GB of physical RAM in [12] reported an upper bound of 271MB.

Morsa is a persistence solution for storing and accessing large models based on on-demand strategies, which is supported by the NoSQL database. Morsa uses MongoDB, a document-oriented database, as its persistence backend. Morsa provides clients with a partial load of large models using a load on-demand mechanism. Morsa satisfies scalability requirements, but it is just a prototype, and there is no update that they plan to consider crucial issues like security in order to deploy this prototype in an industrial context [9]. Also, choosing the cache policy is manual, and the user should select the policy using a GUI [12].

Neo4EMF is a persistence layer for EMF models. It is built on top of the graph-based database Neo4j, as these databases are able to manage large-scale data on highly distributed environments. Moreover, Barmpis and Kolovos [2] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models. Neo4EMF is similar to Morsa in several aspects (notably in on-demand loading), but it aims at exploiting the optimized navigation performance offered by graph-databases. While Neo4EMF is a more performant alternative to XMI due to high-performance access and on-demand loading, its raw performance does not surpass a more mature solution like CDO [3].

SmartSAX is another prototype which was introduced in [17]. It supports partial loading of XMI model files. The main idea is providing in-advance knowledge of a program (which kind of model elements and which of their properties are accessed by the model management program) can be used to partially load only a subset of XMI based EMF model into memory.

SmartSAX also has some limitations. First, it just supports read-only XMI files (does not support changes made to partially loaded models). Also, as partial loading can affect the internal structure of the XMI model, elements should have IDs that do not depend on their position in the containment hierarchy. Finally, SmartSAX currently does not support loading models that are persisted in multiple XMI files. Also, it does not support garbage collection to unload unnecessary parts of model from memory.

Although recent research has made advancements in this area, existing solutions has clear shortcomings in accessing and processing large models. The first shortcoming is about loading models.

Repositories such as Morsa, CDO provide remote accessing of large models and store them in a graph-based or relational database. Still, as some tools are based on EMF, and the common format for storing models is XMI, there is a need for partial access to XMI models, which loads models using on-demand strategies. In addition, loading and storing models by elements is not an efficient way, so the second challenge is about partitioning models. Intelligent strategies for grouping model elements as a partition are needed. Finally, intelligent unloading strategies are needed. Keeping part of models loaded into memory that will not be used further increases the memory footprint unnecessarily. Hence, unloading them when the program does not refer to them anymore would be a solution for reducing the memory consumption of model management programs.

5 CURRENT PROGRESS AND NEXT STEPS

We presented a new approach that will enable model management languages and engines to eliminate the overhead of loading unimportant parts of models (i.e. parts that they will never access) and of unnecessarily keeping obsolete parts (i.e. parts that have already been processed and are guaranteed not to be reaccessed) in memory.

The proposed approach has three main steps. We have made significant progress in implementing the first step. The static analyser for EOL language, which is the core language of Epsilon, was started in [16]. Building on this preliminary work, we added new features to the EOL engine to get more information such as return type compatibility, type compatibility of context and parameters from code at compile time. Compile-time errors are the by-product of its implementation, which shows that we have more accurate information about code.

According to the designed approach, we use a static analyser for extracting the effective metamodel, which has information about referenced model elements in code. We are currently working on extracting the effective metamodel in order to detect relevant model elements starting from the incomplete algorithm introduced in [17].

Now that a static analyser has been implemented for EOL, we plan to extend this facility to other specific languages of the Epsilon framework like the Epsilon Validation Language (EVL) [7]. For models which are stored in repositories, we need to load model elements based on intelligent partitioning of models. Hence, there is a need for finding an algorithm to partition the model in order to load every part of model in an efficient way. This algorithm would be useful in the way that enables the engine to remove elements that are no longer used from memory and only keep parts that are required for executing the rest of the program.

In order to evaluate our project, we will compare our approach to other recent works like Neo4EMF and CDO. The goal of this work is to propose methods for reducing the time of loading and memory footprint when model management tasks are applied on large size models. Hence, memory footprint and loading time are the factors that we compare by running model management programs with and without the intelligent model partitioning and disposal facilities. We have considered the models proposed in the GraBaTs 2009 contest as test cases [14]. The models conform to the JavaMetamodel metamodel. There are five models, from Set0 to Set4, each one larger than its predecessor (from a 8.8MB XMI file with 70447 model elements representing 14 Java classes to a 646MB file with 4961779 model elements representing 5984 Java classes).

6 ACKNOWLEDGMENTS

This research is supported by the Lowcomote Training Network, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska-Curie grant agreement n° 813884.

REFERENCES

[1] Konstantinos Barmpis and Dimitris Kolovos. 2013. Hawk: Towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM Press, 1–9.

[2] Konstantinos Barmpis and Dimitrios S. Kolovos. 2012. Comparative Analysis of Data Persistence Technologies for Large-Scale Models. In *Proceedings of the 2012 Extreme Modeling Workshop (XM '12)*. Association for Computing Machinery, 33–38.

[3] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. 2014. Neo4EMF, a scalable persistence layer for EMF models. In *European Conference on Modelling Foundations and Applications (Lecture Notes in Computer Science, Vol. 8569)*. Springer, 230–241.

[4] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Association for Computing Machinery, 471–480.

[5] Ari Jaaksi. 2002. Developing mobile browsers in a product line. *IEEE software* 19, 4 (2002), 73–80.

[6] Frédéric Jouault, Jean Bézivin, and Mikaël Barbero. 2009. Towards an advanced model-driven engineering toolbox. *Innovations in Systems and Software Engineering* 5', 1 (2009), 5–12.

[7] Dimitrios Kolovos, Richard Paige, and Fiona Polack. 2009. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis (Lecture Notes in Computer Science book)*. 204–218.

[8] Dimitrios Kolovos, Louis Rose, Richard Paige, and Antonio Garcıa-Domınguez. 2010. *The Epsilon Book*. Eclipse.

[9] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. 2013. A Research Roadmap towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering (BigMDE '13)*. Association for Computing Machinery, 1–10.

[10] Juha Kärnä, Juha-pekka Tolvanen, and Steven Kelly. 2009. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*.

[11] Parastoo Mohagheghi and Vegard Dehlen. 2008. Where is the proof?-A review of experiences from applying MDE in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 432–443.

[12] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. 2011. Morsa: A scalable approach for persisting and accessing large models. In *International Conference on Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science, Vol. 6981)*. Springer, 77–92.

[13] Louis Rose, Richard Paige, Dimitrios Kolovos, and Fiona Polack. 2008. The Epsilon Generation Language. In *European Conference on Model Driven Architecture - Foundations and Applications (Lecture Notes in Computer Science book, Vol. 5095)*. Springer, 1–16.

[14] Jean-Sébastien Sottet, Frédéric Jouault, et al. 2009. Program comprehension. In *5th International Workshop on Graph-Based Tools (GraBaTs 2009)*. Citeseer, Zurich (Switzerland).

[15] Eike Stepper. 2016. *CDO*. Retrieved June 5, 2020 from https://wiki.eclipse.org/CDO

[16] Ran Wei and D.S. Kolovos. 2014. Automated analysis, validation and suboptimal code detection in model management programs. In *CEUR Workshop Proceedings*, Vol. 1206. 48–57.

[17] Ran Wei, Dimitrios S. Kolovos, Antonio Garcia-Dominguez, Konstantinos Barmpis, and Richard F. Paige. 2016. Partial Loading of XMI Models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16)*. Association for Computing Machinery, 329–339.

[18] Marc Zeller, Daniel Ratiu, and Kai Höfig. 2016. Towards the Adoption of Model-Based Engineering for the Development of Safety-Critical Systems in Industrial Practice. In *International Conference on Computer Safety, Reliability, and Security (Lecture Notes in Computer Science, Vol. 9923)*. Springer, 322–333.