Proceedings Paper:
Ali, Qurat Ul Ain, Kolovos, Dimitris orcid.org/0000-0002-1724-6563 and Barmpis, Konstantinos (2022) Selective Traceability for Rule-Based Model-to-Model Transformations. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022). Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022), 06-07 Nov 2022 ACM, NZL, pp. 98-109.

https://doi.org/10.1145/3567512.3567521

Selective Traceability for Rule-Based Model-to-Model Transformations

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

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

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

Abstract

Model-to-model (M2M) transformation is a key ingredient in a typical Model-Driven Engineering workflow and there are several tailored high-level interpreted languages for capturing and executing such transformations. While these languages enable the specification of concise transformations through task-specific constructs (rules/mappings, bindings), their use can pose scalability challenges when it comes to very large models. In this paper, we present an architecture for optimising the execution of model-to-model transformations written in such a language, by leveraging static analysis and automated program rewriting techniques. We demonstrate how static analysis and dependency information between rules can be used to reduce the size of the transformation trace and to optimise certain classes of transformations. Finally, we detail the performance benefits that can be delivered by this form of optimisation, through a series of benchmarks performed with an existing transformation language (Epsilon Transformation Language - ETL) and EMF-based models. Our experiments have shown considerable performance improvements compared to the existing ETL execution engine, without sacrificing any features of the language.

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

Keywords: Model-Driven Engineering, Scalability, Model Transformation, Static Analysis

ACM Reference Format:
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 (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3567512.3567521

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

ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567521

1 Introduction

With the growing adoption of MDE for developing large and complex industrial applications [18, 29], MDE tools and technologies are required to handle increasingly large and complex underlying models. While MDE is popular for providing benefits such as increased productivity, maintainability etc. [28], still there are certain limitations, especially when it comes to managing larger models. Scalability can often become a major bottleneck while transforming large models [15, 17], so in order to efficiently use MDE in larger and complex industrial applications, its tools and technologies need to be scalable.

Model-to-model (M2M) transformation is one of the key activities used in a typical MDE workflow. It is essentially used to map one or more input model(s) to one or more output model(s). Various M2M languages like ETL [22] and ATL [19] provide tailored support for automating this task, but they can face scalability issues when it comes to transforming larger models [24]. In this paper, we propose a novel approach that leverages the benefits of static analysis and automated program rewriting to speed up and reduce the memory footprint of model-to-model transformation programs. In our study we target the Epsilon Transformation Language (ETL), however, the proposed approach applies to any rule-based interpreted M2M language that supports imperative constructs. We use map-like data structure to cache the results of imperative operations generated by rules as a transformation trace.

The proposed approach involves statically analysing the M2M transformation, extracting type information of its various constructs and also extracting dependency information between the transformation rules as a dependency graph. Using the information extracted from the static analyser, a rule-based M2M program is then rewritten into an imperative M2M program, where the transformation rules are converted to operations. Moreover, exploiting the dependency graph allows reducing the global transformation trace into a selective trace, lowering its memory footprint. A key novelty of the proposed optimisation approach is that **it does not sacrifice any of the expressiveness** of the M2M language in contrast to e.g. [1], which only supports a subset of ATL. This is because our approach performs in-place

rewriting of rules and calls to *equivalent/equivalents()*, effectively desugaring a rule-based ETL transformation into an imperative form (still in ETL due to the potential presence of *pre/post* blocks). All other constructs of the transformation (e.g. method calls, property call expressions, user-defined operations, instantiation of native types) remain untouched and are executed using the standard ETL interpreter.

Using our proposed approach, performance gains up to 39% in terms of execution time and up to 59% in terms of memory consumption have been achieved in our evaluation experiments.

The remainder of the paper is structured as follows: Section 2 presents the background, tools and technologies used for the implementation of the proposed approach, followed by a motivating example. Section 3, presents the overall architecture of the proposed transformation optimisation approach and then discusses each stage step-by-step. Evaluation of benchmarks 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 transformation optimisation and static analysis. Finally, Section 6 concludes the paper and presents direction for further work.

2 Background

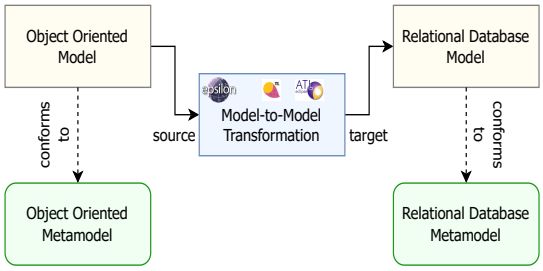

Figure 1. Model-to-Model Transformation Example

This section briefly presents the background and explains the tools and technologies used to implement the approach.

2.1 Model to Model Transformation

An M2M transformation is a program which consumes one or more input models in order to generate one or more output models [12]. A common case is a one-to-one transformation, where one input model is mapped to one output model e.g., mapping an object-oriented model to a relational model, as seen in Figure 1. Still, there can be scenarios where one-to-many, many-to-one and many-to-many transformations (as in the case of model integration) are useful.

2.2 Epsilon

Epsilon [3] is a family of task-specific languages for performing a number of model management tasks like model validation (Epsilon Validation Language - EVL [5]), model-to-model transformation (Epsilon Transformation Language - ETL) and pattern matching (Epsilon Pattern Language - EPL [20]). All these languages extend a core language, the Epsilon Object Language (EOL) [21], which provides imperative constructs such as loops, conditionals and operations (both built-in and user-defined). EOL is inspired by OCL [6], a widely used constraint language, and has a similar syntax. 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 [4].

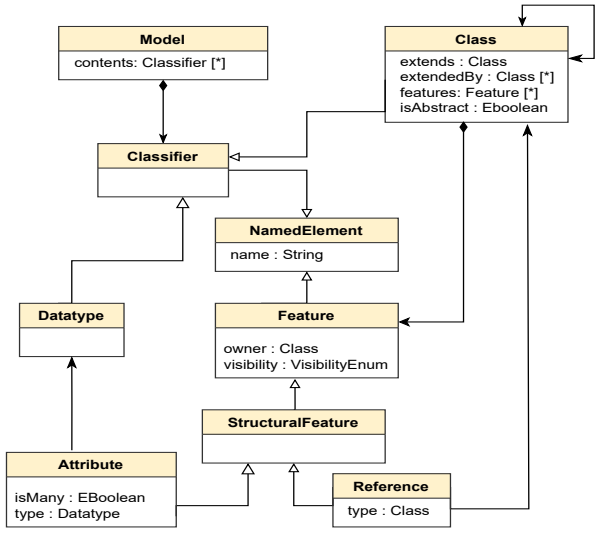

Figure 2. Object Oriented Metamodel

The reason for choosing Epsilon as the basis of this work is that 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 unsupported technologies using its EMC layer.

2.3 ETL

ETL [22] is a hybrid rule-based language for model-to-model transformation in Epsilon. An ETL program (module) takes as input a number of source models and transforms them into a number of target models. The models, as in other Epsilon languages, can be of heterogeneous modelling technologies (e.g., an EMF model can be transformed to a Simulink model or an XML document can be transformed to an Excel spreadsheet). An ETL module can contain a number of transformation rules, transforming source model elements to one or more target model elements. An ETL module can optionally have a *pre* and a *post* block of statements, to be executed before and after the execution of transformation rules respectively. A transformation rule can extend one or more other transformation rules and can be declared as *abstract* or *lazy* through relevant annotations:

- An **abstract rule** must be extended by another transformation rule. Such rules cannot be invoked standalone, they get invoked only when the rule that extends them is invoked.
- A **lazy rule** will get executed only when it is required by another transformation rule.

In a model-to-model transformation, resolving elements created by other transformation rules is quite a common and recurring task. For this resolution ETL provides the *equivalent()*/*equivalents()* operations. The elements returned by these operations follow the respective order of the rules that have created them. An exception to this occurs when one of the rules is declared as *primary*, in which case its results precede the results of all other rules.

2.4 Motivating Example

Let us consider the example of a partial (for conciseness) *OO2DB* transformation. It describes the transformation of a model conforming to an object-oriented schema metamodel, as shown in Figure 2, into a model conforming to a relational database metamodel as shown in Figure 3. This transformation has been adapted from [23] and an excerpt is shown in Listing 1. The transformation contains four transformation rules:

- *Class2Table* to transform all the Classes in the object-oriented model to Tables in the database model;
- *SingleValuedAttribute2Column* to transform single-valued Attributes to Columns in the database model;
- *MultiValuedAttribute2Table* to transform multi-valued Attributes to Tables and foreign key Columns in the database model;
- *Reference2ForeignKey* to transform References in the object oriented model to foreign key Columns in the database model.

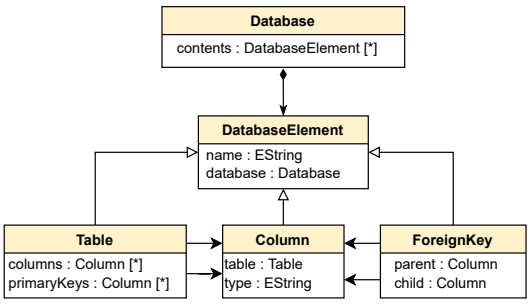

Figure 3. Database Schema Metamodel

The size of the trace of the transformation (as shown in Listing 1), which relates source to target elements in the current implementation of the ETL execution engine will be *O+M+N*, if we evaluate it over a source model containing *O* classes, *M* number attributes and *N* references.

```
1  model  Source driver EMF {
2  nsuri="oo"
3  };
4
5  model  Target driver EMF {
6  nsuri="db"
7  };
8
9  pre {
10     var db : new Target!Database;
11 }
12 rule Class2Table
13   transform c : Source!Class
14   to t : Target!Table{
15       t.name = c.name;
16       t.database = db;
17         if (c.`extends`.isDefined()){
18       var parentTable : Target!Table;
19       parentTable =c.`extends`.equivalent()
             ;
20     }
21 }
22
23 // Transforms a single-valued attribute
24 // to a column
25 rule SingleValuedAttribute2Column
26   transform a : Source!Attribute
27   to c : Target!Column {
28       guard : not a.isMany
29       c.name = a.name;
30       c.table = a.owner.equivalent();
31 }
32
33 // Transforms a multi-valued attribute
34 // to a table where its values are
35 // stored and a foreign key
36 rule MultiValuedAttribute2Table
37   transform a : Source!Attribute
38   to t : Target!Table,
39      fkCol : Target!Column {
40
41       guard : a.isMany
42       fkCol.table = a.owner.equivalent();
43       t.database = db;
44 }
45
46 // Transforms a reference into
47 // a foreign key
48 rule Reference2ForeignKey
49   transform r : Source!Reference
50   to fkCol : Target!Column {
51
52       fkCol.table = r.type.equivalent();
53
54 }
```

Listing 1. Object-oriented 2 Database Transformation

However, at a closer look, in this *OO2DB* transformation, only trace links created by the rule *Class2Table* are needed by the other rules. The remaining trace links, created by the three other rules (*SingleValuedAttribute2Column*, *MultiValuedAttribute2Table* & *Reference2ForeignKey*) are not used anywhere in the transformation and therefore establishing and keeping them in memory is wasteful. This would reduce the size of the transformation trace to *O*.

The first contribution of the paper is an approach for reducing the memory footprint of the transformation trace by selectively tracing only pairs of source-target elements that may be needed elsewhere in the transformation.

This is achieved by computing a dependency graph between rules through static analysis and storing only the traces of a rule that are later needed by another rule. To resolve *equivalent()* operations (as in Lines 18, 30 & 40 in Listing 1), the ETL engine normally triggers a lookup on a global transformation trace, ignoring the fact that the resulting target objects can only have been produced by specific rule(s), in this case the *Class2Table* rule. Hence, one possible optimisation is to benefit from static analysis, discovering which specific rule would provide the resulting target object instead.

The second contribution of this work is an approach for rewriting (desugaring) transformation programs in an imperative form, where rules are turned into operations, and calls to *equivalent/s()* are replaced with calls to appropriate transformation operations determined through static analysis.

3 Proposed Approach

```
1  model  Source driver EMF {
2  nsuri="oo"
3  };
4
5  model  Target driver EMF {
6  nsuri="db"
7  };
8
9  pre {
10   var db : new Target!Database;
11   var cache_rule_Class2Table : Map;
12
13   for (c : Source!Class in
14       Source!Class.all) {
15     c.rule_Class2Table();
16   }
17
18   for (a : Source!Attribute in
19       Source!Attribute.all) {
20    a.rule_SingleValuedAttribute2Column();
21   }
22
23 }
24
25 operation Source!Class rule_Class2Table() :
26 Target!Table {
27
28 if(cache_rule_Class2Table.containsKey(self)
       )
29  return cache_rule_Class2Table.get(self);
30 var t : Target!Table = new Target!Table;
31 t.name = self.name;
32 t.database = db;
33  if (c.`extends`.isDefined()){
34       var parentTable : Target!Table;
35       parentTable =c.`extends`.
             rule_Class2Table();
36     }
37 cache_rule_Class2Table.put(self, t);
38   return t;
39 }
40
41 operation guardSingleValuedAttribute2
42     Column(a : Source!Attribute) :
43         Boolean {
44   return not a.isMany;
45 }
46
47 operation Source!Attribute
48     rule_SingleValuedAttribute2Column() :
49     Collection {
50
51 if (guardSingleValuedAttribute2
52       Column(a)) {
53   var c : Target!Column = new Target!Column
         ;
54   c.name = self.name;
55   c.table = self.owner.rule_Class2Table();
56     return Collection{c};
57   }
58 }
```

Listing 2. Rewritten excerpt of the *OO2DB* Transformation

In this section, we discuss our proposed approach for the efficient execution of rule-based model transformation programs using static analysis and automatic program rewriting. The main goal of this approach is to reduce the execution time and memory footprint of transformations, without changing or compromising any of the language semantics. This approach is illustrated in Figure 4.

The proposed approach contains four main components, with a source metamodel, a source model, a target metamodel and a transformation being its inputs. The *Static Analyser*① component is given the model-to-model transformation and the source and the target metamodel(s), extracting the type information and yielding a type-resolved abstract syntax tree (AST). Then, using the *Dependency Graph Generator*②, we extract the dependencies between the different transformation rules in the transformation: the dependency graph uses

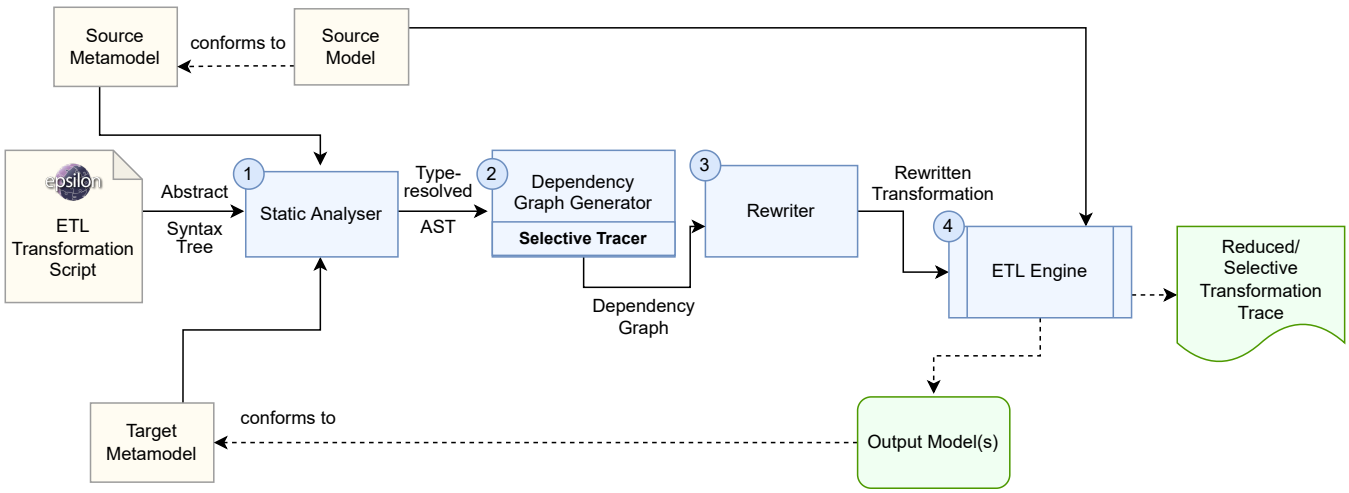

Figure 4. An overview of the proposed approach

the type-resolved AST to populate these dependencies. Following this, we have a *Selective Tracer* to selectively create hash map caches to store the result of source target key-value pairs generated by corresponding operations and use them as a minimal transformation trace. In the next step, we pass this dependency graph to the *Rewriter*③, where the transformation is rewritten, i.e., the transformation rules are converted to the corresponding operations to optimise the resolution of elements by other rules. Finally, we pass the rewritten optimised transformation to the *ETL Engine*④ for execution. ETL Engine is the default engine already provided by Epsilon. We extended the default engine to provide the resolution of operation calls to their corresponding user defined methods mapping provided by the static analyser.

3.1 Static Analysis

Figure 5. Static analysis of Epsilon

In the first step of this approach, the ETL transformation program is parsed into an Abstract Syntax Tree (AST). The static analyser yields a type-resolved AST (an AST augmented with the computed types of expressions), using metamodel introspection and type inference, as shown in Table 1 for the example of Listing 1. Epsilon programs define configuration details of the models they access using model declaration statements (Line 1-3 & Line 5-7 in Listing 1) which are then used by the static analyser to retrieve the available types and typed properties in each model. The relationship between EOL and ETL's static analyser is shown in Figure 5. This static analyser extends the EOL one by including support for analysing expressions inside transformation

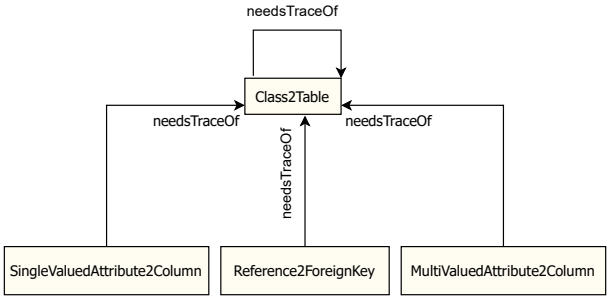

Figure 6. Dependency Graph of Listing 1

rules, their source and target parameters, and for pre and post blocks.

3.2 Dependency Graph

In an ETL transformation, resolving target elements that have been (or can be) transformed from source elements by other rules is a frequent task in the body of a transformation rule. This creates dependencies between these rules, which can be extracted from a type-resolved AST. In the body of a transformation rule say *TRx*, if there is a *equivalent(s)* statement that uses the elements transformed by another transformation rule say *TRy* as depicted in Line 9 of Algorithm 1, we can say that *TRx* is dependent on *TRy*. So, we extract such dependencies as shown in Figure 6, using static analysis, describing which transformation rule is dependent on which other transformation rules for its execution (Line 10). We depict the process of extracting such a dependency graph in Algorithm 1. For example, in Line 18 of Listing 1, there is an *equivalent()* operation. The target expression of equivalent is *a.owner*, the type of which is resolved to *Source!Class* as shown in Table 1. Then a rule whose source parameter is of the same type or a compatible type (super type) will be

Table 1. Resolved types of various constructs in Listing 1

Line#	Expression	Resolved Type
2	db	Target!Database
5	c	Source!Class
6	t	Target!Table
7	t.name	String
7	c.name	String
14	a	Source!Attribute
15	c	Target!Column
16	not a.isMany	Boolean
17	c.name	String
17	a.name	String
18	c.table	Target!Table
18	a.owner	Source!Class
25	a	Source!Attribute
26	t	Target!Table
27	fkCol	Target!Column
29	a.isMany	Boolean
30	fkCol.table	Target!Table
30	a.owner	Source!Class
31	t.database	Target!Database
32	db	Target!Database
37	r	Source!Reference
38	fkCol	Target!Column
40	fkCol.table	Target!Table
40	r.type	Source!Class

searched, which in this case is rule *Class2Table*. Hence an edge is created between *SingleValuedAttribute2Column* and *Class2Table*.

3.3 Selective Traceability

While the resolution of elements using *equivalent/equivalents* operations is explained in Section 2.3, how these equivalent statements are actually executed is defined by the Epsilon execution engine. We decided to follow the approach of completely replacing calls to *equivalent/s()* with calls to operations produced by the corresponding transformation rules. In the running example, the original transformation in line 18 of Listing 1, calls an equivalent operation, while the optimised rewritten program in Listing 2 calls the corresponding operation *rule_Class2Table*. Using the dependency graph, we create caches (*HashMaps*) as shown in Line 11 of Listing 2, which marks them as traceable. Hence, the operation cache also serves as a selective trace for the resolution elements. The cache is populated with the corresponding target element along with source elements as a key as shown in Line 33 of Listing 2. If the cache contains a source element as key then the target elements are retrieved from the cache in the body of the operation as shown in Line 28-30 of Listing 2.

3.4 Transformation Rewriting

After extracting the rule dependency graph, we rewrite the rule-based transformation program into an imperative form, as shown in Listing 2, where all rules are mapped to operations. The detailed process is presented in Algorithm 2. All rules are mapped to operations with the body of the rule mapped to the body of the operation, as in lines 17-20 for rule *Class2Table* and in lines 32-39 for rule *SingleValuedAttribute2Column* (depicted in Line 3-26 of Algorithm 2). If a rule extends other transformation rules, those rules are called in the body of the operation, by setting the source parameter of the rule as a context to the operation (Line 17). The target parameters of a rule are instantiated in the body of the corresponding operation (Line 15), and then returned from the operation (Line 18). If the target parameter is multi-valued, then the resulting values are returned in a *Collection*. If a transformation rule has a guard block (Line 13), the guard block is also mapped to a corresponding operation (Line 4), with the same body (Line 6). The source parameter of the rule is also passed as a parameter (Line 7) of the corresponding operation of the guard block. Table 2 illustrates how the expressions are executed in regular ETL and how they are rewritten using the proposed approach. *single* represents a single source model element while *collection* represents a collection of model elements. If there exists more than one matched rule the results of all the matched rules are combined and executed depending on the equivalent/equivalents call. During the rewriting process, the behaviour of these calls is preserved using operation calls as shown in Table 2.

Secondly, all these converted operations are added to the rewritten ETL transformation (Line 24). Then, we analyse the dependency graph (detailed in Section 3.2), to see if a rule needs to be traced, in which case we create a cache for the respective operation (Lines 11).

Finally, we call the mapped operations in the *pre* block of the ETL transformation (Line 30). All the operations corresponding to non-lazy, non-abstract rules are called in for loops by iterating through all instances of the source parameter of the rule (Line 27-28), setting it as a context to the operation (Line 29). At the end of this process, we remove all the original transformation rules from the ETL transformation (Line 31), as equivalent constructs are already being called as operations in the *pre* block.

3.5 ETL Engine

The rewritten transformation program is executed using a modified version of the ETL engine. This program is semantically equivalent to the original transformation but converted to imperative code, converting rules to operation calls, as discussed above. The ETL engine is the same engine used by the naive ETL, with just one modification in resolving operation

Algorithm 1 Algorithm for extracting dependency graph

```
1:  procedure extractDependencyGraph(a)
2:      Let DG = Dependency graph
3:      Let a = Transformation program
4:      for each rule in a do
5:          add rule as a vertex in DG
6:      for all rule in a.rules do
7:          for all element =elements in body of rule do
8:              if element is an OperationCallExpression then
9:                  if element.name = "equivalent" or "equivalents" then
10:                     type = resolvedType of element.target
11:                     for all r in a.rules do
12:                         if r is not abstract & (source parameter of rś type = type or is supertype of type) then
13:                             add r to rules
14:                     create an edge(s) in DG from rule to rules
15:                     replace element with the corresponding operation call of rule.
```

Algorithm 2 Algorithm for rewriting the transformation

```
1:  procedure rewrite(a)
Require: DG = Dependency graph
2:      Let a = Transformation program
3:      for all rule in a.rules do
4:          Map guard block of rule to an operation op_gd
5:          op_gd.name = operation guard_ruleName
6:          Body of op_gd <- body of guard block
7:          Param of op_gd <- Source parameter of rule
8:          Add op_gd to the ETL module
9:          Map rule to an operation op_rule
10:         op_rule.name = operation rule_ruleName
11:         Body of op_rule<- body of rule
12:         context of op_rule <- Source parameter of rule
13:         if guardBlock exists then
14:             Call op_gd as an if statement
15:         Instantiate target element(s)
16:         Add above as statement(s) to the body of op_rule
17:         Call super rules of rule
18:         Return target element(s) as a Collection
19:         Add above as a return statement in op_rule
20:         Set the type of target element of rule as a return type of op_rule.
21:         If multiple targets set return type as Collection
22:         if rule is traceable according to DG then
23:             declare a HashMap (cache_ruleName) variable in the pre block
24:             add target elements for the corresponding source element in the cache_ruleName
25:             add an if statement to search in cache_ruleName if a key with source element exists
26:         Add op_rule to the ETL module
27:     for all rule in transformation rules do
28:         if rule is not lazy or abstract then
29:             Iterate through all instances of source parameter of rule
30:             Call the corresponding operations of rule in for loop
31:             Set the iterating variable as a context of operation
32:             Add the for statements in the pre block
33:     Clear all transformation rules from ETL module
```

Table 2. Rewriting of *equivalent/equivalents()* operations

Original input expression	Resolution in ETL	Corresponding rewritten expression
single.equivalent()	Return only the first element of the target elements of matched rules	single.matched_rule().first()
single.equivalents()	Return targets of all the matched rules	single.matched_rule()
collection.equivalent()	Return a flattened collection of targets of all the matched rules	collection.collect(x|x.matched_rule()).flatten()
collection.equivalents()	Return targets of all the matched rules for all collection elements	collection.collect(x|x.matched_rule())

calls. Usually operation calls are resolved using Java's reflection API, which can be computationally expensive. Static analysis, as described in Section 3.1, other than resolving types, also monitors which operation call expressions in the program are mapped to which corresponding user defined operations. Hence, we can use this information for providing the exact matched operation, to avoid having to search for it as the program is being executed. This optimisation is not specific to ETL transformations so it can be leveraged by any Epsilon language using user defined operations.

4 Evaluation

This section presents the experimental setup used for evaluating the optimisation of model-to-model transformation programs based on static analysis, explains the methodology used and analyses the results. Finally, it discusses the limitations and possible threats to the validity of the obtained results.

4.1 Experimental Setup

We evaluate the proposed approach against the default ETL execution engine to measure its benefits in terms of execution time and memory footprint. The first contribution of the paper i.e., selective traceability, is expected to substantially improve the memory consumption, by reducing the trace size, while the second one i.e., rewriting, is similarly expected to reduce execution time. We have divided the evaluation into two parts. First, we perform a comparison of the proposed approach with the default ETL engine. Second, we compare the proposed optimisation approach with other state-of-the-art languages for M2M transformation.

The experiment referred to as "ETL" evaluates running the transformation using naive ETL without any optimisations. The "Optimised ETL" one uses our optimisation/rewriting strategy described in Section 3. We also compare our results with two other widely-used model-to-model transformation languages, ATL and YAMTL [7], to position our work in the broader context of M2M languages. For the ATL and YAMTL evaluation, we rewrote the same transformation in ATL and YAMTL and we report on the execution time and memory footprint.

4.1.1 Case Study and Models.

We have used the *OO2DB* transformation as presented in Section 2.4 for evaluating the proposed approach. We executed the *OO2DB* transformation over a set of *OO* models of increasing sizes, as shown in Table 3. These synthetic models conforming to the *OO* metamodel are created using an EOL program which can be found online[1].

Table 3. Sizes of the Object Oriented models used for benchmarking

ID	Model Name	# of model elements
1	OO_10K	140,006
2	OO_15K	210,006
3	OO_20K	280,006
4	OO_25K	350,006

4.1.2 Correctness.

The transformation is rewritten to an efficient form behind the scenes, so it is crucial to ensure that the rewritten transformation is semantically equivalent to the original input program. To gain confidence that our rewritten program is correct, the generated output model(s) should be the same as the ones generated by the original transformation. Using EMFCompare [2], we can check that the output models for ETL and Optimised ETL are the same. For ensuring broad coverage of our tests, we executed seven test ETL scripts mined from GitHub and for all cases we found no differences in the outputs given by the original and the rewritten programs. For this test case, the object oriented to relational database transformation, we matched the generated output models for ATL, ETL, Optimised ETL and YAMTL. After executing these equivalence tests, we are confident of the semantic equivalence of the rewritten transformation and hence of the optimised and selective traceability used in this approach.

4.1.3 Machine Specification.

The benchmark experiments were conducted on a machine with the following specifications: MacBookPro @ 2.8 GHz Quad-Core Intel Core i7, 16

[1]URL suppressed for the reviewing process.

GBs of RAM, macOS Big Sur version 11.1 with JVM MaxHeapSize 4GBs.

4.2 Internal Evaluation

Table 4. Execution time of naive and optimised ETL, in ms

Model size	Execution engine	
	ETL	Optimised ETL
10K	5,899	3,519
15K	9,623	6,423
20K	16,040	10,171
25K	21,836	14,641

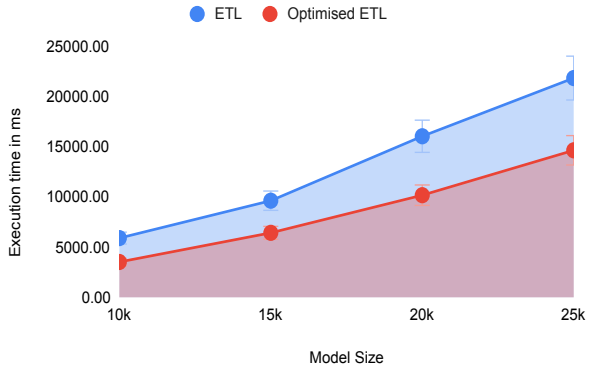

Figure 7. Execution time comparison of Optimised ETL with ETL

The results of the execution time of the naive ETL vs Optimised ETL are reported in Table 4 and visualised in Figure 7.

We can observe that overall ETL's execution time is significantly improved in 'Optimised ETL' version. This is because of the optimisations provided by static analysis and program rewriting to avoid operation call resolutions at runtime and also because of efficient resolution of equivalents before the execution.

As the proposed optimisation approach relies on extracting information (such as static analysis, extracting of dependency graph), it is necessary to compute the incurred overhead of these processes. Static analysis took an average of 50ms, extracting the dependency graph took an average of 35ms, while optimisation and rewriting took 2ms on average for all the experiments. It is worth noting that the size of the models does not affect the time needed to extract this information, as all these steps are performed at the metamodel level, before executing the program itself.

The memory use for the naive ETL and optimised ETL can be seen in Table 5, where we can observe that the Optimised ETL consumes less memory compared to ETL as shown in Figure 8 due to the reduced (selective) transformation trace provided by the selective traceability mechanism we discussed in this paper.

Table 5. Memory consumption of naive and optimised ETL, in MBs

Model size	Execution engine	
	ETL	Optimised ETL
10K	83	30
15K	128	51
20K	175	69
25K	216	85

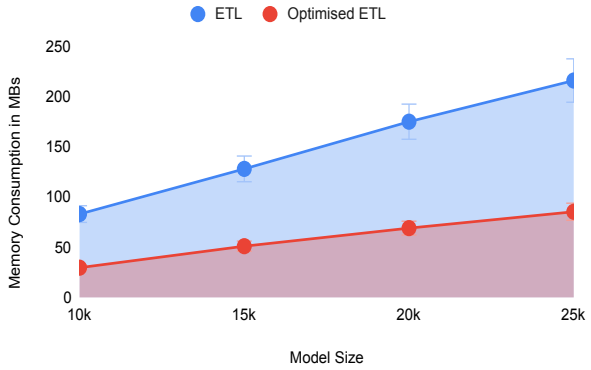

Figure 8. Memory consumption comparison of Optimised ETL with ETL

4.3 External Evaluation

In the MDE ecosystem task specific languages are typically interpreted. But compiled languages can also be used to perform the same tasks and can be faster compared to the interpreted ones, but they are more verbose and less amenable to static analysis. In this section, we will discuss other M2M languages used in the MDE community: ATL, YAMTL and the A2L compiler. We compare our approach with ATL and YAMTL because ATL is a widely used interpreted language, YAMTL is a compiled language that builds on top of Xtend and is state-of-the-art for large-scale model transformations. A2L is a compiler developed for the ATL language that compiles ATL transformations into Java code.

In Tables 6 and 7, we present the results of execution time and memory consumption of our proposed approach in comparison with ATL & YAMTL, respectively (depicted in Figure 9 and 10). We can clearly see YAMTL executes faster than the others, because YAMTL is compiled to Java, while ETL and ATL are interpreted.

On the other hand, YAMTL consumes the most memory, while ATL consumes the least. The excessive memory consumption of YAMTL is explained by the fact it supports incremental execution and hence consumes more memory due to the caching required. We do not present the results of the experiments compared to A2L as we attempted to compile the *OO2DB* case study that we used in the experiments above, but due to certain limitations of A2L (which are discussed below), compilation failed.

Table 6. Execution time of various transformation languages, in ms

Model size	Transformation Language		
	Optimised ETL	YAMTL	ATL
10K	3,519	1,318	10,355
15K	6,423	1,806	15,202
20K	10,171	2,631	84,726
25K	14,641	3,200	103,596

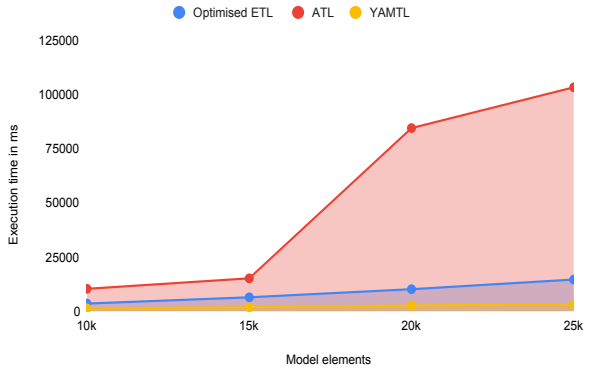

Figure 9. Execution time comparison of Optimised ETL with ATL and YAMTL

While A2L provides a considerable speed up by generating efficient Java code from ATL transformations, still there are certain limitations: Our proposed approach does not limit or change any of the language semantics of an ETL transformation, thus supporting features such as 1) Rule Inheritance 2) Global variables 3) Reflective operations, unlike A2L. We performed preliminary work on optimising builtin operation calls in ETL, but realised that it would have to either limit certain features provided by Epsilon in order to ensure correctness, or require additional constructs (or annotations) to be added to ETL programs. Moreover, the *OO2DB* case study could not be compiled using A2L due to use of a global variable (Line 2 of Listing 1). So, in cases like this, using our approach achieves a significant speedup, without having to alter the original transformation.

Table 7. Memory consumption of various transformation languages, in MBs

Model size	Transformation Language		
	Optimised ETL	YAMTL	ATL
10K	30	131	23
15K	51	200	32
20K	69	267	44
25K	85	337	54

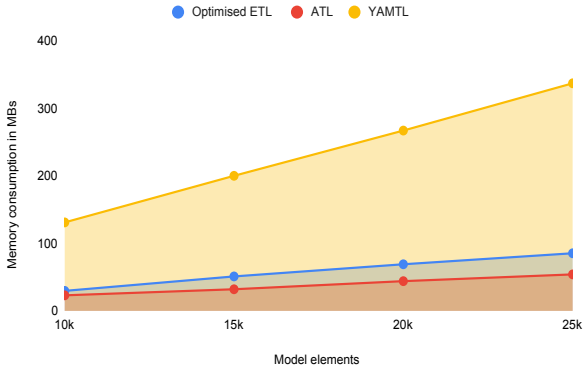

Figure 10. Memory consumption comparison of Optimised ETL with ATL and YAMTL

4.4 Threats to Validity

This experiment uses two metamodels :*OO* metamodel and *DB* metamodel and a set of increasingly large synthetic models conforming to the source *OO* metamodel. Both the metamodels and the transformation were not specifically targeted but were chosen for two reasons. The first was using a well-known transformation, predating this work, as well as the metamodels exercising all core features of Ecore like inheritance, attributes, containment and non-containment references. The second was the generality and ease of understanding of both, as they are in our view generic enough to understand and hence demonstrate the novel work presented in this paper. Nevertheless, we realise that they may play a significant role in determining the results obtained. Hence, we cannot claim that the results obtained are generalisable for every type of transformation and model. The proposed transformation optimisation approach can benefit from experiments performed on more diverse models with a broader range of sizes and more complex transformations, both for investigating semantic equivalence and performance gains.

Another possible threat to the validity of these results is the addition of possibly substantial overheads of this approach when evaluating large enough programs or metamodels: for example, if the selective trace ends up being

almost equal in size to the entire transformation trace (e.g. due to a fully connected dependency graph).

Finally, as the optimisation approach leverages the benefit of information extracted through static analysis, it is crucial to have an accurate static analysis of the transformation. To ensure more complete static analysis information and thus enable efficient program rewriting, we recommend using a more strict coding style, explicitly declaring types where possible, and avoiding *Any* type unless necessary, for more accurate type resolution.

5 Related Work

This section summarises the state-of-the-art within the scope of this article, divided into two main lines of work: Firstly, it lists existing tools that provide static analysis facilities for model management languages, particularly for model transformations; and secondly, it discusses model transformation optimisation strategies used for improving the performance of the engines executing such transformations.

AnATLyzer [13] is a static analysis tool for the ATLAS Transformation Language (ATL) transformations, that provides type checking, problem reporting and quick fixes facilities. It ensures that the transformation is correctly typed according to the source metamodel and identifies any conflicting or missing rules.

In [10], Born et al. extend Henshin, a rule-based model transformation language, adapting graph transformation concepts based on EMF. This extension computes all potential conflicts and dependencies for a set of rules and reports them in the form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency.

In [26], Ujhelyi introduces a static analysis facility for graph transformations. This work uses Constraint Satisfaction Programming (CSP) to provide a type checker for the Viatra2 framework. This type checker is based on CSP, and is not guaranteed to find all the errors in a single run using static analysis.

Static analysis of OCL is presented in [30], where a pseudo-type *OCLSelf* is introduced to infer the type of built-in operations such as *oclAsSet()* and *oclType()*. Willink [31] introduced safe navigation operators in OCL. These operators solve the problem of declaring non-null objects and null-free collections and enable OCL navigation to be fully checked for null safety.

In [25], the A2L compiler is introduced for parallel execution of ATL transformations. It uses static analysis through ATLyzer (discussed above), to generate efficient code at the transformation level. A2L was discussed earlier in Section 4.3.

Static analysis has been used for enabling the translation from EOL to SQL [9], for optimisation of programs over EMF models [27] and for enabling the translation from EOL to Viatra patterns [8].

Gremlin-ATL is another approach presented in [14]. It is a model-to-model transformation framework that translates ATL transformations into Gremlin, a query language supported by several NoSQL databases.

Another model-to-model transformation language, YAMTL, was introduced in [**?**]. YAMTL provides an efficient engine to transform EMF-based models with transformations defined in the internal DSL of Xtend. Support for incremental transformations was also added in [11] using the forward change propagation mechanism.

Several approaches and languages are available for incremental model-to-model transformations, such as the Tefkat tool, by Hearnden et al. in [16]. Here, changes to the source models are directly mapped to their effects on transformation execution, allowing modifications to target models to be computed efficiently.

To summarise, the approach presented in this paper takes the benefit of static analysis to reduce the transformation trace while not sacrificing the language (ETL) expressiveness by compiling it down to a general-purpose programming language such as Java.

6 Conclusions & Further work

In this paper, we presented an approach used to optimise programs written in rule-based M2M transformation languages. The proposed approach resolves the types of various constructs using static analysis and then creates a dependency graph between the transformation rules. Based on this dependency graph, the rule-based transformation program is rewritten to an imperative program that only maintains a selective trace. Our evaluation experiments have demonstrated that the proposed approach can deliver significant performance benefits both in terms of execution time and memory footprint compared to the default ETL execution engine, particularly where larger models are involved.

Directions for future work include conducting experiments to evaluate the proposed approach with other modelling technologies (e.g. Simulink models, repository-based models). Also, providing a disposal facility for the transformation trace can offer further memory footprint reductions. Moreover, using program analysis for detecting additional optimisation opportunities at the expression level is an interesting direction for potentially further improving the performance of such model-to-model transformations.

Acknowledgments

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

References

[1] 2022. A2L GitHub repository. https://github.com/anatlyzer/a2l.git. [Online; accessed 29-April-2022].

[2] 2022. EMF Compare. https://www.eclipse.org/emf/compare/. [Online; accessed 29-April-2022].

[3] 2022. Epsilon. https://www.eclipse.org/epsilon/. [Online; accessed 29-April-2022].

[4] 2022. Epsilon Model Connectivity Layer. https://www.eclipse.org/epsilon/doc/emc/. [Online; accessed 29-April-2022].

[5] 2022. Epsilon Validation Language. https://www.eclipse.org/epsilon/doc/evl/. [Online; accessed 29-April-2022].

[6] 2022. Object Constraint Language. https://www.omg.org/spec/OCL/2.4/About-OCL/. [Online; accessed 29-April-2022].

[7] 2022. Yet Another Model Transformation Language. https://yamtl.github.io. [Online; accessed 29-April-2022].

[8] 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

[9] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. 2020. *Efficiently Querying Large-Scale Heterogeneous Models.* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3417990.3420207

[10] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. 2015. Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–168.

[11] Artur Boronat. 2021. Incremental execution of rule-based model transformation. *International Journal on Software Tools for Technology Transfer* 23, 3 (2021), 289–311.

[12] Marco Brambilla, Jordi Cabot, Manuel Wimmer, and Luciano Baresi. 2017. . Morgan & Claypool. https://ieeexplore.ieee.org/document/7899157

[13] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. AnATLyzer: An Advanced IDE for ATL Model Transformations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 85–88. https://doi.org/10.1145/3183440.3183479

[14] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. 2017. Gremlin-ATL: a scalable model transformation framework. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 462–472.

[15] Raffaela Groner, Katharina Juhnke, Stefan Höppner, Matthias Tichy, Steffen Becker, Vijayshree Vijayshree, and Sebastian Frank. 2022. A Survey on the Relevance of the Performance of Model Transformations. *Software Engineering 2022* (2022).

[16] David Hearnden, Michael Lawley, and Kerry Raymond. 2006. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems* (Genova, Italy) *(MoDELS'06)*. Springer-Verlag, Berlin, Heidelberg, 321–335. https://doi.org/10.1007/11880240_23

[17] Stefan Höppner, Timo Kehrer, and Matthias Tichy. 2022. Contrasting Dedicated Model Transformation Languages versus General Purpose Languages: A Historical Perspective on ATL versus Java Based on Complexity and Size. *Softw. Syst. Model.* 21, 2 (apr 2022), 805–837. https://doi.org/10.1007/s10270-021-00937-3

[18] John Hutchinson, Mark Rouncefield, and Jon Whittle. 2011. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*. 633–642.

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

[20] 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.

[21] 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.

[22] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*. Springer, 46–60.

[23] Michael Lawley, Keith Duddy, Anna Gerber, and Kerry Raymond. 2004. Language features for re-use and maintainability of MDA transformations. In *Workshop on Best Practices for Model-Driven Software Development*.

[24] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. 2016. Scalability of Model Transformations: Position Paper and Benchmark Set.. In *BigMDE@ STAF*. 21–30.

[25] Jesús Sánchez Cuadrado, Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. 2020. Efficient execution of ATL model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering* PP (07 2020), 1–1. https://doi.org/10.1109/TSE.2020.3011388

[26] Zoltán Ujhelyi. 2009. *STATIC ANALYSIS OF MODEL TRANSFORMATIONS.* Master's thesis. Budapest University of Technology and Economics.

[27] 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

[28] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Softw. Syst. Model.* 15, 3 (jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

[29] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2013. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2013), 79–85.

[30] Edward D. Willink. 2011. Modeling the OCL Standard Library. *ECEASST* 44 (2011). https://doi.org/10.14279/tuj.eceasst.44.663

[31] Edward D. Willink. 2015. Safe Navigation in OCL. In *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, Vol. 1512. 81–88.