Efficient Model Partitioning for Distributed Model Transformations

Amine Benelallam, Massimo Tisi, Jesús Sánchez Cuadrado, Juan De Lara, Jordi Cabot

Efficient Model Partitioning for Distributed Model Transformations

Amine Benelallam
Massimo Tisi
AtlanMod team
(Inria, Mines Nantes, LINA), France
{amine.benelallam, massimo.tisi}
@inria.fr

Jesús Sánchez Cuadrado
Juan de Lara
Universidad Autónoma
de Madrid, Spain
{Jesus.Sanchez.Cuadrado,
Juan.deLara}@uam.es

Jordi Cabot
ICREA
Open University
of Catalonia, Spain
jordi.cabot@icrea.cat

Abstract

As the models that need to be handled in model-driven engineering grow in scale, scalable algorithms for model transformation (MT) are becoming necessary. Programming models such as MapReduce or Pregel may simplify the development of distributed model transformations. However, because of the dense inter-connectivity of models and the complexity of transformation logics, scalability in distributed model processing is challenging.

In this paper, we adapt existing formalization of uniform graph partitioning to the case of distributed MTs by means of binary linear programming. Moreover, we propose a data distribution algorithm for declarative model transformation based on static analysis of relational transformation rules. We first extract footprints from transformation rules. Then we propose a fast data distribution algorithm, driven by the extracted footprints, and based on recent results on balanced partitioning of streaming graphs. To validate our approach, we apply it to an existing distributed MT engine for the ATL language, built on top of MapReduce. We implement our heuristic as a custom split algorithm for ATL on MapReduce and we evaluate its impact on remote access to the underlying backend.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/ Specifications—Languages, Tools; C.2.4 [*Distributed Systems*]: Distributed applications

General Terms Languages, Performance

Keywords Model Transformation, ATL, Data Distribution, Static Analysis, MapReduce

1. Introduction

The Model-Driven Engineering (MDE) paradigm has been successfully embraced in several domains, for manufacturing maintainable software while decreasing cost and effort. For instance, recent works have shown its benefits in applications for the construction industry [26] (for communication of building information and interoperation with different tools and actors), modernization of legacy systems [3] (for aiding the migration of large legacy codebases into novel versions meeting particular requirements), learning and big data analytics [9] (for reducing the expertise necessary to implement probabilistic models for machine learning, and speed up development).

Models are the central artefacts in MDE, and Model Transformations (MTs), written in dedicated transformation languages, are the prominent means for automatically manipulating them, e.g. for translation, refinement or code generation. The *relational* paradigm is the most popular among MT languages, based on the definition of rules relating input and output model elements. OMG's QVT and AtlanMod's ATL [17] are the most widespread examples of relational MT languages. However, recent application scenarios like the ones we mentioned are characterized by big amounts of data, that may be represented by very large models (i.e. models with millions of elements). Against models of such size, current execution engines for MT languages easily hit the limit of the resources (IO/CPU) of the underlying machine.

The wide availability of distributed clusters on the Cloud[1] makes distributed computation an accessible solution for transforming large models. A typical data-parallel approach to distribute a model transformation across a cluster involves splitting the input model into chunks, and assigning them

[1] On-demand computing resources dedicated to processing, accessing, and storing data on the Internet

to different machines, following a data-distribution scheme. Each machine is then responsible for transforming a part (i.e. split) of the input model.

Popular distributed programming models like MapReduce [8] may simplify the development of such solutions. However, the characteristics of MT make efficient parallelization challenging. For instance, typical MapReduce applications work on flat data structures (e.g. logs) where input entries can be processed independently and with a similar cost. Hence, a simple data distribution scheme can perform efficiently. Conversely, models are usually densely interconnected, and MTs may contain rules with very different complexity. Moreover, most of the computational complexity of MT rules lies in the pattern matching step, i.e. the exploration of the graph structure. Because of this, model transformations witness higher ratio of data access to computation w.r.t. typical scientific computing applications. With a naive data-distribution scheme, the execution time can be monopolized by the wait time for model elements lookup. With an unbalanced distribution, machines with light workload have to wait for other machines to complete. At other times, machines with heavy workload can crash when hitting their memory limit. Such challenges are not only limited to model transformations, but extend to several distributed graph processing tasks [18].

Several task-parallel [5] and data-parallel [21] distribution techniques have been proposed, but none of them addresses the specific case of relational model transformation. In this paper we argue that when a MT is defined in a relational language, an efficient data-distribution[2] scheme can be derived by statically analysing the structure of the transformation rules.

From static analysis, we derive information on how the input model is accessed by the transformation application, and we encode it in a set of so-called *transformation footprints*. Footprints allow us to compute an approximation of the transformation (task-data) dependency graph. We exploit this graph in a data distribution strategy, minimizing the access to the underlying persistence backend, and hence, improving the performance of our model transformation execution and memory footprint.

We adapt existing formalization of uniform graph partitioning to distributed MTs using binary linear programming. The formalization takes into account task-data overlapping maximization based on the dependency graph. Given a cluster of commodity machines, the objective is to minimize the amount of loaded elements in each of these machines. Then, we propose an algorithm for the extraction of transformation footprints based on the static analysis of the transformation specification. Finally, we propose a fast greedy data-distribution algorithm, which partitions an input model over a cluster, based on the extracted footprints.

To validate our approach, we apply it to a distributed MT engine, ATL-MR [2], that we built in previous work. ATL-MR distributes ATL transformations on top of MapReduce. ATL-MR includes a distributed persistence backend [13] built on-top of HBase [23]. We build our footprints extraction algorithm on top of anATLyzer [7], a static analysis tool for model transformation in ATL. Finally, we implement our heuristic as a custom split algorithm for Hadoop/HBase applications. Execution performance in our experiments is consistently faster (up to 16%) than a random distribution approach.

The rest of the paper is structured as follows. Section 2 introduces the ATL language, and our motivation example. Section 3 describes our system and formalizes the problem. Section 4 introduces our footprints extraction algorithm, and our greedy algorithm for data distribution. Section 5 describes our implementation and discusses experimentation results. Section 6 discusses our approach, and exhibits its limitations. Finally Section 7 compares with the main related works, while Section 8 concludes the paper and draws some future work.

2. Motivating Example

2.1 The ATL Transformation Language

In order to illustrate the ATL transformation language, and motivate the need for efficient data distribution, we use the Class2Relational transformation, a de-facto standard benchmark for MT. Given a class diagram, this transformation generates the corresponding relational model. A subset of the transformation rules as well as an excerpt of its OCL queries (helpers) are shown in Listing 1 and 2 respectively. The full transformation code can be found on the paper's website[3].

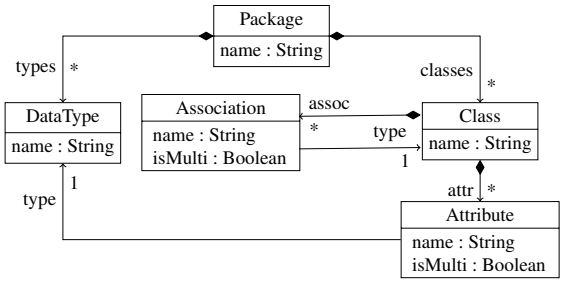
Source and target metamodels of the transformation are shown in Fig. 1. A *Package* (see Fig. 1a) contains *classes* and *types*. A Class in its turn, contains *attributes* and *associations*. An attribute has a *type*, and can be of cardinality *multiValued*. Same for *Associations*. In the Relational metamodel (see Fig. 1b) a *Schema* contains *tables* and *types*. A *Table* contains *columns*, and has *keys*, which can either be a primary or foreign key. Finally, a column has a *type*.

While in this paper we will focus exclusively on the ATL language, our approach can be applied to the whole family of relational MT languages. Relational MT languages are structured in a set of transformation rules encapsulated in a transformation unit. These transformation units are called *modules* in ATL (Listing 1, line 1). The query language used in ATL is the OMG's Object Constraints Language (OCL) [20]. A significant subset of OCL data types and operations is supported in ATL.

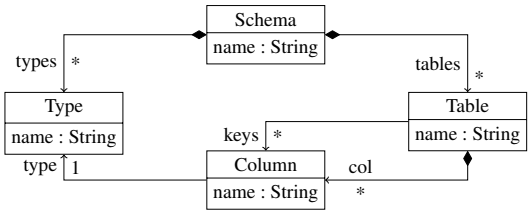
ATL *matched rules* are composed of a source pattern and a target pattern. Both source and target patterns might contain one or many pattern elements. Input patterns are fired auto-

[2] a.k.a. data partitioning

[3] `https://github.com/atlanmod/ATL_MR/`



(a) Class metamodel excerpt



(b) Relational metamodel excerpt

Figure 1: Simplified Class and Relational metamodels

matically when an instance of the source pattern (a match) is identified, and produce an instance of the corresponding target pattern in the output model. Implicitly, transient tracing information is built to associate input elements to their correspondences in the target model.

Source patterns are defined as OCL *guards* over a set of typed elements, i.e. only combinations of input elements satisfying that guard are matched. In ATL, a source pattern lays within the body of the clause *from* (Listing 1, line 14).

Listing 1: Class2Relational - ATL transformation rules (excerpt)

```
1  module Class2Relational;
2  create OUT : Relational from IN : Class;
3  rule Package2Schema {
4    from
5      p : Class!Package
6    to
7      s : Relational!Schema (
8        tables <- p.classes->reject(c | c.isAbstract)
9         ->union(p.getMultivaluedAssocs),
10       types <- p.types
11     )
12 }
13 rule Class2Table {
14   from
15     c : Class!Class (not c.isAbstract)
16   to
17     out : Relational!Table (
18       col <- Sequence{key}
19        ->union(c.attr->select(e|not e.multiValued))
20        ->union(c.assoc->select(e|not e.multivalued)),
21       keys <- Sequence{key}
22        ->union(c.assoc->select(e|not e.multivalued))
23     ),
24     key : Relational!Column (
25       name <- c.name+'objectId',
26       type <- thisModule.getObjectIdType
27     )
```

Listing 2: Class2Relational - OCL helpers (excerpt)

```
1  helper context Class!Package def :
2  getMultivaluedAssocs :Sequence(Class!Association) =
3    self.classes ->reject(c | c.isAbstract)
4      ->collect(cc| cc.assoc->select(a| a.multiValued))
5      ->flatten();
6
7  helper def: getObjectIdType : Class!DataType =
8    Class!DataType.allInstances()
9      ->select(e | e.name = 'Integer')->first();
```

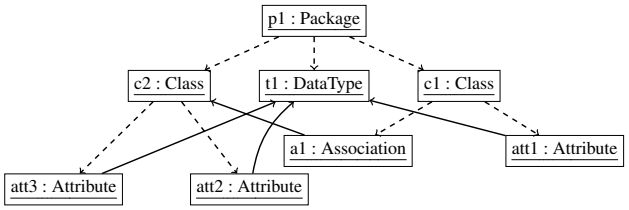


Figure 2: Sample Class model

For instance, in the rule *Class2Table*, the source pattern (Listing 1, line 15) matches an element of type *Class* that is not *abstract*. The output patterns, delimited by the clause *to* (Listing 1, line 16) describe how to compute the model elements to produce when the rule is fired, starting from the values of the matched elements. For example, the *Class2Table* rule produces two elements. The first one represents the *Table*, while the second one represents its *key*. A set of OCL *bindings* specify how to fill each of the features (attributes and references) of the produced elements. The binding at line 18 copies the name of the *Class*, the binding at line 21 computes its corresponding columns (*col*). The rule for transforming packages, generates a schema, and populates its corresponding tables and types (Listing 1, lines 3-12).

OCL helpers enable the definition of reusable OCL expressions. An OCL helper must be attached to a context, that can be a type or global context. Since target models are not navigable, only source types are allowed. Listing 2 shows a simple helper example *getObjectIdType*. It has as context the transformation *module* itself and returns *DataType* (line 7). The second helper instead has as context a *Package* and returns a list of all multivalued associations belonging to the package's classes (line 1).

2.2 Model Partitioning by Example

A source model that conforms to the *Class* metamodel is illustrated in Fig. 2. We refer to containment references using the dashed arrows, and type references by continuous arrows. The model is composed of a package (*p1*) containing two classes (*c1* and *c2*). Table 1 lists, for each source element, the set of elements that the ATL engine will need to read in order to perform its transformation (*Dependencies*). We denote by $d_{(j)}$ the set of dependencies of j, as referred by the column *Dependencies* in Table 1. For instance, $d_{(a1)} = \{a1, c2\}$.

Table 1: Model elements dependencies

MODEL ELMT.	DEPENDENCIES
p1	{p1,c1,a1,c2,t1}
c1	{c1, a1, att1, t1)}
a1	{a1,c2}
att1	{att1,t1}
c2	{c2, att2, att3, t1}
att2	{att2,t1}
att3	{att3,t1}
t1	{t1}

Considering a system $\mathcal{S}$ of 2 machines (m_1 and m_2) computing in parallel a model transformation. A set of input model elements is assigned to each machine. We argue that a random assignment strategy could result in unfavourable performance. Nonetheless, using an adequate data distribution strategy to the cluster nodes, it is possible to optimize the amount of loaded elements, and hence, improve the performance of the model transformation execution.

We consider a random uniform distribution scenario[4] of our sample model over $\mathcal{S}$. We can assume that this model is stored in a distributed persistence backend, permitting the loading of model elements on-demand. We randomly assign elements {p1,att1,c2,t1} to m_1, and the remaining to m_2. The splits assigned to m_1 and m_2 are denoted by $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. We identify the weight of a split assigned to a machine i ($(W)_i$), as the number of elements to be loaded in order to perform its transformation. Assuming that every machine has a caching mechanism that keeps elements that have already been looked-up in memory, loading model elements is considered only once.

Therefore, $\mathcal{W}(\mathcal{A}_1)$ = $|\cup_{e\in\mathcal{A}_i} d_e|$ = |{p1, c1, a1, att1, c2, att2, att3 ,t1}| = 8. Likewise, $\mathcal{W}(\mathcal{A}_2)$ = |{c1, a1, att1, c2, att2, att3, t1}| = 7. As noticed, both machines need almost all the model to perform the transformation of the split assigned to them, resulting in a overall remote access to the underlying persistence backend of 15 (=8+7) elements. However, this naive distribution scenario leads to the worst overall remote access. As the elements {p1,c1,a1,att1} share a heavy dependency between each other, it makes more sense to assign them to the same machine (e.g. m_1). The rest are hence assigned to m_2. This scenario results in better weights. Precisely, $\mathcal{W}(\mathcal{A}_1)$ = 6 and $\mathcal{W}(\mathcal{A}_2)$ = 4, with an overall remote access of 10 (6+4).

Computing such efficient split becomes especially challenging when we consider model graphs with million of nodes. Without a fast heuristic, the cost of traversing these large graphs to compute the splits may overcome any possible benefit of data distribution.

[4] Note that the order in which model elements are transformed does not affect the execution of the transformation.

3. System Formalization

In this section we show how an efficient distribution of a model transformation in a distributed system could be realized to improve the performance of a model transformation execution. We first start by describing our system and introducing some definitions, and environmental assumptions, then we propose an adapted formalization of our system in linear programming. In the rest of the paper, $\mathcal{E}$ denotes a set of model elements, and $\mathcal{M}$ a set of commodity machines in a distributed system $\mathcal{S}$.

3.1 Environmental Assumptions

Distributed Model Transformations

In typical relational MT engines (e.g., the standard ATL and ETL engines), the transformation execution starts by loading the input model. Then the engine applies the transformation by selecting each rule, looking for matches corresponding to the input pattern, and finally generating the appropriate output elements [12]. Each execution of a matched input pattern is called a rule application.

DEFINITION 1. *Let $\mathcal{R}$ be a set of model transformation rules. A rule application is defined by the tuple $(e, rule, d_e)$, where:*

- *$e \in \mathcal{E}$, is the element triggering the execution of the rule application*
- *$rule \in \mathcal{R}$, is the rule whose input pattern matches the element e*
- *$d_e \subseteq \mathcal{E}$, the subset of model elements needed to execute the rule application triggered by e*

Given Definition 1, we consider a MT execution job as the union of elementary rule application execution jobs, where each job is responsible for transforming a single model element. In case of rules with n-ary input pattern (matching a subgraph), we consider the job of applying the rule to be primarily triggered by one input pattern element (e in Definition 1). A rule application as defined in Definition 1 may occur more than once, since a model element might be triggering more than one rule application. Selecting a primary triggering element for each rule application ensures that, after distributing the source model, a rule application occurs in only one machine, i.e. the one responsible for transforming the triggering element.

The distribution of a transformation based on a data-parallel distribution approach over m machines ($m = |\mathcal{M}|$), consists in dividing the input model into m splits, and assigning disjoint submodels of the input model to different machines. Each machine will be then responsible for transforming the assigned subset. In what follows we refer to this set of elements assigned to a machine i by $\mathcal{A}_i$. Given a system $\mathcal{S}$ of m machines, the set of assigned elements has the following property:

PROPERTY 1. *Each element $e \in \mathcal{A}_i$ is assigned to one and only one set ($\bigcap_{i \in \mathcal{M}} \mathcal{A}_i = \emptyset$)*

We also define as $\mathcal{D}_i$, i.e. dependencies of $\mathcal{A}_i$, the set of source elements that need to be accessed to compute the transformation of $\mathcal{A}_i$. This set is determined as the union of the subsets of model elements needed to execute the rule applications triggered by every element e assigned to machine i:

$$\mathcal{D}_i = \bigcup_{e \in \mathcal{A}_i} d_e$$

The presence of $\mathcal{D}_i$ is mandatory for the transformation of the assigned model elements $\mathcal{A}_i$. Consequently, every machine i needs to load all the elements $\mathcal{L}_i$ belonging to $\mathcal{A}_i \cup \mathcal{D}_i$. Our objective is to balance the charge of loaded elements while maximizing the overlapping of dependencies. This formalization is similar to a well-known problem on graphs, *Graph-Based Data Clustering with Overlaps* [11]. This problem allows clusters overlapping by duplicating (to a particular extent) graph vertices or edges. In our system, the overlapping elements w.r.t. to i are $\mathcal{O}_i = \mathcal{L}_i \setminus \mathcal{A}_i$ (Property 1).

Distributed Model Persistence

Our work assumes the existence of an underlying model-persistence framework that is well-suited for distributed model transformation. We define the properties of such framework:

PROPERTY 2. on-demand loading *to ensure that only needed elements are loaded, especially when the model is too big to fit in memory*

PROPERTY 3. fast look-up of already loaded elements*, this can be implemented using caching and/or indexing mechanisms*

PROPERTY 4. concurrent read/write *to permit different machines accessing and writing into the persistence backend simultaneously*

Balanced Partitioning Formalization

As discussed in our definition, the set of elements to be loaded ($\mathcal{A}_i \cup \mathcal{D}_i$) by a machine i is controlled by the elements to be assigned to i for transformation. The number of loaded elements is determined by the cardinality of $|\mathcal{A}_i \cup \mathcal{D}_i|$. Intuitively we want elements that share most dependencies to be assigned to the same split. This implies minimizing model elements being loaded in different machines, and perform it only once in a single machine (Property 2). Hence, by assigning elements to machines in an appropriate manner, it is possible to reduce the amount of elements to be loaded by each machine. On the other side, it is also important to balance the load across the cluster in order to enable a good scalability of the transformation application.

Balanced graph partitioning is a problem that consists in splitting the graph into partitions of about equal weight, while minimizing the number of cross-partitions edges. In our approach, we define this weight as the cardinality of the set of elements fetched from the persistence backend, and needed to transform the assigned subset. This set is computed using the *transformation dependency graph*. The *transformation dependency graph* is the directed graph representing the rule application dependency between model elements. Each element e, has outgoing edges towards all the elements belonging to his set of needed elements d_e.

Finally, we determine the weight associated to an element e and a machine i, respectively as:

$$\begin{aligned} \mathcal{W}(e) &= |e \cup (d_e)| \\ \mathcal{W}(\mathcal{A}_i) &= |\mathcal{L}_i| = |\mathcal{A}_i \cup \mathcal{D}_i| \end{aligned} \tag{1}$$

By the formalization above, our objective is to find a model partitioning instance that balances $\mathcal{W}(\mathcal{A}_i)$ for $i \in \mathcal{M}$ by maximizing $\mathcal{O}_i$.

3.2 Balanced Model Partitioning for Distributed Model Transformation

As discussed previously, our objective is to assign model elements in a way to reduce (minimize) the amount of elements loaded in each machine. As rule applications share transformation dependency to model elements, one way to achieve this objective is by minimizing the amount of loaded elements while maximizing the overlapping of transformation dependencies. Consider the transformation of a model of size n ($n = |\mathcal{E}|$) over $\mathcal{S}$ of m machines. Our problem can be defined in linear programming as follows:

$$minimize \quad f(X) = \max_{i \in \mathcal{M}} \Big\{ \sum_{j=1}^{n} \big(\bigvee_{j \in \mathcal{E}} (X_{i,j} \times \mathcal{D}_j) \big) \Big\} \tag{2}$$

where,

$$X_{i,j} = \begin{cases} 0 & \text{if j is not assigned to the machine i ,} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

and,

$$D_{j,k} = \begin{cases} 0 & \text{if j does not need element k for its transformation,} \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

subject to:

$$\forall j \in \mathcal{E}, \sum_{i \in \mathcal{M}} X_{i,j} = 1, \tag{5}$$

$$\mid \frac{n}{m} \mid \leq \sum_{j=1}^{n} (X_{i,j}), \tag{6}$$

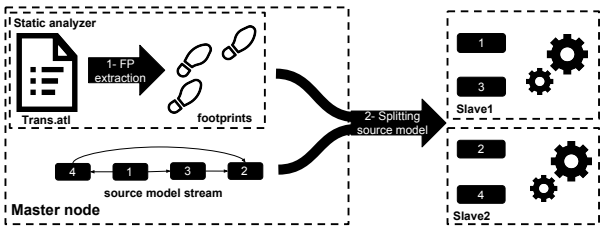


Figure 3: Execution overview of our proposed solution

and

$$| \frac{n}{m} | +1 \geq \sum_{j=1}^{n} (X_{i,j}) \quad (7)$$

In this formulation, we denote *the transformation dependency graph* by a matrix ($n \times n$), where $\mathcal{D}_{j,k}$ equals to 1 if j needs k, and 0 otherwise. As to $\mathcal{D}_j$, it is a boolean vector representing the projection of this matrix on a single element j (Equation 4). The operation $\bigvee$ plays the role of an inclusive or over the resulting vector, and returns one if the element is needed at least once, and 0 otherwise. This ensures that the loading of a given element is considered only once (as stated in Property 3). Finally, the constraint (5) is responsible for making sure that a model element is assigned to one and only one machine. Whilst constraints (6) and (7) restrict the search space of the elements assigned to each machine X_i to even number of elements over all the machines.

4. Data Partitioning for Distributed Model Transformation

Although such a problem can be solved by building a full dependency graph and using existing linear programming solvers, the computational cost is not suitable to our scenario. In case of very large graphs, the global execution time would be dominated by (i) the construction of the full dependency graph and (ii) the data distribution algorithm. In this section, we propose a greedy algorithm instead that can efficiently provide a good approximation of the ideal split.

We rely on static analysis of the transformation language to compute an approximated dependency graph. The building process of this graph relies on transformation footprints. These footprints represent an abstract view of the navigation performed by a given rule application. Footprints are a simplification of the actual computation, and they originate a dependency graph that is a super-graph approximation of the actual one. We will then apply greedy data-distribution algorithms from recent related work to the approximated dependency graph.

The execution overview of our proposed solution is depicted in Fig. 3. The framework starts by passing the transformation to the static analyser in the master node for footprints extraction. Footprints are fed to the splitting algorithm to assist its assignment decision making. As the model stream arrives, the master decides to which split a model element should be assigned. Once the stream is over, the slave nodes proceed with the transformation execution.

Table 2: Footprints of Class2Relational transformation

RULES	FOOTPRINTS
Package2Schema	Package.classes.attr Package.types
Class2Table	Class.assoc Class.attr DataType.allInstances
Attribute2Column	Attribute.type
MVAttribute2Column	Attribute.type Attribute.owner
Association2Column	DataType.allInstances
MVAssociation2Column	Association.type DataType.allInstances

4.1 Static Analysis of Model Transformation: Footprint Extraction

We analyse rule guards and bindings to extract their footprints. Since we are only interested in navigating the source model, we consider mainly *NavigationCallExp*. The extraction is achieved by traversing the tree representation of the OCL expression symbolizing either the guard or the binding. For instance, Fig. 4 shows a simplified AST representation of the getMultivaluedAssocs() helper, which is illustrated in Listing 2. Notice that the AST traversal follows a left-to-right depth-first order traversal.

Our footprints extraction process is described in Algorithm 1. It proceeds by traversing all the rules of an ATL module (line 1), and computing the footprint of each rule for both guards and bindings (lines 2..5). For each rule guard footprints and bindings footprints are aggregated. The function *extractRuleFootprint* (line 6) is the one responsible for computing the footprint of an OCL expression. It recursively traverses the AST representation of an OCL expression and returns the corresponding footprint according the node type. Fig. 4 shows an example resulting from traversing the AST tree of the *getMultivaluedAssocs* helper.

We use three different symbols to represent different operations in our algorithm. The operator $\lhd$ is used to denote navigation call chaining. It is used to chain the *source* of a *NavigationCallExp* with its corresponding referred type. For example, the footprint of the *NavigationCallExp* '*classes*' (in Fig. 4) is, $Package \lhd classes$. In our representation, the $\lhd$ is denoted by '.', as in OCL (see Table 2). Whilst, the operation $\oplus$ is used to decouple the expression to two separate footprints, one corresponding to the LHS of the operator, and the second to its RHS. This operator is mainly used in '*ConditionalExps*', and '*BinaryOperators*'. The $\otimes$ operator instead, behaves like $\lhd$ operator when the chaining is possible, otherwise it behaves as a $\oplus$ operator. A chaining is feasible when the type of the OCL expression in the LHS, corresponds to the initiating type in the RHS. For instance, the operator $\otimes$ footprint of the *IteratorCallExp collect* behaves like $\lhd$. This happens because the type of

Algorithm 1: Footprint extraction algorithm

Input : ATLModule $module$
Output : Set⟨ Set⟨ FootPrint⟩ ⟩ $perRuleFps$

foreach $rule \in$ `getRules`$(module)$ **do**
  foreach $guard \in$ `getGuards`$(rule)$ **do**
    $perRuleFps[rule] \cup$ {`extractFootprint`$(guard)$}
  foreach $binding \in$ `getBindings`$(rule)$ **do**
    $perRuleFps[rule] \cup$ {`extractFootprint`$(binding)$}

Function `extractFootprint` *(OCLExp exp)*

  if `isInstanceOf`$(exp, LiteralExp)$ **then**
    if `hasSource`(exp) **then**
      $fps :=$ `extractFootprint`$(exp.source)$
    else
      $fps := \emptyset$
  if `isInstanceOf`$(exp, NavigationOrAttributeCallExp)$ **then**
    if `isAttributeCall` *(exp)* **then**
      $fps := \emptyset$
    else if `isHelperCall`(exp) **then**
      $helper :=$ `getHelperByName`$(exp.name)$
      $fps :=$ `extractFootprint`$(exp.source) \otimes$ `extractFootprint`$(helper)$
    else //isNavigationCall
      $fps :=$ `extractFootprint`$(exp.source) \triangleleft exp.refferedProperty$
  if `isInstanceOf`$(exp, OperatorCallExp)$ **then**
    $fps :=$ `extractFootprint`$(exp.firstOperand)$
    if `isBinaryOperator`(exp) **then**
      $fps := fps \oplus$ `extractFootprint`$(exp.secondOperand)$
  if `isInstanceOf`$(exp, OperationCallExp)$ **then**
    $fps :=$ `extractFootprint`$(exp.source)$
  if `isInstanceOf`$(exp, VariableExp)$ **then**
    if `hasContainer`(exp) **then**
      $fps :=$ `getType`(exp)
    else
      $fps := \emptyset$
  if `isInstanceOf`$(exp, IteratorExp)$ **then**
    $fps :=$ `extractFootprint`$(exp.source) \otimes$ `extractFootprint`$(exp.body)$
  if `isInstanceOf`$(exp, IfExp)$ **then**
    if `hasElseClause`(exp) **then**
      $fps :=$ `extractFootprint`$(exp.cond) \otimes$ (`extractFootprint`$(exp.then) \oplus$ `extractFootprint`$(exp.else)$)
    else
      $fps :=$ `extractFootprint`$(exp.cond) \otimes$ `extractFootprint`$(exp.then)$

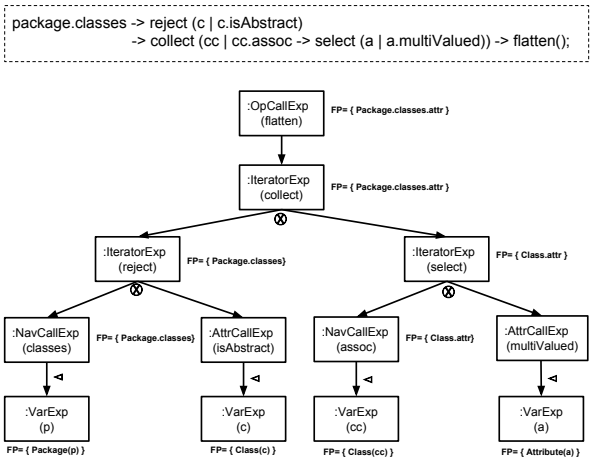


Figure 4: Simplified AST representation helper 'getMultivaluedAssocs' augmented with footprint values. For simplicity sake, we add the helper's definition at the top.

the collection of the footprint '*Package.classes*' (in LHS), coincides with the initiating type of the footprint '*Class.attr*' (in RHS). As noticed, the algorithm does not cover all the set of *OCLExpressions*. Operations on collection types such as *flatten()*, *first()*, or *last()*, are omitted as they don't affect the computation of the transformation footprints.

The set of footprints resulting after using our algorithm in the *Class2Relational* transformation is summarized in Table 2. This set is organized by rules. We represent footprints as a series of *NavigationCallExpression* starting from a particular Type. They take the form of $[source.Type][.'('?[propertyName][')'?*]?]^+$. For instance, the binding '*tables*' in rule *Package2Schema* contains a multi-level navigation '*{Package.classes.attr}*'. A chunk of the footprint can take part in a recursive call. We use '$*$' to denote it. We argue that multi-step navigation calls provide a better approximation of the dependency graph. The same argument holds for recursive navigation calls.

4.2 A Greedy Model-Partitioning Algorithm for Distributed Transformations

Although the footprints of the transformation help us approximate a transformation dependency graph, data distribution necessitates traversing the model according to these footprints. This traversal process can be expensive as it may possibly visit model elements more than once. In our approach, we propose a greedy partitioning algorithm which visits every model element only once. For each element, we only visit the first degree neighbours as stated by the footprints with initiating type. Thanks to this algorithm, it is possible to instantly decide to which machine a model element should be assigned.

We first decompose all existing multi-level navigation footprints into single-navigation footprints in order to visit only first degree neighbours. Then, we order them according

Algorithm 2: Data distribution algorithm

```
Input : Stream<Vertex> stream, int avgSize, int bufCap, int m
1  assignElementToMachine (stream.first(), 1)
2  while stream.hasNext() do
3      element := stream.next()
4      if isHighPriorityElement(element) then
5          machineId := ComputeMachineId(element)
6          storeElement-MachineDeps(element, clusterId)
7          assignElementToMachine(element, clusterId)
8      else
9          addToBuffer(element)
10     if |buffer| == bufCap then
11         processBuffer()
12 processBuffer()
13 Function ComputeMachineId (element)
       Output : int index
14     index := max_{i∈m} (|dependentElementsInMachine(element, i)| *
       (1 − currentSize(i)/avgSize))
15 Function storeElement-MachineDeps (element, clusterId)
16     foreach
       fp ∈ getFirstOrderFPsForType(element.type) do
17         dependent_elements :=
           getPropertyElements(element, fp.property) →
           reject(alreadyAssigned)
           addDependencies(clusterId,
           dependent_elements)
18     foreach fp ∈ getFPsFromDeps(element, clusterId)
       do
19         dependent_elements :=
           getPropertyElements(element, fp.property) →
           reject(alreadyAssigned)
           addDependencies(clusterId,
           dependent_elements)
20 Function processBuffer ()
21     while buffer.hasNext() do
22         element := buffer.next()
23         clusterId := ComputeMachineId(element)
24         storeElement-MachineDeps(element, clusterId)
25         assignElementToMachine(element, clusterId)
```

to the navigation call chain in the original footprint. Single-level footprints have by default an order equals to 1. For example , the footprint {Package.classes.attr} decomposes to two single-level footprints with the corresponding order, FP_1 = {{Package.classes,1}, {Class.attr,2}}. Later, we show how the order of footprints is used to enable visiting only first degree neighbours.

The initiating type of the resulting footprints is inferred from the navigation call referring type. The order of the decomposed footprints is important, it enables processing the right single footprint, and discard the succeeding ones. Moreover, it allows keeping track of the next footprint to be taken into consideration during the partitioning process. Later, we show how, by using these footprints, we monotonously build our transformation dependency graph.

It is theoretically impossible to have a good streaming graph partitioning algorithm regardless of the streaming order. The ordering on which model elements arrive, may sometimes cause it to perform poorly. Moreover, sometimes, it can be discouraged to decide to which partition a model element should be assigned. Hence, the use of a buffer to store these model elements for a future processing is favoured. The buffer should have a capacity that, once is reached (hopefully, after the master has enough information to decide to which machine these elements should be assigned), it gets cleared, and belonging elements should be processed.

To alleviate this process, we distinguish between high-priority and low-priority elements. High-priority elements are elements that can participate in building the transformation dependency graph. These elements are immediately processed and assigned to a split. Whilst, low-priority elements do not directly participate in building the transformation dependency graph. For example, the elements of a type that does not figure in the LHS of any footprints maybe considered low-priority element. Low-priority elements are stored in a buffer for further processing.

Algorithm 2 describes our greedy data distribution approach. It takes as input the graph stream, the average machine size '*avgSize*', the buffer capacity '*bufCap*', and finally the number of commodity machines '*m*'. The algorithm starts by assigning the first element to the first machine. Next, for each streamed element, if the element is high-priority (need to be assigned immediately), then it computes the index of the appropriate machine to be assigned to (line 5). Later, it stores its dependencies (line 6). Finally, the element is assigned to the given machine (line 7). If the element is low-priority, it is directly stored in the buffer (line 9).

In order to monotonously build our transformation dependency graph, the dependencies of each element are stored after its assignment. A dependency has the following structure, {element_id, next_footprint, split_id}. This reads like, the element with 'ID' *element_id*, along with the elements referred to by *next_footprint*, are needed in the split with 'ID' *split_id*. The dependencies storage process happens in two

phases (line 15). It starts by recovering all footprints with initiating type as the element type and order equals to 1 (line 16). For each footprint, we check if it has any successor. If so, the dependency is stored together with the next footprint to be resolved if the dependent element is assigned to the same machine. In case the footprint is originally single, we only store the dependency of the element to the machine. Dependencies referring to elements that have been already assigned are not created (line 16).

For example, after assigning the element 'p1', supposedly to split m_1, we collect its corresponding footprints, which are, {*Package.classes, Package.types*}. For the footprint '*Package.classes*', we generate the following dependencies: {*c1, Class.attr,* m_1} and {*c2, Class.attr,* m_1}. The elements *c1* and *c2* are resulted from visiting '*classes*' property. As for the footprint *Package.types*, only one dependency is generated, which is, {*c1, null,* m_1}. The value of *next_footprint* is null, as *Package.types* is originally single-level.

In the second step, we collect all the dependencies corresponding to the current element and the assigned split, with a non-null value of *next_footprint* (line 18). If any, additional dependencies are created, similarly to the previous step. This step enables building the dependecy graph while visiting only first degree neighbours. For example, if the element 'c1' was assigned to m_1, then, we recover the dependency {*c1, Class.attr,* m_1}, and we generate the additional dependencies. After the dependencies creation, existing dependencies referring to the current element are deleted.

The function *ComputeMachineId* (line 13) is responsible for deciding the appropriate machine. It assigns a model element to the machine with the highest number of elements depending on it, and adds it to the list of already visited elements, thanks to the dependencies generated at every model assignment. The amount of dependencies to an element e in a particular machine m refers to, the number of dependencies with *element_id* equals to e , and *split_id* equals to m. This value is weighted by a penalty function based on the average size a machine can take, and penalizing larger partitions. If the splits have similar score, then the element is assigned to the split with the lowest index. While different penalty functions can be used, we go for a linear weighted function, as it has experimentally shown good performance results [21]. This forces our distribution to be uniform. Experimenting our algorithm with different penalty functions is left for future work.

5. Validation

5.1 Implementation

As proof of concept, we use the ATL [17] (AtlanMod Transformation Language), a relational model-to-model transformation language, along with ATL-MapReduce (ATL-MR) [2], a prototype distributed engine, built on top of MapReduce [8]. MapReduce is a programming model and software framework developed at Google in 2004. It allows easy and transparent distributed processing of big data sets while concealing the complex distribution details a developer might cross. Inspired from the map and reduce primitives that exist in functional languages, MapReduce has two constructs, *Map*, and *Reduce*. Each receives a sequence of ⟨key, value⟩ pairs called records, and produces other records in response. Records are organized in splits, and a split represents a chunk of input data.

ATL-MapReduce (ATL-MR) [2] is a prototype distributed engine for running complex ATL transformations on top of MapReduce [8]. ATL-MR is implemented as an extension of an existing ATL VM [24], and runs in two steps, *LocalMatchApply* (map), and *GlobalResolve* (reduce). Each mapper is assigned a subset of model elements using a splitting process. For each model element, if it matches the input pattern of an existing rule, then a target element is created together with tracing information, and target properties corresponding to elements locally assigned are resolved. Information about non-local elements is stored in traces and sent to the reduce phase for a global resolve. This step is responsible for composing the sub-models resulting from the previous phase into a single global output model. More elaborated description of the algorithm can be found in the approach's paper [2].

ATL-MR is coupled with a decentralized persistence backend, NEOEMF/HBASE, built on on top of HBase [23] and ZooKeeper [22]. It offers a lightweight on-demand loading and efficient garbage collection and model changes are automatically reflected in the underlying storage, making changes visible to all the clients. It also supports pluggable caching mechanisms. Therefore, NEOEMF/HBASE answers perfectly to the assumptions stated in Sec. 3.1.

In order to perform an automated footprint extraction operation, we rely on a static analysis tool of ATL transformations, anATLyzer [7]. Although, it was initially implemented to automate errors detection and verification in ATL transformations, in our solution, we use anATLyzer internals, especially, typing information inference of OCL expressions, in order to construct the footprints of ATL rules.

The greedy algorithm is implemented on top of a Hadoop *TableInpuFormat* class. This class is the one responsible for partitioning HBase tables. We override its default behaviour and enable the use of our custom splitting solution. The implementation relies on the HBase storage scheme, where, ⟨key, value⟩ pairs are grouped by family and stored in increasing lexicographical order. Table splits are defined by a start key and end key. In our solution, we associate to each split an *ID*. Later on, we use this id to salt the row keys belonging to the same splits. We applied our splitting approach to the *LocalResolve* phase of ATL-MR algorithm. The Global resolve uses a random assignment approach.

5.2 Experiments

We evaluate the performance of our proposed solution while running it on a well known model transformation,

Table 3: Remote access count (per properties) : Random Vs. Greedy

DIST. MODES	SIZES	SPLITS						
		2	3	4	5	6	7	8
Random	5000	10655	10089	8968	8087	8085	8320	8377
	10000	19871	19286	17198	18529	16390	17342	16386
	15000	31402	28582	26937	25214	24785	24617	24520
	20000	35060	35053	38325	42405	38109	35045	35045
Greedy	5000	9853 (7.5%)	8698 (13.7%)	8146 (9.17%)	8088 (-0%)	8086(-0%)	8493 (-2.0%)	8083 (3.5%)
	10000	17211 (13.3%)	16396(14.9%)	16493(4.1%)	16395(11.5%)	16673(-1.7%)	17227(0.0%)	16949(-3.4%)
	15000	27863(11.2%)	23912(16.3%)	24850(7.7%)	23908(5.1%)	23905(3.5%)	23904(2.9%)	23900(2.5%)
	20000	32855(6.29%)	32691(6.74%)	36207(5.53%)	38688(8.77%)	35232(7.55%)	32362(7.67%)	32361(7.66%)

Class2Relational. We use as input randomly generated models with diverse sizes (we provide generation seeds to make the generation reproducible). The generator is highly configurable. The default configuration takes as input the model size, the density of references, and the variation. The variation applies to both the model size, and the number of references to be generated per property. In our experiment, we used a density of 8, and a variation of 10%.

The Hadoop framework comes with tools to help analysing data being processed. This feature gathers statistics about the job, for different purposes like quality control or application level statistics. Each counter is maintained by a mapper or reducer, and periodically sent to the application master so it can be globally aggregated. Counter values are definitive only once a job is successfully completed. Hadoop allows user code to define user-defined counters, which are then incremented as desired in the mapper or reducer. We defined a Hadoop counter for globally reporting metrics about data access and update. This counter was integrated with our distributed persistence backend, NEOEMF/HBASE. It globally reports *remote access count, local access count* at the end of every successful job. We believe that such metrics are relevant as they can be decoupled of the internal complexities of both the transformation and persistence engines.

The scenario of our experimentation is described by Algorithm 3. For each model size, we generate three random models. Each execution of the transformation on the generated model, is launched with a fixed number of nodes (2..8).

Algorithm 3: Data distribution algorithm

Input : Stream<Vertex> $stream$, int $avgSize$, int $bufCap$, int m

```
foreach size ∈ sizes do
    foreach pass ∈ 1..3 do
        input ← generateInputModel(size)
        foreach map ∈ 2..8 do
            transformUsingRandomDist(input, map)
            transformUsingGreedyDist(input, map)
            cleanRound()
    deleteInputModel()
```

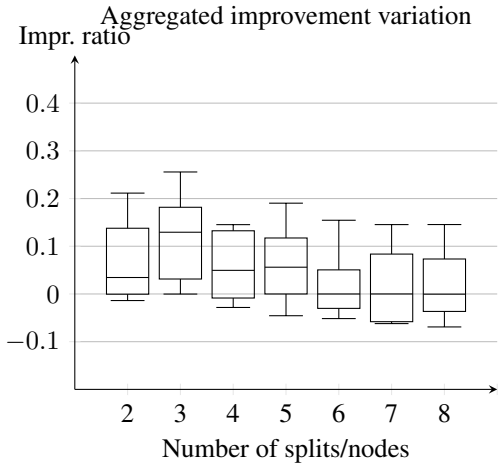


Figure 5: improvement obtained in the experiment (Table 3)

The average improvement results expressed in term of number of accesses to the underlying backend is summarized in Table 3. Each access either fetches an element or its properties. Our distributed solution shows an average improvement up to 16.7%. However, in some cases, the random distribution performed better than our greedy algorithm, mostly for the set of model with sizes 5000 and 10000. For the smallest set of models, it did not bring any improvement, in average, with 5 and 6 splits respectively.

Fig. 5 shows the performance's distribution of our algorithm per number of splits. For the 'Class2Relational' transformation and models with sizes ranging between 5000 and 20000, partitioning the model into 2 to 5 splits, guarantees better performance, where it reached up to more than 26%.

6. Discussion and Limitations

A success key to our algorithm is the approximation quality of our dependency graph. The graph structure grows gradually as model elements arrive. Each element may or may not participate in the construction process of the graph (i.e. the element has dependencies or not). The sooner (during our partitioning process) we have a good quality approximation of our dependency graph, the better splitting we can achieve.

Our partitioning algorithm is subject to various internal as well as external factors that may impact its performance. Inconveniently, these factors are completely orthogonal to each other. More importantly, the order in which elements arrive impacts the quality of the dependency graph approximation, and hence, the performance of our algorithm. Hereafter, we elaborate on the repercussions of these factors.

The choice of the appropriate parameters is crucial. On one hand, both of the average size (which implicates the number of splits) and the buffer capacity depend on the memory capacity of a single commodity machine[5]. On the other hand, the size of the buffer plays an important role in delaying the assignment of some model elements – low-priority ones. This delay is beneficial when the dependency matrix evolves from the time a model element was buffered until it was cleared from the buffer. With big enough buffer capacity, the master node can improve the decision making of the assignment to a split after having a better approximation of the dependency graph. However, the performance of the algorithm can be compromised when receiving a stream of low-priority elements in a row, until the buffer reaches its limit. In this case, the quality of the dependency graph does not improve.

Furthermore, in the default behaviour of our algorithm, we define a high-priority model element as any element that participates in the construction of the approximated dependency graph. Still, participating elements can have a different priority. The priority order can be quantified by the amount of contribution to the construction of the graph approximation. Elements with a bigger amount of dependency elements contribute more. In some cases, where the model transformation and the possible topology can be known in advance, this definition can be customized to improve the performance of our algorithm. In our running example, elements of type *Package* and *Class* can have high-priority as they are the more luckily to have a big number of dependencies. Nonetheless, we showed in our experiments that the default behaviour can bring a considerable improvement in any case.

Finally, the performance of our approach may be reduced when (i) having elements with big amounts of dependencies (sometimes exceeding the average size of the split), (ii) having approximate graph containing false positive dependencies, and (iii) having an unfavourable order of streamed elements. The first limitation is typically encountered when having complex OCL expressions starting with the operation *allInstances()* or when having recursive helpers. The second limitation happens when having OCL expressions involving several occurrences of filtering operations (*select()* or *reject*) followed by a collect. While the *collect()* operation operates only on the filtered model elements, in our approach, non-filtered elements are also traversed. In case the number of filtered elements is considerable, our algorithm reports elements as false positive. This may lead to grouping non-dependent elements in the same split.

[5] Assuming that the cluster is homogeneous

7. Related work

Performance improvement in distributed systems is a well-studied problem. Impulsed by different and yet sound motivations, different families of approaches emerge. In what follows we introduce generic related work, afterwards, we introduce existing approaches to improve model transformation and query in the MDE field. Finally, we describe other applications of static analysis of model transformation.

7.1 Task-Driven

Observing that the order on which tasks are executed influences the overall completion time, some works [6, 27, 28] proposed some scheduling heuristics to improve the system makespan, especially on MapReduce. Each one of these heuristics considers one or more specific factors such as the cost matrix, task dependency, and machines heterogeneity.

Assuming that map tasks are usually parallelizable but not reduce tasks, Zhu et al. [28] introduce an offline and bi-objective scheduler to minimize makespan and total completion time of reduce tasks. Algorithms for both of preemptive and non-preemptive have been developed proving a performant worst ratio in accordance with the number of machines in the system.

HaSTE [27] is a Hadoop YARN scheduling algorithm aiming at reducing the MR jobs' makespan based on task dependency and resource demand. The scheduler consists of two components, an initial task assignment and a real time task assignment. The first one is intended to assign the first batch of tasks for execution while the rest remains pending. A greedy algorithm using dynamic programming is used for this regard. The real time task assignment component is triggered with resource capacity update. The scheduler then decides with tasks of the remaining ones to assign considering two metrics, the resource availability and the dependency between tasks.

In other work, Chen et al. [6] a task scheduler that joins all three phases of MR process. The problem assumes that tasks are already assigned to the machines. Though it only deals with tasks execution ordering. A precedence graph among tasks is used to represent tasks that should start after others finish. The model system is formulated in linear programming and solved with column generation.

While in our approach we consider a fine-grained task definition (rule application), in these approaches, it is considered at a larger scope (map reduce tasks). Moreover, these approaches do not consider data locality and complex data dependencies. As these approaches are generic, we believe that our approach can be coupled with them.

7.2 Data-Driven

Stanton and Kliot [21] compared a set of lightweight Streaming Graph Partitioning for Large Distributed Graphs and

compare their performance to some well-known offline algorithms. They run their benchmark on large collections of datasets, and showed up to 76% of average gain.

Charalampos et al., introduce a framework for graph partitioning, FENNEL [25]. The framework uses a formulation that relaxes the hard constraint on edge cuts, and replaces it by two separate costs, the edges cut and the cluster size. The formulation provides enables the accommodation of already existing heuristics. The framework was also integrated to Apache Giraph, an open-source implementation of Pregel, a distributed graph processing framework.

Kyrola et al. describes GraphChi, large-scale graph computation system that runs on a single machine by developing an method called Parallel Sliding Windows. It process a graph with from disk, with only a limited of number of non-sequential disk accesses, while supporting the asynchronous model of computation. The approach uses graph partitioning to maximize data locality and minimize disk access counts.

The main difference characterizing our approach, is that, our partitioning formalization and algorithm are based on task-data dependency graph, while these approaches rely on the graph data itself.

7.3 Distributed Model Querying Frameworks

In [15], Izso et al. present an incremental query engine in the cloud, called IncQuery-D . This approach is based on a distributed model management middleware and a stateful pattern matcher framework using the RETE algorithm. The approach has shown its efficiency, but it addresses only distributed model queries while we focus on declarative model transformations.

Hawk [1] is a model indexing framework that enables that enables efficient global model-element-level queries on collections of models stored in file-based version control systems (VCS). It can operates with different file-based VCSs while providing a query API that can be used with by model management tools. Hawk is shipped with features enabling its robustness. Namely, incremental model updates, derived attributes indexed attributes, and scoped queries. Hawk showed improvement in querying models ranging in 22.5% and 76.4% while growing the size of the models being queried.

While both approaches focus on performance improvement of model querying, our solution is designed for improving distributed model transformations specified in relation MT languages. Moreover, both tools use built-in indexers to improve the query execution performance. Instead, in our approach we rely efficient data partition to improve the performance of transformation execution.

7.4 Static Analysis of Model Transformations

Static analysis have been used in model transformations for different purposes, such as proving properties of model transformations like the absence of rule conflicts [10], to detect errors [7] or to compute and visualize tracing information [14] to enhance maintainability.

Meta-model footprints are computed using the information gathered via static analysis. In [16] a technique to estimate the model footprint of an operation is presented. Essentially, a given model is just filtered based on the computed static meta-model footprint. In our case, the model footprint is streamed as a the original model is traversed. In [19] the type information available the abstract syntax tree of Kermeta programs is used to generate test models. [4] computes footprints of ATL rules in order to determine which rules are involved in the violation of a transformation contract.

8. Conclusion and future work

In this paper, we argue that distributed model transformations in relational languages can be improved by using efficient distribution strategies. To cope with this, we first showed that our problem can be related to the problem of uniform graph partitioning. Later, we adapted existing formalization to our system. Then, we proposed a greedy data distribution algorithm for declarative model transformations, based on the static analysis of transformation rules. We introduced an algorithm to extract the knowledge on how the input model is accessed by the transformation application. Later, we use this knowledge to help out the algorithm deciding the appropriate assignment. In our experimentation, we showed that thanks to our greedy algorithm, we improved the access to our distributed backend by up to 16%.

In our future work we intend to improve the efficiency of our distributed transformation engine by exploring the following lines:

- Extending our work to balanced edge partitioning and conducting a thorough study on the impact of the model density on the partitioning strategy.
- Improving the distribution of, not only the input model of the transformation, but also its intermediate transformation data (tracing information).
- Applying our approach to distributed model persistence for any model transformation-based application. We plan to design a model-persistence framework that would decide, at model-element creation time, which machine should host the element, by exploiting knowledge about its possible future transformations.

Acknowledgments

This work is partially supported by the MONDO (EU, Sevent Framework programme ICT-611125) project.

References

[1] K. Barmpis and D. Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 6. ACM, 2013.

[2] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributed Model-to-model Transformation with ATL on MapReduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 37–48, New York, NY, USA, 2015. ACM.

[3] H. Bruneliére, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[4] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.

[5] Y. Chawla and M. Bhonsle. A Study on Scheduling Methods in Cloud Computing. *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)*, 1(3):12–17, 2012.

[6] F. Chen, M. Kodialam, and T. Lakshman. Joint Scheduling of Processing and Shuffle Phases in MapReduce Systems. In *Proceedings of The 31rd Annual IEEE International Conference on Computer Communications*, pages 1143–1151, March 2012.

[7] J. S. Cuadrado, E. Guerra, and J. d. Lara. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 34–44, Nov 2014.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *ACM Communication*, volume 51, pages 107–113, NY, USA, 2008. ACM.

[9] B. Dominic. Towards Model-Driven Engineering for Big Data Analytics – An Exploratory Analysis of Domain-Specific Languages for Machine Learning, Jan 2014.

[10] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[11] M. R. Fellows, J. Guo, C. Komusiewicz, R. Niedermeier, and J. Uhlmann. Graph-based Data Clustering with Overlaps. *Discrete Optimization*, 8(1):2–17, 2011.

[12] C. Gomes, B. Barroca, and V. Amaral. Classification of Model Transformation Tools: Pattern Matching Techniques. In *Proceedings of 17th International Conference Model-Driven Engineering Languages and Systems*, pages 619–635. Springer International Publishing, Sep 2014.

[13] A. Gómez, A. Benelallam, and M. Tisi. Decentralized Model Persistence for Distributed Computing. In *Proceedings of 3rd BigMDE Workshop*, volume 1406. CEUR Workshop Proceedings, July 2015.

[14] V. Guana and E. Stroulia. ChainTracker, a Model-Transformation Trace Analysis Tool for Code-Generation Environments. In *ICMT*, pages 146–153. Springer, 2014.

[15] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. IncQuery-D Incremental Graph Search in the Cloud. In *Proceedings of the Workshop on Scalability in MDE*, BigMDE '13, pages 4:1–4:4, New York, NY, USA, 2013. ACM.

[16] C. Jeanneret, M. Glinz, and B. Baudry. Estimating Footprints of Model Operations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 601–610. ACM, 2011.

[17] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. Special Issue on 2nd issue of experimental software and toolkits (EST).

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[19] J. M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static Analysis of Model Transformations for Effective test Generation. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 291–300. IEEE, 2012.

[20] Object Management Group. Object Constraint Language, OCL, May, 2016. URL: `http://www.omg.org/spec/OCL/`.

[21] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[22] The Apache Software Foundation. Apache ZooKeeper, 2015.

[23] The Apache Software Foundation. Apache HBase, May, 2016. URL: `http://hbase.apache.org/`.

[24] The Eclipse Foundation. ATL EMFTVM, May, 2016. URL: `https://wiki.eclipse.org/ATL/EMFTVM/`.

[25] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

[26] R. Volk, J. Stengel, and F. Schultmann. Building Information Modeling (BIM) for Existing Buildings: Literature Review and Future Needs. *Automation in Construction*, 38(0):109–127, 2014.

[27] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi. HaSTE: Hadoop YARN Scheduling Based on Task-Dependency and Resource-Demand. In *Proceedings of 7th IEEE International Conference on Cloud Computing (CLOUD)*, pages 184–191, Jun 2014.

[28] Y. Zhu, Y. Jiang, W. Wu, L. Ding, A. Teredesai, D. Li, and W. Lee. Minimizing Makespan and Total Completion Time in MapReduce-like Systems. In *Proceedings of The 33rd Annual IEEE International Conference on Computer Communications*, pages 2166–2174, Apr 2014.