Proceedings Paper:
Jahanbin, Sorour, Kolovos, Dimitris orcid.org/0000-0002-1724-6563 and Gerasimou, Simos orcid.org/0000-0002-2706-5272 (2023) Towards Memory-Efficient Validation of Large XMI Models. In: Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2023, 01-06 Oct 2023 Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. Institute of Electrical and Electronics Engineers Inc., SWE, pp. 241-250.

https://doi.org/10.1109/MODELS-C59198.2023.00053

Towards Memory-Efficient Validation of Large XMI Models

1st Sorour Jahanbin
Dept. Computer Science
University of York
York, United Kingdom
sorour.jahanbin@york.ac.uk

2nd Dimitris Kolovos
Dept. Computer Science
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

3rd Simos Gerasimou
Dept. Computer Science
University of York
York, United Kingdom
simos.gerasimou@york.ac.uk

***Abstract*—Model validation is a common activity in model-driven engineering, where a model is checked against a set of consistency rules (also referred to as constraints) to assess whether it has desirable properties further to those that can be expressed by the metamodel that it conforms to (e.g. to check that all states in a state machine are reachable or that no classes in an object-oriented model are involved in circular inheritance relationships). Such constraints can be written in general-purpose (e.g. Java) or in task-specific validation languages such as the Object Constraint Language (OCL) or the Epsilon Validation Language (EVL). To check a model that is serialised in the OMG-standard XMI format against a set of constraints, the current state of practice requires loading the entire model into memory first. This can be problematic in cases where loading the model into memory requires more memory (heap space) than is available in the host machine, and is sub-optimal when carrying out distributed model validation over a number of machines. In this paper, we present an approach that uses static analysis to split sets of model validation constraints into sub-groups that operate on smaller subsets of the model. Combined with existing XMI partial loading capabilities, the proposed approach makes it possible to check larger XMI-based models on a single machine and to potentially improve efficiency when checking models in a distributed setting.**

***Index Terms*—model validation, partial loading, memory management, file-based model, XMI model, model-driven engineering**

I. Introduction

Model-driven engineering relies on models as the primary artefact, which are manipulated by model management programs performing various tasks, including model validation. However, when dealing with large file-based models such as those serialized in XMI format, scalability issues arise in the validation of such models.

XMI (XML Metadata Interchange)[1] is an Object Management Group standard for model persistence and it is a native format for model persistence in the Eclipse Modeling Framework (EMF). EMF's default XMI parser loads the entire model into memory and occupies memory with the parts of model that are not referenced by the program. This inefficient interaction results in wasted memory and can cause execution to fail when the model exceeds the available heap memory, preventing the execution engine from loading the model and running the program on a single machine.

[1]https://www.omg.org/spec/XMI/

Similar issues arise when attempting to execute constraints in parallel. In this scenario, all constraints are distributed across multiple machines, and the entire model is loaded on each machine, regardless of which constraints they are assigned to execute. Consequently, the loading and validation of models in the execution engine lack efficiency, leading to increased model loading time and unnecessary memory consumption in each machine.

In this paper, an approach is presented for improving the performance of execution engines when dealing with the validation of large models. By leveraging static analysis and advance knowledge about the program, the proposed approach partitions the constraints into groups. The execution engine employs a partial XMI parser, allowing it to selectively load the relevant parts of the model that are expected to be accessed by each group. Additionally, the execution engine can discard model sections that are no longer referenced by the program, thereby freeing up memory for the remaining execution.

Applying the proposed approach, execution engines of validation programs can handle large models more efficiently. The use of static analysis allows for better resource allocation, thus reducing unnecessary memory consumption for loading and validating models.

The main contributions of this work are:

- An algorithm for grouping constraints using static analysis.
- An efficient execution plan for validating constraints on XMI models.
- A prototype implementation of the algorithm using the Eclipse Epsilon family of model management languages and an existing partial XMI parser.

The remainder of the paper is structured as follows. Section II provides an explanation of the challenges of interest through a motivating example. In Section III, the proposed approach is presented and discussed in detail. Also, the limitations of our approach are acknowledged. Related work is reviewed in Section V, providing an overview of existing research and approaches that have addressed similar or related problems. Section VI reports on the results of the evaluation of our approach, and finally, Section VIII summarises the contributions of our work and outlines directions for future work.

II. Motivating Example

Figure 1 shows the metamodel of a simplified component-connector language, which is used as a motivating example in this paper. The metamodel has a *Component* class which has a *name* and consists of *ports*, which is shown by a containment reference from *Component* to the *Port* class. Each *Port* has a *name* and a *type*. Also, it has a reference to its *Component*. There are two types of *Ports*, *InPort* and *OutPort* that are connected to each other by a *Connector*. The *source* of a *Connector* is always an *OutPort* of a *Component*, and the *target* of *Connector* is an *InPort*.

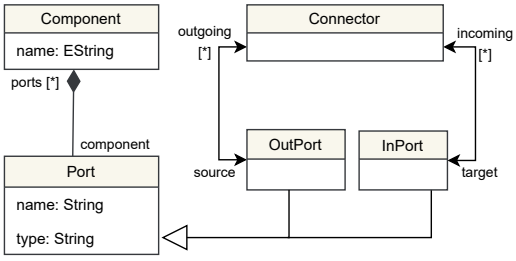

Fig. 1. Component Language Metamodel

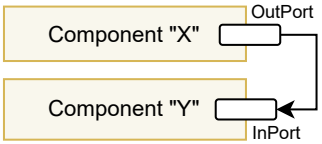

Fig. 2. Sample model that conforms to the metamodel of Figure 1

Consider a model that conforms to the Component language (a Component diagram is shown in Figure 2), which we would like to validate with additional constraints. These constraints could be written in general-purpose (e.g. Java) or in task-specific validation languages such as the Object Constraint Language (OCL) or the Epsilon Validation Language (EVL) [1]. For this work, we have selected Epsilon's EVL; however, the proposed approach also applies to other types of validation languages. Listing 1 contains the additional constraints expressed in EVL.

Listing 1. EVL constraints to validate Component Language instances

```
context Component {
  constraint hasValidName {
    check: self.name = self.name.ftuc()
    message: self.name + " should start with an
        upper-case letter"
  }

  constraint hasUniqueName {
    check: Component.all.select
      (c:Component|c.name = self.name).size() == 1
    message: "Duplicate component name" + self.name
  }
}

context Connector {
  constraint portTypesMatch {
    check:self.source.type=self.target.type
    message:"The types of the source and target
        ports don't match"
  }
}
```

The first constraint (*hasValidName*) checks that the *name* of each *Component* should start with an upper case letter (using EVL's *ftuc()*[2] method), while the second one (*hasUniqueName*) makes sure that *Component* names are globally unique. The third constraint (*portTypesMatch*) checks that the *type*s of two *Port*s connected by a *Connector* are the same.

According to Listing 1, the information that is required for executing the *hasValidName* constraint is the *name* attribute of all instances of *Component* in the model. The *hasUniqueName* constraint also needs the same information. For running the *PortTypeMatch* constraint, the *source* and *target* references of all instances of *Connector* are of interest and the *type* attribute of *Port*, too. It is worth noting that some parts of the model, such as the *name* attribute or the *component* reference of *Port*, are not accessed by any of these constraints.

To run these constraints, if the execution engine uses EMF's default XMI parser, the whole model will be loaded into memory. Hence, some memory will be occupied by model elements or features not required for executing the program (such as the *name* attribute or the *component* reference of *Ports*). Also, all loaded elements will be kept in memory until the end of execution, which is not optimal. For example, after running the second constraint (*hasUniqueName*), instances of the *Component* class are no longer needed by the program; there is no need to keep these model elements and their features in memory while executing the third constraint.

To improve efficiency, it is possible to identify groups of constraints that access the same sub-sets of the model. For instance, the *hasValidName* and *hasUniqueName* constraints in the example require the same information (the *name*s of all *Component*s in the model). In our approach, by recognising this, the execution engine can execute these constraints together without the need for additional loading and unloading overhead. The necessary information, such as the *name* attribute of *Component*, can be loaded into memory, the constraints can be executed, and the information can be unloaded once it is no longer needed for the *PortTypeMatch* constraint. By selectively loading and unloading the relevant information, the execution engine can reduce the loading time and overall execution time.

The current execution engine of EVL programs is not aware of features of required model elements that have to be fetched from the XMI model and loaded into memory. Also, there is no hint for the execution engine to dispose of the parts of models from memory that are no longer required by the program. Hence, a static analyser becomes essential to analyse the EVL program in order to obtain advanced knowledge about the program. This knowledge allows the execution engine to selectively load relevant information and group the constraints based on this acquired understanding.

III. Approach

The memory management and loading time when working with large XMI models can be improved through the

[2]first to upper case

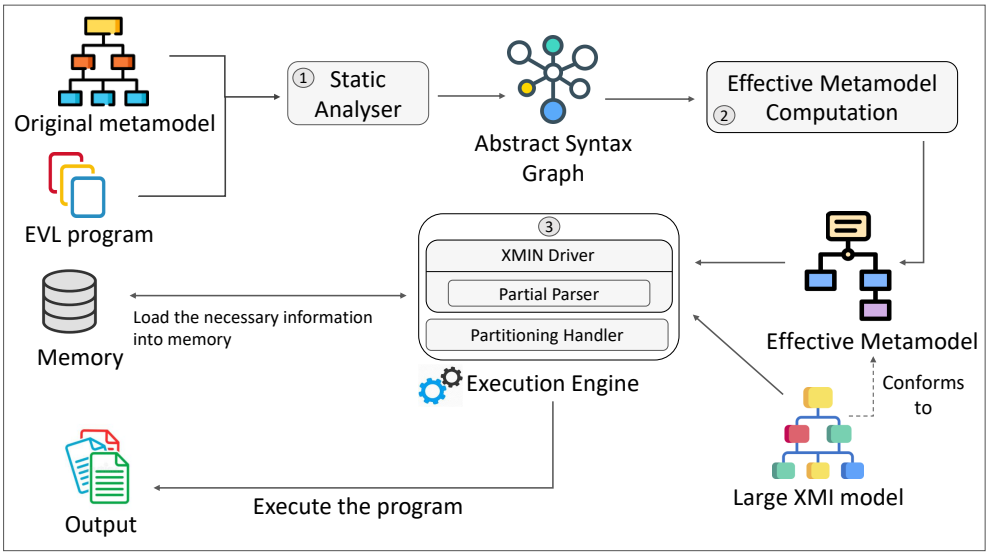

Fig. 3. The proposed approach

implementation of two key features in the EVL execution engine: partial loading and partitioning. In our approach, we combine these capabilities to enable the execution engine to load only specific subsets of the model that are required for executing the program, resulting in a reduced memory footprint. Additionally, splitting constraints into sub-groups (which is referred as partitioning the constraints in this paper) allows for selective loading and unloading subsets of the model based on the constraints being executed, further improving the overall efficiency of the program. Together, these features can facilitate more efficient memory utilization and improved performance in running the program.

In terms of concrete technologies, we use XMI as a file-based model format and programs written in the EVL language of the Epsilon platform, but the approach can be extended to other languages of Epsilon, and it is also applicable to other similar technologies (e.g. OCL). A high-level overview of our approach is presented in Figure 3. The main components of this approach are represented in grey and labelled with numbers 1 to 3.

A. Static Analyser

In our work, we use a static analyser for Epsilon programs[3] that applies variable and type resolution to derive an abstract syntax graph from the abstract syntax trees of EVL constraints. The inputs of the static analyser are an EVL program (set of constraints) and the metamodels of the models it consumes (first step of our approach in Figure 3).

In-advance knowledge about the program (i.e., types and properties accessed by the program) is extracted by the static analyser using the abstract syntax graph. The extracted information is in the form of an *effective metamodel* for every model the constraints access. The effective metamodel is a subset of the model's original metamodel, consisting only of types and properties likely to be accessed by the program [2] (see Section III-B).

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

In the first step of our approach, the static analyser sets the resolved types of expressions to types from the respective metamodels, to primitive types (e.g., String, Integer) or to collection types.

For example, in line 3 of Listing 1, the resolved type of *self* variable is equal to *Component* as the context of constraint is *Component* (Line 1). In line 3, the property call checks the *name* of each *Component*. The *name* is an attribute of *Component*, and the resolved type is String. The type of the *self.name* in line 3 (as a target of *ftuc()* operation call) and in line 4 (as a property call) is resolved as String (as same as property call in line 3).

In the following constraint (*hasUniqueName*), the target of the *select* operation call is a property call which retrieves the *name* of all *Component*s. In line 8, variable *c* is resolved as model element (*Component*) as the target of the *select* operation is all instances of *Component*. The type of *self.name* is resolved as String, as same as in the *hasValidName* constraint.

In line 14, the types of *self.source.type* and *self.target.type* are resolved as String. The type of the *self* variable is resolved to *Connector*, and the *source* and *target* are the references of *Connector*s which are resolved as model elements (source is resolved to *OutPort* and the target is *InPort* type) (see the metamodel in Figure 1). Table I shows the resolved types of expressions extracted from Listing 1 by the static analyser.

[3]https://github.com/epsilonlabs/static-analysis

TABLE I
RESOLVED TYPES CALCULATED BY THE STATIC ANALYSER

Line number in Listing 1	Expression	Resolved Type
1	Component	Model Element (Component)
3, 4, 9	self.name	Property call expression (String)
3, 4, 9	self	Model Element (Component)
3	self.name.ftuc()	Operation call expression (String)
8	Component.all.select(c | c.name = self.name).size()	Operation call expression (Integer)
8	Component.all.select(c | c.name = self.name)	Operation call (Collection<Component>)
8	c.name	Property call expression (String)
8	c	Model Element (Component)

B. Effective Metamodel Computation

The extraction of the effective metamodel of constraints is the second step of the approach. The concept of the effective metamodel is introduced by Wei et al. [3], and it is constructed using the algorithm discussed in [4]. This algorithm uses the resolved types of expressions from an abstract syntax graph (that is created by the static analyser). It contains only types which are necessary for executing the program from which it is extracted. In our implementation, effective metamodels are only computed for EMF-based models, but in principle, this approach can also be applied to other metamodelling technologies.

As shown in Figure 4, an effective metamodel consists of an *EffectiveMetamodel* class with *name* and namespace URI (*nsuri*, the unique ID of the metamodel in terms of EMF terminology) 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 to specify class instances that should be loaded only when they appear in the containment references of model elements.

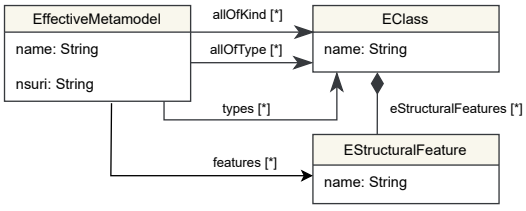

Fig. 4. The structure of effective metamodel (adapted from [3])

For every class used in the program (such as *Component* and *Connector*), 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.

Figure 5 illustrates the effective metamodel extracted from the EVL program in our motivating example (Listing 1).

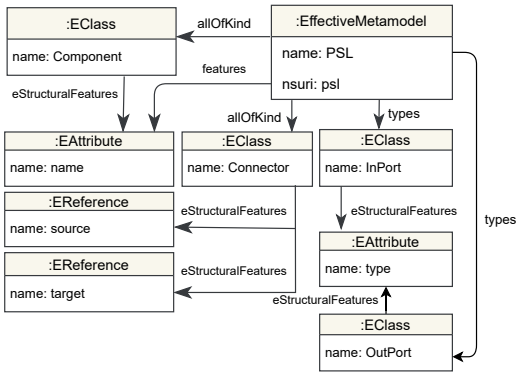

Fig. 5. The effective metamodel extracted from Listing 1

In Figure 5, 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 EVL program in Listing 1, all instances of *Component* and *Connector* must be loaded. The *Component* and *Connector* classes are added to *EffectiveMetamodel* under the *allOfKind* references. The *name* attribute of *Component* is added to the *EffectiveMetamodel* as well. The *source* and *target* references of *Connector* are also required (line 15 of Listing 1), hence, they are added to *Connector* as *EReference*s using *EStructuralfeatures* references.

The resolved type of *source* reference is equal to *OutPort* (see Table I), so the *OutPort EClass* is added to effective metamodel using the *types* reference which is the same for the *target* reference that is resolved to *InPort EClass*. The last attribute is *type* attribute which is added to *InPort* and *OutPort EClasses*.

C. XMIN Driver

In Epsilon's architecture, there is the Epsilon Connectivity Layer (EMC)[4]) which enables Epsilon programs (including EVL constraints) to interact with models in different modelling technologies in a uniform manner by defining drivers (e.g., EMF, CDO, NeoEMF).

Our frugal file-based model loading approach has been implemented in the form of a new Epsilon driver called XMIN, which is an extension of the existing EMF driver and provides facilities for Epsilon programs to load XMI-based models partially.

1) Partial Parser: As EMF's default XMI parser loads the entire model into memory, for partial loading, a custom XMI

[4]https://www.eclipse.org/epsilon/doc/emc/

parser is needed to load models based on the information provided by an effective metamodel. For this purpose, we use the parser described in [3] after fixing some bugs and making further improvements to its performance. Therefore, in step 3 of Figure 3, the partial parser will load the model based on the effective metamodel.

This parser scans the whole XMI file and checks the tags of elements with the effective metamodel classes and features. If the element's tag is represented in the effective metamodel, then the element is loaded; otherwise, the parser ignores the element, pushes a placeholder in the parser stack and moves on to the next element in the file.

Using a partial parser in the XMIN driver saves time in the loading process and saves memory [3] because less information is loaded compared to the default XMI parser.

Considering the example in Listing 1, in step 3 of our approach, the partial parser loads the XMI models based on the extracted effective metamodel (Figure 5). All instances of *Component*s with their *name* attribute, all instances of *Connector* with their *source* and *target* references, and *InPort* and *OutPort* instances with their *type* attribute are loaded. Other information, like the names of ports, is skipped by the parser.

However, by only using the partial parser, the necessary information for executing all constraints is kept in memory until the end of execution. The execution engine performance can be improved by keeping this information in memory only while it is required and then disposing it. This feature can be supported by partitioning the constraints, loading the part of the model required for executing the group and disposing of it after execution is finished.

D. Partitioning Handler

Loading all necessary information from the model upfront and running the program would be efficient in terms of loading time as it would avoid the cost of re-parsing a model but it would keep elements and their property values in memory for longer than needed, which is not efficient in terms of memory footprint. Hence, if the machine's memory is insufficient to accommodate all the necessary model elements, the execution engine will be unable to load the model, resulting in a failure to execute the program. Partitioning the constraints and loading information for each constraint separately can be a solution for this issue. Although loading and unloading information for each constraint is not considered optimal (discussed below), it serves as an intermediate step to introduce our approach.

In this solution, instead of extracting an effective metamodel for the whole program and loading information based on that in one go, an effective metamodel is extracted for each constraint separately. After executing the constraint, all model elements will be unloaded, and the memory becomes available to run the next constraint.

In Listing 1, there are three constraints to validate the model. Figure 6 shows three effective metamodels that are extracted for each constraint. An overlap outlines how a specific task or program will be executed by the system. Figure 7 shows an overview of the overlap for running an EVL program in the intermediate solution.

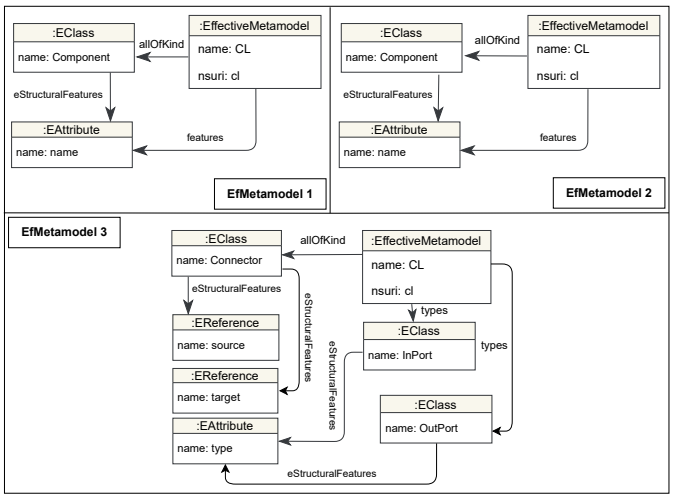

Fig. 6. Effective metamodels of constraints in Listing 1

To run the EVL constraints in listing 1, using the intermediate solution, the EVL execution engine loads the model based on *hasValidName*'s effective metamodel (EfMetamodel 1 in Figure 6) and runs the constraint using loaded model elements and properties. In the next step, the cache is cleared, and all information is unloaded from the memory. The same process happens for the rest of the program until the execution is finished.

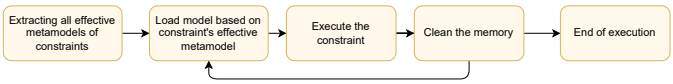

Fig. 7. Overview of an overlap

1) Constraint Grouping: Partitioning the program based on constraints is efficient regarding memory footprint as model elements are not kept in the memory for the entire execution. However, it is more time-consuming as the model is loaded multiple times.

Figure 6 shows that the effective metamodels of *hasValidName* and *hasUniqueName* are identical (EfMetamodel 1 and EfMetamodel 2, and these two constraints require the same data. Hence, disposing of the information from memory after running *hasValidName* constraint and loading the same information again to execute *hasUniqueName* is sub-optimal.

By grouping the *hasValidName* and *hasUniqueName* constraints and associating their effective metamodel with the group, the execution engine can identify that these constraints can be executed using the same data. As a result, the execution engine optimizes the process by avoiding the unnecessary unloading of information after loading model elements for the *hasValidName* constraint. Instead, it loads the required information based on the effective metamodel, executes both constraints and unloads the information.

Therefore, adopting a strategy of grouping constraints based on their effective metamodel, specifically when they require the same information or a subset of information for execution,

presents a promising approach to minimize loading time more than the intermediate solution. By identifying these relationships and grouping the constraints accordingly, the execution engine can optimize the loading process by reusing the already loaded information, thereby decreasing the overall loading time. In our approach, the partitioning handler is a component in the execution engine which is responsible for grouping the constraints.

Algorithm 1 describes an algorithm to group the constraints. The strategy of this algorithm is to make a group of constraints with the same effective metamodel, or the constraints that their effective metamodels are a subset of each other.

In line 3 of Algorithm 1, the algorithm goes through all constraints and computes their effective metamodels. In line 6, the algorithm searches in constraints to evaluate if there are other constraints, the effective metamodel of which is a subset or a super-set of this constraint. If it is found, the second constraint will be added to the same group as the first one, and the super-set effective metamodel will be mapped to the group (lines 7 to 9).

For example, if a constraint accesses the *target* reference of *Connector* class, then the effective metamodel is a subset of EfMetamodel 3 (as EfMetamodel 3 includes the *target* reference of *Connector* class to execute the constraint), and it would be efficient to put them in the same group.

Another possible case for grouping constraints is when the constraints' effective metamodels are not the same or superset/subset of each other but there is a significant overlap between them (the elements that are required for running the constraints). In this situation, grouping constraints becomes less straightforward.

On the one hand, grouping constraints saves time in the loading/unloading process, but on the other hand, loading more information can increase the memory footprint of the loaded model. For example, in Figure 8, there are two constraints that both require all instances of *InPort* class with the *name* attribute (they are shown by green colour). Constraint 1 needs the *incoming* reference of *InPort* class, while Constraint 2 requires the *type* attribute to be executed. In this scenario, the partitioning handler can take two different strategies:

- Loading model elements for each constraint and executing the two constraints separately. The execution engine loads the *name* attribute and *incoming* reference of *InPort* class, executes the Constraint 1 and unloads the model elements. Then, it loads the *name* and *type* attributes of *InPort* class again and executes Constraint 2.
- Grouping the two constraints, merging their effective metamodels and executing the constraints in sequence (or in parallel). The execution engine loads the *name* and *type* attributes and the *incoming* reference of *InPort* class and executes both constraints.

Therefore, there is a trade-off between loading more information in one go, having less memory available for execution and saving time, and loading elements separately and spending more time on loading but making more space available in memory for running the constraint. For example, in Figure 8,

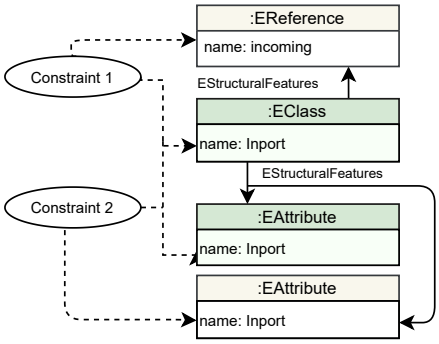

Fig. 8. Constraints with overlap

for executing Constraint 1, if the number of *InPort* instances is large enough to occupy most of the memory, then it is highly possible that loading an additional attribute (*type* attribute) would not be efficient (execution needs the memory). If the number of elements occupies less memory, then loading more data and running more constraints would be recommended.

The partitioning algorithm needs more information about the model to decide on partitioning and automatically evaluate which strategy is the optimal option. This information is not available in our approach, as the static analyser gains in-advance knowledge about the program. It works on the metamodel abstraction level and has no information about the model and its content. Therefore, the lack of this knowledge is to be compensated by the user.

To address this limitation, we empower users to group the constraints themselves using annotations in the EVL program. By leveraging the expertise and insights of the user, the constraints can be grouped in a way that takes into account the specific characteristics and dependencies of the model. This enables users to make informed decisions about constraint grouping, leveraging their domain knowledge and understanding of the model structure.

As shown in Listing 2, we consider a new annotation (*@group*) in EVL language. Using this annotation, users can group constraints manually. In Listing 2, the *@group* annotation is followed by an id. Constraints with the same group id are in the same group. Therefore, constraint *hasValidName* and *hasUniqueName* are grouped as they have the same id, and *portTypeMatch* is executed separately.

Listing 2. Groupig EVL constraints to validate Component Language instances

```
1  context Component {
2
3    @group gp1
4    constraint hasValidName {
5      check: self.name = self.name.ftuc()
6      message: self.name + " should start with
7               an upper-case letter"
8    }
9
10   @group gp1
11   constraint hasUniqueName {
12     check: Component.all.select
13       (c:Component|c.name = self.name).size() == 1
14     message: "Duplicate component name" + self.name
15   }
16 }
17
```

Algorithm 1 Partitioning Algorithm

```
let Map<Constraint, EffectiveMetamodel> constraintSets
let Map<Set<Constraint>, EffectiveMetamodel> constraintgroups
for all constraint1 in constraintSets do
  let gpEfModel = getEffectiveMetamodel(constraint)
  let group = New Set<Constraint>
  for all constraint2 in constraintSets do
    if isSubSet(gpEfModel, getEffectiveMetamodel(constraint2)) then
      group.add(constraint2)
      geEfModel = MergeMetamodel(gpEfModel, getEffectiveMetamodel(constraint2))
    end if
    constraintgroups.add(group, geEfModel)
  end for
end for
```

```
context Connector {

  @group gp2
  constraint PortTypesMatch {
    check:self.source.type=self.target.type
    message:"The types of the source and target
        ports don't match"
  }
}
```

By allowing users to manually group constraints, we provide a flexible and customisable approach that empowers users to optimize the performance of the constraint execution process based on their knowledge and understanding of the model.

IV. Limitations

Our proposed approach has two notable limitations that should be highlighted. Firstly, it is designed specifically for read-only input models in model management programs. This means that our approach does not support any modifications to the models such as updates, deletions, or additions of model elements.

Secondly, from a technological perspective, this approach has been implemented and evaluated only for EMF-based models. Extending it to support other model types is part of our future work.

V. Related Work

In [5], an approach is proposed for parallelization of Eclipse OCL (Object Constraint Language) [6] constraints utilizing Communicating Sequential Processes (CSP). In this work, expressions are executed in parallel and their results are combined through binary operations. The authors demonstrated the equivalence of behaviour between the parallel and sequential representations of OCL CSP to prove the correctness of their approach. Their implementation utilized CSP as an intermediate representation, which was then transformed into C# code. However, users were required to manually specify the expressions to be parallelized.

In [7] Madani et al. demonstrated how executing an EVL program concurrently and in a distributed setting can result in a proportional decrease in execution time with more machines and larger models. Their methodology involves breaking down each EVL program into an ordered sequence of rule-element pairs. The EVL distributed execution engine replicates the program execution environment across multiple computers and assigns a subset of this sequence to each computer for independent execution. The main limitation of this approach is that it requires all workers to have a full copy of both the models and the program. Our work can address this limitation by distributing constraint groups to multiple machines and loading models based on the group's effective metamodels. In this way, each machine can load only the necessary information for executing the assigned constraints which is more efficient in terms of loading time and memory footprint.

Another optimisation technique is incremental execution. Incremental execution reduces execution time by avoiding unnecessary re-computation through caching. Cabot and Teniente [8] developed an algorithm for incremental model validation that guarantees the generation of the most efficient expression to validate a specific constraint when the model undergoes changes (such as Create/Read/Update/Delete events). They approach the problem of model validation from a conceptual standpoint and demonstrate that their solution automatically generates the most optimal expression for incremental validation in response to a CRUD event. This ensures that the least amount of work is required to execute the constraint.

While recent research has made progress in this field, existing solutions still have notable limitations when it comes to accessing and processing large models. To the best of our knowledge, none of the model validation optimisation techniques in the literature incorporates static analysis to enable constraint grouping.

The primary work on partial loading of XMI models is SmartSAX, which was introduced by Wei et al. [3]. SmartSAX enables the partial loading of XMI model files. In our approach, we leverage SmartSAX to load models partially based on effective metamodels, which is automatically extracted through the static analysis of EVL constraints. However, it is important to note that SmartSAX does not provide support for garbage collection to unload parts of a model from memory when they are no longer required.

In a wider context, several approaches have been proposed

TABLE II
EVALUATED MODELS

Model name	Number of elements	Size
model1	1,000,000	212.9 MB
model2	3,000,000	563.6 MB
model3	3,700,000	806.6 MB
model4	5,000,000	1.05 GB
model5	6,000,000	1.29 GB

for static analysis of model management programs such as AnATLyzer [9] for ATL, and [10] for Viatra2, however they focus on program correctness and completeness rather than on execution planning and optimisation.

VI. EVALUATION

In this section, we report on the results of experiments that measure the performance of different approaches discussed in Section III. By evaluating and comparing partitioning techniques, we aimed to provide insights into the advantages and time-saving potential associated with the constraint grouping approach.

We evaluated our approach on a system using Java VM 11.0.10 with Intel(R) Core(TM) i7, 16 GB memory and CPU @ 2.80 GB running Mac OS X Catalina.

For our experiments, we generated random large models using EMF (pseudo) random instantiator[5] developed by the AtlanMod Team[6]. The models conform to an Ecore-based metamodel of the Component language (see Figure 1). We validated five XMI models, from model1 to model5 (from a 212.9 MB XMI file with 1 million model elements to a 1.29 GB file with 6 million model element). The models are listed in Table II.

We have also implemented 10 constraints[7] in the Epsilon Validation Language (EVL) to validate these models.

Figure 9 illustrates the Component Language metamodel and all constraints that are used to validate the models conforms to this metamodel. Every constraint is represented by a coloured dot with a number, and classes and features of the metamodel required for its execution, are marked by the same colour.

For example, *ComponentValidName* constraint requires all instances of *Component* class and their *name* attribute. In Figure 9, *ComponentValidName* constraint is represented by a dark blue dot (number 2). Hence, there is a dark blue dot in the *Component* class compartment and its *name* attribute.

Within this structure, when multiple dots always appear together in the metamodel, it means they are using the same information and can be grouped. In Figure 9, *ComponentValidName* belongs to the same group as the *ComponentIsConnected* and *ComponentUniqueName* constraints, as they require a subset of the same information to be executed. By following the colour of these constraints (dark green (1), dark blue (2) and red (3)), it is shown that they appear in Figure 1 next to each other. Therefore, they are allocated to the same group according to Algorithm 1.

The execution time was measured for three partitioning modes. The modes are mentioned below:

1. Partitioning the model based on constraints and running each constraint independently (intermediate solution): In this mode, the model is partitioned based on individual constraints, and each constraint is executed separately. By running constraints independently, any dependencies or overlaps between constraints are ignored.

2. Partitioning the model based on groups of constraints using our proposed algorithm: In this mode, we use our developed algorithm to group the constraints based on specific criteria (see Section III-D). The algorithm determines which constraints should be executed together as a group. By grouping constraints, we aim to optimise the execution time by reducing the overhead of repeatedly loading the same information. This approach considers the overlaps between constraints.

3. Partitioning the model based on groups of constraints specified by the user: In this mode, the user defines how constraints should be grouped based on their understanding of the model and the relationships between constraints.

To evaluate the execution time in each partitioning mode, we measured the time taken to execute the constraints for each mode separately. By comparing the execution times across the three modes, we can assess the efficiency and effectiveness of each approach in terms of constraint execution time.

The results are shown in Figure 10, and they were computed after 3 warm-up iterations and represent the average over 10 executions of the program.

The obtained results are illustrated in Table III, showing some findings.

TABLE III
LOADING TIME AND EXECUTION TIME FOR MODEL4 (IN MILLISECONDS)

	Time in Mode 1		Time in Mode 2	
Constraint	**Loading**	**Execution**	**Loading**	**Execution**
CUN	13.012	10,299	17.238	73.786
CIC	17.338	58.141		
CVN	14.107	7.466		

In the experiments, it is observed that the execution engine, under the first mode of partitioning, requires approximately 13 seconds to load the necessary model elements and properties for executing the *CUN* (*ComponentUniqueName*) constraint. Similarly, it takes roughly 17 seconds to load the required information for the *CIC* (*ComponentIsConnected*) constraint and about 14 seconds for the *CVN* (*ComponentValidName*) constraint. In Figure 10, the blue column represents execution time in mode 1 for model1 to model5.

In the second mode, the constraints are grouped based on Algorithm 1. The *ComponentIsConnected* constraint required all instances of *Component* class with the *name* attribute and *ports* reference. Furthermore, this constraint belongs to the same group as the *ComponentValidName* and *ComponentUniqueName* constraints. These constraints are in the same

[5]https://github.com/atlanmod/mondo-atlzoo-benchmark/tree/master/fr.inria.atlanmod.instantiator

[6]https://www.imt-atlantique.fr/fr

[7]https://eclipse.dev/epsilon/playground/?858b6314

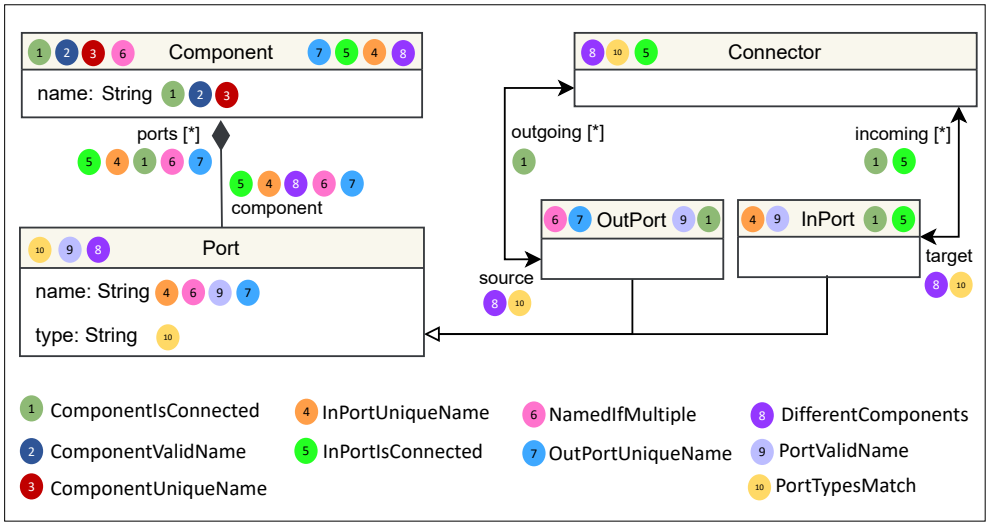

Fig. 9. The coverage of evaluated constraints

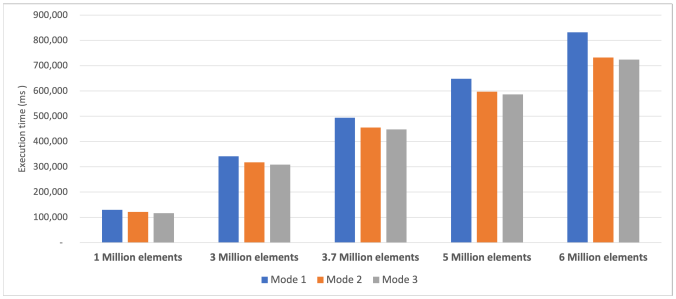

Fig. 10. Execution time for different models

group as they require a subset of the same information to be executed (See Figure 9). As shown in Table III, the time for loading the necessary information for all of these constraints is about 17 seconds (which is equal to the loading time of *ComponentIsConnected* constraint). The loading time for the two other constraints can be saved by grouping these constraints together, as they can now reuse the information loaded for the *ComponentIsConnected* constraint. As demonstrated in Table IV, constraints are divided into seven groups in mode 2.

Reviewing the execution time in both modes in Table III, it is worth noting that there is no execution time overhead observed when executing the constraints. This implies that the constraint execution time remains consistent regardless of the constraint grouping method applied (only the model loading times differ).

TABLE IV
PARTITIONS IN TWO MODES

groups in mode 2	groups in mode 3
ComponentUniqueName ComponentValidName ComponentIsConnected	ComponentUniqueName ComponentValidName ComponentIsConnected
outPortUniqueName NamedMultiple	outPortUniqueName NamedMultiple
InportIsConnected	InportIsConnected
InportUniqueName	InportUniqueName
DifferentComponents	DifferentComponents
PortTypeMatch	PortTypeMatch
PortValidName	PortValidName

In Figure 10, the orange column represents the execution time in mode 2 for the XMI models, while the blue column represents the execution time in mode 1. By comparing these two columns, it becomes evident that the grouping of constraints has a noticeable impact on the execution time.

In the third mode of partitioning, two groups remain the same as in mode 2 (first and second rows in Table IV), based on Algorithm 1. However, an additional group is added by the user (third row of mode 3 in Table IV). The information needed for executing the *InPortUniqueName* constraint is similar to that for the *InPortIsConnected* constraint, with just one additional attribute needed for executing the former. Upon examining our test models, it was realised that, for example, out of the 5 million elements, there are over 2 million instances of InPort. Considering this insight, unloading 2 million elements and loading them again would likely be more time-consuming compared to loading one more attribute.

To address this, in this experiment, we grouped the *InPortUniqueName* and *InPortIsConnected* constraints together to save loading time for the *InPortUniqueName* constraint. By grouping these constraints, a part of the loading time for loading the *InPortUniqueName* constraint's required information is saved, as the necessary information is already loaded while considering the additional attribute. This grouping strategy takes advantage of the machine's available memory and assumes that accommodating one more attribute is feasible.

In Figure 10, the grey column represents the execution time in the third mode, and this grouping approach effectively decreases the overall execution time.

As a general observation in our experiment, considering the models and constraints that we evaluated, grouping constraints can result in a time saving of approximately 10% to 12% in the execution time of EVL constraints.

Comparing the three columns for each model in Figure 10

demonstrates that when constraints are grouped together, there is a significant improvement in the overall execution time. It shows how manual grouping of constraints based on insights about the model and the trade-off between loading elements versus loading additional attributes can lead to significant further time savings in constraint execution. Also, the increasing differences between the columns as the model gets larger demonstrate that our approach is capable of handling larger models more efficiently. It highlights the scalability of our approach and indicates that larger models benefit even more from the grouping of constraints.

Regarding the correctness of our approach, we conducted validation by executing the EVL constraints using both the default EVL execution engine and our approach. The aim was to ensure that the output generated by both execution methods is identical in terms of the number of satisfied and unsatisfied constraints, regardless of whether partial loading and partitioning were applied. This validation process ensures that our approach does not introduce conflict or any inconsistencies in the constraint execution process compared to the default execution engine. By establishing the equivalence of the outputs, we can conclude that our approach for partial loading and partitioning maintains the correctness of the constraint evaluation, producing results consistent with the default execution engine.

VII. Threats to validity

To address construct validity threats, we specifically consider large models generated by a random generator that conform to the Component Language metamodel. The results presented in this paper are based on these test cases.

To mitigate internal validity threats, we ensure reliable and consistent results by executing three warm-up iterations of the program before reporting the final results. Additionally, we take an average of 10 executions to minimize any potential impact on memory usage and execution time caused by JVM startup and initialization.

To minimize external validity threats, we build our approach on top of mature and robust Model-Driven Engineering (MDE) technologies, such as the Epsilon suite of model management programs and the XMI format. These technologies provide a solid foundation for our approach. We acknowledge that extending our approach to support other technologies is relatively straightforward with reasonable effort, as discussed in Section III. However, further experiments are needed to determine the applicability and scalability of our approach in domains and with metamodels/models and constraints that have different characteristics from those used in our experimental evaluation. By addressing these threats and taking measures to ensure reliability and generalizability, we strive to provide a comprehensive evaluation of our approach in this paper.

VIII. Conclusion

We proposed an approach for grouping the EVL constraints using static analysis of EVL programs, and we conducted evaluations using five large randomly generated models. The results demonstrate that grouping constraints can significantly reduce the loading and overall execution time of EVL programs compared to a non-grouping approach without any impact on program behaviour or output.

In terms of future work, we intend to enhance user support by providing additional information about the model. This will enable users to make more informed decisions when grouping constraints, leading to further reductions in execution time. Additionally, we are interested in exploring the parallel execution of constraints and executing them on distributed setups. By loading information based on the constraints that will be executed on each machine, we can reduce loading time and optimise the execution process.

These future directions will contribute to improving the efficiency and scalability of the partitioning approach, ultimately benefiting users by reducing the time required for executing EVL constraints on large models.

Acknowledgment

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] D. S. Kolovos, R. F. Paige, and F. A. Polack, "On the evolution of ocl for capturing structural constraints in modelling languages." Springer, 2009, pp. 204–218.

[2] R. Wei and D. Kolovos, "Automated analysis, validation and suboptimal code detection in model management programs," in *CEUR Workshop Proceedings*, vol. 1206, 01 2014, pp. 48–57.

[3] 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*, ser. MODELS '16. Association for Computing Machinery, 2016, p. 329–339.

[4] S. Jahanbin, D. Kolovos, S. Gerasimou, and G. Sunyé, "Partial loading of repository-based models through static analysis," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, 2022, pp. 266–278.

[5] T. Vajk, Z. Dávid, M. Asztalos, G. Mezei, and T. Levendovszky, "Runtime model validation with parallel object constraint language," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, 2011, pp. 1–8.

[6] J. Cabot and M. Gogolla, *Object Constraint Language (OCL): A Definitive Guide*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90.

[7] S. Madani, D. Kolovos, and R. F. Paigea, "Towards optimisation of model queries: A parallel execution approach," *Journal of Object Technology*, 2019.

[8] J. Cabot and E. Teniente, "Incremental evaluation of ocl constraints," in *Advanced Information Systems Engineering: 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006. Proceedings 18*. Springer, 2006, pp. 81–95.

[9] J. S. Cuadrado, E. Guerra, and J. de Lara, "Anatlyzer: An advanced ide for atl model transformations," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 85–88. [Online]. Available: https://doi.org/10.1145/3183440.3183479

[10] Z. Ujhelyi, "Static analysis of model transformations," Master's thesis, Budapest University of Technology and Economics, May 2009.