Efficient execution of ATL model transformations using static analysis and parallelism

Jesús Sánchez Cuadrado, Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo

Abstract—Although model transformations are considered to be the heart and soul of Model Driven Engineering (MDE), there are still several challenges that need to be addressed to unleash their full potential in industrial settings. Among other shortcomings, their performance and scalability remain unsatisfactory for dealing with large models, making their wide adoption difficult in practice. This paper presents A2L, a compiler for the parallel execution of ATL model transformations, which produces efficient code that can use existing multicore computer architectures, and applies effective optimizations at the transformation level using static analysis. We have evaluated its performance in both sequential and multi-threaded modes obtaining significant speedups with respect to current ATL implementations. In particular, we obtain speedups between 2.32x and 38.28x for the A2L sequential version, and between 2.40x and 245.83x when A2L is executed in parallel, with expected average speedups of 8.59x and 22.42x, respectively.

Index Terms—Model transformation, MDE, ATL, Performance, Scalability, Parallelization

1 Introduction

The progressive adoption of Model-Driven Engineering (MDE) [1] approaches for developing better and more efficient software is posing different kinds of challenges to current MDE methods and tools. Despite the potential benefits of MDE technologies to significantly reduce time to market and improve product quality, they still suffer from some limitations that may hinder their full adoption by industry (see, e.g., [2]–[4]). In particular, the scalability, usability and performance of model transformations (MT) are crucial issues that need to be tackled if they are to be effectively used to address scenarios such as model-driven modernization of legacy systems and the engineering of large and complex applications in, e.g., the automotive, biology or aerospace domains.

At this moment, ATL [5] and QVT [6] are the most widely-used model transformation languages [7]. Although they provide powerful abstractions to specify and implement transformations between models and to generate model views, their implementations have limited scalability, and thus the execution time of transformations may become prohibitive with large input models (e.g., in the order of millions of elements), or even medium-size input models if the transformation has complex model navigations. One reason for this lack of scalability is due to the fact that most transformation engines are implemented as simple interpreters and they barely use static analysis information to apply compile time optimizations or to improve their scheduling. Moreover, although multicore computers are widely available, there are very few engines that implement parallel transformation algorithms.

The contribution presented in this paper addresses the engineering of an efficient model transformation engine for the particular case of the ATL model transformation language. We have developed a new compiler for ATL, called A2L, which provides several novel features with respect to state-of-the-art approaches, namely:

- A2L uses static analysis information provided by AnATLyzer [8] to compile ATL transformations to the Java Virtual Machine (JVM), applying optimizations for OCL expressions and for transformation rule handling.
- We present a novel algorithm which enables the parallel execution of the transformation, using data parallelism. This allows A2L to achieve an effective distribution of the parallel jobs, thus outperforming other parallel ATL engines which are based on task parallelism [9], [10].
- A2L is integrated with the ATL/AnATLyzer IDE and Eclipse Java Development Toolkit (JDT), which enables the development of transformations using the facilities provided by AnATLyzer, e.g., quick fixes [11] and visualizations [12]. Moreover, the compiled code can be seamlessly integrated with existing Java code.

A2L has been validated for correctness using the regression tests defined for the ATL virtual machine [13] and supports the majority of the constructs of ATL, including all types of rules (matched, lazy and called), module and context helpers, imperative blocks, all datatypes including collections, maps and tuples, and the standard OCL library. We have run several benchmarks that show significant performance improvements when compared with the existing ATL engines.

This paper is organized as follows. Sect. 2 introduces the ATL model transformation language and describes the limi-

- *Jesús Sánchez Cuadrado is with the Universidad de Murcia, Dept. Informática y Sistemas. Campus de Espinardo, Murcia, Spain E-mail: jesusc@um.es*
- *Loli Burgueño is with the Open University of Catalonia, IN3, Barcelona, Spain, and Institut LIST, CEA, Université Paris-Saclay, París, France Email: lburguenoc@uoc.edu*
- *Antonio Vallecillo is with ITIS Software, Universidad de Málaga, Bulevar Louis Pasteur, 35, 29071, Malaga, Spain E-mail: av@lcc.uma.es*
- *Manuel Wimmer is with the Johannes Kepler Universität, Business Informatics – Software Engineering, Linz, Austria E-mail: manuel.wimmer@jku.at*

tations of current transformation engines through a running example. Then, Sect. 3 describes the architecture of A2L, the compiler we have developed to compile and execute in parallel existing ATL programs. The more prominent features of A2L are described in Sections 4 and 5, which present, respectively, the algorithm used to execute ATL transformations in parallel, and the A2L optimization strategies and mechanisms enabled by the use of AnATLyzer static typing information. Sect. 6 describes the evaluation that we have conducted to validate our proposal. Finally, Sect. 7 discusses related work, and Sect. 8 concludes with an outline on future work.

2 Motivation and background

Performance and scalability of model transformations is deemed as one of the most important challenges in MDE since it enables the use of model transformation technology to handle large models appearing in scenarios like reverse engineering, model analysis, or data engineering. It is also key to apply MDE to other engineering disciplines, such as construction [14] or automotive engineering [15].

Model transformation languages, notably those with a declarative form, have the potential to tackle this challenge because they provide an abstraction to write transformations which are independent of the execution mechanism. A good compiler should generate e cient code by analysing the structure and relationships of the transformation. However, this possibility has not been exploited in state-of-the-art MT languages, resulting in poor performance. In fact, a recent study [7] has revealed that, although MT users value the advantages of using MT languages, the poor performance and scalability issues of MT engines are hampering their use and forcing them to develop their transformations in general-purpose languages (even though if it makes the task more cumbersome and error-prone).

Our working hypothesis is twofold. First, by using static analysis information, it is possible to compile declarative transformations to produce high-performance code; moreover, recurrent transformation idioms can be optimized by the compiler. Second, since a declarative transformation does not prescribe the execution order, it is possible to seamlessly execute a transformation in parallel, if the adequate transformation algorithm is implemented. More precisely, to achieve e cient parallel execution of ATL programs, such an algorithm should limit the number of dependencies between parallel processes to maximize concurrency, while load balancing between processes should aim at preventing processes from becoming idle if they nish before others. To this end, we propose the use of data-based parallelism, whereby the model is split into chunks of elements that are transformed by the processes, all running the complete transformation in parallel.

This paper describes our proposed parallelization algorithm, its main features and characteristics, the optimizations we have applied, and the performance gains it achieves over existing ATL model transformation engines.

2.1 ATL

ATL [5] is a hybrid model transformation language that allows both declarative and imperative constructs. A transformation consists of a set of rules that speci es which elements of the output model are created from which ones of the input model.

Fig. 1. Excerpt of the Java metamodel (MoDisco).

Fig. 2. Excerpt of the UML metamodel.

Listing 1 shows an excerpt of an ATL transformation taken from the ARTIST project [16] that generates a UML class diagram (a dependency view) from a Java project. Excerpts from the input and output meta-models of this transformation are depicted in Fig. 1 and 2, respectively. In ATL, the main type of rule is the so-called matched rule. It consists of an input pattern that might have a lter condition which is matched on the source model, and an output pattern that produces a set of elements in the target model for each matched input pattern. OCL expressions [17] are used to calculate the values of features of the target elements. In this excerpt, we have included two matched rules, Package2Package and Class2Class , which take package and class elements respectively from the Java model and convert them to the corresponding counterparts in the UML model, but ltering proxies out (not s1.proxy). A rule body consists of binding elements. A binding either assigns a primitive value (e.g., name s1.name) or resolves the source values appearing in its right-hand side (RHS) to target values generated by other rules. For example, the binding in line 15 retrieves and assigns the subpackages mapped by rule Package2Package, and the binding in line 16 retrieves and assigns all non-proxy classes mapped by rule Class2Class .

The ATL transformation algorithm works in two phases, which are graphically illustrated in Fig. 3. The left-hand side shows a sample input model. In the rst phase each matched rule accesses the input model to get all elements whose type is compatible with its input element (speci ed in the from part of the rules). The lter is used to rule elements out. In the example, elements jp1 and jp3 are retrieved by the rule Package2Package but only jp1 satis es the lter and is matched. When an element is matched, the target elements speci ed in the to part of the rules are created and a traceability link is established (depicted by a dashed arrow in the image) which includes a reference to the rule producing the link. The second phase of the algorithm consists of traversing all traceability links and resolving each rule binding in order. To resolve a binding, its OCL expression in the RHS

Listing 1. Excerpt of the Java2UML transformation.

```
1  -- @nsURI UML=http://www.eclipse.org/uml2/3.0.0/UML
2  -- @nsURI JAVA=http://www.eclipse.org/MoDisco/Java/0.2.incubation/java
3  module java2uml;
4  create OUT : UML from IN : JAVA;
5
6  helper context JAVA!Package def: nonProxyClasses : Sequence(JAVA!ClassDeclaration) =
7    self.ownedElements-> select (e | not e.proxy)-> select (e | e.oclIsTypeOf(JAVA!ClassDeclaration));
8
9  helper context JMM!ClassDeclaration def : getRefClassFields : Sequence(JMM!FieldDeclaration) = ...
10
11 rule Package2Package {
12   from p : JAVA!Package (not p.proxy)
13   to t : UML!Package(
14     name <- p.name,
15     packagedElement <- p.ownedPackages->select (e | not e.proxy)-> select (e | e.oclIsTypeOf(JAVA!Package)),
16     packagedElement <- p.nonProxyClasses,
17     packagedElement <- p.nonProxyClasses
18         ->select (p2 | not p2.getSuperClass.oclIsUndefined() )
19         ->collect(p2 | thisModule.createGeneralizationDependency(p2)),
20     packagedElement <- p.nonProxyClasses
21         -> collect(p2 | p2.getRefClassFields)->flatten()->collect(e|thisModule.createUsageDependency(e)) )
22 }
23 rule Class2Class {
24   from c : JAVA!ClassDeclaration (not c.proxy)
25   to t : UML!Class ( name <- c.name )
26 }
27 lazy rule createGeneralizationDependency {
28    from class : JAVA!ClassDeclaration
29    to d : UML!Dependency (
30      supplier <- Sequence { class.superClass.type },
31      client <- Sequence { class }
32    )
33 }
34 lazy rule createUsageDependency {
35    from field : JAVA!FieldDeclaration
36    to d : UML!Dependency (
37      supplier <- Sequence { field.type.type },
38      client <- JAVA!ClassDeclaration.allInstances()->      select (cd | cd.bodyDeclarations->includes(field))
39     )
40 }
```

Fig. 3. Representation of a sample transformation execution.

is evaluated. If the result is a primitive type, the value is directly assigned to the feature in the left hand side. If it is an object (or a collection of objects), the internal trace is looked up to retrieve the corresponding target element and it is assigned to the left hand side (if it is a collection, the value is added). In the example, to resolve the binding in line 16, the engine retrieves and assigns target objects c1 and c2 from source objects jc1 and jc2 respectively.

ATL also supports rules which must be explicitly invoked. This is the case of lazy rules. A lazy rule can be seen as a global function that takes model elements as parameters and returns a target model element, which is created and initialized by the rule. In the example, the lazy rule createUsageDependency (line 38) de nes a dependency between the class that de nes a eld and the eld type. In line 21, the rule is invoked. Since the lazy rule generates the target element, it can be assigned directly in the corresponding binding (i.e., no binding resolution is needed).

The ATL code is compiled into bytecode for its execution using two main runtime engines: the default ATL virtual machine [5], which was released along with the ATL language; and EMFTVM [18], which provides performance improvements as well as other advanced language features such as the possibility to execute in-place model transformations.

2.2 Static analysis of model transformations: AnATLyzer

A model transformation is typed against its input and output meta-models. This means that the types and features used in the transformation program must exist in the corresponding meta-model. This can be enforced dynamically (at runtime) or statically (at compilation time). Some languages, like QVT-Operational, enforce this statically, while others, like ATL and Epsilon ETL [19], do it dynamically. In addition, data dependencies between the input model and the rules which match the elements, as well as among the transformation rules (i.e., established by means of bindings in the case of ATL), may exist. In this work, we have used AnATLyzer [8], [12] to

statically analyse ATL transformations and bene t from this information to implement our compiler.

Fig. 4 illustrates some of the static analysis information made available by AnATLyzer . First, every node of the abstract syntax tree of the transformation is annotated with its type (i.e., a reference to the corresponding meta-model element). There are three bindings to initialize packagedElement, whose semantics is to add elements (i.e., the second binding does not override the previous setting because it is a collection). For example, the type of the rst binding is Sequence(JAVA!Package) because AnATLyzer recognizes the use of oclIsTypeOf and performs an implicit casting. From the inferred types, AnATLyzer builds a graph to make dependencies among rules explicit. For instance, the rst binding packagedElement can only be resolved by rule Package2Package because of the implicit casting that determines that the RHS of the binding will only have JAVA!Package elements. The second binding is resolved by the rule Class2Class because the return type of nonProxyClasses is Sequence(JAVA!ClassDeclaration) . Finally, the third binding does not need to be resolved because it directly assigns target elements generated by the createUsageDependency rule.

Fig. 4. Static analysis of the example transformation.

2.3 Limitations of current approaches

The original ATL transformation algorithm, based on the two phases described above, and its implementations (both in the standard ATL Virtual Machine (VM) [5] and EMFTVM [18]), can cope with scenarios involving small or medium-size models. However, their performance and scalability rapidly degrade as the size of the input models grows. Among other reasons, they fail to exploit a variety of interesting performance and optimization opportunities, which are described next.

Limited parallelism . The algorithm and its current implementations are sequential. A relatively simple approach to make the algorithm parallel is to use task parallelism [10], in which the parallelisation unit is the transformation rule. However, this approach is sub-optimal since it su ers from lock contention and unbalanced loads (i.e., some threads will be idle if they nish their tasks earlier than others). Using a data parallelism approach, all processes perform the same task, but on di erent chunks of data|it is the data that is split. Contrarily, in task parallelism, it is the model transformation that is split into separate smaller processes (e.g., a rule) and all of them work on the same data. As demonstrated in [20], using data parallelism to implement concurrent model transformations can produce signi cantly better results.

Fig. 5. Compiler architecture.

Ine cient model access . A transformation engine which does not exploit type information (such as the existing ATL virtual machines) does a \blind access" to the input model. This means that after loading the input, it cannot discard unused parts of the model. For instance, in the example (see Fig. 3), objects of types JAVA!Field and JAVA!Method will never be matched by a rule, and thus, they could be ruled out in the loading phase. We will improve rule matching by considering, in a pre-processing step, only those elements that are relevant for the transformation.

Expensive runtime checks . The ATL compiler does not perform any type checking, which means that it needs to insert code to perform dynamic checks, including, e.g., the cardinality of the LHS of the binding, or calls to helper methods. Moreover, it can only use the re ective EMF API, which also imposes an additional overhead. We will use the information made available by the type checker to avoid these kinds of overheads.

Lack of OCL optimizations . Complex transformations typically contain many OCL expressions and operation helpers. These expressions are often devoted to navigating collections. A good implementation of OCL is critical to achieve a satisfactory performance on large models|especially when collection operations are involved. It has already been reported that the standard ATL VM does not handle large collections e ciently [21] and it is the EMFTVM engine which does provide a better implementation. However, both engines have not addressed optimizations yet. For instance, the expression

```
s1.nonProxyClasses->collect(p2| p2.getRefClassFields)
    ->flatten()->collect(e|thisModule.
    createUsageDependency(e))
```

requires two intermediate collections to be created (for the rst collect and the flatten). An optimizer could identify this pattern and evaluate the expression without creating unnecessary intermediate collections.

To the best of our knowledge, there is no transformation engine that makes use of static analysis information to improve its performance, and combines this with parallelism to take advantage of all the computing power of current CPUs.

3 A2L: A Compiler for ATL

Our technical approach to address the limitations presented above is based on a compiler from ATL to Java. Fig. 5 shows its architecture.

First, it performs static analyses using AnATLyzer . This produces an extended ATL abstract syntax model

(AST), which includes type information, control flow and data flow information. This will be used throughout the compilation process.

The optimization phase (described in detail later in Sect. 5) is in charge of detecting common transformation patterns that can be particularly managed to generate efficient code. To implement the optimizations, special nodes are added to the AST. Each of these nodes represents an optimization pattern. The optimizer is in charge of detecting the patterns and replacing the AST nodes. Then, the compiler has specific extensions to produce specific code for these nodes. This optimization phase is optional and can be disabled.

The compilation phase generates all the code and related artefacts needed to execute the transformation in the JVM. An essential feature of A2L is that it targets a transformation algorithm specifically designed to execute the model transformations concurrently, using data parallelism to achieve adequate performance results (the algorithm is described in detail later in Sect. 4, and the gains in performance are presented in Sect. 6).

As a result of these three steps, the compiler generates four main artefacts (Fig. 5). The Runner allows the user to configure the transformation execution programatically (e.g., to set the input models, to configure the number of threads, etc.). The Pre-processor is in charge of filtering the input models to optimize the rule matching by considering only those elements required by the transformation rules. The Transformation contains the actual transformation behavior which will be executed by the parallel processes. Finally, the Post-processor is in charge of combining the results of all the processes that have been working in parallel to realize the transformation, and to generate the output models.

The following sections describe in more detail these features of the A2L compiler. We begin explaining our parallel transformation algorithm. Then, we explain the optimization strategies and mechanisms to implement them.

4 Parallel execution of ATL transformations

To address the lack of parallelism of ATL we have designed a new transformation algorithm. The algorithm is intended to respect the semantics of the original one but, in addition, it enables data-based parallelism and focuses on minimizing the amount of lock contention among the worker threads.

In data-based parallelism, all threads execute the same code but on different chunks of data. The advantage over task-based parallelism, as proposed in [10] for ATL, is that processors are less prone to be idle. We have redesigned the ATL transformation algorithm to make it amenable to data-based parallelism, generalizing the approach proposed in [20]. Our algorithm is presented in Algorithm 1. It works in three phases, pre-processing (line 2), execution (line 5) and post-processing (line 11). The architecture to execute these phases in parallel is illustrated in Fig. 6 and it is described next.

```
1  def transform(model, transformation) :
2      // Step 1: Pre-processing
3      types = footprint(transformation)
4      buffer = preprocess(model, types)
5      // Step 2: Parallel execution
6      // This loop does sequential execution,
7      // parallel execution requires assigning jobs to threads
8      foreach e in buffer do
9          execute(element)
10     end
11     // Step 3: Post-processing
12     foreach b in pendingBinding do
13         resolve(b)
14     end
15 end
16 def execute(element) :
17     foreach rule in transformation.rules do
18         if rule.filter(element) then
19             executeRule(rule, element)
20             break
21     end
22 end
23 def executeRule(rule, element) :
24     foreach type in rule.outputElements do
25         target = createObject(type)
26         create trace link (element, target)
27     end
28     foreach binding in rule.bindings do
29         right = evaluate(binding) if binding is primitive then
30             target."binding.feature" = right
31         else
32             // Resolve the binding
33             foreach resolving in binding.resolvingRules do
34                 if resolving.filter(element) then
35                     add to pendingRules (target, binding, right)
36                     break
37             end
38         end
39     end
40 end
```

Algorithm 1: Data-oriented ATL algorithm.

Pre-processing. The input model is read from some source (label 1). Its elements are placed in a buffer which will be used by worker threads in the next phase when fetching work. However, not all model elements are required by all worker threads, only those whose type is declared by the matched rules in their source patterns. Thus, this set of types is extracted from the static analysis of the transformation, and used to filter the source model (lines 3-4). In the transformation example, this set consists of Package and ClassDeclaration types. The intended effect is to reduce the size of the buffer and to speed up rule matching, since there are less elements to consider. Although this step is done sequentially, the actual overhead due to the filtering is small since the engine needs to load and prepare the input model in any case.

Execution. This phase is in charge of executing the transformation logic. In the sequential version each element in the buffer is processed one after another. In the parallel version, we spawn worker threads (label 2). A worker obtains a chunk of data from the buffer (label 4), which is split into chunks of a given size (e.g., 512 model elements). Each worker has a counter to represent the chunk that is currently transforming. When it finishes, it asks for the next chunk to the scheduler, which uses an atomic integer variable to represent the last chunk given to a worker. The scheduler uses an atomic operation to increment the counter (label 3), which means that the increment operation for the chunk counter does not need to be guarded by a lock because it is done atomically. This way, there are no locks involved in the algorithm and we expect less contention. There are several possible strategies to split to decide the chunk size. The simplest one is to use a fixed chunk size. The larger the chunk size the less competition to get more work. However, it may happen that some threads are idle at the end of the transformation execution (i.e., load imbalance). On the contrary, setting a small chunk size would lead to more contention. The alternative is a dynamic scheduling policy in which chunks are larger at the beginning and smaller towards

Fig. 6. Components of the technical realization of the parallel transformation algorithm.

the end, but this implies some overhead. Therefore, we have implemented a mixed strategy in which we split the bu er in two parts. The rst one is statically divided into chunks of size 0:75 bu er _size=num _workers, which are large enough for each worker to start transforming elements. As soon as these chunks are nished, the algorithm continues with a dynamic scheduling strategy that uses smaller chunks (10 elements in the current con guration) to prevent workers from being idle.

Each worker uses a new instance of the transformation, given that we are using data-based parallelism, which has its own local state so that threads do not compete to access shared resources (label 6). For each element of a chunk, we try to nd a matching rule. In practice, rule matching consists of checking the rules of the transformation in some order. Actually, in ATL the order is irrelevant because a given source element can only be matched by one rule, otherwise a runtime error is raised. Algorithm 2 illustrates the style of the code generated for matching the rules of the running example.

```
1 def transform(element) :
2   if model2model _match(element) then
3     model2model _execute(element)
4   else if package2package _match(element) then
5     package2package _execute(element)
6   else if class2class _match(element) then
7     class2class _execute(element)
8   end
9 end
```

Algorithm 2: Example of rule matching

If a matching rule is found, the output elements are created and a trace record is generated to establish a mapping between the input and output elements. Thus, each transformation instance contains a partial trace (LocalTrace in the gure) to store such records locally to avoid any locking situation. In addition, the RHS of each binding is evaluated. Primitive bindings are assigned directly, but non-primitive bindings (i.e., those whose RHS evaluation returns a set of model elements) cannot be evaluated because the corresponding target models may not have been transformed yet. Thus, we delay this task by recording the fact that such dependency has yet to be resolved (PendingBindings in Fig. 6). This approach is a generalization of [20] in which special identi ers are used to construct model references which are yet to be resolved. The advantage in this case is that it is independent of the meta-modelling framework and the subsequent resolution does not require to traverse the full target model, but can be done by a constant-time look up.

Post-processing . The main task of this phase is to resolve bindings, given that now all target model elements are available. For each unresolved binding its RHS is used to look up the partial traces in order to retrieve the corresponding target elements and assigning them to target features (label 7). In the sequential scenario, looking up a single trace has a constant cost O(1) because the trace model can be indexed by source element using a hash map. However, an undesirable e ect of the parallelization is that, now, each trace lookup has a cost proportional to the number of worker threads, because forp threads we may needp accesses to the partial traces (that is, the cost is O(p). To mitigate this shortcoming, we also execute the post-processing in parallel. This requires classifying unresolved references into overlapping and non-overlapping. Two references are overlapping if they may potentially cause a race condition when set in parallel.[1] This is the case of opposite references and references that are set in more than one location in the transformation. In the example, client and supplier are non-overlapping, but packagedElement is overlapping because there are several assignments in the same rule. At the end of this phase, the output model may be con gured as required by the underlying meta-modeling framework (label 8). In particular, if the target model is in EMF format, we need to establish the root elements of the resulting EMF resource.

This algorithm exhibits almost the same functional behavior of the original ATL algorithm, but it enables data parallelism. The only observable di erence is that the order in which root elements appear in the model is not deterministic (i.e., elements which are not assigned to any containment reference) because each run may allocate chunks di erently. Sect. 6 evaluates the speed up that can be obtained by applying it to exploit multi-core CPUs. However, it is possible to achieve even greater performance by optimizing the compilation of the sequential part of the transformation.

5 Optimizations

The availability of typing information allows our compiler to target a typed runtime environment like the JVM. This

1. Some meta-modelling frameworks, including EMF, are not thread-safe.

already provides a performance improvement over dynamically typed languages like ATL or ETL, and over interpreter-based approaches like most QVT implementations. In addition, another signi cant increase in performance is possible by applying a number of optimizations to handle common transformation scenarios and idioms. The most relevant optimizations that A2L implements are described next.

5.1 Optimizations at the transformation level

These optimizations are intended to improve the performance of the execution of the transformation rules. They include the process of matching input elements and the binding execution.

Matched rule ordering . Our algorithm tries to match each input element against the input pattern of each matched rule. Algorithm 2 shows the generated code. The sooner the matching rule is identi ed, the more e cient the process is, because it avoids checking unnecessary conditions. Finding the matching rule as soon as possible depends on three main factors: the order in which the patterns are checked, the complexity of the rules' lters, and the structure and contents of the input model (some rules are matched more frequently than others depending on the particular model).

Our approach is to heuristically prioritize rules according to their lters. We count the number of OCL elements to be evaluated as part of the lter execution, and check the rules with fewer elements rst. The rationale is that, in the worst case, all rules must be checked and thus it is preferable to check the cheaper ones rst. If a given element is eventually matched by a rule with a costly lter, the time spent on checking the wrong rule lters rst is low. However, other heuristics are possible, such as estimating the chances of matching an element by considering the operations used in the lter (e.g., equality operation would match less elements than inequalities).

Transformation footprinting . We use the transformation footprint to lter the input model in order to consider only those elements that may actually be matched by some matched rule. Notably, we only use the partial footprint, meaning that we are only interested on the types declared in the source pattern of the rules. In the running example, the partial footprint of that transformation consists of the set footprint= f Package,Class g. This is speci ed in line 3 of Algorithm 1. We intend to signi cantly reduce the bu er size in scenarios in which the transformation only matches a small subset of the model. Although the default ATL algorithm behaves di erently (it computes an associative table to gather objects per type) it would also bene t from this optimization since the table size could be smaller by only recording needed elements. Moreover, we use this step to pre-compute global data such as ClassDeclaration.allInstances() , thus avoiding another model traversal.

Binding handling . ATL relies on binding resolution to assign target references implicitly. To resolve a binding, ATL checks if the the value of the RHS is a primitive value, an object, or a collection of primitive values and/or objects. If it is a collection, it might be the case that it is a nested collection, in which case it needs to be attened. A2L does not need to check these conditions at runtime since it knowns at compile time whether the RHS contains primitive or object values, and whether collections are nested or not.

Trace footprint reduction . ATL relies on recording trace links between the input elements matched by a rule and the corresponding elements created upon execution. A transformation engine without access to information about rule relationships will generate trace links even in cases where they are not needed. In A2L, we apply an optimization to reduce the trace memory footprint, namely, we analyse which rules may need to resolve a given binding. There are two main scenarios: a) if a matched rule is never used to resolve a binding, the link between the input element and the primary output pattern element does not need to be recorded, and b) if a matched rule has more than one output pattern element, there is no need to record the trace link for the secondary elements (i.e., all elements except the rst one) unless there is a resolveTemp operation which retrieves them (in ATL, the resolveTemp operation is used to explicitly retrieve a target element from a given source element).

This optimization is useful to reduce the memory footprint when there are rules that create, for a single input element, a large connected set of elements, but other rules only need to link a single element|typically the parent of these elements.

5.2 OCL-related optimizations

As mentioned in the introduction, A2L supports all ATL and OCL datatypes including collections, maps and tuples, and the standard OCL library. ATL makes heavy use of OCL expressions for navigating the models, selecting elements in the rule lters, and for calculating the values of the target elements' features. However, the evaluation of these OCL expressions is often ine cient, in particular regarding the use of collections. This is due to the fact that OCL datatypes are immutable, which means that each operation over a collection needs to return a new collection. The transformation engine should therefore use internally a library for immutable collections, but this is sometimes not enough when dealing with very large models. The problem is that these auxiliary collections can be huge and they can be unnecessarily created several times during an OCL expression evaluation. Moreover, the typical OCL access patterns for collections are not well suited for immutable collections (e.g., in sequences including appends an element, but in an immutable list prepend is typically more e cient).

To address this issue, we have a two-step approach. In the rst step, we perform escape analysis to check whether a given collection may be modi ed in more than one location. In this case, it is not possible to apply any optimization and we resort to immutable collections. However, if a collection is not going to be shared, we mark each involved (sub-)expression as mutable, so that in the second step the compiler is free to generate code using mutable collections. For instance, in the following listing, the rst example shows a piece of ATL for which it is possible to generate code using mutable collections, whereas in the second example we must use immutable collections.

```
1 -- Intermediate collections do not escape the
2 -- expression
3 s1.ownedPackages->select (p | not p.proxy)
4                   ->collect(p | p.ownedElements)
5 -- The pkgs collection is shared
6 let pkgs : s1.ownedPackages-> select (p | not p.proxy)
7   in if  pkgs->including(aPkg)->size() > 2
8      then pkgs else Set { } endif
```

TABLE 1
Summary of the most relevant OCL optimisations. The target code is written in a Java-like pseudocode for simplicity.

Name	Example	Generated code

Mutable addition	`aPkg.ownedElements`
	`->select(p|p.isProxy)`
	`->including(aClass)`
		`res = new ArrayList<AbstractTypeDcl>()`
		`for (o : aPkg.getOwnedElements())`
		`if (o.isProxy())`
		`res.add(o)`
		`res.add(aClass)`

Filter and check existence	`aPkg.ownedElements`
	`->select(p|p.oclIsKindOf(JAVA!ClassDeclaration))`
	`->exists(c|c.isAbstract)`
		`boolean result = false`
		`for (o : aPkg.getOwnedElements())`
		`if (o instanceof ClassDeclaration)`
		`ClassDeclaration cd = (ClassDeclaration)o`
		`if (cd.isAbstract())`
		`result = true`
		`break`

Filter and count	`aClass.bodyDeclarations`
	`->select(d|not d.isProxy)`
	`->select(d|d.modifier. static )`
	`->size()`
		`size = 0`
		`for (d : aClass.getBodyDeclarations())`
		`if (!d.isProxy() &&`
		`d.getModifier().isStatic())`
		`size++`

Filter and map	`aPkg.ownedElements`
	`->select(p|not p.proxy)`
	`->collect(p|p. class )`
	`->collect(c|c.name)`
		`res = new ArrayList<String>()`
		`for (p in aPkg.getOwnedElements())`
		`if (!p.isProxy())`
		`tmp1 = p.getClass()`
		`tmp2 = tmp1.getName()`
		`res.add(tmp2)`
		`// or res.addAll(tmp2) if flatten`

Collection conversion	`classes`
	`->collect(c| c.name)`
	`->asSet()`
		`// Target collection is created beforehand`
		`res = new HashSet<String>()`
		`for (c in classes)`
		`res.add(c)`

Indexing	`(1) JAVA!ClassDeclaration.allInstances()`
	`->exists(p | p.name = name)`
	`(2) JAVA!ClassDeclaration.allInstances()`
	`->select(p | p.name = name)`
		`// A global object is initialized in`
		`// the pre-processing phase with`
		`// (1) Set<String>`
		`// (2) Map<String, List<ClassDeclaration>>`
		`globalContext.existsIndex.contains(name)`
		`globalContext.selectIndex.get(name)`

In a second step, we use the results of the escape analysis We try to optimize certain access patterns for which we can generate optimized code that avoids redundant creation of temporary collections by using mutable collections. For instance, the evaluation of the following OCL expression using immutable collections requires traversing two intermediate collections (one for select and another for collect) whose size is equal to the number of UML!Class instances.

```
UML!Class.allInstances()->
  select(c| not c.owningPackage.oclIsUndefined())->
  collect(c| c.owningPackage())->asSet()
```

However, if the resulting collection is not shared we can generate more e cient code which avoids unnecessary processing and memory usage. Our compiler detects this particular pattern and generate speci c code for it, using only one traversal of the source collection and without intermediate collections. The following listing illustrates the code that would be generated.

```
Set<Package> result = new HashSet<>();
for (Class c : model.allInstancesOf(Class. class )) {
    if (! (c.getOwningPackage() == null )) {
        result.add(c.getOwningPackage())
    }
}
```

Table 1 shows a summary of the most relevant optimizations, described by means of their context, a prototypical example of each one, and how they are implemented in A2L, i.e, the Java code generated for them.

5.3 Automatic caching

This optimization deals with OCL code that computes the same value several times, and such computation can be potentially time consuming. It is possible to increase the execution performance if such computations are cached so that they are reused in subsequent accesses. ATL supports caching by factorising code in attribute helpers, but this requires the developer to identify which code locations should be cached. Our compiler detects some of these locations and generates code that caches repeated results. In particular, we consider a hot spot a sub-expression within a nested loop such that it starts with a variable that is independent of the outermost loop. For instance, in the following code the value c2.allSuperClasses()->reject(...) is reused across iterations of the outer forAll because it is cached.

```
-- Classes in pkg must have a non-abstract subclass
pkg.ownedClasses->forAll(c1|
  UML!Class.allInstances()->exists(c2|
    c2.allSuperclasses()->reject(c | c.isAbstract)->
      contains(c1))
```

The last row of Table 1 shows another form of automatic caching, in which certain access patterns are compiled as an indexing operation. The index is lled in the pre-processing phase in order to provide fast access during the transformation execution.

6 Validation

To evaluate our approach, we have de ned four research questions regarding the correctness of the obtained transfor-

mation output models, the completeness of the ATL language support, the speedup compared to existing ATL engines and, nally, the scalability of A2L, i.e., the speedup gained when raising the number of available cores. To answer these questions, we carried out an empirical case study [22] by following the guidelines for conducting empirical explanatory case studies by Roneson and Horst [23]. Moreover, the implementation, case studies and scripts to reproduce our results are available at http://github.com/anatlyzer/a2l.

In the next subsections, we describe our research questions and the case studies and metrics we have used to answer these questions. Finally, we discuss the answer to each research question and the overall threats to validity of our proposal.

6.1 Research Questions

Our study addresses the following four research questions. With these questions, we aim to justify the use of our compilation strategy and our parallelization approach in order to signi cantly improve the performance of ATL transformations. At the same time, we argue about the correctness and completeness of our approach.

- RQ1 Compiler Correctness : Does the code generated by the A2L compiler exhibit the same functional behavior as the standard ATL engine? To validate the correctness, we compared the results of running transformations compiled with A2L against the results of running the same transformations on the standard ATL VM. We resort to an available set of regression tests already used by ATL transformation engines [13] to test their correctness.
- RQ2 Compiler Completeness : How much of the ATL language is the A2L compiler able to deal with? To validate its completeness, we evaluated the coverage of the ATL language, de ned by the ATL metamodel, for which the A2L compiler provides support. Moreover, we have manually checked against the ATL documentation which features are actually supported by A2L.
- RQ3 Performance : What is the gain in performance when compared with ATL VM and EMFTVM? To evaluate the performance of the implementation we have used seven case studies, which exercise several transformation styles and ATL constructs. We compared the execution times of different A2L versions (non-optimized, optimized, sequential and parallel) with the standard ATL VM and EMFTVM. EMFTVM is a newer ATL engine that compiles to Java bytecode on the y and it is reported to achieve gains of 80% in basic benchmarks. In the experiments, we used A2L in both sequential and parallel mode. The sequential execution allows us to show the gains obtained only by the use of static analysis and optimizations. The parallel execution aims to validate our parallelization strategy. Since the optimizations are optional, we also compared the executions with and without the optimizations, to assess their impact on performance.
- RQ4 Parallelism : What are the e ects of adding more cores? We analyse how the number of cores in uences the execution times of ATL transformations parallelized by A2L. We have executed our case studies using an increasing number of threads and recorded the obtained speedups.

6.2 Experimental Setup

6.2.1 Case Studies

RQ1 and RQ2 are evaluated using the same set of 24 regression tests [13] that the ATL team used to validate the correctness of two consecutive versions of the ATL Virtual Machine. Each of these tests consists of one model transformation and all the necessary artifacts needed to execute the transformation, i.e., the input and output metamodels, and a sample input model. Since none of these tests provides large models, to answer RQ3 and RQ4, we considered seven additional case studies: java2uml (reverse engineering Java code into UML models), java2graph (creates a graph of dependencies between Java classes), dblpv1 (query the DBLP database to obtain authors and associated information), dblp2bibtex (map DBLP entries to BibTeX records), identity (copy transformation of the IMDB database), ndcouples (extracts actors from IMDB who played together) and airquality (queries weather data obtained from sensors). For these transformations, we have models with up to 5.6 millions elements, and 1.2 GB when serialized and stored in disk.

In order to characterize our benchmark, we have considered six dimensions, which are used in Table 2 to summarize the main characteristics of the performance case studies:

1) I/O size is the expected size of the output model with respect to the size of the input model.
2) Matching cost is the expected cost of rule matching, in particular the complexity of the rule lters.
3) Rule cost is the expected cost of rule execution, which is the complexity of the RHS of the bindings.
4) FP size is the footprint size with respect to the input model size, that is, if the transformation has rules that attempt to match all input elements.
5) OCL refers to the dominant OCL elements in the transformation: collection intensive, usage of allInstances , conditionals, etc.
6) Finally, we count the transformation elements in order to have an indication about its \size": number of matched rules (M_R), number or called or lazy rules (L_R), number of helpers (H), dependencies between rules as the number of bindings for references (B), number of imperative blocks (I).

All transformations except identity and dblp2bibtex generate output models which are smaller, in terms of number of elements, than their corresponding input models (I/O size). This means that they either rule out many elements in the rule lters (they have a high matching cost) or its footprint with respect to the original meta-model is small (they purposely lack rules to match certain elements). Both identity and dblp2bibtex exercise the ability of the engine to handle many binding resolutions. Regarding the matching cost, we have that airquality, dblpv1 and dblp2bibtex have at least one rule lter which traverses collections or accesses all instances of a given type, so it is expected to be costly, whereas the other four transformations have very simple or no rule lters.

TABLE 2
Main features of the case studies.

Name	M_R	L_R	H	B	I	I/O size	Match. cost	Exec. cost	FP size	OCL elements
airquality	1	0	0	0	0	$I_{size} > O_{size}$	high	low	equal	collections, allInstances
dblpv1	1	0	1	0	0	$I_{size} > O_{size}$	medium	medium	equal	collections
dblp2bibtex	7	2	1	1	0	$I_{size} = O_{size}$	medium	medium	equal	allInstances
ndcouples	3	1	2	5	2	$I_{size} > O_{size}$	low	high	equal	set operations
identity	5	0	0	8	0	$I_{size} = O_{size}$	low	low	equal	property access
java2graph	2	0	3	2	0	$I_{size} > O_{size}$	low	medium	smaller	conditionals
java2uml	3	5	2	6	0	$I_{size} > O_{size}$	low	medium	smaller	collections

The execution of cost of airquality and dblpv1 is low because they are "query transformation" whose target elements have simple initialisations. In this respect identity also has simple initialisations, but it has more binding dependencies which makes the post-processing phase time consuming.

6.2.2 Evaluation Metrics

To answer our research questions, we use several metrics depending on the nature of the research question.

Model Comparison Metrics (RQ1): To evaluate research question RQ1, correctness, we need to compare the resulting models after running the code produced by the A2L compiler, with those obtained from the execution of the standard ATL VM. For this, we used EMF Compare, a model comparison framework that compares two models and reports differences between them, such as additions, deletions, and updated elements.

Language Coverage Metrics (RQ2): To evaluate RQ2, we computed the footprints of the ATL transformations with respect to the ATL metamodel. This gives an estimation of how many features are tested by the test cases.

Execution Performance Metrics (RQ3 and RQ4): To evaluate research questions RQ3 and RQ4, we calculated the execution time of the seven case studies listed in Table 2, using a large model as input. We run the experiments on a desktop machine with Ubuntu 18.04 and kernel 5.3.0, a i7-5820K CPU, with 6 cores at 3.30GHz and hyper-threading (12 threads) and 16 GB of RAM, which is expected to be representative of a typical setup of a professional developer. We have used Java 8 (OpenJDK 1.8.0_252) configured with the default options except for the heap size which was set to 8 GB (-Xms=8196m -Xmx=8196m) except for dblp2bibtex which was set to 12GB. Each case study is run 10 times with the different engines, discarding the first two runs (as warm up). We perform the 10 executions together in the same VM instance, but after each execution we wait until the garbage collector has released the used memory. We report the average results.

6.3 Result Analysis

6.3.1 Results for RQ1

We executed the 24 regression test cases and compared the output models produced by the standard ATL engine and by A2L. All test cases produced the same results, apart from five of them that could not be directly executed.

We could not compile the ATL2Problem transformation with A2L because its typing is too convoluted for AnATLyzer and it cannot properly infer the type of a couple of expressions. Another two unsupported transformations were DSL2XML and KM32DSL, because they set global variables using some reflective operations which are currently unsupported by A2L. Similarly, the SpreadsheetMLSimplified2Trace transformation uses global attributes in a way that adds serious performance penalties to the parallel algorithm. It is worth noting that all of these four transformations could be rewritten so that they could be compiled by A2L. For instance, when there are global variables involved, a strategy could be to use helpers to compute the global information from the source model each time (possibly with some penalty in the execution time). When the problem is related to typing, it might be possible to insert specific annotations (e.g., a dummy version of oclAsType) in dedicated places to guide the type inference performed by AnATLyzer [24]. Finally, the ATL and A2L output models of the XML2DSLModel transformation were slightly different because this transformation suffers from child stealing, i.e., an element is set to a containment reference more than once.

6.3.2 Results for RQ2

The test cases used in our experiments cover 88% of the ATL meta-model. The missing 12% belongs to meta-model elements used to represent declaration of libraries and query modules, unique lazy rules, entry point rules, rule inheritance and map types. We have separate tests for all of these elements, except query modules and rule inheritance which are currently not supported. Although we do not foresee any difficulty in supporting rule inheritance in the future, we decided not to implement it at this stage because it is rarely used in practice [25].

We have used the ATL manual as reference for our implementation. Table 3 shows the language features covered by A2L. We support all forms of matched rules (i.e., 1:1, 1:N, N:1 and N:N) and all major ATL features.

6.3.3 Results for RQ3

Table 4 shows the execution times of the case studies in our desktop machine. It compares two versions of the ATL engine (the standard VM and EMFTVM) against four different configurations of A2L: sequential mode without optimizations (seq O), sequential mode with optimizations (seq O+), parallel (using 12 cores) without optimizations enabled (par O), and parallel (using 12 cores) with optimizations (par O+) | or simply A2L, since this is the default A2L mode. The figures shown in Table 4 correspond to the average execution time in seconds of eight runs of each transformation. Note that the execution time excludes model loading, but in the case of A2L, it includes the three phases of the algorithm: pre-processing (filling the buffer after model loading), transformation execution, and post-processing. We did not include the parallel ATL [9] implementation (pATL) because it is not supported anymore and we could not make it work reliably.

TABLE 3
ATL coverage.

Feature	Support	Observations
Matched rules	Yes	
Input elements = 1	Yes	
Input elements > 1	Yes	Via rewriting
Output elements = 1	Yes	
Output elements > 1	Yes	
Rule inheritance	No	
Binding resolution	Yes	
resolveTemp	Yes	Secondary elements resolution
Lazy rules	Yes	
Unique lazy rules	Yes	
Called rules	Yes	Also end/entry point rules
Helpers	Yes	Context and global helpers
OCL		
Collection types	Yes	
Tuple types	Yes	
Collection iterators	Yes	
Iterate operation	Yes	
Re ective operations	No	

Table 5 shows the speedups obtained by the di erent transformation engines and A2L execution modes. For each one, we calculated the individual speedups achieved in all the case studies (excluding the dblp2bibtex model transformation in the comparisons between ATL/EMFTVM and A2L because it could not be executed in ATL and EMFTVM). Every cell shows a tuple with the minimum (left) and maximum (right) values, as well as its geometric mean (center), which is the most informative way to represent the average speedup as expected by users [26]. The standard ATL engine is consistently slower than the other options. This is due to its sub-optimal implementation of immutable collections which hinders its ability to handle scenarios with extensive processing of large collections [21]. In particular, the airquality test case heavily exercises this feature and shows that EMFTVM is far more e cient (50 vs. 1,280 secs). This is because it uses a custom implementation of immutable (and lazy) collections. Moreover, EMFTVM compiles to JVM bytecode on the y, which provides additional gains. These two features combined makes it normally much faster than the standard ATL engine (except for the ndcouples example).

A2L obtains signi cant gains over both engines in all execution modes. Next, we only discuss improvements w.r.t. EMFTVM since they also imply gains over standard ATL.

In the slowest A2L execution mode (A2L seq O{), the speedup w.r.t. EMFTVM is between 1.37x and 10.4x depending on the application. Given that both A2L and EMFTVM target JVM bytecode, this improvement can be explained (in addition to di erences in the engine internals) by the fact that A2L does not need to generate code for dynamic checks and model accesses. Table 4 shows that all case studies bene t from this improvement. The ndcouples case study is the slowest, with a relatively modest performance gain of 1.37x. The main bottleneck is a nested loop which computes the same value in di erent rule applications. This shortcoming is addressed by our optimizer through automatic caching. Given these results, it can be stated that it is possible to have a signi cant performance improvement by changing the design of the transformation language from dynamically to statically typed. All other A2L execution modes (with optimizations and in parallel) outperform EMFTVM even more, with speedups that range between 2.32x and 38.28x (A2L seq O+), 1.79x and 21.43x (A2L par O), and even between 2.4x and 245.83x in the case of the A2L default behavior (A2L par O+), depending on the test case. The geometric means indicate expected average speedups of A2L against EMFTVM of 3.37, 8.59, 6.49 and 22.42, respectively.

We also wanted to investigate the individual e ects of optimization and parallelization in A2L. This is why we compared the performance of the four possible execution modes of A2L: A2L seq O , A2L seq O+, A2L par O , and A2L par O+.

Using as baseline the sequential mode without optimizations (seq O), the speedup obtained by enabling optimizations (seq O+) ranges between 0.84x and 66.76x depending on the case study. The worse case is the identity application, in which the execution time even increases. This is because the partial footprint of the transformation is equal to the source meta-model, and therefore our footprint ltering optimisation only adds overhead to the optimised version but does not reduce the bu er size (there are no other optimisations in that transformation). Therefore, when the partial footprint is equal to the meta-model, it is better not to activate this optimisation. The best case occurs in the dblp2bibtex transformation, where optimizations achieve a speedup of 66.76x due to the automatic indexing optimization. The speedup obtained by the use of optimizations in the airquality case study is 5.2x, because this transformation is particularly well-suited for collection optimizations given that it applies several OCL iterators to the set of objects returned by allInstances (i.e., it needs to traverse the complete model). The ndcouples case study gets a speedup of 3.84x thanks to the automatic caching optimization. The rest of the transformations obtain speedups of 1.64x, 6.31x and 1.65x, respectively, when optimizations are enabled, with a geometric average of 4.07x.

To analyse the e ects of parallelization on the performance of A2L (using the 12 threads in our experimental desktop machine), we compared our baseline A2L seq O with A2L par O . The resulting speedup ranges between 1.15x and 4.66x, which represents a signi cant improvement, although not as noticeable as the one obtained with the optimizations. This is clear in the comparison between optimizations (seq O+) and parallelization (par O), where the former outperforms the latter 2.14x (' 1.0/0.47x) on average.

The speedup obtained by combining optimizations and parallelization ranges between 1.11x and 92.02x depending on the case study, compared to the sequential baseline (seq O). Again, the identity transformation gets the smallest improvement (1.11x) while dblp2bibtex obtains the largest gain (92.02x). The rest of the test cases achieve speedups of 32.23x, 22.66x, 1.70x, 14.19, and 4.47x, respectively.

Finally, if we compare the execution times of the two parallel modes (with and without optimizations), the one with optimizations obtain speedups that range between 0.96x (identity) and 53.48x (dblp2bibtex). This means that optimizations also have a positive e ect in the parallel results, mainly because they reduce memory usage, thus reducing the pressure over the garbage collector (i.e., the less garbage collection pauses the better, because a pause stops all threads and causes an important degradation in the speedup).

In summary, A2L achieves signi cant speedups compared to ATL and EMFTVM. By combining both optimizations and parallel execution, A2L is able to outperform EMFTVM

TABLE 4
Performance comparison. Execution on an i7-5820K CPU@3.30GHz - 6 cores (12 threads). Time in seconds.

	#elements (millions)	ATL	EMFTVM	A2L: seq O (1 thread)	A2L: seq O+ (1 thread)	A2L: par O (12 threads)	A2L: par O+ (12 threads)
airquality	0.1M	1280.40	50.70	6.65	1.32	3.39	0.21
ndcouples	3.5M	1765.09	1908.89	1395.66	363.28	299.54	61.60
identity	3.5M	269.00	71.73	11.63	13.87	10.08	10.48
java2graph	4.4M	146.33	2.32	1.65	1.00	1.30	0.97
java2uml	4.4M	49.18	42.05	27.28	4.32	14.59	1.92
dblpv1	5.6M	75.66	48.68	4.68	2.84	2.27	1.05
dblp2bibtex	5.6M	-	-	783.07	11.73	455.15	8.51

TABLE 5
Speedups achieved between the transformation engines: ATL, EMFTVM and the different A2L options.

Speedups	EMFTVM	A2L seq O	A2L seq O+	A2L par O	A2L par O+
ATL	[0.92; 4.64; 63.06]	[1.26; 15.63; 192.6]	[4.86; 39.90; 966.63]	[3.37; 30.15; 377.25]	[25.58; 104.14; 6207.98]
EMFTVM	{	[1.37; 3.37; 10.4]	[2.32; 8.59; 38.28]	[1.79; 6.49; 21.43]	[2.40; 22.42; 245.83]
A2L: seq, O		{	[0.84; 4.07; 66.76]	[1.15; 1.90; 4.66]	[1.11; 9.69; 92.02]
A2L. seq, O+			{	[0.03; 0.47; 1.38]	[1.04; 2.38; 6.42]
A2L: par, O				{	[0.96; 5.11; 53.48]

between 2.4x and 245.83x, with an expected average of 22.42x. In this way, A2L execution times for large models become acceptable in all cases, which is an indication that ATL can become a competitive model transformation language when compiled with A2L. In addition, A2L is capable of running transformations such as *dblp2bibtex* that may not be executed with the other engines because of the excessive usage of collection operations over very large collections. In practice, these results also mean a much better developer experience because the transformations can be used as part of the development process without incurring long waiting times.

6.3.4 Results for RQ4

To answer this question, we have executed the case studies using an increasing number of cores, from 1 to 12. Our algorithm has three phases: pre-processing, execution and post-processing, but only the execution phase is expected to have a speedup due to parallelism, because the parallel execution of the post-processing phase only amortizes the cost of accessing the partial traces created by each transformation instance. The speedups obtained in the execution phase for each case study are shown in Fig. 7. We can see how two of the test cases (*airquality* and *ndcouples*) scale up very well, close to the theoretical limit, which is 6x with 6 threads. Then, *dblpv1* and *identity* also scale well, with speedups around 4.5x with 6 threads. However, the other three stop scaling soon. This can be caused by a variety of reasons. On the one hand, the rules of these transformations have a modest computation cost which may increase the competition for obtaining the next chunk from the scheduler. On the other hand, parallel computations are highly sensitive to external factors. For instance, a stop-the-world execution of the Java GC would provoke an important decrease in the speedup. Also, in Java and EMF in particular there is little control over the memory layout which may cause an increase in cache misses.

A related question is what is the scalability when taking into account all the phases of the algorithm, or in other words, how much the pre-processing and post-processing phases limit the parallel speedup. Table 6 shows the breakdown of the execution times of the case studies in sequential and in parallel mode with 6 threads. In some transformations, like *airquality*, *java2graph*, *java2uml* and *dblpv1*, the post-processing time is negligible because they do not resolve many bindings. However, transformations with more bindings or whose bindings have many objects in the RHS, incur in large postprocessing times. Therefore, the post-processing phase represents the main bottleneck for scalability in case there are many dependencies among the elements in the rules, because the traces are distributed in the threads and there is a $O(numThreads)$ cost to access them.

Fig. 7. Speedup of the execution phase of the transformations.

Overall, it can be claimed that our algorithm exhibits good scalability, particularly for transformations whose rules have a high computational cost, but relatively few rule dependencies.

6.3.5 Further findings

Compatibility with ATL. The fact that ATL is a dynamic language poses the challenge of correctly inferring type information for its compilation to a typed target language such as Java. In practice this means that we require the transformation to be considered well-typed by AnATLyzer. However,

TABLE 6
Execution times for A2L O+.

	Sequential			Par. 6 threads		
	Pre.	Exec.	Post.	Pre	Exec.	Post.
airquality	0.01	1.31	0.00	0.01	0.25	0.00
ndcouples	0.60	353.65	9.03	0.61	68.64	8.48
identity	0.66	3.88	9.33	0.65	0.87	9.14
java2graph	0.94	0.05	0.01	0.93	0.01	0.01
java2uml	1.10	3.13	0.08	1.06	0.91	0.07
dblpv1	0.73	2.11	0.00	0.74	0.47	0.00
dblp2bibtex	1.88	4.70	5.15	1.84	1.72	5.20

when evaluating RQ1, we already found that ATL2Problem and those transformations using reflective operations could not be compiled. The fact that we have been able to compile the rest of the test cases make us believe that this is a minor issue. Besides, it is always possible to write an equivalent program compatible with A2L.

At the execution level, there might be differences in the order in which the root elements appear in the serialized model, notably when executed in parallel mode. Nevertheless, this is not a real incompatibility since this order is not prescribed by the ATL manual. There are also some differences in the way errors are handled. In particular, ATL signals rule conflicts with a runtime exception, whereas A2L ignores them and relies on AnATLyzer for signalling them at compile time.

Additional optimizations. In the experiments, we have observed that the post-processing phase is sometimes a bottleneck when a transformation needs to initialize many target references. It would be interesting to find out ways to reduce its impact. At the pre-processing level, it would be possible to filter the input model at loading time using approaches like partial model loading [27]. This would enable the engine to start the transformation without waiting for the model to be fully loaded. Regarding the evaluation of OCL expressions, more specific optimizations for common access patterns can be developed. A2L is prepared for this with a dedicated extension mechanism.

Using GPLs vs. ATL. There is a recent trend in the modeling community to use general purpose programming languages (GPL), such as Java or Scala, to develop and execute model transformations. There are several reasons that justify such a decision. For example, (a) IDE features available for GPLs such as live error reporting, quick fixes and debugging have been traditionally missing for transformation languages like ATL; (b) the performance and scalability of ATL, when compared with those of GPL, are much worse; (c) ATL can only deal with EMF models, which is not the format in which many models are stored (for instance, in biology or automotive applications), being rather inefficient, too. The declarative nature of ATL for specifying the transformations and the facility to navigate models using OCL was at the beginning a strong point against Java, but this is not the case any more after Java 8 supported lambda expressions. Scala was also strong in this respect, and its efficient performance has made it a good recent replacement for ATL, too.

However, our goal with A2L is to improve the situation for model transformation languages in general, and ATL in particular, showing that our design can make MT languages competitive again against these GPLs. First, the AnATLyzer tool provides very helpful and useful analyses of the ATL code which are not possible if the transformation is written in a GPL. This, together with the quick-fixing and visualization capabilities that AnATLyzer provides [11], [12] help solving the first shortcoming. Second, we have seen how the performance and scalability of the A2L generated code can be quite acceptable for transforming large models, with competitive execution times. Furthermore, the specific optimizations of A2L for the ATL code and the OCL expressions can provide interesting advantages over GPLs since navigation code could be written in the most readable manner without impact in its performance. A transformation engine also hides the technological complexity related to the modelling platforms. For instance, we internally optimize certain type of accesses to multi-valued features for which we discovered poor performance of the default EMF's internal iterators.[2] Moreover, the declarative nature of ATL enables the automatic parallelization of the transformation code. If a transformation is written in a GPL, this improvement has to be done manually, which is typically difficult, cumbersome, and error prone.

6.4 Threats to Validity

In this section, we cover the four basic types of validity threats that can affect the validity of our study [28].

6.4.1 Conclusion Validity

Conclusion validity affects the ability to draw correct conclusion about the relations between the treatment and the outcome, i.e., how reasonable the conclusion is. Examples that influence this threat to validity include the choice of sample size, and the measurement of the experiment. In this respect, the correctness and coverage of the compiler have been assessed using the standard regression test suite for ATL [13]. The size of this suite is relatively low, but it covers a large part of the ATL language. Regarding the performance evaluation, there might be specific transformations for which A2L performs worse than ATL and EMFTVM. To mitigate this threat, we have used transformations which exercise different types of scenarios and its analysis shows that they cover different transformation styles. Moreover, the large improvents obtained with A2L makes us confident that improvements will be achieved even in unforeseen scenarios.

6.4.2 Construct Validity

Construct validity refers to the extent to which the experiment setting reflects the theory, i.e., whether the research/tests are well-constructed using established standards and methods. For example, whether the type of samples are representative of the population or not; or whether the number of classes taken reflects common experience. Again, by using a standard test suite for ATL, we aimed at minimizing this threat, since the transformations that comprise the suite provide a representative subset of all kinds of transformations that can be written with ATL. Likewise, the seven additional transformations were carefully selected so that they contain the main features that can have impact on the performance and scalability of the results, and therefore can constitute a representative

2. This happens because EListIterator.hasNext() invokes List.size() instead of having its own size property, which prevents JVM optimizations

sample of the kinds of ATL transformations in which we are interested. Moreover, the sizes of the input models were also selected according to the model sizes used in similar tests [9], [20] in order to extract comparable conclusions.

6.4.3 External Validity

This kind of threat limits the ability to generalize the results beyond the experiment context. In this respect, the fact that we have used representative model transformations gives us some con dence that the performance results can be generalized to the rest of the ATL model transformations. Regarding how our results could be applicable to other languages, we believe that for model-to-model rule-based languages like ETL and RubyTL, the applicability would be straightforward, provided that an appropriate type checker is available. QVT-Operational is also rule-based but the rule structure is more explicit, which implies that there might be less opportunities for data-based parallelism at the matched rule level as we do. Nevertheless, the OCL compiler and optimizer could be adapted. In particular, applying these ideas to QVT-Operational is part of our future work.

Moreover, we have used di erently sized input models for our study to cover diverse scenarios. However, there may be scenarios where even larger input models are required to be processed by model transformations. We cannot generalize our results for very large models (going beyond 10 millions of model elements such as present in the Train Benchmark [29] for continuous model queries) based on our performed study and leave this as subject to future work.

6.4.4 Internal Validity

Internal validity checks whether the test or instrument measures what it is supposed to. This threat can a ect the independent variable with respect to causality. That is, the results may indicate a causal relationship, although there is none. In this respect, the optimizations at the transformation and OCL expression level, as well as the parallelization algorithm used to execute the transformation have proved to be e ective to signi cantly improve its performance. The speedup results obtained independently for each separated feature seem to corroborate our hypotheses. The compiler was designed so that these features could be independently enabled or disabled, precisely to facilitate these kinds of analyses.

7 Related Work

With respect to the contribution of this paper, namely the e cient execution of model transformations by data parallelism and optimizations based on static analysis information, we identify three lines of related work. First, we discuss general approaches for speeding up model transformations which apply smart execution strategies for di erent contexts. Second, we discuss research dedicated to the evaluation of model transformation engines. Third, we discuss approaches focusing on the parallel execution of model transformations.

7.1 Model transformation execution strategies

In this paper, we have investigated the creation of output models from input models by following the classical batch transformation strategy, i.e., the transformation run is considered as a fresh one which is creating a new output model from scratch for a given input model [30]. In addition to this strategy, there are several other strategies for executing model transformations in particular contexts. First, incremental transformations [31]{[33] have been proposed for cases where output models are already available from previous transformation runs. In such cases, the transformation engine may only propagate the changes of the input models to the existing output models. Second, lazy transformations [34] have been proposed for cases where only subsets of output models are needed in a rst step. In such cases, the transformation engine only creates these subsets in a rst phase, while other elements are created just-in-time when they are requested. Third, streaming transformations [35] have been proposed for transforming so-called streaming models, i.e., models are considered as an open and continuous stream of model elements opposed to a xed set of elements given at once in a closed input model. Fourth, patch transformations [36] are used in cases where transformations are evolving and the existing output models have to be migrated to newer versions of the transformations.

All of the mentioned execution strategies mainly focus on not having to re-execute the whole transformation by providing some kind of reactivity, e.g., to changes in the input models and model transformations or read access to the output models. Needless to say, identifying the transformation statements that are unnecessary to be re-executed is probably the best optimization for running transformations outside the classical batch transformation area.

While we have focused on batch transformations in this work, we do not see any major obstacle to adopt the presented optimizations also for other kind of transformation execution strategies used for the discussed contexts. On the contrary, we see the introduction and evaluation of the presented optimization techniques for these additional transformation strategies as an interesting line for future research.

In addition to the already mentioned model transformation execution strategies, in recent years there have been also major e orts in optimizing the execution of full batch transformations by following the distributed execution paradigm for transforming large models which are fragmented over di erent computing nodes. For instance, MapReduce has been exploited for running ATL transformations on distributed models [9]. Camargo et al. presented a data-centric famework for distributed model transformations reusing the Bloom platform [37]. In one of our previous work, we have employed Linda-based tuple spaces to run distributed model transformations [38]. Finally, graph queries (comparable to the matching part of model transformation rules such as the in-pattern of ATL rules) are executed for large models in a distributed manner by combining incremental graph search techniques and cloud computing technologies [39] as well as by providing dedicated optimization concepts such as node sharing for e ciently executing partially redundant code fragments [40].

In this paper, we did not consider the distribution aspect of models but rather focused on supporting the scenario of running model transformations directly on developer-scale machines having the models stored in-memory. Thus, we consider investigations on the distribution aspect in combination with our presented optimization techniques as a subject for future work.

7.2 Performance evaluations of transformation engines

There is work on evaluating the performance of state-of-the-art model transformation engines. For instance, Amstel et al. [41] compared the runtime performance of transformations written in ATL and in QVT, nding that the standard ATL engine outperforms the available QVT engines. From these works we can conclude that A2L may also outperform existing QVT engines, although this needs to be con rmed with the more recent versions of the QVT engines. In [42], several implementation variants of the ATL language, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their di erent runtime performance was compared. However, both mentioned works [41], [42] only consider the traditional sequential execution engines.

A performance evaluation of the standard ATL engines with respect to running the transformations within database technologies is performed in [43]. In our work, we currently do not use a dedicated database technology for storing the models to be transformed. However, the presented optimization techniques for ATL rules and OCL expressions based on static analysis information may also help in cases where dedicated database technologies are used to produce even more e cient code that is subsequently executed inside the databases. An interesting future work line in this respect is to study the combination of the presented approach of this paper with current advances achieved for graph databases (cf. [44] for a survey) as well as investigations of data-parallel operations inside databases (cf. [45] for a comparison of di erent execution models) to allow even more e cient transformations of very large models.

Finally, we also like to mention the Transformation Tool Contest (TTC) [3] that is now running for 13 years. Every year, there are dedicated transformation cases developed and submitted by the community for the community. In previous years, there have also been some transformation cases that focused on performance and scalability of model transformations. In particular, worth to mention are the Train Benchmark case, the Movie Database case and the Program Comprehension case. We have reused and partially adapted the Movie Database case and the Program Comprehension case for our evaluation, as they represent outplace batch transformations that we are considering in this paper.

7.3 Parallel model transformations

In recent years, there is an increased interest in parallelizing di erent types of model transformations which resulted in dedicated extensions for graph transformation languages, model management languages based on Epsilon, and also ATL which we discuss in the following.

First, there is work in the eld of graph transformations where multi-core platforms are used for parallel execution of graph transformation rules [46], [47]. In these papers, speci c focus is put on the parallel execution of the matching phase of the left-hand sides of graph transformation rules, which is considered to be the most expensive part. Another work exploits the Bulk Synchronous Parallel model for executing transformations based on the Henshin graph transformation framework [48]. To make use of the Bulk Synchronous Parallel model, the Henshin graph transformation rules are compiled to Apache Giraph.

Secondly, e orts have been made to speed up di erent types of model management programs (which can be considered as speci c kinds of model transformations) for the Epsilon framework [19], [49] available in Eclipse. In particular, parallel execution support for the di erent model management languages provided by Epsilon has been presented in [50], [51]. For instance, the authors provide data and rule parallelization approaches for the Epsilon Validation Language (EVL). However, the use of static analysis information for automatic performance improvement is only mentioned as future work.

Thirdly, there have been pioneering approaches for the parallel execution of ATL transformations. In previous work, we have presented LinTra [20], an approach for running ATL transformations on Linda-based platforms following the data-based parallelization approach. Tisi et al. [10] presented another approach for the parallel execution of ATL transformations, using a task-based approach for parallelization, as already mentioned in Sect. 2.3.

7.4 Synopsis

While there are several approaches available for speeding up model transformations, the scalability of model transformations is still considered as a major challenge in MDE [52]. We are not aware of any approach that uses the information from static analyses to nd improved (parallel) execution strategies for model transformations. Thus, we are con dent that in this paper we provide an important cornerstone for scalable and high-performance execution of model transformations, which may also help to improve other model transformation engines beyond ATL. This line of research is considered critical to the long-term success of model transformation tools, as a recent survey has revealed [30].

8 Conclusions

This paper has presented A2L, a compiler for the ATL model transformation language that aims at achieving e cient transformations of large models. Improved performance and scalability is accomplished by two of the main features of A2L: the use of static information to achieve e ective optimizations on both rule execution and the evaluation of the OCL expressions; and the use of data parallelism in the algorithm that implements and executes the transformation. The results show that A2L produces large performance gains with respect to existing ATL engines in both sequential and parallel modes. The gures presented in this paper should be a baseline for the expected performance of future transformation languages.

Our work can be extended along di erent lines of research. First, further optimizations can be pursued, mainly for domain-speci c transformations where the semantics of the particular domains can be taken into account. Second, we plan to study how our work can be generalised to other model transformation languages, notably QVT-Operational. To this end we aim at designing an intermediate representation from which it is easy to reuse most of the infrastructure. Implementation-wise, it would be interesting to create an intermediate representation in which it is easier to perform

3. https://www.transformation-tool-contest.eu

rewritings and make our optimizations composable. Handling models from different modeling platforms, beyond EMF, or even stored as plain Java objects, is another research line we are also working on, with the goal of widening the usability of A2L. This line of research may also require the development of a dedicated debugger which allows the observation and control of the executing transformations directly on the ATL code level [53]. Finally, we would like to explore the possibilities of creating a streaming transformation engine on top of A2L as well as of employing emerging database technologies for executing our transformations for very large models.

Acknowledgments

This work was partially funded by Spanish Research Projects PGC2018-094905-B-I00, TIN2015-73968-JIN (AEI/FEDER/UE), a Ramón y Cajal 2017 research grant, and TIN2016-75944-R. Furthermore, this work was partially supported and funded by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and by the FWF under the Grant Numbers P28519-N31 and P30525-N31.

References

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed. Morgan & Claypool Pub., 2017.

[2] B. Selic, "What will it take? A view on adoption of model-based methods in practice," *Software and System Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[3] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. Sánchez Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model driven engineering," in *Proc. of the Workshop on Scalability in Model Driven Engineering (BigMDE 2013)*. ACM, 2013, pp. 2:1–2:10.

[4] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos, "Engineering model transformations with transML," *Software & Systems Modeling*, vol. 12, no. 3, pp. 555–577, 2013.

[5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[6] OMG, *MOF QVT Final Adopted Specification*, Object Management Group, 2005, OMG doc. ptc/05-11-01.

[7] L. Burgueño, J. Cabot, and S. Gérard, "The future of model transformation languages: An open community discussion," *Journal of Object Technology*, vol. 18, no. 3, pp. 7:1–11, Jul. 2019.

[8] J. S. Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Trans. Software Eng.*, vol. 43, no. 9, pp. 868–897, 2017.

[9] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot, "Distributing relational model transformation on MapReduce," *Journal of Systems and Software*, vol. 142, pp. 1–20, 2018.

[10] M. Tisi, S. M. Pérez, and H. Choura, "Parallel Execution of ATL Transformation Rules," in *Proc. of MoDELS 2013*, ser. LNCS, vol. 8107. Springer, 2013, pp. 656–672.

[11] J. S. Cuadrado, E. Guerra, and J. de Lara, "Quick fixing ATL transformations with speculative analysis," *Software and Systems Modeling*, vol. 17, no. 3, pp. 779–813, 2018.

[12] ——, "AnATLyzer: an advanced IDE for ATL model transformations," in *ICSE'18 Companion Proceedings*. ACM, 2018, pp. 85–88.

[13] F. Jouault, "Regression Tests for the ATL Virtual Machine," 2013. [Online]. Available: https://wiki.eclipse.org/ATL_VM_Testing

[14] C. Eastman, P. Tiecholz, R. Sacks, and K. Liston, *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*, 2nd ed. John Wiley, 2011.

[15] Y. Dajsuren and M. van den Brand, Eds., *Automotive Systems and Software Engineering – State of the Art and Future Trends*. Springer, 2019.

[16] J. Troya, H. Brunelière, M. Fleck, M. Wimmer, L. Orue-Echevarria, and J. Gorroñogoitia, "ARTIST: Model-Based Stairway to the Cloud," in *Proc. of the Projects Showcase @ STAF*, vol. 1400. CEUR-WS.org, 2015, pp. 1–8.

[17] Object Management Group, *Object Constraint Language (OCL) Specification. Version 2.2*, Feb. 2010, OMG Document formal/2010-02-01.

[18] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," in *Proc. of MODELS 2011*, ser. LNCS, vol. 6981. Springer, 2011, pp. 623–637.

[19] D. Kolovos, L. Rose, R. Paige, and A. García-Domínguez, *The Epsilon Book*. Eclipse, 2010. [Online]. Available: https://www.eclipse.org/epsilon/doc/book/

[20] L. Burgueño, M. Wimmer, and A. Vallecillo, "A Linda-based platform for the parallel execution of out-place model transformations," *Information and Software Technology*, vol. 79, pp. 17–35, 2016.

[21] J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin, "Optimization patterns for OCL-based model transformations," in *Proc. of Workshops and Symposia at MODELS 2008*, ser. LNCS, vol. 5421. Springer, 2008, pp. 273–284.

[22] A. S. Lee, "A scientific methodology for MIS case studies," *MIS Quarterly*, vol. 13, no. 1, pp. 33–50, 1989.

[23] P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[24] "AnATLyzer tutorial," 2017, https://github.com/jesusc/anatlyzer-models17.

[25] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software and Systems Modeling*, vol. 14, no. 2, pp. 537–572, 2015.

[26] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proc. of SC'15*. ACM, 2015, pp. 73:1–73:12.

[27] R. Wei, D. S. Kolovos, A. Garcia-Dominguez, K. Barmpis, and R. F. Paige, "Partial loading of XMI models," in *Proc. of MODELS 2016*. ACM, 2016, pp. 329–339.

[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012.

[29] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, "The train benchmark: cross-technology performance evaluation of continuous model queries," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1365–1393, 2018.

[30] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software and Systems Modeling*, vol. 18, no. 4, pp. 2361–2397, 2019.

[31] T. L. Calvar, F. Jouault, F. Chhel, and M. Clavreul, "E□cient ATL incremental transformations," *Journal of Object Technology*, vol. 18, no. 3, pp. 2:1–17, 2019.

[32] A. Razavi and K. Kontogiannis, "Partial evaluation of model transformations," in *Proc. of ICSE 2012*. IEEE Computer Society, 2012, pp. 562–572.

[33] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software and Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[34] M. Tisi, S. M. Perez, F. Jouault, and J. Cabot, "Lazy execution of model-to-model transformations," in *Proc. of MODELS 2011*, ser. LNCS, vol. 6981. Springer, 2011, pp. 32–46.

[35] J. S. Cuadrado and J. de Lara, "Streaming model transformations: Scenarios, challenges and initial solutions," in *Proc. of ICMT 2013*, ser. LNCS, vol. 7909. Springer, 2013, pp. 1–16.

[36] A. Bergmayr, J. Troya, and M. Wimmer, "From out-place transformation evolution to in-place model patching," in *Proc. of ASE 2014*. ACM, 2014, pp. 647–652.

[37] L. C. Camargo and M. D. D. Fabro, "Applying a data-centric framework for developing model transformations," in *Proc. of SAC 2019*. ACM, 2019, pp. 1570–1573.