Proceedings Paper:
Jahanbin, Sorour, Kolovos, Dimitris orcid.org/0000-0002-1724-6563, Gerasimou, Simos orcid.org/0000-0002-2706-5272 et al. (1 more author) (2022) Partial Loading of Repository-Based Models through Static Analysis. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022). Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022), 06-07 Nov 2022 ACM , NZL , 266–278.

https://doi.org/10.1145/3567512.3567535

Partial Loading of Repository-Based Models through Static Analysis

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

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

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

Gerson Sunyé
gerson.sunye@ls2n.fr
University of Nantes
Nantes, France

Abstract

As the size of software and system models grows, scalability issues in the current generation of model management languages (e.g. transformation, validation) and their supporting tooling become more prominent. To address this challenge, execution engines of model management programs need to become more efficient in their use of system resources. This paper presents an approach for partial loading of large models that reside in graph-database-backed model repositories. This approach leverages sophisticated static analysis of model management programs and auto-generation of graph (Cypher) queries to load only relevant model elements instead of naively loading the entire models into memory. Our experimental evaluation shows that our approach enables model management programs to process larger models, faster, and with a reduced memory footprint compared to the state of the art.

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

Keywords: partial loading, memory management, repository-based model, cypher language, model-driven engineering

ACM Reference Format:
Sorour Jahanbin, Dimitris Kolovos, Simos Gerasimou, and Gerson Sunyé. 2022. Partial Loading of Repository-Based Models through Static Analysis. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3567512.3567535

SLE '22, December 06–07, 2022, Auckland, New Zealand

1 Introduction

In MDE, models are manipulated using model management programs that carry out different tasks such as model transformation, model merging, model validation, etc. As models grow in size, MDE tools face scalability problems. One of them is the ability of execution engines of model management languages to scale up for models of growing size in terms of execution time and memory usage [8].

The primary reason for this scalability issue lies in the way that most contemporary 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), then all the required information from it needs to be read upfront, before the execution of the program. If the model is stored in a database (e. g. Neo4j), then we can issue multiple queries to the database, to fetch model elements of interest (and their properties) progressively, as they are needed for the execution of the program.

To summarise, the interaction of model management program execution engines with large models (file-based or repository-based) can be too "short-sighted" in the absence of static-analysis-based model loading and caching mechanisms. This can result in increased model loading times and unnecessary memory consumption. This paper introduces an approach that can help execution engines of model management programs handle larger models more efficiently. In our approach, by using in-advance knowledge about the program provided by static analysis, execution engines are able to identify and load only parts of model that are likely to be accessed by the program.

The main contributions of this work are:

- An algorithm for extracting the effective metamodel of model management programs, which are executed against models in graph-based model repositories.
- An algorithm for translating a program's effective metamodel into a set of efficient graph queries that only return elements, relationships and properties that the program is likely to exercise at run-time.

- A prototype implementation of these algorithms using the Eclipse Epsilon family of model management languages and the Neo4J graph database.

The remainder of the paper is structured as follows. In Section 2, a motivating example is provided that explains the challenges of interest in more detail. In Section 3, the proposed approach is discussed and the limitations of our approach are mentioned. Related work is reviewed in Section 4. Section 5 reports on the result of the evaluation of our approach compared to the state of the art, and finally, Section 6 concludes the paper and outlines directions for future work.

2 Motivating Example

As a motivating example, consider a model that conforms to a contrived Project Scheduling Language (PSL), the UML class diagram of which is shown in Figure 1. According to the PSL metamodel, each *Project* has a title and a description, and it consists of *Task*s and *Person*s. *Task*s can be completed through automated means (*AutomaticTask*s) or manually (*ManualTask*s). All *Task*s have a title but only *ManualTask*s have a duration, and a start time. Also, each *ManualTask* specifies the *Effort* that different *Person*s in the project will contribute to it (as a percentage of their time).

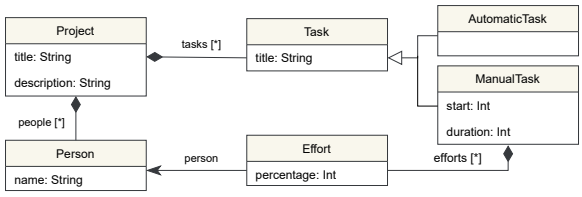

Figure 1. PSL Metamodel

Consider a model that conforms to the PSL language (see Figure 1), and on which we would like to print the number of people who contribute to each *ManualTask*. This program could be written in a language such as the Epsilon Object Language (EOL)[1] [7] as shown in Listing 1.

Listing 1. EOL program to print number of people who contribute to each task

```
1 for(task:ManualTask in ManualTask.all()){
2   task.title.print();
3   task.efforts.person.asSet().size().println();
4 }
```

Epsilon[2] is a platform that provides task-specific languages for common model management activities such as model transformation, code generation, merging, validation, and refactoring. The core language of the platform is the Epsilon Object Language (EOL), which is a model-oriented programming language, that provides common facilities for developing task-specific model management languages.

In Epsilon's architecture, there is an Epsilon Connectivity Layer (EMC)[3]) which enables Epsilon programs to interact with models in different modelling technologies in a uniform manner by defining the drivers (e. g., EMF, CDO, NeoEMF).

In Listing 1, line 1 defines the *task* variable and it goes through all instances of *ManualTask*. In lines 2-3, the *title* of each *task* and the number of people who contribute to each *task* are printed.

The program shown in Listing 1 only accesses the *title* attribute and *efforts* reference of *ManualTask*, the *person* reference of *Effort* and no other properties of the PSL metamodel. To run this program against a repository-based model (e. g. stored in a database-backed repository such as CDO or NeoEMF), the EOL engine uses the respective driver to fetch all instances of model element types and properties of model elements on demand. Therefore, without using in-advance static analysis of the EOL program, there is no way to tell before executing the program which features of the required model elements should be retrieved from the repository. In this situation, there are two alternatives at runtime: either *greedily* fetch all properties and attributes of model elements retrieved from the database or *lazily* fetch attributes and references 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, which can be detrimental to performance.

Considering Listing 1 that uses the *title* attribute and *efforts* reference of *ManualTask*, the *person* reference of *Effort* and no other attributes or references, the two strategies are sub-optimal:

- Greedy: When all instances of *ManualTask* are fetched in line 1, all their attributes and references would be fetched too (including *ManualTask.duration*). As *ManualTask.duration* is not accessed by the EOL program, fetching its value from the repository and maintaining it in memory is wasteful.
- Lazy: Using this approach, in line 1, only skeletons of *ManualTask* elements would be initially fetched from the database. Then in line 3, for each *ManualTask*, the program would need to go back to the database and fetch the value of its *efforts* reference. So, multiple round-trips to the repository to fetch the value of the attribute or references of each model element are required. These trips are time-consuming.

3 Proposed Approach

The overall goal of this work is to reduce the loading time and memory footprint of repository-based models consumed by model management programs through static analysis of said programs. In our approach, a static analyser can determine

[1] https://www.eclipse.org/epsilon/doc/eol/
[2] https://www.eclipse.org/epsilon/
[3] https://www.eclipse.org/epsilon/doc/emc/

which features or all instances of which specific types (e. g., *ManualTask*) are likely[4] to be accessed by the program in advance.

This information can be used to fetch a subset of the model that is likely to be accessed by the program from the database in one go just before the program is executed (e. g., populate the *efforts* reference of each *ManualTask* in one go, but leave out the *start* attribute which is not required). Our expectation is that this approach will be more efficient in terms of memory and time compared to the greedy and lazy strategies described above.

In terms of concrete technologies, we use Neo4J as a graph-based model repository[5] and model management programs written in languages of the Epsilon platform—but the approach is also applicable to other similar technologies (e. g., OrientDB[6] and OCL [2], ATL [6] or Acceleo[7]). A high-level overview of our approach is presented in Figure 2. The main components of this approach are represented in grey colour and they are labeled with numbers 1 to 3.

3.1 Static Analysis

In the first step of our approach (see Figure 2), a model management program and the metamodels of the models it consumes are provided as the input to a static analyser. The static analyser computes the abstract syntax tree of the program. Then, resolution algorithms, including variable resolution and type resolution, are applied to derive an abstract syntax graph [12]. Using the abstract syntax graph, the static analyser can extract relevant information (i.e., types and properties accessed by the program).

The output of the static analyser is an *effective metamodel* for every model accessed by the program. The effective metamodel is a subset of the model's original metamodel, which consists only of types and properties that are likely to be accessed by the program [12] (see Section 3.2).

While static analysis supports multiple models (and therefore produces multiple effective metamodels), in the remainder of the paper, we will only consider programs with one model (and therefore one effective metamodel). To illustrate how every step of the approach works, we use the motivating example from Section 2.

In the first step of our approach, the static analyser sets the resolved types of expressions to types from the respective metamodels or to primitive types (e. g., String, Integer). For example, in line 1 of Listing 1, the resolved type of *task* variable is equal to *ManualTask*. In line 2, the property call is supposed to print the *title* of each *task*. The *title* is an attribute of *task* and the resolved type is *String*. Then, line 3 accesses *task.efforts.person* and *task.efforts* is a property call where the target of this call is a model element (*ManualTask*) and the feature which is called is *efforts*. In this case, *efforts* is a reference of *ManualTask* of *Effort* type. Table 1 shows the resolved types of expressions which are extracted from Listing 1 by the static analyser.

3.2 Effective Metamodel Computation

The second step of the approach is the extraction of the effective metamodel of the model consumed by the program from its abstract syntax graph. The concept of effective metamodel was introduced by Wei et al. [13]. The effective metamodel is constructed using Algorithm 1 (described later in the paper) that uses the resolved types of expressions, and contains only types which are necessary for executing the program from which it is extracted. In our prototype implementation, effective metamodels are only computed for EMF-based models, but in principle, this approach can be applied to other metamodelling technologies too.

As shown in Figure 3, an effective metamodel consists of an *EffectiveMetamodel* class with *name* and *nsuri* attributes. The *EffectiveMetamodel* class is connected to an *EClass* that has *EStructuralFeature*s. The *EffectiveMetamodel* class is connected to an *EClass* by *allOfKind*, *allOfType* and *types* references.

The *allOfKind* and *allOfType* references specify the instances of types that the execution engine should load. The difference between these two references is that *allOfKind* is used when all instances of a class (including subclasses) should be loaded. In contrast, *allOfType* reference means the execution engine should consider only the elements that are direct instances of the class (without considering any of its subclasses). The *types* reference is used for specifying class instances of which should be loaded only when they appear in the references of model elements of interest.

For every class used in the program (such as *ManualTask* and *Effort*), an *EClass* is added in the respective effective metamodel. The *EClass* contains collections of *structural features* that reflect the attributes and references of the type accessed by the program.

The process which extracts this effective metamodel from an EOL program is described in Algorithms 1 and 2. This algorithm is easily extendable for other Epsilon languages but considering the motivating example, we will discuss the version that works with EOL program in this section.

In Algorithms 1 and 2, the Abstract Syntax Graph, which is extracted by the static analyser, is visited. Algorithm 1 is interested in calls of *all()* and *allInstances()* operations and *property call*s (such as *ManualTask.efforts*) as they are the only way to navigate to model elements in Epsilon programs. In lines 7-11, the *all()* and *allInstances()* operation calls are handled by Algorithm 1 to add the respective *EClass*es to the effective metamodel. If the target of the operation call is a model element type, then it will be added to an *allOfKind*

[4] In some cases, the execution engine needs to load unnecessary model properties as well because of a lack of information before the execution (see Section 3.2.1).
[5] https://neo4j.com
[6] https://orientdb.org/
[7] https://www.eclipse.org/acceleo/

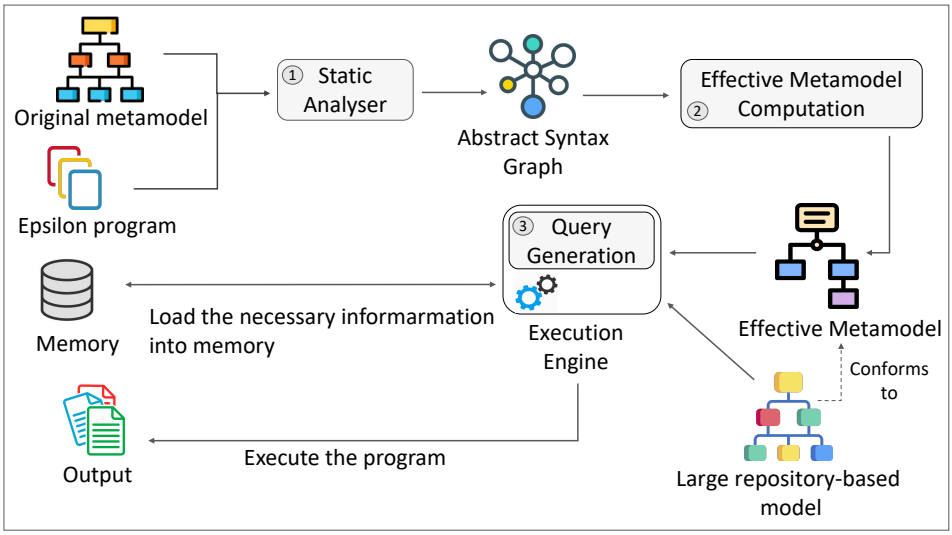

Figure 2. The proposed approach

Table 1. Resolved Types Calculated by the Static Analyser

Line number in Listing 1	Expression	Resolved Type
1	task	Model Element (ManualTask)
1	ManualTask.allInstances()	Operation call expression (Sequence<ManualTask>)
2	task.title	Property call expression (String)
2	task	Model Element (ManualTask)
3	task.efforts.person.size()	Operation call expression (Integer)
3	task.efforts.person	Property call expression (Sequence<Person>)
3	task.efforts	Property call expression (Sequence<Effort>)
3	task	Model Element (ManualTask)

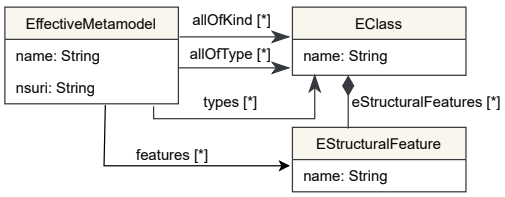

Figure 3. The structure of effective metamodel (adapted from [13])

reference of the effective metamodel (line 11) and, if it already exists in the effective metamodel under the *allOfType* or *types* references, the *EClass* is moved to the *allOfKind* reference.

In lines 15-28, Algorithm 1 handles property calls to further populate the effective metamodel. In Algorithm 2, if the accessed property is *all* (it is an alias for *allInstances()*), then the target element type will be treated identically to the *allOfKind* operation call (lines 2-3). When the target of property call is a model element, if an attribute of the model element is accessed, it is added to the effective metamodel as an *EAttribute* (lines 8-9), or if it is a reference, then it is added as an *EReference* (lines 10-11).

Figure 4 illustrates the effective metamodel extracted from the EOL program in our motivating example (Listing 1). In Figure 4, the attributes of the *EffectiveMetamodel* class are filled by the original metamodel, which are the *name* and the *nsuri* of the metamodel. For running the EOL program, all instances of *ManualTask* must be loaded. The *ManualTask* class is added to *EffectiveMetamodel* under the *allOfKind* reference according to lines 7-11 of Algorithm 1. The *title* attribute of *task* is added to the *EffectiveMetamodel* according to lines 8-9 of Algorithm 2. The *efforts* reference of *ManualTask* is also required (line 3 of Listing 1), hence, it is added to *ManualTask* as an *EReference* according to lines 10-11 in Algorithm 2. The resolved type of *efforts* reference is equal

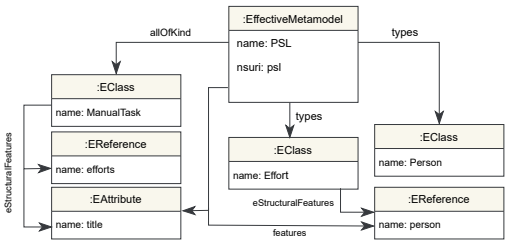

Figure 4. Effective metamodel of the EOL program shown in Listing 1

to *Effort* (see Table 1), so according to lines 12-13 in Algorithm 2, the *Effort EClass* is added to effective metamodel using the *types* reference and the *Person EClass* is added to the effective metamodel using the *types* reference.

3.2.1 Accommodating Untyped Variables and Expressions. Algorithm 1 visits the abstract syntax graph to consider all statements and expressions in the code. Thus, constructing the effective metamodel relies on the ability of the static analyser to precisely resolve their types. If in a property call expression the resolved type of the left hand side is unknown (*"Any"* in terms of the Epsilon type-system), then the name of the property is added to an *unresolvedProperties* set. After the effective metamodel has been extracted, the *unresolvedProperties* set is used to augment the effective metamodel with additional *FeatureAccess* elements for all the types of the effective metamodel that have features matching properties in the set. For example, in Listing 2, the *title* of the first *Project* of the input model is printed. All instances of *Project* with *title* attribute and all instances of *Task* are required for running this part of program.

Listing 2. EOL Example Code

```
1 var p = Project.all().first();
2 var t = Task.all().first();
3 p.title.println("Title: ");
```

In the first line of Listing 2, the first item of all instances of *Project* in the model is assigned to *p* variable. As the type of *p* is undefined, the resolved type of *p* is considered as *Any*. Hence, the condition in line 22 of Algorithm 1 is not satisfied, and the *title* attribute of *Project* is not added to the effective metamodel in the first iteration.

Algorithm 1 handles this situation by considering possible (instead of precise) types for variables and expressions. While this is the case for this minimal example, in the general case, the type of variables assigned in more than one places in a program cannot be resolved reliably.

In lines 1-2 of Listing 2, according to lines 9-16 of Algorithm 1, *Project* and *Task EClass*es are added to the effective metamodel. Then, the *title* attribute is accessed by the program but as the resolved type of *p* is *Any*, it adds the *title* attribute to all *EClass*es that are already in the effective metamodel. Hence, according to lines 8-9 in Algorithm 2, *title* will be added to the effective metamodel for the *Project* and *Task* EClasses and the execution engine will load the *title* of all instances of *Project* and the *title* of all instances of *Task* into memory.

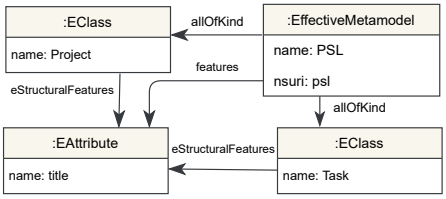

Figure 5. Extracted effective metamodel for Listing 2

As shown in Figure 5, the effective metamodel includes an attribute which is not necessary for running the program (*title* of *Task*), but because of lack of information before the execution and to be on the safe side, the execution engine loads more information from the repository. Loading more information and running the program is preferable to loading less information than required at runtime. As during the execution of the Epsilon program, more *EClass*es may be added to the effective metamodel, in line 2 of Algorithm 1, there is a loop which calculates the effective metamodel until the point that no further changes are made to it. As the time consumption of extracting the effective metamodel is negligible, repeating the algorithm will not have a great impact on the efficiency of our approach.

3.3 Query Generation

3.3.1 Mapping EMF Models to Graph Databases. In this work we are concerned with the efficient management of models that reside in graph-based model repositories as these have been shown to outperform model repositories backed by relational/document databases [1, 4]. A state-of-the-art graph-based model repository is NeoEMF, which enables the persistence of EMF-based model in Neo4J graph databases (among others). Our first attempt was to implement our efficient model loading approach on top of NeoEMF, however the framework cannot support partial model element loading without a significant amount of refactoring. Therefore, we chose to implement a custom mapping of EMF models to Neo4J databases, which we discuss in this section. Figure 6 shows a model which conforms to the PSL metamodel in Figure 1. The *project* named as "ACME" consists of three *task*s: "Design" is a *ManualTask* which has to be completed by Bob (40 % *effort*) and Alice (60 % *effort*); "Implementation" is also a *ManualTask* equally split among Alice and Bob (50 % each) and "Meeting organisation", which is an *Automated Task*.

In order to map EMF-based models such as this one to a Neo4J graph, we use the mapping strategy illustrated in

Algorithm 1 EOL Effective Metamodel Extraction Algorithm (1 of 2)

```
1:  procedure computeEffectiveMetamodel
2:      while No further changes are made to the effective metamodel do
3:          let EM = New effective metamodel;
4:          for all operation call expression do
5:              if IsModelElement(operation.target) then
6:                  let EC = target.type;
7:                  if operation.name.equals(all or allOfKind or allInstances) then
8:                      if allOfType.contains(EC) or types.contains(EC) then
9:                          move EC under EM's allOfKind reference;
10:                     else
11:                         allOfKind.add(EC);
12:                 else if operation.name.equals(allOfType then
13:                     if not allOfKind.contains(EC) and not allOfType.contains(EC) then
14:                         allOfType.add(EC);
15:         for all property call expression do
16:             if IsModelElement(propertyCall.target) then
17:                 let EC = target.type
18:                 handlePropertyCallExpression(propertyCallExpression,EC)
19:             else if IsCollection(propertyCall.target) then
20:                 if IsModelElement(collection.content) then
21:                     let EC = collection.content.type
22:                     handlePropertyCallExpression(PropertyCallExpression,EC)
23:                 else if IsAny(collection.content) then
24:                     for all EClasses in EM do
25:                         handlePropertyCallExpression(PropertyCallExpression,EClass)
26:             else if IsAny(propertyCall.target) then
27:                 for all EClasses in EM do
28:                     handlePropertyCallExpression(PropertyCallExpression,EClass)
```

Algorithm 2 EOL Effective Metamodel Extraction Algorithm (2 of 2)

```
1:  procedure handlePropertyCallExpression(propertyCallExpression, EClass)
2:      if the property.name.equals(all) then
3:          Go to line 7 Algorithem 1
4:      else
5:          let EC = EClass
6:          if not allOfKind.contains(EC) and not allOfType.contains(EC) then
7:              types.add(EC);
8:          if IsAttribute(property) then
9:              EC.attributes.add(property)
10:         else if IsReference(property) then
11:             EC.references.add(property)
12:             let EType = reference.type
13:             types.add(EType)
```

Figure 7. Using this strategy, every model element is mapped to a *Node* in the graph and the properties of the node are set to the values of the attributes of the element. For example in Figure 7, *ACME* is an instance of *Project* in the model which has a *title* attribute; hence a corresponding node is created in the graph and the *title* property of the node is set to *ACME*. For *Design*, which is an instance of *ManualTask*, its *duration* and *start* attributes are copied to the respective node.

In Neo4j graphs, nodes can have labels. In our approach, the label of each node is set to the type of the respective model element and its super types. Thus, in Figure 7, the label of *ACME* is set to *Project* and the labels of *Design* are set

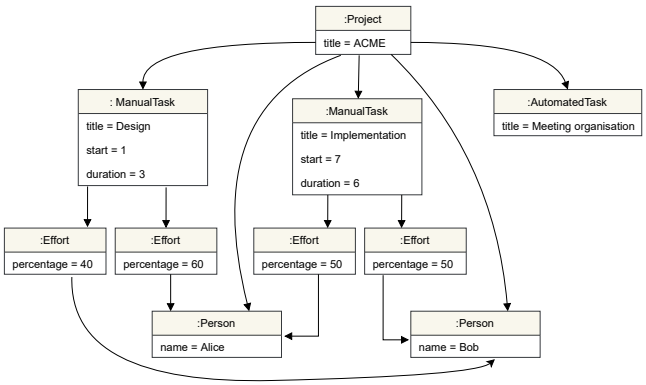

Figure 6. PSL model of the ACME project

to *Task* and *ManualTask*. As the labels of nodes are unordered, to understand which label corresponds to the exact type of the node, we can load all labels of each node and by using the structure of the metamodel, find the exact type of node. However, it is more efficient to connect each node to another node that has the name of its type, using an *instanceOf* edge to capture the type of the node. In Figure 7, there is an *Edge* in the graph which connects the created node to the *Project* node to capture the type of the node and an edge which connects the *Design* node to *ManualTask*. Algorithm 3 describes the mapping process in detail.

3.3.2 Model Loading. In the third step of our approach, we wish to generate queries in Neo4J's Cypher query language, that will fetch (1) only the part of the model that conforms to the effective metamodel extracted in step 1 and (2) do this as efficiently as possible (after statically analysing the program and identifying the part of the model it is likely to access at runtime in the form of an effective metamodel).

Cypher[8] is a language for querying Neo4j databases. Hence, by generating Cypher queries based on the effective metamodel, EOL's execution engine will be able to load the required parts of the model for running the program.

The Cypher expressions that are used for loading information in our approach are listed in Table 2 (*var* stands for variable). The first column of Table 2 lists the Cypher expressions, the second column is the pattern that each expression follows and the third column is the functionality of the expression.

Using these three expressions, the queries of Listing 3 are generated based on the effective metamodel in Figure 4.

Considering the effective metamodel in Figure 4, all instances of *ManualTask* are required for running the program (EOL code in Listing 1). Hence, all nodes with the *ManualTask* label should be loaded. The *MATCH* keyword matches all nodes with the specified label in the graph. The generated query in line 1 of Listing 3 loads all *ManualTask* nodes.

[8]https://neo4j.com/developer/cypher/

Algorithm 3 EMF Model to Graph Conversion Algorithm

```
1:  let visitedElements = keep the EMF model elements with corresponding nodes
2:  let source = Node, target = Node, newNode = Node
3:  for all model elements of EMF model do
4:      let ME = model element
5:      if visitedElements.contains(a node corresponding to ME) then
6:          source ←node
7:      else newNode = CreateNode(ME)
8:          setNodeProperties(newNode,ME.attributes)
9:          newNode.labels.add(ME.type.name)
10: for all supertypes of ME.type do
11:     newNode.labels.add(supertype.name)
12:     visitedElements.add(newNode)
13:     source ←newNode
14: for all ME.references do
15:     if visitedEl.contains(reference.value) then
16:         target ←visitedElements.get(reference.value)
17:     else newNode = CreateNode(reference.value)
18:         setNodeProperties(newNode,reference.value.attributes)
19:     visitedElements.add(newNode.labels)
20:     target ←newNode
21:     Edge e = CreateEdge(source, target)
22:     e.name = reference.name
```

Listing 3. Generated Cypher Queries According to Effective Metamodel in Figure 4

```
1 MATCH (task:ManualTask)
2 RETURN ID(task), task.title
3 OPTIONAL MATCH (task:ManualTask)-[taskins:
      instanceOf]->(taskType)
4 RETURN taskType.name
5 OPTIONAL MATCH (task:ManualTask)-[effortRefTask:
      effort]->(effort:Effort),(effort)-[effortins
      :instanceOf]->(effortType)
6 RETURN ID(effort), effortType.name
7 OPTIONAL MATCH (effort:Effort)-[personRefEffort:
      person]->(person:Person),(person)-[personins
      :instanceOf]->(personType)
8 RETURN ID(person), personType.name
```

After matching all *ManualTask* nodes, the *id* of each *task* has to be loaded in order to distinguish between different *ManualTask* instances. Line 2, shows the *Return* query to fetch the *id* and the *title* attribute of *ManualTask* nodes from the database.

Beyond the *ID*, the exact type of *ManualTask* instances is also needed, which is specified by the *"instanceOf"* edge. In the case of references, the *MATCH* and *OPTIONAL MATCH* expressions are used to match the edges of the source node and return the respective target nodes. Using *MATCH* is a

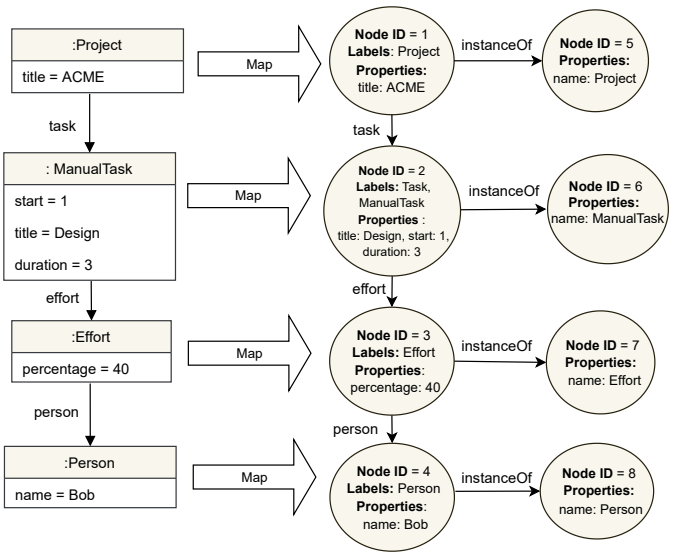

Figure 7. Mapping the model of Figure 6 into a Neo4J graph

Table 2. Cypher Expressions

Cypher expression	Pattern	Functionality
MATCH	(var: label of node)	loading all nodes with the same label
OPTIONAL MATCH	(var:source node)- [var:edge]-> (var:target node)	return the target node of matched edge
RETURN	node.property	return the specific property(ies) of the matched nodes

strict condition, and if there are no matches in the database, the query will fail to run and it does not return any result. With *OPTIONAL MATCH* on the other hand, if there is no match, the query will be run and "null" will be returned as a value of the edge which is not matched. Hence, *OPTIONAL MATCH* is a better fit for matching the references of each node. The query for requesting the type of *ManualTask* nodes, is shown in line 3 and the name property of *taskType* is returned using the *RETURN* expression in line 4.

One reference of *ManualTask* instances is required according to the effective metamodel. The *efforts* reference is an edge between *ManualTask* and *Effort* nodes and the query in line 5 matches this reference. In lines 6-7, the *instanceOf* reference is matched to find the type of the *Effort* node. According to the effective metamodel, attributes of *Effort* are not required, so only the *id*s and *type*s of *Effort* nodes are returned in line 6.

The *person* reference and the *instanceOf* reference of *Effort* are matched in line 7 and the query that returns the *id* and *type* of *Person* is shown in line 8.

3.3.3 Query Optimisation. The goal of our approach is to reduce the number of database hits and load as much information as possible in each access to the database to minimise the overall execution time. The Neo4j documentation[9] offers some recommendations to reduce the execution time of Cypher queries. We have applied some of them to optimise the automated query generation in our approach.

- **Using Labels**: Nodes are labeled by the type and super types of their corresponding model element. These labels are then used by generated queries to efficiently match nodes of different types.
- **Avoid Cartesian Products**: If two different node labels without any relationships between them are matched in a Cypher query, it is considered as a *disconnected pattern*. Generating a query for disconnected patterns will build Cartesian products between two node types, which would not be efficient as it will return a number of records which are not necessary for executing the program. For example, considering the PSL metamodel (Figure 1), there is no relationship between nodes with *AutomatedTask* and *Person* labels. So, in Listing 4, Neo4j matches each *AutomatedTask* node with all *Person* nodes. Suppose that the number of nodes with *AutomatedTask* label is equal to m and the number of nodes with *Person* label is equal to n. The number of returned records from database will be

[9]https://neo4j.com/blog/tuning-cypher-queries/

equal to *n*m*. However, if Listing 4 is separated into two *MATCH* clauses, then the number of records will be *(n+m)*, which is more efficient.

Listing 4. Query Generating Cartesian Products

```
1 MATCH (autoTask:AutomatedTask), (p: Person)
2 RETURN autoTask.title, p.name
```

Thus, we consider a trade-off between generating fewer queries and avoiding Cartesian products. In our approach, *MATCH* clauses are generated for *allOfKind* types in the effective metamodel. This is efficient as in each MATCH clause, the label of each node will be matched and then all required properties and references of a node in the effective metamodel will be returned. Considering the generated queries in Listing 3, there are three *MATCH* clauses that are related so they can be combined in one query for efficiency. The combined query is shown in Listing 5.

Listing 5. Optimised queries of Listing 3

```
1 MATCH (task:ManualTask)
2 OPTIONAL MATCH (task)-[taskins:instanceOf
    ]->(taskType),(task)-[effortRefTask:
    effort]->(effort:Effort),(effort)-[
    effortins:instanceOf]->(effortType),(
    effort)-[personRefEffort:person]->(
    person:Person), (person)-[personins:
    instanceOf]->(personType)
3 RETURN ID(task), task.title, taskType.name,
     ID(effort), effortType.name,ID(person)
    , personType.name
```

- **Reduce Cardinality**: Since nodes are labelled by the type of their respective model element and its super types, the results of some queries can overlap. For example, in Listing 6 all nodes with *Task* and *ManualTask* label are matched and the *title*s of matched nodes are returned. Considering the part of the graph in Figure 8, the nodes returned in line 2 are "Design", "Implementation" and "Meeting organisation" nodes since they are labeled as *Task*. The titles of the nodes that are returned in line 4, are "Design" and "Implementation" nodes which are *ManualTask*-labeled nodes. The "Design" and "Implementation" nodes are returned twice (in lines 2 and 4). This redundancy is because of querying two classes (*Task* and *ManualTask*) that have an inheritance relationship. Therefore, it is more efficient to execute only the first query in lines 1-2, which covers all nodes that are loaded by both MATCH clauses.

Listing 6. Queries with Overlap

```
1 MATCH (task:Task)
2 RETURN task.title
3 MATCH (manualTask:ManualTask)
4 RETURN manualTask.title
```

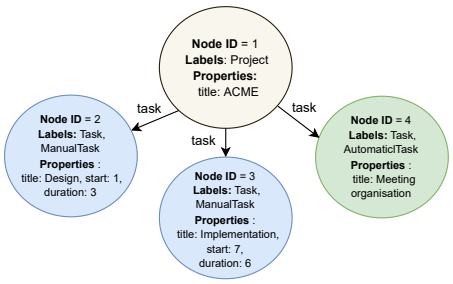

Figure 8. A part of graph model

Algorithm 4 generates Cypher queries automatically based on the effective metamodel.

Algorithm 4 Cypher Queries Generation (1 of 2)

```
1:  let EM = calculated effective metamodel
2:  for all EClass in EM.allOfKind do
3:      let matchString = "";
4:      let visitedClasses = Set of EClasses
5:      let optionalMatch = Set<String>
6:      let return = Set<String>
7:      matchString ←EClass.name
8:      return.add(EClass.id)
9:      return.add(EClass.getReference(instanceOf).name)
10:     Call HANDLEFEATURES (EClass)
11:     let query =""
12:     query ← "MATCH"+ query
13:     for all item in match array do
14:         query ←item + ","
15:     query ← "OPTIONALMATCH"+ query
16:     for all item in optionalMatch array do
17:         query ←item + ","
18:     query ← "RETURN"+ query
19:     for all item in return array do
20:         query ←query + item + ","
```

There are three collections to record MATCH clauses, OPTIONAL MATCH clauses and RETURN clauses in Algorithm 4. In lines 1-9 these three collections are filled and in lines 10-19, the query is generated by combining these clauses together using the appropriate Cypher expressions. One query for each EClass is generated where all instances of that class are required by the program.

In line 2, the classes which are under the *allOfKind* reference in the effective metamodel are considered by Algorithm 4. In line 7 the variable definition for the EClass is added to *matchString* and the ID of the matched node is added to the *return* collection (lines 7-8). In line 9, the *HANDLEFEATURES* procedure is called to handle the features of the EClass. In Algorithm 5, which shows the *HANDLEFEATURES* procedure, each attribute of the EClass is added to *return* collection (in lines 4-5). To follow the references of

Algorithm 5 Cypher queries generation (2 of 2)

```
1:  procedure HANDLEFEATURES(EClass)
2:      visitedClasses.add(EClass)
3:      for all features in EClass.eStructuralFeatures do
4:          if IsAttribute(feature) then
5:              return.add(feature)
6:          else if IsReference(feature) then
7:              let type = reference.type
8:              return.add(reference.target.id)
9:              optionalMatchString ← (EClass.name) −
    [reference.name] → (type)
10:             if not visitedClasses.contains(EClass) and not
    allOfKind.contains(EClass) then
11:                 handleFeatures(type)
```

nodes in the graph, an OPTIONAL MATCH pattern is needed. So, in lines 6-9, an OPTIONAL MATCH pattern is created and added to the *optionalMatch* collection. The OPTIONAL MATCH pattern matches edges that have the same name as the reference of the EClass and it connects the matched node to the target nodes (it is kept in the *type* variable).

As discussed in Section 3.3.3, it is more efficient to generate one query and load all relevant information in one go. So, in lines 10-11 the *HANDLEFEATURES* method is called on *type* to follow the features of the target node. Thus, the Algorithm 5 uses this recursive method to load a sub-graph which consists of the nodes and their properties and edges.

This recursive call returns when a cycle is found or when there are no further references to follow. In line 2 of Algorithm 5, the *visitedClasses* is a collection to keep track of visited classes. So if a *type* is visited once, it will not be visited again according to the *if* condition in line 10 to avoid infinite loops.

After handling all features in Algorithm 5, the query is generated for the EClass and then the process is repeated for the next EClass under the effective metamodel's *allOfKind* reference. The result returned from the database after executing the generated query is shown in Table 3.

3.3.4 Program Execution. After retrieving information from the database, this information is used by the execution engine to run the EOL program. In this approach, we use the EMF driver of Epsilon to run the EOL program to avoid implementing a new driver with almost the same functionality. Therefore, in the last step of our approach, in the execution engine, the results of queries are wrapped as an EMF input and the program is run by the Epsilon EMF driver.

3.4 Limitations

There are two noteworthy limitations in our proposed approach. First, our approach is limited to read-only input models of model management programs. Changing models (updating, deleting or adding model element) is not supported in our approach. Also, our prototype implementation does not attempt dead code elimination, which means that the extracted effective metamodel can contain types and features that may never be accessed at runtime.

4 Related Work

To achieve partial loading, some of the related work proposes database-backed persistence technologies. The most mature ones are Morsa [9], CDO [11] and Neo4EMF [1].

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, and supports partial loading of large models using a load on-demand mechanism. More specifically, it uses The *single load on-demand* which can be considered as a lazy loading and *partial load on demand* which follows the greedy loading mechanism.

The Connected Data Object (CDO) is a model repository for EMF models. Metamodels and models can be stored in all kinds of database backends, including major relational databases and NoSQL databases.

Scalability in CDO is achieved by object loading based on on-demand strategies and caching the objects in the application. Hence, it does not keep the objects which are no longer referenced by the application, and they are collected from memory automatically.

Neo4EMF is a persistence layer for EMF models. It is built on top of the graph-based database Neo4j, as graph-based databases are able to manage large-scale data in highly distributed environments. Neo4EMF is similar to Morsa in several aspects (notably in on-demand loading), but it aims at exploiting the optimised navigation performance offered by graph databases. Neo4EMF is a more preferred alternative to XMI and CDO; due to high-performance access and on-demand loading, its raw performance does not surpass a more mature solution like CDO [4].

SmartSAX is another prototype which was introduced by Wei et al. [13]. It supports partial loading of XMI model files. SmartSAX currently does not support loading models that are persisted in multiple XMI files. Also, it does not support garbage collection to unload parts of a model from memory when they are no longer needed.

In [3], Daniel et al. propose PrefetchML, a domain-specific language that describes prefetching and caching rules over models. PrefetchML is a suitable solution to improve query execution time on top of scalable model persistence frameworks. The rules to describe the event conditions to activate prefetching, the objects to prefetch, and the customisation of the cache policy are defined by designers in PrefetchML. PrefetchML can take advantage of our work to use the static analyser to define the rules instead of depending on the designers.

Table 3. Result for the Execution of Generated Query in Listing 5

ID(task)	task.title	taskType.name	ID(effort)	effortType.name	ID(person)	personType.name
2	"Design"	"ManualTask"	10	"Effort"	9	"Person"
2	"Design"	"ManualTask"	12	"Effort"	7	"Person"
6	"Implementation"	"ManualTask"	13	"Effort"	7	"Person"
6	"Implementation"	"ManualTask"	14	"Effort"	9	"Person"

The idea of using static analysis to equip execution engines and improve the performance of model management programs is discussed in our preliminary work [5]. However, in this work, we present only an abstract overview of the idea. There is no evaluation and no results are presented to show the efficiency of the approach.

Although recent research has made advancements in this area, existing solutions have apparent shortcomings in accessing and processing large models.

To the best of our knowledge, none of the model repositories we are aware of, perform query analysis to enable efficient partial model loading. Repositories such as Morsa, CDO provide remote access to large models and store them in a document-based or graph-based database, but as discussed in Section 1, in the absence of static analysis, greedy and lazy strategies are not efficient. Using the greedy approach, all properties and references of the model element are loaded which is not efficient in terms of memory consumption, while in the lazy approach, multiple trips to the database render the approach very time-consuming.

We should note here that we initially investigated implementing our approach on top of the CDO and NeoEMF repositories. However, CDO is designed and implemented in a way that provides facilities to access model element properties greedily (requesting all properties) and lazily (asking for one property in each access to the database). Similarly, NeoEMF only supports lazy loading. Therefore, none supports requesting specific properties from the database based on the footprint (effective metamodel) of the program which is the base of our approach. We hope that our demonstration of the efficiency benefits of partial model loading (see Section 5), can motivate the developers of repositories such as NeoEMF and CDO to add support for such a feature in the future.

5 Evaluation

In this section, we report on the results of experiments that measure the performance of our approach against that of NeoEMF. We have chosen NeoEMF because it outperforms other repositories [4]. NeoEMF follows the lazy loading strategy.

We evaluated our approach on a system using Java VM 14.0.1 with Intel(R) Core(TM) i7, 16 GB memory and CPU @ 2.80 GHz running Mac OS X Catalina.

For our experiments, we have used the models proposed in the GraBaTs 2009 contest [10]. The models conform to an Ecore-based metamodel of the Java programming language and have been reverse-engineered from open-source Java projects. There are five XMI models, from Set0 to Set4, each one larger than its predecessor (from a 8.8 MB XMI file with 70 447 model elements representing 14 Java classes to a 646 MB file with 4 961 779 model elements representing 5984 Java classes). We produced Neo4j graphs from the Grabats XMI files and saved them in the Neo4j (version 4.4.3) embedded databases.

We have also implemented a query in the Epsilon Object Language (EOL) inspired by one of the Grabats test cases[10]. This query finds all classes that declare public static methods whose return type is the containing class itself (i. e. like the *getInstance* method of the *Singleton* pattern).

For our experiments, we ran the EOL program against Set0-Set4 graphs in the database using our approach and NeoEMF and measured the execution time and memory consumption. The results are shown in Table 4. The results were computed after 5 warm-up iterations and represent the average over 10 executions of the program. For instance, for Set1, it takes 8.3 s and 118 MB of memory to run the Grabats query using our approach while for NeoEMF, the average time is equal to 9.5 s and the memory consumption is 656.2 MB which is about 5.5 times higher than our approach. The most significant difference in memory usage is in Set3, where the memory footprint of our approach is 93 % lower than NeoEMF.

The charts illustrated in Figures 9 and 10 show the linear behaviour of our approach and NeoEMF. In Figure 9, in the Set0 model, as the model is not very large, the overhead of effective metamodel extraction and loading data from database in our approach is not compensated at runtime and the execution time is better for NeoEMF. As the size of the model grows, the slope of the diagram is greater in NeoEMF compared to our approach, which means our approach is more efficient in terms of execution time. The highest percentage of time saving is 74 % for Set2.

In Figure 10, both approaches have a linear behaviour, and our approach consumed less memory compared to NeoEMF.

[10] http://https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=GraBaTs_2009_Case_Study

Table 4. Experiment Results

Model	Our Approach		NeoEMF	
	Execution Time (s)	Execution Memory (MB)	Execution Time (s)	Execution Memory (MB)
Set0 (70 447 model elements)	8.1	108	5.7	319
Set1 (198 466 model elements)	8.3	118	9.5	656.2
Set2 (2 082 841 model elements)	12.2	210	61.5	2537.5
Set3 (4 852 855 model elements)	37.3	232.8	114.6	3718
Set4 (4 961 779 model elements)	41.5	318.1	126.4	2987.2

From Set0 to Set4, as the size of model increased, the memory consumption grew in two approaches.

On average, using our approach, memory consumption is lower by 84 % and execution time is lower by 37 % compared to NeoEMF.

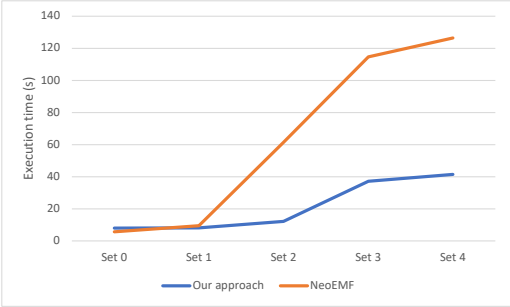

Figure 9. Execution time comparison of our approach and NeoEMF

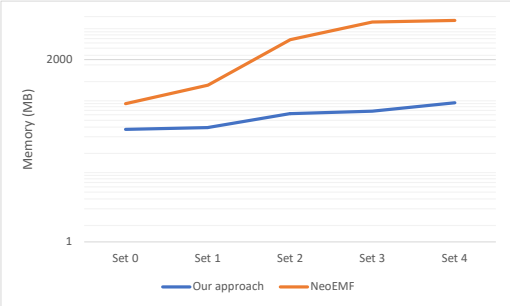

Figure 10. Memory comparison of our approach and NeoEMF (logarithmic scale)

Regarding correctness, we validated all models by executing each program with two approaches (our approach and NeoEMF) and verify that the output produced by all execution pairs should be equivalent (e. g., in the number of people for each task in EOL and NeoEMF).

5.1 Threats to Validity

We limit construct validity threats by considering big models from the widely used GraBaTs 2009 contest [10] that conform to the Grabats metamodel. The results reported in this paper consider these test cases.

We limit internal validity threats by reporting results after executing 5 warm-up iterations of the program, thus reducing the potential impact in memory and execution time of starting and initialising the JVM.

We reduce external validity threats by building our approach atop mature and robust MDE technologies, including the Epsilon suite of model management programs and the Neo4j graph database. As discussed in Section 3, extending our approach to support other technologies is relatively straightforward with modest effort. However, more experiments are required to establish the applicability and scalability of our approach in domains and metamodels/models with characteristics different than those used in our experimental evaluation.

6 Conclusion

We introduced an approach for partial loading of repository-based models based on information extracted through static analysis of model management programs. We evaluated our approach against large models from the Grabats test suite. The results demonstrate that program-aware partial loading can significantly reduce the time and memory required to run model management programs against repository-based models when only a subset of the model's elements is accessed by the program, without otherwise affecting the behaviour or the output of the program.

As future work, we plan to investigate how the results of static analysis can also improve memory footprint when the execution engine unloads obsolete parts of the model in memory (i. e., parts that have already been processed and are guaranteed not to be accessed again) instead of keeping them for the duration of the execution of the program. In this way, resources will be freed, thus enabling management programs to accommodate even larger models.

Acknowledgments

This research is supported by the Lowcomote Training Network, which has received funding from the European Union's Horizon 2020 Research and Innovation Program under the Marie Skłodowska-Curie grant agreement no 813884.

References

[1] 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.

[2] Jordi Cabot and Martin Gogolla. 2012. *Object Constraint Language (OCL): A Definitive Guide*. Springer Berlin Heidelberg, Berlin, Heidelberg, 58–90. https://doi.org/10.1007/978-3-642-30982-3_3

[3] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2016. Prefetchml: a framework for prefetching and caching models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 318–328.

[4] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. 2017. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming* 149 (2017), 9–14.

[5] Sorour Jahanbin, Dimitris Kolovos, and Simos Gerasimou. 2020. Intelligent run-time partitioning of low-code system models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–5.

[6] Frédéric Jouault and Ivan Kurtev. 2005. Transforming models with ATL. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 128–138.

[7] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. The epsilon object language (EOL). In *European conference on model driven architecture-foundations and applications*. Springer, 128–142.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.