UNIVERSITY of York

This is a repository copy of *Towards Efficient Model Comparison using Automated Program Rewriting*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/204203/

Version: Accepted Version

Proceedings Paper:
Ali, Qurat Ul Ain, Kolovos, Dimitris orcid.org/0000-0002-1724-6563 and Barmpis, Konstantinos (2023) Towards Efficient Model Comparison using Automated Program Rewriting. In: Saraiva, Joao, Degueule, Thomas and Scott, Elizabeth, (eds.) Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2023):SPLASH 2023. Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2023), 22-27 Oct 2023 SLE 2023 - Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with: SPLASH 2023 . ACM , PRT , pp. 181-193.

https://doi.org/10.1145/3623476.3623519

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Towards Efficient Model Comparison using Automated Program Rewriting

Qurat ul ain Ali
quratulain.ali@york.ac.uk
University of York
UK

Dimitris Kolovos
dimitris.kolovos@york.ac.uk
University of York
UK

Konstantinos Barmpis
konstantinos.barmpis@york.ac.uk
University of York
UK

Abstract

Model comparison is a prerequisite task for several other model management tasks such as model merging, model differencing etc. We present a novel approach to efficiently compare models using programs written in a rule-based model comparison language. As the comparison is done at the model element level, and each element needs to be traversed and compared with its corresponding elements, the execution of these comparison algorithms can be computationally expensive for larger models. In this paper, we present an efficient comparison approach which provides an automated rewriting facility to compare (both homogeneous and heterogeneous) models, based on static program analysis. Using this analysis, we reduce the search space by pre-filtering/indexing model elements, before actually comparing them. Moreover, we reorder the comparison match rules according to the dependencies between these rules to reduce the cost of jumping between rules. Our experiments demonstrate that the proposed model comparison approach delivers significant performance benefits in terms of execution time compared to the default ECL execution engine.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: Model-Driven Engineering, Scalability, Model Comparison, Static Analysis, Program Analysis

ACM Reference Format:
Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2023. Towards Efficient Model Comparison using Automated Program Rewriting. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623519

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623519

1 Introduction

While there is increased adoption of Model-Driven Engineering (MDE) principles, tools and technologies in industry [10], going forward scalability of these tools remains one of the key challenges [9]. To enable the use of MDE in large-scale applications it is essential to make MDE tools and technologies scalable. Model management languages are often interpreted and hence slower compared to general-purpose programming languages [22]. So, optimising these model management languages can deliver performance benefits on the top of already provided underlying dedicated task-specific support.

Model comparison is usually a prerequisite to various other key model management activities such as model differencing, model versioning, etc. It involves establishing matches/correspondences between elements of two models. There are different possible ways to compare models, such as traditional text-based comparison, comparison based on unique identifiers, model-to-model (M2M) transformation to establish comparison as in [11], or to use a dedicated comparison language, such as the Epsilon Comparison Language (ECL), which supports specifying matching criteria. Such model comparison can be computationally very expensive because each element of the first model needs to be traversed and compared to a corresponding element of the second model, which does not scale well.

In this paper, we introduce an efficient model comparison approach based on static program analysis and automated program rewriting. We have developed a prototype implementation of the proposed approach that can rewrite ECL programs, which operate on models with Ecore-based metamodels. According to the current ECL engine, all elements of one type are compared against the elements of their matching type based on the provided comparison logic by the developer. Using program analysis, we pre-filter the elements to be compared, index them and then compare the pre-filtered instances rather than all instances. These pre-filtered elements are automatically embedded into the original ECL program, using program rewriting, and executed using the traditional ECL engine. Also, we ensure that all the rules that are needed for the execution of a particular rule have already been executed, by reordering the rules, to avoid extra overhead of finding the appropriate rule to invoke. The output of an ECL program is a match trace that

contains all the established results of matches between elements of two models. Our approach yields a reduced match trace, by omitting any unsuccessful matches, whenever it was possible to identify these beforehand through program analysis.

Our proposed approach has shown performance gains up to 95% in terms of execution time in the experiments we have conducted.

The rest of the paper is structured as follows: Section 2 presents the background concepts, tools and technologies used for the implementation of the proposed approach, followed by a running example. Section 3, presents the overview of the proposed comparison optimisation approach and then discusses each component step-by-step. Experiments, case studies and the obtained results are presented and analysed in Section 4. Section 5 discusses the relevant state-of-the-art in the field of model comparison optimisation and static analysis. Finally, Section 6 concludes the paper and presents direction for future work.

2 Background & Motivation

This section provides the background concepts and a brief overview of the technologies used to implement the proposed approach. It also presents a running example which will be used to motivate this work.

2.1 Model Comparison

Model comparison is one of the fundamental model management tasks, usually a prerequisite for other tasks such as versioning, model merging, model differencing and model transformation testing. Model comparison establishes correspondences between matching elements of two models[14]. Such comparison can be performed both on homogeneous and heterogeneous models. One example scenario could be to identify matching elements before merging two models. Such correspondences can also be used to test model-to-model transformation pairs (source and corresponding target elements). Moreover, model comparison can be used in order to establish matching elements before calculating the differences between two models.

2.2 Epsilon

Epsilon [4] is a family of task-specific languages for performing several model management tasks, such as model merging (Epsilon Merging Language - EML [17]), model validation (Epsilon Validation Language - EVL [1]), model-to-model transformation (Epsilon Transformation Language - ETL [18]) and pattern matching (Epsilon Pattern Language - EPL [15]). All these languages extend a core language, the Epsilon Object Language (EOL) [16], which provides imperative constructs such as loops, conditionals and operations (both built-in and user-defined). All languages of Epsilon support managing models from a number of modeling technologies (and their respective persistence formats), through a uniform interface, the Epsilon Model Connectivity (EMC) layer [5].

The reason for choosing Epsilon as the basis of this work is twofold. Firstly, Epsilon provides a dedicated language for model comparison. Secondly, the developed optimisation facilities can be leveraged by a wide range of modelling technologies, as Epsilon supports languages like EMF, Simulink and XML, and can be further extended to work with currently unsupported technologies using its EMC layer.

2.3 Epsilon Comparison Language

The Epsilon Comparison Language (ECL)[1] is a hybrid rule-based dedicated model comparison language, provided by the Epsilon framework. ECL lets developers specify custom comparison algorithms in a rule-based script to identify matching elements between homogeneous and heterogeneous models. An ECL program contains a number of *MatchRules* and optional pre and post-block(s) executing before and after the rules respectively. A *MatchRule* enables developers to specify comparison logic between model elements at a high level of abstraction. *MatchRules* consist of a declared name along with two parameters (left and right) to specify the types of elements they can compare. A *MatchRule* can also optionally extend a number of match rules and can be labelled as *abstract*, *lazy* and/or *greedy* using corresponding annotations.

- An **abstract match rule** must be extended by other *MatchRule*s. Abstract match rules cannot be invoked standalone, they get invoked only when the rules that extend them are invoked.
- A **lazy match rule** will get executed only when it is required by another *MatchRule*, using the *matches* operation.
- A **greedy match rule** is executed for *all* pairs that have a kind-of relationship with the types specified by the left and the right parameters of the *MatchRule*.

The execution engine automatically evaluates non-abstract, non-lazy match-rules in two passes, starting with the order in which they appear.

2.4 Motivating Example

In this paper, as a running example, we consider comparing class diagrams with sequence diagrams. Figure 1 is illustrates the metamodel of a class diagram language. The class diagram shows a structural view of the system containing the classes, their attributes and their operations.

Then we consider a metamodel of a sequence diagram, an excerpt of which is shown in Figure 2. Sequence diagrams show the interaction between objects of a system - its intended behaviour.

[1] https://www.eclipse.org/epsilon/doc/ecl/

```
model Left driver EMF{
nsuri = "sd"
};

model Right driver EMF{
nsuri = "cd"
};

rule Lifeline2Class
  match l : Left!Lifeline
  with r : Right!Class {
  compare : l.type = r.name
}

rule Message2Operation
  match l : Left!Message
  with r : Right!Operation {

  compare : l.`operation` = r.name
    and (l.`to`.matches(r.class) or l.`to`.
        matches(r.class.superTypes)) and l.
        parameters.matches(r.parameters)
}

rule Param2Param
  match l: Left!Parameter
  with r: Right!Parameter {

  compare : l.name = r.name and l.type = r.
      type.name
}

operation String matchOperation(others :
    Collection<Right!Operation>) : Boolean
    {
   return others.exists(o|o.name = self);
}
```

Listing 1. Example ECL script before optimisation

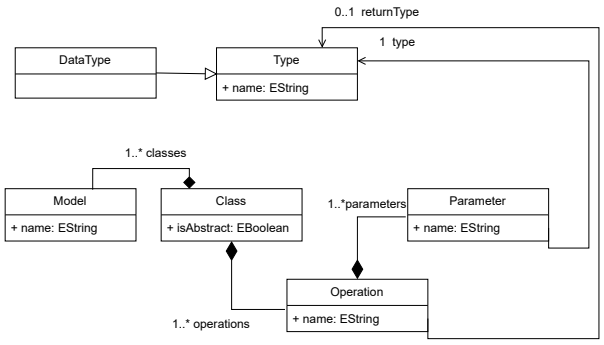

Figure 1. An excerpt of the Class Diagram metamodel

Now, as a sequence diagram depicts the interaction between objects and a class diagram represents the classes and their features, we can establish correspondences between the two, which can be used for downstream activities such as validation, model merging etc.

```
model Left driver EMF {
nsuri = "sd"
};

model Right driver EMF {
nsuri = "cd"
};

pre {
  var Lifeline2ClassMap = Right!Class.all.
      mapBy(param|param.name);
  var Message2OperationMap = Right!
      Operation.all.mapBy(param|param.name)
      ;
  var Param2ParamMap = Right!Parameter.all.
      mapBy(param|param.name);
}

rule Lifeline2Class
    match l : Left!Lifeline
    with r : Right!Class
    from : Lifeline2ClassMap.get(l.type) ?:
        Sequence{}{
   compare : true
}

rule Param2Param
    match l : Left!Parameter
    with r : Right!Parameter
    from : Param2ParamMap.get(l.name) ?:
        Sequence{}{
     compare : true and l.type = r.type.
        name
}

rule Message2Operation
    match l : Left!Message
    with r : Right!Operation
    from : Message2OperationMap.get(l.`
        operation`) ?: Sequence{} {
     compare : true and (l.`to`.matches(r.
        class) or l.`to`.matches(r.class.
        superTypes)) and l.parameters.
        matches(r.parameters)
}

operation String matchOperation(others :
    Collection(Right!Operation)) : Boolean
    {
  return others.exists(o : Right!Operation|
      o.name = self);
}
```

Listing 2. Example ECL script after optimisation

Table 1. Match trace produced from the execution of Listing 1 on the models in Figure 3 and Figure 4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (qa: User)	Class (ATM)	False
3	Lifeline (qa: User)	Class (Card)	False
4	Lifeline (hsbc: ATM)	Class (User)	False
5	Lifeline (hsbc: ATM)	Class (ATM)	True
6	Lifeline (hsbc: ATM)	Class (Card)	False
7	Message (enterPin)	Operation (verifyPin)	False
8	Message (enterPin)	Operation (dispenseCash)	False
9	Message (enterPin)	Operation (enterPin)	True
10	Message (enterPin)	Operation (depositCash)	False
11	Message (enterPin)	Operation (withdrawCash)	False
12	Message (enterPin)	Operation (activate)	False
13	Message (verifyPin)	Operation (verifyPin)	True
14	Message (verifyPin)	Operation (dispenseCash)	False
15	Message (verifyPin)	Operation (enterPin)	False
16	Message (verifyPin)	Operation (depositCash)	False
17	Message (verifyPin)	Operation (withdrawCash)	False
18	Message (verifyPin)	Operation (activate)	False

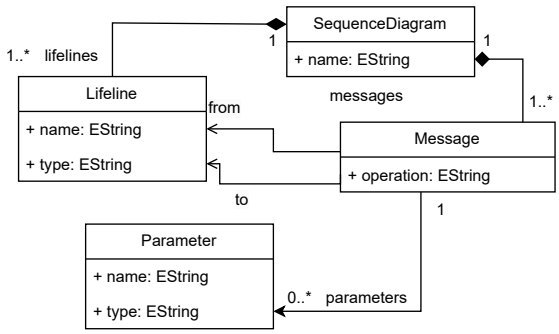

Figure 2. An excerpt of the Sequence Diagram metamodel

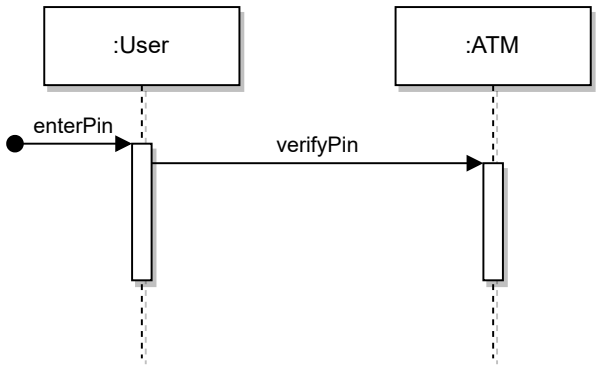

Figure 3. Sequence Diagram of ATM

A custom comparison algorithm written in ECL is shown in Listing 1. For this comparison, we have the following basic criteria:

- A lifeline matches a class when the type of the lifeline is the same as the name of the class in class diagram.
- A message matches an operation when the operation of the message is the same as the name of the operation. Also, the class corresponding to the "to" lifeline of the message or one of its supertypes should contain the operation.
- The parameters of the message need to be matched with the parameters of the operation.

We discuss some builtin operations supported by ECL and EOL that are used in the running example.

EOL supports a safe navigation operator ?., for making the null checks more concise. The use of the safe navigation operator is shown in the listing below, where we return *someProperty* if the *a* has a non-null value and returns *anotherProperty* if *a* contains a null value. `var result = a?.someProperty?.anotherProperty;`

Map (*mapBy(iterator : Type | expression)*) is a function that returns a map containing the results of the expression as keys and the respective items of the collection or collections of elements as values.

ECL provides a built-in operation *matches(right :Any)* for model elements and collections. When invoked, the *matches()* operation returns the cached result, if the elements have been already matched, otherwise, it finds rules that can compare the elements, executes them, and returns the result. In this ECL program, we have three match rules: *Lifeline2Class* (Line 9-13), which compares the type of lifeline to the class name (Line 12), *Message2Operation*, which compares the operation of *Message* with *Operation* name (Line 20). *Message2Operation* also compares whether the operation's owner class is the same as the *to* (Lifeline) of the *Message*. Finally, *Param2Param* compares the name and type of the parameters of both models (Line 27).

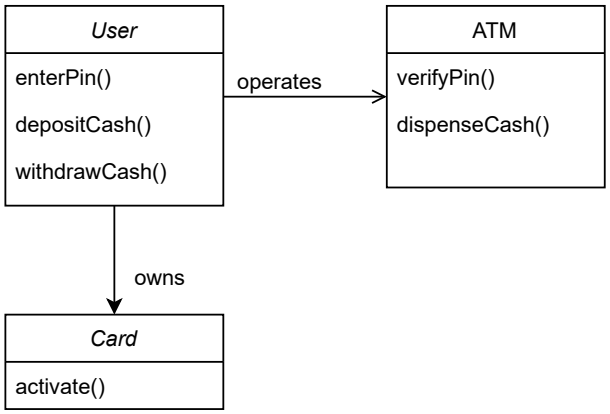

Figure 4. Class Diagram of ATM

As an example, let us consider matching a class diagram of an ATM system as shown in Figure 4) with its corresponding sequence diagram as shown in Figure 3. If we execute the ECL program (Listing 1) over these two models it would produce the match trace shown in Table 1. As we can see, it returns all matches of each element with its corresponding type and a boolean indicating if the element were matched or not.

The default execution engine of ECL will compare each instance of the left parameter (i.e., *Lifeline*) to all the instances of the right parameter (i.e., *Class*). The complexity of this rule here would be $O(M \times N)$, if there are M number of *Lifeline*s and N number of *Class*es. Using program analysis, we could index the instances by analysing these compare blocks as shown in Listing 2. Considering example sequence and class diagrams in Figure 3 and Figure 4, as there are 2 Lifelines and 3 Classes so there will be 6 matches for the rule *Lifeline2Class*. In the rule *Lifeline2Class*, we can filter the *Class* instances only keeping ones where the name of the class is equal to the *type* of the *Lifeline*. These indices can be pre-computed once, and then used as required. This could reduce the complexity to $O(M)$, considering the complexity of the hash function to be $O(1)$. Again considering the example models, if we execute Listing 2, the resultant match trace would be the same as shown in Table 2. This can be observed that there are only two matches for the same *Lifeline2Class* rule. Hence, the idea of this work is to analyse the ECL matching program and to automatically replace it with an efficiently rewritten program, to reduce the complexity of (some of) the comparisons.

3 Proposed Approach

In this section, we present proposed approach, an overview of which is illustrated in Figure 5. The idea is to optimise ECL matching programs automatically using program analysis. The developer writes the comparison algorithm in ECL to compare two models, say left and right. The expected outcome is a match trace resulting from computing the compare block of each match rule. The match trace contains a number of matches, each match contains the two objects that were matched and a boolean to indicate if the match was successful or not. So using the proposed optimisation approach we generate a match trace, which is a reduced or pre-filtered version, containing a significantly smaller number of unsuccessful matches. We have hence reduced the search space, making the comparison faster. This is because we do not compare all instances of left parameter to all instances of right parameter (which is done in existing ECL execution), rather we compare instances of left parameter to pre-filtered/pre-indexed instances of the right parameter.

The first step in our proposed approach is the (1) static analyser, a block used to populate the Abstract Syntax Tree (AST) of the ECL matching program, with the respective type information. This type resolved AST is then used for two purposes: i) For the (2) dependency graph extractor, a block that extracts the dependencies between different match rules of an ECL program by analysing compare blocks and *matches()* operations. Dependency here means that if a rule *MRx* invokes another rule *MRy* then the rule *MRx* would be dependent on *MRy*. The dependency graph is then used by the (4) rule scheduler to efficiently reorder the execution of rules. So that if a rule invokes another rule like in Line 20 of Listing 1, rule *Message2Operation* is dependent on rule *Lifeline2Class* and *Param2Param*. Both the rules on which *Message2Operation* is dependent should be executed before the execution of *Message2Operation*. ii) For the (3) optimisable rule detector, a block for program analysis to

Table 2. Match trace produced from the execution of Listing 2 on the models in Figure 3 and Figure 4

S #	Left	Right	Matching
1	Lifeline (qa: User)	Class (User)	True
2	Lifeline (hsbc: ATM)	Class (ATM)	True
3	Message (enterPin)	Operation (enterPin)	True
4	Message (verifyPin)	Operation (verifyPin)	True

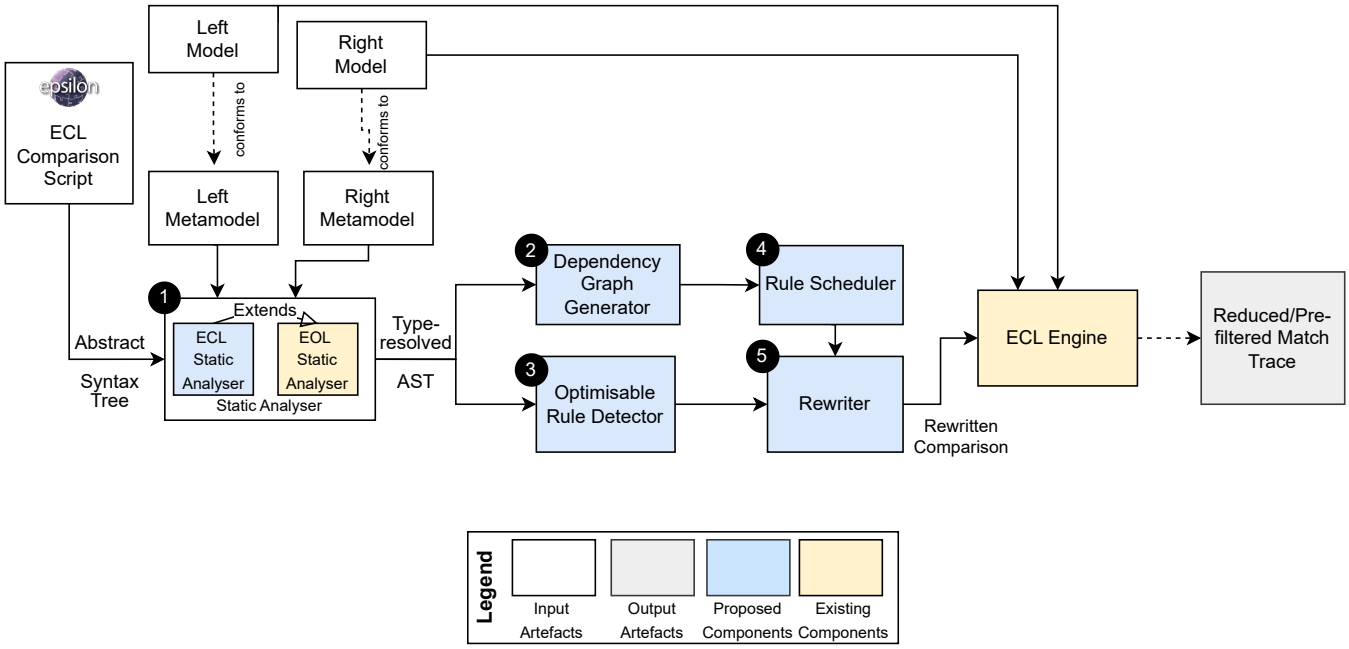

Figure 5. An overview of the proposed approach

identify the match rules which can be optimised based on the expressions in compare block. Here, optimisable rules mean the rules which are matching two elements on the basis of a specific property and can be indexed. So, this step identifies optimisable match rules along with the specific property name. Finally, (5) the rewriter block will replace the original program with a rewritten optimised program along with the new order of the match rules. This optimised comparison program will then be executed by the existing ECL engine. The resultant match trace would be a subset of the trace that would have been produced by the original comparison program. This subset trace would exclude the matches which would not satisfy the domain (an EOL expression to narrow the search space), while including all positive matches.

3.1 Static Analysis

Static analysis is the first step of our proposed approach workflow. It analyses the ECL program's abstract syntax tree (AST) and computes the types of all expressions in it. This type information is extracted using metamodel introspection, type resolution and type inference. *ModelDeclarationStatements* in Lines (1-3 and 5-7) in Listing 1 actually access the metamodel structure and help retrieve the types and their hierarchy available in the metamodel. To statically analyse ECL programs, we extended the already available EOL static analyser [2] by adding language specific support (e.g. analysing *MatchRules*, compare blocks etc.). The resolved types of various constructs in Listing 1 are shown in Table 3. The outcome of the static analyser block is a type-resolved AST, which is just the input AST with its nodes populated with their respective types.

3.2 Dependency Graph

When *matches()* operation is invoked, it returns the cached result, if the elements have been already matched. Otherwise, it finds the rule comparing the same two elements and

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

Table 3. Resolved types of various constructs in Listing 1

Line#	Expression	Resolved Type
13	l	Left!Lifeline
14	r	Right!Class
15	l.type	String
15	r.name	String
19	l	Left!Message
20	r	Right!Operation
21	l.operation	String
21	r.operations	Collection<Right!Operation>
21	r.superTypes.operations	Collection<Right!Operation>
29	l.parameters	Collection<Left!Paramter>
29	r.parameters	Collection<Right!Paramter>
33	l	Left!Parameter
34	r	Right!Parameter
36	l.name	String
36	r.type.name	String

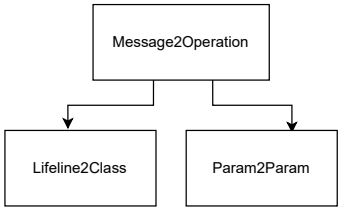

Figure 6. Dependency graph of Listing 1

then returns its results. Due to these rule invocations, rules can be dependent on one another. These dependencies can be extracted by the help of the type resolved AST, as done for model-to-model transformations in [8]. To construct a dependency graph, we create a vertex for each *MatchRule* declared in the ECL program. If a rule *MRx* has a statement in its compare block that calls a *matches()* operation which invokes another rule, say *MRy*. The resolution of which rule is invoked by the *matches()* operation is done by finding the rule where the type of the left and right parameters of rule is the same as the type of the target and parameter expressions of the *matches()* operation. Then, we create an edge from the vertex corresponding to *MRx*, to the vertex corresponding to *MRy*. If there are multiple rules invoked by the *matches()* operation, we create multiple edges from *MRx*. For example, as in rule *Message2Operation* Line 19 of Listing 1 there is a call to the *matches()* operation with *l.to* (resolved type: *Lifeline*) as the target expression and *r.class* (resolved type: *Class*). This means that this *matches* operation call will invoke a rule which is matching *Lifeline* with *Class* i.e., rule *Lifeline2Class*. So, we create an edge from *Message2Operation* to *Lifeline2Class* as shown in Figure 6. The reason for extracting the dependency graph is to reorder the rules in a way that if *MRx* is invoked by a rule *MRy* then *MRy* is scheduled before *MRx*. When *x.matches(y)* is called in ECL, if x and y have not already been matched, the ECL engine needs to find rule(s) that can match them, invoke these rules and return the result to *matches(...)*. This can have a non-negligible cost for large models and sets of match rules. By reordering rules to maximise the number of pairs of x and y that have been already matched before *x.matches(y)* is called, we reduce that cost of *jumping* between rules. This rescheduling can help improve performance, because it can reduce the number of attempts needed to find the appropriate rules to invoke. Any rule invocation using a *matches()* operation can use the cached results in the match trace, if the rules have been reordered properly. We do not create an edge when a rule invokes itself, as it does not affect the reordering for which we extract dependency graph. However, ECL provides a mechanism to avoid an infinite loop, in case of a cyclic invocation of a rule i.e., two rules implicitly invoking each other. ECL maintains a temporary trace along with the primary trace. In a primary trace the matching value is added after the execution of compare block, while the matching value is set to true in the temporary trace before the execution of the compare block. In case of another attempt to match elements from already invoked rules, these rules would not be re-invoked. Finally, the temporary trace is reset when a top-level rule returns.

3.3 Identifying Optimisable *MatchRules*

This is the third step of the approach that takes in a type-resolved AST as an input with the aim to identify the rules which can be optimised. By optimisable rules, we mean the rules which are comparing the elements of the two models based on a specific property. This is done by traversing the compare block of each *MatchRule* and finding expressions where two elements are compared on the basis of a specific property or attribute. Currently, the rewriting approach only considers equality operators, as checking for name/id-like attribute equality is very common in model matching in our experience, but it can be extended to support other operators in the future too. In this case, the elements can be indexed based on that property. The process for identifying such optimisable rules is specified in Algorithm 1. The algorithm traverses a set of Match rules and its compare block. Then, in a compare block all DOM elements are traversed to identify cases where a *PropertyCallExpression* is used within an *EqualsOperatorExpression* and it records the relevant Match rules and properties in a HashMap for later use in indexing. With one exception, if there is a logical operator between equals expression, we just record the index if it is an and operator. For instance, in Listing 1 Line 12 the rule *Lifeline2Class* is

comparing the *Lifeline* from Sequence diagram to *Class* from Class diagram on the basis of the property *name*, in this class. So the Algorithm 1, would return the hashmap containing rule *Lifeline2Class* with the respective property "*name*".

Algorithm 1 Algorithm for Identifying optimisable rules

```
1:  Let op = HashMap<rule, NameExpression>
2:  for all Matchrules rule do
3:      Visit all DOM elements (elem) of compare block of rule
4:      if elem instanceof PropertyCallExpression and !(op.contain(rule)) then
5:          parent ← elem.parent
6:          if parent instanceof EqualsOperatorExpression then
7:              parent ← parent.parent
8:              if parent instanceof OperatorExpression then
9:                  if parent instanceof AndOperatorExpression then
10:                     op ← rule and elem.NameExpression
11:                 end if
12:             else
13:                 op ← rule and elem.NameExpression
14:             end if
15:         end if
16:     end if
17: end for
```

3.4 Program Rewriting

The final step is the rewriting phase illustrated in Algorithm 2, now that we have all the program analysis in place. As discussed in the previous step, we have identified the optimisable rules say *MR1, MR2.., MRn* along with the specific properties say *p1, p2.., pn* on the basis of which we are comparing the elements in the compare block. We index all instances of the right parameter of the identified rule *MRn* on the basis of the respective property *pn*. This is done using a built-in method called *mapBy* (Line 10 in Listing 2), which returns a map containing the results of the parameter expression as keys and the respective items of the target collection as values. The *mapBy* operation is called with all instances of the identified rule's right parameter and assigned to a newly declared variable (Line 4-10 of Algorithm 2). The naming convention of these variables is the rule name concatenated with the string "Map". So, a Map for the rule *Lifeline2Class* will be called as *Lifeline2ClassMap* (Line 11 of Algorithm 2). These variable statements are then added to the *pre* block of the ECL program (Line 13 of Algorithm 2). The *Pre* block is a set of EOL statements that are executed before the execution of match rules in ECL. This can be seen in Listing 2 (Line 10-12).

The next step is to utilise these pre-computed hashmaps (indices). For this, we have added the facility of specifying domains in ECL. Each parameter in an ECL rule can define a domain, which is an EOL expression that yields a set of model elements, allowing the developers to narrow down the search space. We support two types of domains in ECL. Static domains which are computed once for one match rule and are independent of bindings of the other parameter of the *MatchRule*. Static domains are denoted by the "*in*" keyword) and dynamic domains which are recomputed every time the other parameter value is changed. Dynamic domains are dependent on the other parameter values and are denoted by the "*from*" keyword. So we use these hashmap variables added in the pre block, as a dynamic domain for the right parameter of the corresponding *MatchRule*. For instance in Line 18 of Listing 2, we retrieve the value from the corresponding hashmap i.e., *Lifeline2ClassMap* using the left parameter's compared property (identified in the previous step) as a key.

Hashmaps return null values if they don't contain the mapping for a particular key, so to cater for possible null pointer exceptions, we use a safe navigation operator. The use of the safe navigation operator is shown in Line 18 of Listing 2, where we return an empty *Sequence* if the *get()* operation returns a null value. `var result = a?.someProperty?.anotherProperty;`

If *a* is not null, *someProperty* would be assigned to *result*, otherwise, *anotherProperty* would be assigned.

The last step of the rewriting phase is to rewrite the order of the rules as described in the Rule Scheduler step. The reordering is done on the basis of the dependency graph as in Figure 6, so that dependency-free rules can be executed first and then the ones dependent on them. Now, instead of the ECL engine executing the original program written by the developer, as listed in Listing 1, the automatically rewritten program as in Listing 2 will be executed. During execution, to minimize the storage of unnecessary unsuccessful match traces, only the unsuccessful traces that are required based on the dependency graph (i.e., if there are no corresponding *matches()* calls are saved.

4 Evaluation

In this section, we first present the experimental setup, including the case study and the models used for our benchmarks, and then we present the results of the conducted experiments. Finally we conclude the section by analysing and then stating any threats to the validity of the presented results.

4.1 Experimental Setup

To evaluate the proposed approach, we measured the execution time of the original ECL programs using the existing ECL engine with the rewritten ECL programs (also using the existing ECL engine). Since Epsilon already supports parallel execution of ECL programs, we conducted all these experiments with the parallel execution mode. Program rewriting

Algorithm 2 Algorithm for Program Rewriting

1: Let *op* = HashMap of rules with the corresponding properties as in Algorithm 1
2: *DG* = Dependency Graph
3: **for all** *rule* in *op* **do**
4: Construct property call expression (*pce*)
5: target ← type of right parmeter of *rule*
6: property ← all
7: Construct operation call expression (*oce*
8: target ← *pce*
9: operation ← mapBy
10: expression ← *op*.get(*rule*)
11: declare variable (*v*) with name *rule*.getName()+"Map"
12: *v* ← *oce*
13: add *v* to *pre* block
14: add domain block with expression *v*.get(leftParameter.property)
15: **end for**
16: reorder rules according to topological order of *DG*

with the help of dependency graph, identifies the independent rules that can be executed in parallel. First, we measured the execution time for running the comparison program with the existing ECL engine (without any optimisations) in parallel mode and we refer this as ECL in all the results tables and graphs. Second, we use the proposed approach to automatically rewrite the ECL program (as described in Section 3 and execute the rewritten program using the existing ECL engine in parallel mode. We refer to this as *Optimised ECL* in the results tables and graphs.

Table 4. Sizes of the models used for benchmarking

	No of model elements					
ID	**OO**	**DB**	**OO+DB**	**Seq**	**Class**	**Class+Seq**
1	287	184	471	305	356	661
2	357	229	586	417	356	773
3	427	274	701	417	469	886
4	497	319	816	342	356	698
5	567	364	931	342	356	698
6	637	409	1046	305	469	774
7	707	454	1161	342	469	811

4.1.1 Case Study & Models. For evaluating our approach, we used two case studies: one is the class and sequence diagram comparison as shown in Listing 1, the second is the comparison of object oriented (OO) models with database (DB) models. We have used the class and sequence diagram models of different sizes conforming to these metamodels publicly available on GitHub [13]. OO & DB are the synthetic models generated in [8]. The number of elements of different models are mentioned in Table 4. The point to note is that the sizes of the models that we are using are not very large but the comparison of these models still becomes computationally very expensive. Hence, a notable performance gain can be observed in these models.

```
rule Class2Table
  match l : OO!Class
  with r : DB!Table{

  compare : l.name = r.name
}

rule Attribute2Column
  match l : OO!Attribute
  with r : DB!Column
  {
  compare : l.name = r.name and l.owner.
       matches(r.table)
}
```

Listing 3. ECL comparison program for OO-DB models

To compare OO models with DB ones, we used a simple comparison algorithm (depicted in Listing 3) to establish matches between tables and classes, when their names are same. In the second rule, we compare attributes with columns on the basis of the property *name*, and also whether they belong to same class and table respectively. This example is quite simple but we have used this as a case study to show the substantial performance benefits observed even for simpler matching programs, with increasing model sizes. The comparison program in Listing 3 would be optimised and rewritten as represented in Listing 4.

4.1.2 Correctness. As the approach is based on automatic rewriting of the program, it is crucial that the rewritten program preserves the semantics of the original program. To ensure this, we use equivalence testing to compare the match trace for both the original and the rewritten programs. We used several ECL comparison programs mined from GitHub to compare models both conforming to same and different metamodels and then compared their output match traces. Mostly, comparison programs available on GitHub were comparing models from the same modelling language. We verified that the number of successful matches in both the optimised and the unoptimised version remained the same, as shown in Table 5 and 6. While the number of successful matches are the same, one can observe the difference in number of unsuccessful matches in the Table 5 and 6. This is because of the successful pre-filtering/pre-indexing in the proposed approach. We filter some of the instances which, using the static program analysis, can be categorised

as unsuccessful, before actually running the comparison algorithms.

```
pre {
  var Class2TableMap = DB!Table.all.mapBy(
      param|param.name);
  var Attribute2ColumnMap = DB!Column.all.
      mapBy(param|param.name);
}

rule Class2Table
match l : OO!Class
with r : DB!Table
from : Class2TableMap.get(l.name) ?:
    Sequence{} {
  compare : true
}

rule Attribute2Column
match l : OO!Attribute
with r : DB!Column
from : Attribute2ColumnMap.get(l.name) ?:
    Sequence{} {
  compare : true and l.owner.matches(r.
      table)
}
```

Listing 4. ECL rewritten program for OO-DB models

4.1.3 Machine Specification.

The set of evaluation experiments presented in this paper were performed on a MacBookPro @ M2 Core i7, 24 GBs of RAM, Mac operating system Ventura version 13.0, and Java 17 on JDK 17.0.6 with JVM MaxHeapSize 6GBs.

4.2 Results

In this section, we present the results from the conducted experiments. Table 7 presents the execution time in milliseconds for the OO and the DB model comparison, and the Class and Sequence Diagram model comparison respectively. This execution time also includes the time taken for indexing. The rewriting and reordering of rules are done before the execution and takes negligible amount of time (≈2ms). The results can also be visualised for the OO & DB comparison in Figure 7 and the Class and Sequence diagram in Figure 8.

Table 7. Execution time of existing ECL and optimised ECL, in ms

	OO - DB		CL - SEQ	
ID	ECL	Optimised	ECL	Optimised
1	1962	535	3287	196
2	3488	781	3109	194
3	6745	1238	3894	205
4	14051	1735	4046	188
5	22044	1924	4286	287
6	31611	3705	4342	199
7	52159	4520	5050	250

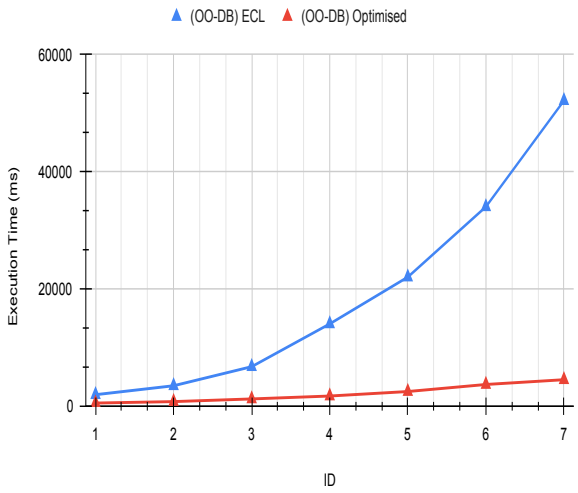

Figure 7. Comparison of Execution time in OO DB Comparison

As seen in Table 4, the OO and the DB models are of increasing sizes, while this is not the case with the Class and Sequence Diagram models. Keeping these sizes of models in mind, we can see a continuous rise in performance gain as the model size increases (Figure 7). While in Figure 8, we can see almost a constant performance gain compared to the existing ECL engine. This suggests that our performance benefits are proportional to model size.

This performance gain is achieved by reducing the search space needed for matching. We can clearly observe in the match traces produced for both case studies in the Tables Table 5 and Table 6 that the number of unsuccessful matches are significantly reduced in our proposed approach.

Another important factor to notice here is that this approach might not bring performance benefits when comparing very small models. As the proposed approach provides

Table 5. Match Trace of Class and Sequence Diagram Comparison

ID	ECL (All)	Optimised (All)	ECL (Successful)	Optimised(Successful)
1	29400	92	38	38
2	29400	92	38	38
3	33700	78	0	0
4	33284	54	0	0
5	33284	54	0	0
6	33700	78	0	0
7	38100	208	72	72

Table 6. Match Trace of OO and DB Comparison

ID	ECL (All)	Optimised (All)	ECL (Successful)	Optimised(Successful)
1	18060	4120	60	60
2	28200	6400	75	75
3	40590	9180	90	90
4	55230	12460	105	105
5	72120	16240	120	120
6	91260	20520	135	135
7	112650	25300	150	150

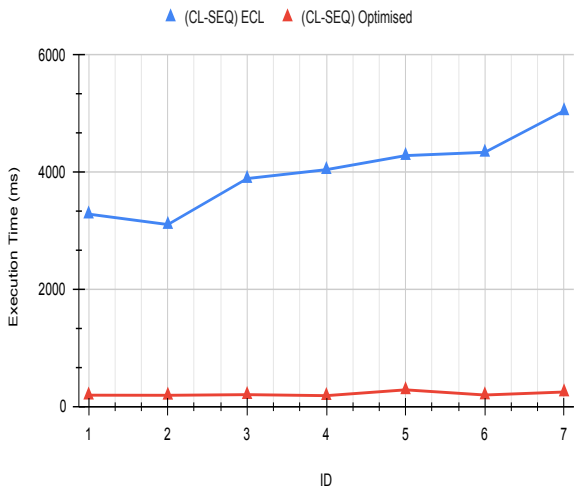

Figure 8. Comparison of Execution time in Class Sequence Diagram Comparison

a caching mechanism, the optimization indeed comes at the expense of increased memory footprint. As the size of the computed caches can be estimated from the number of rules/indexed properties in a straightforward manner. Precomputing the indices (as mentioned in the program rewriting section) has an overhead, which is paid off for larger models, and we expect to see a much clearer improvement in performance when it comes to larger models.

4.3 Threats to Validity

A primary threat to the validity of the results presented here, is that the measured performance may be particular to the models that were created for the tests, to the kind of model, or to the comparison programs that were proposed. A key challenge identified in MDE research is a lack of publicly accessible real-world models [20]. Although we used both synthetic models in the OO-DB case studies and publicly available models for the class and sequence diagrams one, this can still affect the measured performance benefits. To further generalise the results, we would need to perform experiments with different models and comparison programs as well as with different modeling technologies such as Simulink and CDO to demonstrate the scalability of our proposed approach, especially for larger models.

As the rewriting is based on static analysis, we recommend explicitly stating the types of the constructs wherever possible, to allow accurate type resolution and enable automated rule optimisation (as described in Section 3).

5 Related Work

Model comparison deals with finding similarities and differences between elements of different models. This comparison can be done on the basis of structure, semantics and metrics etc., [12]. In the context of this paper we will be stating the use of program rewriting in optimisation and also the state of the art that involves structural model comparison.

Program rewriting has proven to be beneficial for various optimization purposes, as demonstrated in [24] where it was utilized for optimizing type level model queries. Additionally, rewriting has played a crucial role in translating EOL expressions to Viatra for incremental evaluation [6], as well as converting them to MySQL queries for efficient execution on relational databases [7].

It has been demonstrated in [23] that conventional text-based comparison and differencing techniques are insufficient for model comparison due to the structured nature of models.

Model-to-model transformations have shown to be used for comparing models as in [11]. As M2M languages are not tailored for model comparison task and hence generally very verbose as M2M languages do not have constructs tailored for model comparison activities.

Change-based model comparison was presented in [25] where the comparison is done only for the model elements that have been changed since the previous version which is quite efficient compared to state-based comparison.

EMF Compare [2] & EMF Diff Merge[3] are two tools available to compare and then merge two models. EMF Compare uses built-in heuristics for model element references and attribute values while a tailored language like ECL lets you write custom matching rules for different model elements.

There are other comparison approaches as shown in [21] that demonstrates comparison between different UML models but the approach is only limited to models conforming to a single metamodel. Additionally as mentioned in [14], most similarity-based approaches such as SiDiff [23] and DSMDiff [19], have limited support when it comes to heterogeneous models which is supported by ECL, where one can specify complex matching algorithms for models conforming to different metamodels.

6 Conclusions & Future Work

We have presented an approach for efficiently comparing models using programs written in rule-based model comparison language. This efficient comparison approach incorporates an automatic rewriting facility to speed up the model comparison (both homogeneous and heterogeneous) based on static analysis. The rewriting automatically extracts dynamic domains to provide pre-filtering of model elements before actually comparing them. Additionally, static analysis also helps reorder the rules based on the dependencies identified between these match rules through the creation of a dependency graph. This enables us to execute independent rules before those dependent on them, optimizing the comparison process by reducing the cost of jumping between comparison rules. Through experiments, we demonstrate that our approach significantly improves execution time compared to the default ECL execution engine, providing substantial performance benefits.

In future work, the proposed approach can be potentially used to provide correspondence between models from heterogeneous modelling technologies. For instance, it can facilitate the comparison between Simulink models and EMF models. Moreover, this automatic domain rewriting facility can be integrated with other rule-based languages such as Epsilon's pattern matching language (EPL).

Acknowledgments

This research is supported by the Lowcomote project, funded by the European Union's H2020 Research and Innovation Programme under the Marie Skłodowska-Curie GA n°813884.

References

[1] 2022. Epsilon Validation Language. https://www.eclipse.org/epsilon/doc/evl/. [Online; accessed 29-April-2022].

[2] 2023. Eclipse EMF Compare. https://projects.eclipse.org/projects/modeling.emfcompare. [Online; accessed 10-April-2023].

[3] 2023. EMF DiffMerge. https://wiki.eclipse.org/EMF_DiffMerge. [Online; accessed 10-April-2023].

[4] 2023. Epsilon. https://www.eclipse.org/epsilon/. [Online; accessed 26-March-2023].

[5] 2023. Epsilon Model Connectivity Layer. https://www.eclipse.org/epsilon/doc/emc/. [Online; accessed 26-March-2023].

[6] Qurat Ul Ain Ali, Benedek Horváth, Dimitris Kolovos, Konstantinos Barmpis, and Ákos Horváth. 2021. Towards Scalable Validation of Low-Code System Models: Mapping EVL to VIATRA Patterns. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 83–87. https://doi.org/10.1109/MODELS-C53483.2021.00019

[7] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2020. Efficiently Querying Large-Scale Heterogeneous Models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 73, 5 pages. https://doi.org/10.1145/3417990.3420207

[8] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2022. Selective Traceability for Rule-Based Model-to-Model Transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 98–109. https://doi.org/10.1145/3567512.3567521

[9] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19, 1 (2020), 5–13.

[10] Justin Cooper, Alfonso De la Vega, Richard Paige, Dimitris Kolovos, Michael Bennett, Caroline Brown, Beatriz Sanchez Piña, and Horacio Hoyos Rodriguez. 2021. Model-Based Development of Engine Control Systems: Experiences and Lessons Learnt. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and*

Systems (MODELS). 308–319. https://doi.org/10.1109/MODELS50736.2021.00038

[11] Marcos Didonet Del Fabro and Patrick Valduriez. 2007. Semi-Automatic Model Integration Using Matching Transformations and Weaving Models. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea) *(SAC '07)*. Association for Computing Machinery, New York, NY, USA, 963–970. https://doi.org/10.1145/1244002.1244215

[12] Lucian Gonçales, Kleinner Farias, Murilo Scholl, Toacy Oliveira, and Mauricio Veronez. 2015. Model Comparison: a Systematic Mapping Study. https://doi.org/10.18293/SEKE2015-116

[13] Faezeh Khorram, Masoumeh Taromirad, and Raman Ramsin. [n. d.]. SeGa4Biz: Model-Driven Framework for Developing Serious Games for Business Processes. ([n. d.]).

[14] Dimitrios S. Kolovos. 2009. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Model Driven Architecture - Foundations and Applications*, Richard F. Paige, Alan Hartman, and Arend Rensink (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 146–157.

[15] Dimitris S Kolovos and Richard F Paige. 2017. The epsilon pattern language. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, 54–60.

[16] 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.

[17] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2006. Merging models with the epsilon merging language (eml). In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*. Springer, 215–229.

[18] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The epsilon transformation language. In *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings 1*. Springer, 46–60.

[19] Yuehua Lin, Jeff Gray, and Frédéric Jouault. 2007. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems* 16, 4 (2007), 349–361.

[20] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2021. Towards the Characterization of Realistic Model Generators using Graph Neural Networks. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 58–69. https://doi.org/10.1109/MODELS50736.2021.00015

[21] Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between Versions of UML Diagrams. *SIGSOFT Softw. Eng. Notes* 28, 5 (sep 2003), 227–236. https://doi.org/10.1145/949952.940102

[22] Massimo Tisi, Salvador Martínez, and Hassene Choura. 2013. Parallel execution of ATL transformation rules. In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*. Springer, 656–672.

[23] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. 2007. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 295–304.

[24] Qurat Ul Ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2021. Identification and Optimisation of Type-Level Model Queries. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 751–760. https://doi.org/10.1109/MODELS-C53483.2021.00121

[25] Alfa Yohannis, Rodriguez Hoyos Rodriguez, Fiona Polack, and Dimitris Kolovos. 2019. Towards Efficient Comparison of Change-Based Models. *Journal of Object Technology* 18, 2 (July 2019), 7:1–21. https://doi.org/10.5381/jot.2019.18.2.a7 The 15th European Conference on Modelling Foundations and Applications.

Received 2023-07-07; accepted 2023-09-01