Distributed Model-to-Model Transformation with ATL on MapReduce

Amine Benelallam, Abel Gómez, Massimo Tisi
AtlanMod Team
Inria, Mines-Nantes, Lina
4 rue Alfred Kastler
44307 Nantes, France
{ amine.benelallam | abel.gomez | massimo.tisi } @inria.fr

Jordi Cabot
ICREA - UOC
Av. Carl Friedrich Gauss, 5
08860 Castelldefells, Spain
jordi.cabot@icrea.cat

ABSTRACT

Efficient processing of very large models is a key requirement for the adoption of Model-Driven Engineering (MDE) in some industrial contexts. One of the central operations in MDE is rule-based model transformation (MT). It is used to specify manipulation operations over structured data coming in the form of model graph. However, being based on computationally expensive operations like subgraph isomorphism, MT tools are facing issues on both memory occupancy and execution time while dealing with the increasing model size and complexity. One way to overcome these issues is to exploit the wide availability of distributed clusters in the Cloud for the distributed execution of MT.

In this paper, we propose an approach to automatically distribute the execution of model transformations written in a popular MT language, ATL, on top of a well-known distributed programming model, MapReduce. We show how the execution semantics of ATL can be aligned with the MapReduce computation model. We describe the extensions to the ATL transformation engine to enable distribution, and we experimentally demonstrate the scalability of this solution in a reverse-engineering scenario.

Keywords

Model Transformation, Distributed Computing, MapReduce, ATL, Language Engineering

1. INTRODUCTION

Model-Driven Engineering (MDE) is gaining ground in industrial environments, thanks to its promise of lowering software development and maintenance effort [17]. It has been adopted with success in producing software for several domains like civil engineering [31], car manufacturing [19] and modernization of legacy software systems [4]. Core concepts of MDE are the centrality of (software, data and system) models in all phases of software engineering and the automation of model processing during the software life-cycle.

Model Transformation (MT) languages have been designed to help users specifying and executing these model-graph manipulations. They are often used in implementing tooling for software languages, especially domain-specific [33], e.g. in reverse engineering [4]. The AtlanMod Transformation Language (ATL) [16] is one of the most popular examples among them, and a plethora of transformations exist addressing different model types and intentions[1].

Similarly to other software engineering approaches, MDE has been recently facing the growing complexity of data and systems, that comes in MDE in the form of Very Large Models (VLMs) [8]. For example, the Building Information Modeling language (BIM) [31] contains a rich set of concepts (more than eight hundred) for modeling different aspects of physical facilities and infrastructures. A building model in BIM is typically made of several gigabytes of densely interconnected graph nodes. Existing MDE tools, including MT engines, are based on graph matching and traversing techniques that are facing serious scalability issues in terms of memory occupancy and execution time. This stands especially when MT execution is limited by the resources of a single machine. In the case study that we selected for our experimentation, we show how typical MT tasks in the reverse-engineering of large Java code bases take several hours to compute in local.

One way to overcome these issues is exploiting distributed systems for parallelizing model manipulation (processing) operations over computer clusters. This is made convenient by the recent wide availability of distributed clusters in the Cloud. MDE developers may already build distributed model transformations by using a general-purpose language and one of the popular distributed programming models such as MapReduce [10] or Pregel [22]. However such development is not trivial. Distributed programming (i) requires familiarity with concurrency theory that is not common among MDE application developers, (ii) introduces a completely new class of errors w.r.t. sequential programming, linked to task synchronization and shared data access, (iii) entails complex analysis for performance optimization.

In this paper we show that ATL, thanks to its specific level of abstraction, can be provided with semantics for *implicit distributed execution*. As a rule-based language, ATL allows the declarative definition of correspondences and data flows between elements in the source and target model. By

[1]The Eclipse Foundation, ATL transformation zoo, `http://www.eclipse.org/atl/atlTransformations/`, visited on June 2015

our proposed semantics, these correspondence rules can be efficiently run on a distributed cluster. The distribution is implicit, i.e. the syntax of the MT language is not modified and no primitive for distribution is added. Hence developers are not required to have any acquaintance with distributed programming.

The semantics we propose is aligned with the MapReduce computation model, thus showing that rule-based MT fits in the class of problems that can be efficiently handled by the MapReduce abstraction. We demonstrate the effectiveness of the approach by making an implementation of our solution publicly available[2] and by using it to experimentally measure the speed-up of the transformation system while scaling to larger models and clusters. We identify specific properties of the ATL language that make the alignment possible and the resulting solution efficient. In future work we plan to study how our approach may be generalized to other MT languages (e.g. QVT [25] and ETL [18]) that share some properties with ATL.

The interest of this work extends also outside the MDE community, as we perform the first steps for proposing the rule-based MT paradigm as a high level abstraction for data transformation on MapReduce. High-level languages have already been proposed for data querying (as opposed to data transformation) on MapReduce [7, 26, 29]. In the MapReduce context, they allow for independent representation of queries w.r.t. the program logic, automatic query optimization and maximization of query reuse [21]. We want to extend this approach, aiming at obtaining similar benefits in data transformation scenarios.

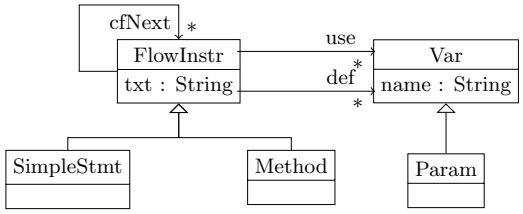
The paper is structured as follows. Section 2 describes the running case of the paper, and uses it to introduce the syntax of ATL and its execution semantics. Section 3 describes the distributed execution of ATL over MapReduce. Section 4 details the implementation of our prototype engine. Section 5 discusses the evaluation results of our solution. Section 6 discusses the main related works, while Section 7 wraps up the conclusions and future works.

2. RUNNING EXAMPLE: MODEL TRANSFORMATION FOR DATA-FLOW ANALYSIS

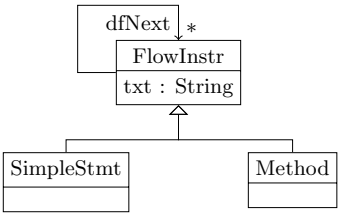
While our distribution approach is applicable to model transformations in any domain, to exemplify the discussion we refer throughout the paper to a single case study related to the analysis of data-flows in Java programs. The case study is well-known in the MDE community, being proposed by the Transformation Tool Contest (TTC) 2013 [13] as a simple but computationally demanding benchmark for MT engines. We focus on one phase of the scenario, the transformation of the control-flow diagram of a Java program into a data-flow diagram. Such task would be typically found in real-world model-driven applications on program analysis and reverse engineering of legacy code [4].

Excerpts of the source and target metamodels of this step are shown in Fig. 1. In a control-flow diagram (Fig. 1a), a *FlowInstruction* (FlowInstr) has a field *txt* containing the textual code of the instruction, a set of variables it defines or writes (*def*), and a set of variables it reads (*use*). A *FlowInstruction* points to the potential set of instructions that may

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



(a) ControlFlow metamodel excerpt



(b) DataFlow metamodel excerpt

Figure 1: Simplified ControlFlow2DataFlow metamodels

be executed after it (*cfNext*). *Method* signatures and *SimpleStatements* (SimpleStmt) are kinds of *FlowInstruction*. A *Parameter* is a kind of *Variable* that is defined in method signatures.

The data-flow diagram (Fig. 1b) has analogous concepts of *FlowInstruction*, *Method* and *SimpleStatements* but a topology based on the data-flow links among instructions (*dfNext*). For every flow instruction n, a *dfNext* link has to be created from all nearest control-flow predecessors m that define a variable which is used by n. Formally [13]:

$$\begin{aligned} m \rightarrow_{dfNext} n \iff\ & def(m) \cap use(n) \neq \emptyset \ \wedge \\ & \exists\ Path\ m = n_0 \rightarrow_{cfNext} \ldots \rightarrow_{cfNext} n_k = n : \\ & (def(m) \cap use(n)) \backslash \Big(\bigcup_{0<i<k} def(n_i) \Big) \neq \emptyset \end{aligned} \tag{1}$$

Fig. 2 shows an example of models for each metamodel, derived from a small program calculating a number factorial. As it can be seen in the figure, the transformation changes the topology of the model graph, the number of nodes and their content, and therefore can be regarded as a representative example of general transformations. In this paper we refer to an ATL implementation of the transformation named *ControlFlow2DataFlow* and available at the article website[2].

Model transformations in ATL are unidirectional. They are applied to read-only source models and produce write-only target models. ATL developers are encouraged to use declarative rules to visualize and implement transformations. Declarative rules abstract the relationship between source and target elements while hiding the semantics dealing with rule triggering, ordering, traceability management and so on. However, rules can be augmented with imperative sections to simplify the expression of complex algorithms. In this paper, we focus on declarative-only ATL.

Languages like ATL are structured in a set of transformation rules encapsulated in a transformation unit. These transformation units are called *modules* (Listing 1, line 1). The query language used in ATL is the OMG's Object Con-

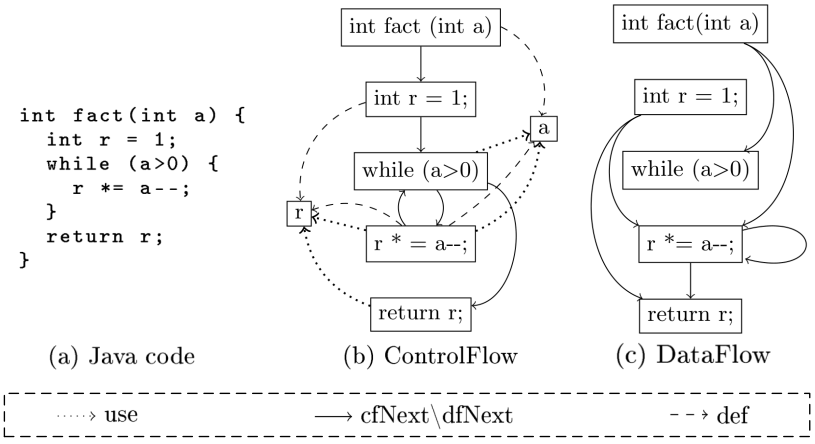


Figure 2: ControlFlow2DataFlow transformation example

straints Language (OCL) [24]. A significant subset of OCL data types and operations is supported in ATL. Listing 1 shows a subset of the rules in the *ControlFlow2DataFlow* transformation and Listing 2 an excerpt of its OCL queries (helpers).

ATL *matched rules* are composed of a source pattern and a target pattern. Both of source and target patterns might contain one or many pattern elements. Input patterns are fired automatically 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 input elements satisfying that guard are liable. In ATL, a source pattern lays within the body of the clause '*from*' (Listing 1, line 14). For instance, in the rule *SimpleStmt*, the source pattern (Listing 1, line 15) matches an element of type *SimpleStmt* that defines or uses at least a variable. Output patterns, delimited by the clause

Listing 1: ControlFlow2DataFlow - ATL transformation rules (excerpt)

```
1  module ControlFlow2DataFlow;
2  create OUT : DataFlow from IN : ControlFlow;
3  rule Method  {
4    from
5      s : ControlFlow!Method
6    to
7      t : DataFlow!Method (
8        txt <- s.txt,
9        dfNext <- s.computeNextDataFlows()
10     )
11 }
12
13 rule SimpleStmt {
14   from
15     s : ControlFlow!SimpleStmt (not(s.def->
16           isEmpty() and s.use->isEmpty()))
17   to
18     t : DataFlow!SimpleStmt (
19       txt <- s.txt,
20       dfNext <- s.computeNextDataFlows()
21     )
22 }
```

Listing 2: ControlFlow2DataFlow - OCL helpers (excerpt)

```
1  helper Context ControlFlow!FlowInstr def :
       ↪computeNextDataFLows() : Sequence (
       ↪ControlFlow!FlowInstr) =
2      self.def ->collect(d | self.users(d)
3           ->reject(fi | if fi = self then   not
                ↪fi.isInALoop else false endif )
4           ->select(fi | thisModule.isDefinedBy(
                ↪fi,Sequence{fi},self, Sequence
                ↪{}, self.definers(d)->excluding(
                ↪ self))))
5           ->flatten();
6
7  helper def : isDefinedBy(start : ControlFlow!
       ↪FlowInstr, input : Sequence(ControlFlow!
       ↪FlowInstr), end : ControlFlow!FlowInstr,
       ↪visited :Sequence(ControlFlow!FlowInstr),
       ↪ forbidden : Sequence(ControlFlow!
       ↪FlowInstr)) : Boolean =
8          if input->exists(i | i = end) then true
9          else let newInput : Sequence(
                ↪ControlFlow!FlowInstr) = input ->
                ↪collect(i |i.cfPrev) ->flatten()
                ↪->reject(i | visited ->exists(v |
                ↪ v = i) or forbidden ->exists(f|
                ↪f = i)) in
10           if newInput ->isEmpty() then false
11           else  thisModule.isDefinedBy(start,
                   ↪newInput, end, visited->union(
                   ↪newInput)->asSet() ->asSequence
                   ↪(), forbidden)
12           endif
13         endif;
```

'*to*' (Listing 1, line 17), they describe how to compute the model elements to produce when the rule is fired, starting from the values of the matched elements. E.g., the *SimpleStmt* rule produces a single element of type *SimpleStmt*. A set of OCL *bindings* specify how to fill each of the features (attributes and references) of the produced elements. The binding at line 19 copies the textual representation of the instruction, the binding at line 20 fills the *dfNext* link with values computed by the *computeNextDataFlows* OCL *helper*. The rule for transforming methods is analogous (Listing 1, lines 3-11).

OCL helpers enable the definition of reusable OCL expressions. An OCL helper must be attached to a context, that

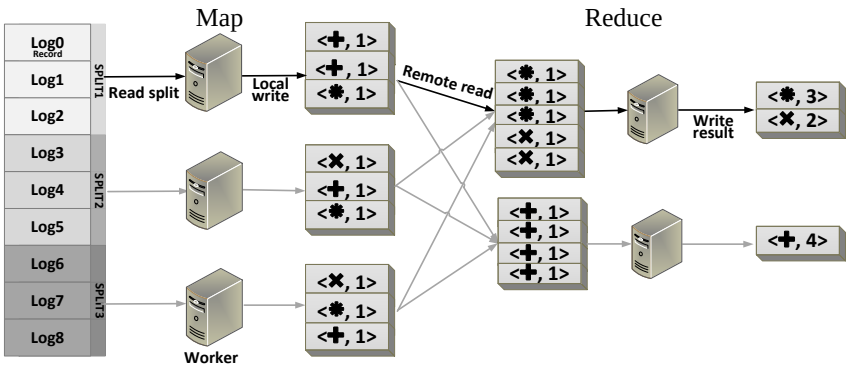


Figure 3: MapReduce programming model overview

can be a type or global context. Since target models are not navigable, only source types are allowed. Listing 2 shows our implementation of the *computeNextDataFlows* helper derived by the direct translation in OCL of the data-flow definition we gave in Equation 1. It has as context *FlowInstr* and returns a sequence of same type (Listing 2, line 1).

ATL matched rules are executed in two phases, a *match phase* and an *apply phase*. In the first phase, the rules are applied over source models' elements satisfying their guards. Each single match, corresponds to the creation of an explicit traceability link. This link connects three items: the rule that triggered the application, the match, and the newly created output elements (according to the target pattern). At this stage, only output pattern elements type is considered, bindings evaluation is left to the next phase.

The apply phase deals with the initialization of output elements' features. Every feature is associated to a binding in an output pattern element of a given rule application. Indeed, a rule application corresponds to a trace link. Features initialization is performed in two steps, first the corresponding binding expression is computed. Resulting in a collection of elements, it is then passed to a resolution algorithm (called *resolve algorithm*) for final update into the output model. The *resolve algorithm* behaves differently according to the type of each element. If the type is primitive (in case of attributes) or target, then it is directly assigned to the feature. Otherwise, if it is a source element type, it is first resolved to its respective target element – using the tracing information – before being assigned to the feature. Thanks to this algorithm we are able to initialize the target features without needing to navigate the target models.

It is noteworthy that the helpers' implementation illustrated in the paper is compact and straightforward (for an OCL programmer at least) but it has quadratic time complexity in the worst case (as the definition in Equation 1)[3]. As a consequence it does not scale to inter-procedural data-flow analysis of large code-bases like the ones typically found during modernization of legacy systems [4].

3. ATL ON MAPREDUCE

MapReduce is a programming model and software framework developed at Google in 2004 [10]. It allows easy and transparent distributed processing of big data sets while concealing the complex distribution details a developer might cross. MapReduce is inspired from the map and reduce primitives that exist in functional languages. Both *Map* and *Reduce* invocations are distributed across cluster nodes, thanks to the *Master* that orchestrates jobs assignment.

Input data is partitioned into a set of chunks called *Splits* as illustrated in Fig. 3. The partitioning might be monitored by the user throughout a set of parameters. If not, *splits* are automatically and evenly partitioned. Every *split* comprises a set of logical *Records*, each containing a pair of ⟨key, value⟩.

Given the number of *Splits* and idle nodes, the Master node decides the number of workers (slave machines) for the assignment of *Map* jobs. Each *Map worker* reads one or many *Splits*, iterates over the *Records*, processes the ⟨key, value⟩ pairs and stores locally the intermediate ⟨key, value⟩ pairs. In the meanwhile, the *Master* receives periodically the location of these pairs. When *Map workers* finish, the *Master* forwards these locations to the *Reduce workers* that sort them so that all occurrences of the same key are grouped together. The *mapper* then passes the key and list of values to the user-defined reduce function. Following the reduce tasks achievement, an output result is generated per reduce task. Output results do not need to be always combined, especially if they will subsequently be processed by other distributed applications.

Let's take a closer look to the MapReduce programming model by means of a simple example, depicted in Fig. 3. Assume we have set of log entries coming from a git repository. Each entry contains information about actions performed over a particular file (creation $\rightarrow +$, deletion $\rightarrow X$, or modification $\rightarrow *$). We want to know how many times each action was performed, using MapReduce. The master evenly splits the entries among workers. For every record (log entry), the map worker extracts the action type and creates a ⟨key,value⟩ pair with a key the action itself and value '1'. In the reduce phase, pairs with the same key are grouped together. In our example, the modification and deletion go to the first reducer, while the creation goes to the second one. For each group, the reducer combines the pairs, and creates a ⟨key,value⟩ pair, but this time with value the sum of the values with same key. This value refers to how many times the action occurred in the logs.

Much of the interest of MapReduce is due to its fault-tolerant processing. The *Master* keeps track of the evolution

[3]An algorithm with better efficiency, is described e.g. in the Dragonbook [20], Chapter 9.1, and is implemented with ATL in [9].

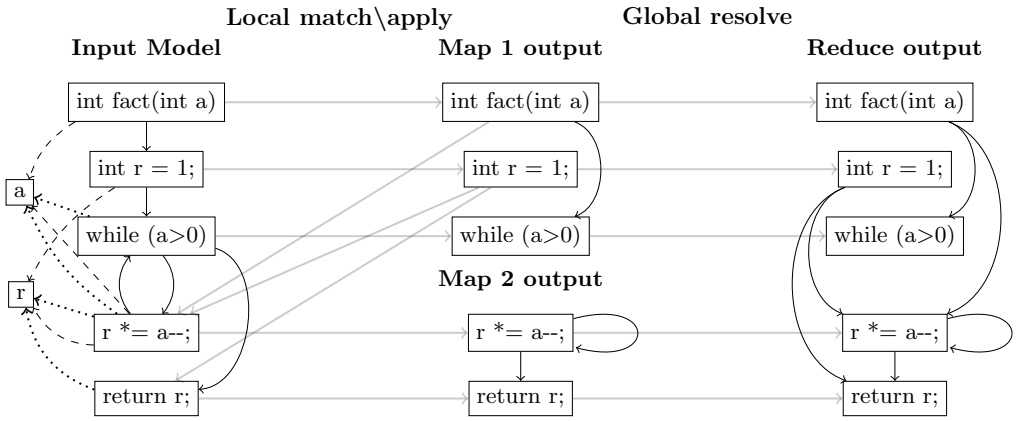


Figure 4: ControlFlow2DataFlow example on MapReduce

of every worker execution. If after a certain amount of time a worker does not react, it is considered as idle and the job is re-assigned to another worker.

3.1 ATL and MapReduce alignment

Transformations in ATL could be regarded as the union of elementary transformation tasks, where each takes charge of transforming a single model element. The approach we are proposing follows a data-distribution scheme, where each one of the nodes that are computing in parallel takes charge of transforming a part of the input model. This is made possible, thanks to the semantics alignment for ATL distributed execution with MapReduce described in this section.

However, implicit data distribution is not trivial for transformation languages where rules applied to different parts of the model can interact in complex ways with each other. As result of ATL's execution semantics, especially four specific properties of the language (below), we argue that inter-rule communication is made discernible. More precisely, interaction among ATL transformation rules are reduced to bindings resolution, where a target element's feature needs to reference to other target elements created by other rules:

1. *Locality.* Each ATL rule is the only responsible of the computation of the elements it creates, i.e., the rule that creates the element is also responsible of initializing its features. In case of bi-directional references, responsibility is shared among the rules that create the source and the target ends of the reference.

2. *Single assignment on target properties.* The assignment of a single-valued property in a target model element happens only once in the transformation execution. Multi-valued properties can be updated only for adding values, but never deleting them.

3. *Non-recursive rule application.* Model elements that are produced by ATL rules are not subject to further matches. As a consequence, new model elements can not be created as intermediate data to support the computation. Target models in ATL are read-only. This differentiates ATL from typically recursive graph-transformation languages. Not to confuse with recursion in OCL helpers, that are responsible of intermediate computations over the source models, not target ones.

4. *Forbidden target navigation.* Rules are not allowed to navigate the part of the target model that has already been produced, to avoid assumptions on the rule execution order. Thanks to the resolve algorithm along with the trace links that it is made possible.

These properties strongly reduce the possible kinds of interaction among ATL rules, and allow us to decouple rule applications and execute them in independent execution units, as explained in the following.

As an example, Fig. 4 shows how the ATL transformation of our running example could be executed on top of a MapReduce architecture comprising three nodes, two map and and one reduce workers. The input model is equally split according to the number of map workers (in this case each map node takes as input half of the input model elements). In the map phase, each worker runs independently the full transformation code but applies it only to the transformation of the assigned subset of the input model. We call this phase *Local match-apply*. Afterwards each map worker communicates the set of model elements it created to the reduce phase, together with trace information. These trace links (grey arrows in Fig. 4) encode the additional information that will be needed to *resolve* the binding, i.e. identify the exact target element that has to be referenced based on the tracing information. The reduce worker is responsible of gathering partial models and trace links from the map workers, and updating properties value of unresolved bindings. We call this phase *Global resolve*.

In the following (i) we describe the distributed execution algorithm we propose for ATL decomposing it in the *Local match-apply* phase assigned to mappers and the *Global resolve* phase assigned to reducers; (ii) we define the trace information that needs to be passed between mappers and reducers to allow the re-composition of the global model after distribution.

Local match-apply.

At the beginning of the phase, input splits are assigned to map workers. Each one of these splits contains a subset of the input model for processing. Although, each worker has a full view of the input models in case it needs additional data for bindings computation. Note that while intelligent assignment strategies could improve the algorithm efficiency by increasing data locality, in this paper we perform a ran-

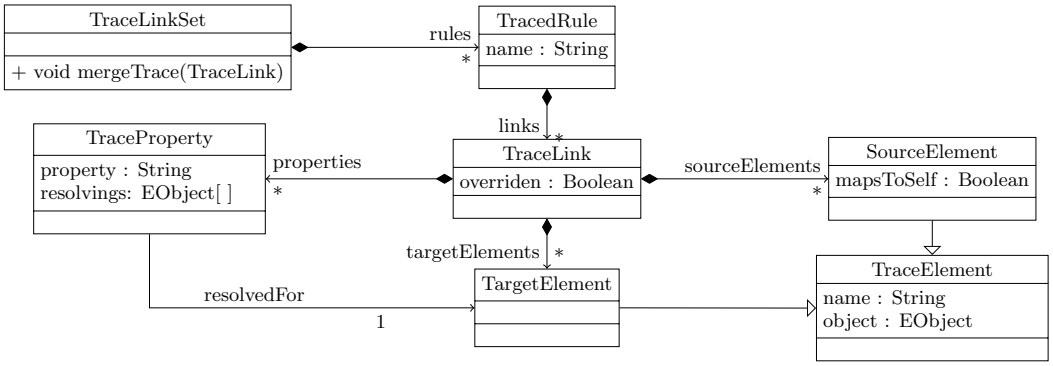
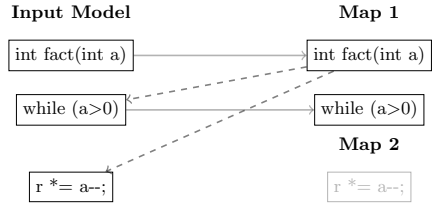


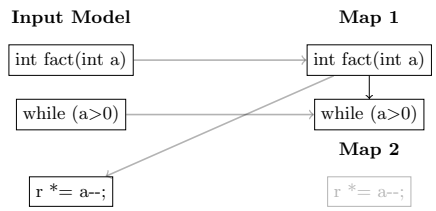
Figure 5: Extended Trace metamodel

Algorithm 1: Map function

```
   input: Long key, ModelElement elmt
1  if isMatched(element) then
2      link ← createLink(elmt);
3      foreach binding ∈ getBindings(link) do
4          if isAttribute(binding) then
5              apply (binding);
6          else
7              foreach
               element ∈ computeBindingExp(binding) do
8                  if isLocal(elmt) then
9                      addElementToTarget(elmt, binding);
10                 else
11                     trgElement ← resolveTarget(elmt);
12                     addElementToTrace(trgElmt, binding);
13     storeLink(moduleName, link);
```



(a) results of dfNext binding for {int fact (int a)}



(b) Resolving the results of dfNext binding for {int fact (int a)}

Figure 6: Local resolve of dfNext for {int fact (int a)}

dom assignment. Intelligent assignment strategies for model elements, especially based on static analysis of the transformation code, are left to future work.

Pseudo-code for the processing in this phase is given in Algorithm 1. For every model element, the map function verifies if a rule guard matches and in this case instantiates the corresponding target elements (line 2), same as in the regular execution semantics (Sec. 2). For instance, in Fig. 2 the *Method* and *FlowInstr* rules instantiate the method signature and the instructions that define or use variables (all the instructions of the example). Variables (a and r) are not instantiated since no rule matches their type. For each instantiated output element, a trace link is created connecting binding source and target elements of the applied rule.

Subsequently, the algorithm starts processing the list of property bindings for the instantiated target elements. We extended the behavior of the *resolve algorithm* to enable handling elements transformed in other nodes, we call this algorithm *local resolve*. In the case of attribute bindings, the same standard behavior is preserved, the OCL expression is computed and the corresponding feature is updated accordingly (lines 4– 5). While bindings related to references connect elements transformed by different rule applications, potentially in different nodes, the resolution is performed in two steps: (i) the OCL expression of the binding computes to a set of *elements in the source model* and ATL connects the bound feature to these source elements using trace links; (ii) the source-models elements are resolved, i.e. substituted with the corresponding target element according the rule application trace links. If source and target elements of the reference are both being transformed in the same node, both steps happen locally (lines 8– 9), otherwise trace links are stored and communicated to the reducer, postponing the resolution step to the *Global resolve* phase (lines 11– 12).

For example, executing the binding *dfNext* over the method *fact(int a)*, results in {*while(a)>0*, *r*=a--;*} (dashed grey arrows in Fig. 6 (a)). Since *while(a)* and *fact(int a)* reside in the same node, a *dfNext* reference between them is created in the target model. Instead, a trace property is created between *fact(int a)* and *r*=a--;* because they belong to different nodes(Fig. 6 (b)).

Global resolve

At the beginning of the reduce phase, all the target elements are created, the local bindings are populated, and the unresolved bindings are referring to the source elements to be

resolved. This information is kept consistent in the tracing information formerly computed and communicated by the mappers. Then it *resolves* the remaining reference bindings by iterating over the trace links, as depicted in Algorithm 2. For each trace link, the reducer iterates over the unresolved elements of its property traces (line 3), resolves their corresponding element in the output model (line 4), and updates the target element with the final references(line 5). In the right-hand side of Fig. 2 all the trace properties have been substituted with final references to the output model elements.

Algorithm 2: Reduce function

```
input: String key, Set<TraceLink> links
1 foreach link ∈ links do
2     foreach prop ∈ getTraceProperties(link) do //
      unresolved properties
3         foreach element ∈ getSourceElements(prop) do
4             targetElement ← resolveTarget(element);
5             updateUnresolvedElement(pro, targetElement);
```

Trace metamodel

MT languages like ATL need to keep track during execution of the mapping between source and target elements [32]. We define a metamodel for transformation trace information in a distributed setting (see Fig. 5).

As in standard ATL, traces are stored in a *TracelinkSet* and organized by rules. Each *TracedRule* is identified by its name, and may contain a collection of trace links. A link maps a set of source pattern elements to a set of target pattern elements. Both source and target pattern elements are identified by a unique name within a trace link (same one in the rule). Likewise, source elements and target elements refer to a runtime object respectively from input model or output model. This layered decomposition breaks down the complexity of traversing/querying the trace links.

This trace information (source elements, rule name, and target elements) is not sufficient for the distributed semantics, that requires to transmit to the reducer trace information connecting each unresolved binding to the source elements to resolve. Thus, we extended the ATL trace metamodel with the *TraceProperty* data structure. Trace properties are identified by their name that refers to the corresponding feature name. They are contained in a trace link, and associated to the source elements to be resolved along with the target element to be updated.

4. TOOL SUPPORT

4.1 Distributed transformation engine

We implemented our approach as a Distributed ATL engine, whose source code is available at the paper's website. The engine is built on top of the ATL Virtual Machine (EMFTVM [28]) and Apache Hadoop [2]. Hadoop is the leading open-source implementation of MapReduce and comes with the Hadoop Distributed File System (HDFS) that provides high-throughput access to application data and data locality optimization for MapReduce tasks.

In ATL VM, the transformation engine iterates the matched rules, and looks for the elements that match its application condition (guard). Instead, our VM iterates over each input model element, and checks if it is matched by an existing rule (`matchSingle(EObject)` in Table. 1). In this perspective we extended the ATL VM with a minimal set of functions (see Table 1) allowing the VM to run either in standalone or distributed mode. In particular, the distributed VM is required to factorize and expose methods for launching independently small parts of the execution algorithms. For instance the distributed VM exposes methods to perform the transformation of single model elements. Typically the methods `localApplySingle(EObject)` and `globalResolve()` that we call at the map and reduce functions respectively.

Each node in the system executes its own instance of the ATL VM but performs either only the local match-apply or the global resolve phase. The standalone and distributed execution modes share most of the code and allow for a fair comparison of the distribution speedup. Configuration information is sent together with the tasks to the different workers, so that they can be able to run their local VMs independently of each other. This information includes the paths of transformation, models and metamodels in the distributed file system. More information about the tool usage and deployment can be found at the tool's website[4].

4.2 Data distribution

Data locality is one of the aspects to optimize in distributed computing for avoiding bottlenecks. In Hadoop, it is encouraged to run map tasks with input data residing in HDFS, since Hadoop will try to assign tasks to nodes where data to be processed is stored. In Distributed ATL we make use of HDFS for storing input and output models, metamodels and transformation code.

Each mapper is assigned a subset of model elements by the splitting process. In Distributed ATL we first produce a text file containing model elements URIs as plain strings, one per line. This file will be splitted in chunks by Hadoop. Hadoop provides several input format classes with specific splitting behavior. In our implementation we use an *NLineInputFormat*, that allows to specify the exact number of lines per split. Finally, the default record reader in Hadoop creates one record for each line of the input file. As a consequence, every map function in Distributed ATL will be executing on a single model element.

Choosing the right number of splits has significant impact on the global performance. Having many splits means that the time taken to process each split will be small compared to the time to process the whole input. On the other hand, if splits are too small, then the overhead of managing the splits and creating map tasks for each one of them may dominate the total job execution time. In our case we observed better results where the number of splits matches the number of available workers. In other words, while configuring Distributed ATL, the number of lines per split should be set to $\frac{model\ size}{available\ nodes}$.

4.3 Tool limitations

Currently, our ATL VM supports only the default EMF serialization format XMI (XML Metadata Interchange). This file-based representation faces many issues related to scalability. In particular, models stored in XMI need to be fully loaded in memory, but more importantly, XMI does not sup-

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

Table 1: API extension

Class name	Operation	Description
ExecEnvironment	matchSingle(EObject)	Matching a single object
	localApplySingle(EObject)	Matching and Applying if possible
	globalResolve()	Resolving unresolved bindings and assignements in the global scope
TraceLinkSet	mergeTrace(TraceLink)	Add traceLink if does not exist and resolve input and output cross references

port concurrent read/write. This hampers our tool at two levels, first, all the nodes should load the whole model even though if they only need a subset of it. This prevents us from transforming very big models that would fit in memory. The second one concerns the reduce phase parallelization, and this is due to the fact that only one mapper can write to the output XMI file at once. In a recent work, we extended an existing persistence backend NeoEMF [11] with support for concurrent read/write [12] on top of Apache HBase [27]. NeoEMF is an extensible and transparent persistence layer for modeling tools designed to optimize runtime performance and memory occupancy. In future work, we plan to couple it with our VM to solve these two particular issues.

5. EXPERIMENTAL EVALUATION

We evaluate the scalability of our proposal by comparing how the transformation of our running example performs in different test environments. The transformation covers a sufficient set of declarative ATL constructs enabling the specification of a large group of MTs. It also contains an interesting number of OCL operations, recursive helper's call included.

We use as input different sets of models of diverse sizes. The original case study [13] already includes a set of input models for the benchmark. These models are reverse-engineered from a set of automatically generated Java programs, with sizes up to 12 000 lines of code. For our benchmark we used the same generation process but to stress scalability we produced larger models with sizes up to 105 000 lines of code. We consider models of these sizes sufficient for benchmarking scalability in our use case: in our experimentation, processing in a single machine the largest of these models takes more than four hours. All the models we generated and the experimentation results are available at the article website.

In what follows we demonstrate the scalability of our approach through two different but complementary experiences. The first one shows a quasi-linear speed-up w.r.t. the cluster size for input models with similar size, while the second one illustrates that the speed-up grows with increasing model size.

5.1 Experiment I: speed-up curve

For this experiment we have used a set of 5 automatically generated Java programs with random structure but similar size and complexity. The source Java files range from 1 442 to 1 533 lines of code and the execution time of their sequential transformation ranges from 620s to 778s. The experiments were run on a set of identical *Elastic MapReduce* clusters provided by *Amazon Web Services*. All the clusters were composed by 10 EC2 instances of type *m1.large* (i.e. 2 vCPU, 7.5GB of RAM memory and 2 magnetic Hard Drives). Each execution of the transformation was launched in one of those clusters with a fixed number of nodes – from 1 to 8 – depending on the experiment. Each experiment has been executed 10 times for each model and number of nodes. In total 400 experiments have been executed summing up a total of 280 hours of computation (1 120 normalized instance hours [1]). For each execution we calculate the distribution speed-up with respect to the same transformation on standard ATL running in a single node of the cluster.

Fig. 7 summarizes the speed-up results. The approach shows good performance for this transformation with an average speed-up up between 2.5 and 3 on 8 nodes. More importantly, as it can be seen in upper side, the average speed-up shows a very similar curve for all models under transformation, with a quasi linear speedup indicating good scalability w.r.t. cluster size. We naturally expect the speed-

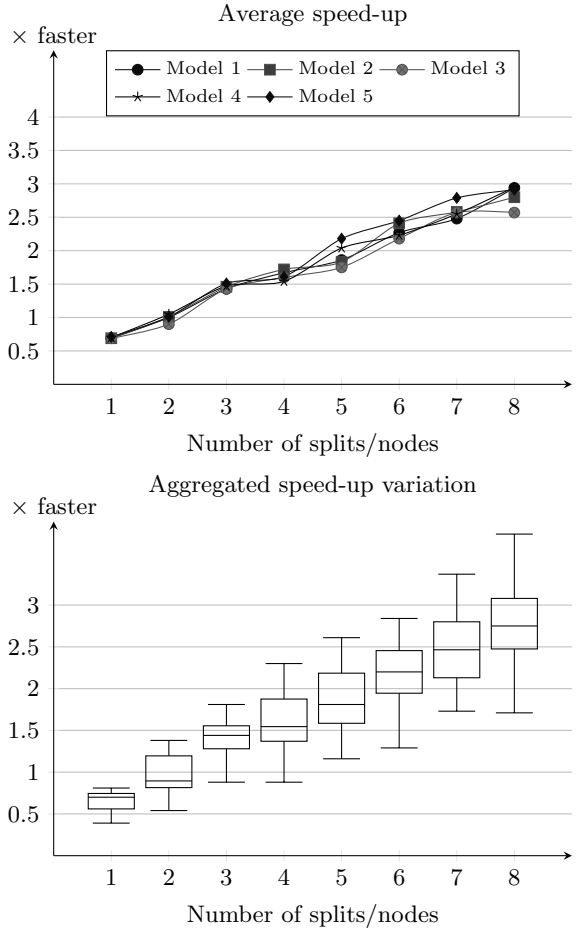


Figure 7: Speed-up obtained in Experiment I

Table 2: Execution times and speed-up (between parentheses) per model

#	Size	Elts	Std. VM	Distributed VM using x nodes (time and speed-up)							
				1	2	3	4	5	6	7	8
1	∼4MB	20 706	244s	319s (×0.8)	165s (×1.5)	128s (×1.9)	107s (×2.3)	94s (×2.6)	84s (×2.9)	79s (×3.1)	75s (×3.3)
2	∼8MB	41 406	1 005s	1 219s (×0.8)	596s (×1.7)	465s (×2.2)	350s (×2.9)	302s (×3.3)	259s (×3.9)	229s (×4.4)	199s (×5.1)
3	∼16MB	82 806	4 241s	4 864s (×0.9)	2 318s (×1.8)	1 701s (×2.5)	1 332s (×3.2)	1 149s (×3.7)	945s (×4.5)	862s (×4.9)	717s (×5.9)
4	∼32MB	161 006	14 705s	17 998s (×0.8)	8 712s (×1.7)	6 389s (×2.3)	5 016s (×2.9)	4 048s (×3.6)	3 564s (×4.1)	3 050s (×4.8)	2 642s (×5.6)

up curve to become sub-linear for larger cluster sizes and very unbalanced models. The variance among the 400 executions is limited as shown by the box-plots in the lower side.

5.2 Experiment II: size/speed-up correlation

To investigate the correlation between model size and speed-up we execute the transformation over 4 artificially generated Java programs with identical structure but different size (from 13 500 to 105 000 lines of code). Specifically, these Java programs are built by replicating the same imperative code pattern and they produce a balanced execution of the model transformation in the nodes of the cluster. This way, we abstract from possible load unbalance that would hamper the correlation assessment.

This time the experiments have been executed in a virtual cluster composed by 12 instances (8 slaves, and 4 additional instances for orchestrating Hadoop and HDFS services) built on top of OpenVZ containers running Hadoop 2.5.1. The hardware hosting the virtual cluster is a Dell PowerEdge R710 server, with two Intel® Xeon® X5570 processors at 2.93GHz (allowing up to 16 execution threads), 72 GB of RAM memory (1 066MHz), and two hard disks (at 15K rpm) configured in a hardware-controlled RAID 1.

As shown in Fig. 8 and Table 2, the curves produced by Experiment II are consistent to the results obtained from Experiment I, despite the different model sizes and cluster architectures. Moreover, as expected, larger models produce higher speed-ups: for longer transformations the parallelization benefits of longer map tasks overtakes the overhead of the MapReduce framework.

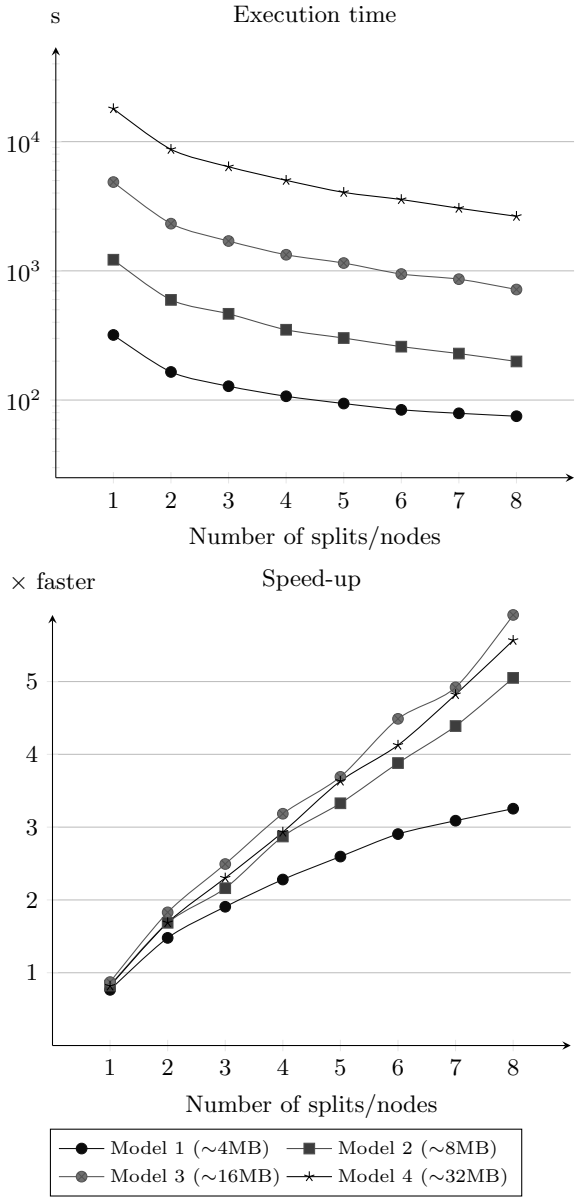


Figure 8: Execution times and speed-up on Experiment II

6. RELATED WORK

To our knowledge, our work is the first applying the MapReduce programming model to model transformation. The only other proposal addressing MT distribution is Lintra, by Burgueño et al. [5], based on the Linda coordination language. Lintra uses the master-slave design pattern for their execution, where slaves are in charge of applying the transformation in parallel to submodels of the input model. The same authors propose a minimal set of primitives to specify distributed model transformations, LintraP [6]. With respect to our approach, Lintra requires to explicitly use distribution primitives, but it can be used in principle to distribute any transformation language by compilation. However no compiler is provided, and it is difficult to compare their performance results with ours, since they only perform a local multi-threaded experimentation in a case with low speed-up (maximum 3.4 on 16 nodes).

Among distributed graph transformation proposals, a re-

cent one is Mezei et al. [23]. It is composed of a transformation-level parallelization and a rule-level parallelization with four different matching algorithms to address different distribution types. Unlike our approach, their main focus is on the recursive matching phase, particularly expensive for graph transformations, but less significant in MT (because of Property 3). In [15], Izso et al. present a tool called IncQuery-D for incremental query in the cloud. 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 transformation rules.

Shared-memory parallelization is a closely related problem to distribution. For model transformation, Tisi et al. [30] present a systematic two-steps approach to parallelize ATL transformations. The authors provided a multi-threaded implementation of the ATL engine, where every rule is executed in a separate thread for both steps. The parallel ATL compiler and virtual machine have been adapted to enable a parallel execution and reduce synchronization overhead. A similar approach for parallel graph transformations in multicore systems [14] introduces a two-phase algorithm (matching and modifier) similar to ours. Bergmann et al. propose an approach to parallelize graph transformations based on incremental pattern matching [3]. This approach uses a message passing mechanism to notify of model changes. The incremental pattern matcher is split into different containers, each one is responsible for a set of patterns. The lack of distributed memory concerns make these solutions difficult to adapt to the distributed computing scenario. Moreover in these cases the authors investigate task distribution, while we focus on data distribution, especially for handling VLMs.

While MapReduce lacks a high-level transformation language, several high-level query languages have been proposed. Microsoft SCOPE [7], Pig Latin [26], and HiveQL [29] are high level SQL-like scripting languages targeting massive data analysis on top of MapReduce. Pig Latin and SCOPE are hybrid languages combining both forces of a SQL-like declarative style and a procedural programming style using MapReduce primitives. They provide an extensive support for user defined functions. Hive is a data warehousing solution built on top of Hadoop. It comes with a SQL-like language, HiveQL, which supports data definition statements to create tables with specific serialization formats, and partitioning and bucketing columns. While all these query languages compile down to execution plans in the form of series of MapReduce jobs, in our approach each node executes its own instance of the transformation VM, re-using the standard engine. However our approach computes single transformations in only two MapReduce rounds, while these language may compile in multi-round MapReduce chains. We also manipulate EMF model elements instead of tool-specific data representations, hence leaning on a standardized way to represent data structure.

7. CONCLUSION AND FUTURE WORK

In this paper we argue that model transformation with rule-based languages like ATL is a problem that fits in the MapReduce execution model. As a proof of concept, we introduce a semantics for ATL distributed execution on MapReduce. We experimentally show the good scalability of our solution. Thanks to our publicly available execution engine, users may exploit the availability of MapReduce clusters on the Cloud to run model transformations in a scalable and fault-tolerant way.

In our future work we plan to improve the efficiency of our approach, by addressing related research aspects. We aim to investigate:

- I/O optimization of model processing in MapReduce by coupling with the transformation engine a distributed model-persistence backend supporting concurrent read-/write;
- parallelization of the *Global Resolve* phase, made possible by high-performance I/O;
- efficient distribution of the input model over map workers aiming to optimize load balancing and minimize workload, relying on a static analysis of the transformation;
- global optimization and pipelining for transformation networks on MapReduce.

Acknowledgments

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

8. REFERENCES

[1] Amazon Web Services, Inc. Amazon EMR FAQs, June, 2015. URL: `http://aws.amazon.com/elasticmapreduce/faqs`.

[2] Apache Software Foundation. Apache Hadoop, June, 2015. URL: `http://hadoop.apache.org/`.

[3] G. Bergmann, I. Ráth, and D. Varró. Parallelization of Graph Transformation Based on Incremental Pattern Matching. In *Electronic Communications of the EASST*, volume 18, 2009.

[4] 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.

[5] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. On the Concurrent Execution of Model Transformations with Linda. In *Proceeding of the First Workshop on Scalability in MDE*, BigMDE '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[6] L. Burgueño, E. Syriani, M. Wimmer, J. Gray, and A. Moreno Vallecillo. LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra. In *Proceedings of 2nd BigMDE Workshop*, volume 1206. CEUR Workshop Proceedings, 2014.

[7] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[8] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, Denmark, 2012. Springer.

[9] V. Cosentino, M. Tisi, and F. Büttner. Analyzing Flowgraphs with ATL. *arXiv preprint arXiv:1312.0343*, 2013.

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

[11] A. Gómez, M. Tisi, G. Sunyé, and J. Cabot. Map-based transparent persistence for very large models. In A. Egyed and I. Schaefer, editors, *Proceedings of Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2015.

[12] A. Gómoz, A. Benelallam, and M. Tisi. Decentralized Model Persistence for Distributed Computing. In *Proceedings of 3rd BigMDE Workshop*. CEUR Workshop Proceedings, July 2015. to appear.

[13] T. Horn. The TTC 2013 Flowgraphs Case. *arXiv preprint arXiv:1312.0341*, 2013.

[14] G. Imre and G. Mezei. Parallel Graph Transformations on Multicore Systems. In *Multicore Software Engineering, Performance, and Tools*, volume 7303 of *LNCS*, pages 86–89. Springer, 2012.

[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] 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).

[17] J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *9th OOPSLA workshop on Domain-Specific Modeling*. Helsinki School of Economics, 2009.

[18] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] A. Kuhn, G. Murphy, and C. Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MDE Languages and Systems*, volume 7590 of *LNCS*, pages 352–367. Springer, 2012.

[20] M. Lam, R. Sethi, J. Ullman, and A. Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[21] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel Data Processing with MapReduce: A Survey. In *SIGMOD Rec.*, volume 40, pages 11–20, New York, NY, USA, 2012. ACM.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceeding of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, Indianapolis, Indiana, USA, 2010. ACM.

[23] G. Mezei, T. Levendovszky, T. Meszaros, and I. Madari. Towards Truly Parallel Model Transformations: A Distributed Pattern Matching Approach. In *IEEE EUROCON 2009*, pages 403–410. IEEE, 2009.

[24] Object Management Group. Object Constraint Language, OCL, June, 2015. URL: `http://www.omg.org/spec/OCL/`.

[25] Object Management Group. Query/View/Transformation, QVT, June, 2015. URL: `http://www.omg.org/spec/QVT/`.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[27] The Apache Software Foundation. Apache HBase, June, 2015. URL: `http://hbase.apache.org/`.

[28] The Eclipse Foundation. ATL EMFTVM, June, 2015. URL: `https://wiki.eclipse.org/ATL/EMFTVM/`.

[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[30] M. Tisi, S. Martínez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 656–672. Springer, 2013.

[31] 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.

[32] A. Yie and D. Wagelaar. Advanced Traceability for ATL. In *Proceeding of the 1st International Workshop on Model Transformation with ATL (MtATL)*, pages 78–87, Nantes, France, 2009.

[33] S. Zschaler, D. Kolovos, N. Drivalos, R. Paige, and A. Rashid. Domain-specific metamodelling languages for software language engineering. In M. van den Brand, D. Gašević, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 334–353. Springer Berlin Heidelberg, 2010.