ATL-MR: Model Transformation on MapReduce

Amine Benelallam, Abel Gómez, Massimo Tisi

ATL-MR: Model Transformation on MapReduce

Amine Benelallam
AtlanMod Team, Inria, Mines-Nantes,
Lina, 4 rue Alfred Kastler, 44307
Nantes, France
amine.benelallam@inria.fr

Abel Gómez
Departamento de Informática e
Ingeniería de Sistemas. Universidad
de Zaragoza. Zaragoza, Spain
abel.gomez@unizar.es

Massimo Tisi
AtlanMod Team, Inria, Mines-Nantes,
Lina, 4 rue Alfred Kastler, 44307
Nantes, France
massimo.tisi@inria.fr

Abstract

The Model-Driven Engineering (MDE) paradigm has been successfully embraced for manufacturing maintainable software in several domains while decreasing costs and efforts. One of its principal concepts is rule-based Model Transformation (MT) that enables an automated processing of models for different intentions. The user-friendly syntax of MT languages is designed for allowing users to specify and execute these operations in an effortless manner. Existing MT engines, however, are incapable of accomplishing transformation operations in an acceptable time while facing complex transformations. Worse, against large amount of data, these tools crash throwing an out of memory exception.

In this paper, we introduce ATL-MR, a tool 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 briefly present an overview of our approach, we describe the changes with respect to the standard ATL transformation engine, finally, we experimentally show the scalability of this solution.

Keywords Model Transformation, Distributed Computing, Tool, ATL, MapReduce

1. Introduction

Model-Driven Engineering (MDE) is gaining ground in industrial environments, thanks to its promise of lowering software development and maintenance effort [10]. It has been adopted with success in producing software for several domains such as the modernization of legacy software systems [3]. 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. The AtlanMod Transformation Language (ATL) [9] is one of the most popular examples among them, and a plethora of transformations exist addressing different model types and intentions[1].

In recent years, MDE has been witnessing the increasing complexity of systems and data that comes in MDE in the form of Very Large Models (VLMs) [4]. Existing MDE tools, especially MT engines, are based on graph matching and traversing techniques, that makes them exposed to serious scalability issues in terms of memory occupancy and execution time. This stands in particular when MT execution is limited by the resources of a single machine.

With the broad availability of distributed clusters and programming models such as MapReduce [5], a natural way to overcome these issues would be exploiting distributed systems for parallelizing model processing operations.

In this paper we introduce ATL-MR[2], a tool enabling the automatic execution of model transformations in ATL on top of MapReduce. The distribution is implicit, i.e. the syntax of the ATL is not modified and no primitive for distribution is added. Hence, no familiarity with distributed programming is required.

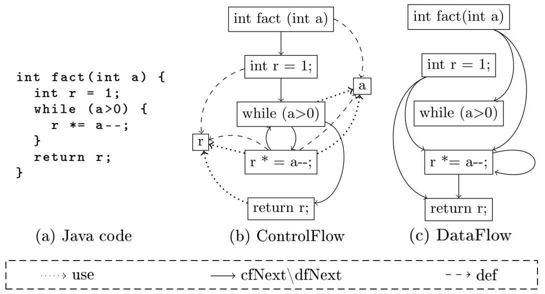
The paper is structured as follows. Section 2 introduces the syntax of ATL and its execution semantics by means of a running example. Section 3 describes an overview of our approach and details the implementation of our prototype engine. Section 4 discusses the evaluation results of our solution. Finally, Section 5 wraps up the conclusions and future works.

2. Background

In this section we give a brief introduction to ATL by means of model-driven reverse engineering case study. This case study was proposed as a simple but computationally expensive benchmark for MT engines [8]. We focus on one

[1] `http://www.eclipse.org/atl/atlTransformations/`

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



```
int fact(int a) {
  int r = 1;
  while (a>0) {
    r *= a--;
  }
  return r;
}
```

Figure 1: ControlFlow2DataFlow transformation example

Listing 1: ControlFlow2DataFlow - ATL rules (excerpt)

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

phase of the scenario, the transformation of the control-flow diagram of a Java program into a data-flow diagram.

Fig. 1 shows an example of models, derived from a small program calculating a number factorial. Instructions are represented by rectangles, and variables by squares. An instruction points to the set of variables it defines or writes *(def)*, and a set of variables it reads *(use)*. The links *cfNext* and *dfNext* refer to the next control flow and data flow instructions respectively[3]. As it can be seen in the figure, the transformation changes the topology of the model graph.

2.1 The AtlanMod Transformation Language

In this paper we refer to an ATL implementation of the transformation named *ControlFlow2DataFlow* and available at the article website[2]. Listing 1 illustrates a rule of the transformation.

Model transformations in ATL [9] are unidirectional. They are applied to read-only source models and produce write-only target models. Declarative rules abstract the relationship between source and target elements while hiding the semantics dealing with rule triggering, ordering, traceability management and so on.

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 [13] *guards* over a set of typed elements, i.e. only input elements satisfying that guard are matched. In ATL, a source pattern lays within the body of the clause '*from*'. For instance, in the rule *SimpleStmt*, the source pattern (Listing 1, lines 4-6) matches an element of type *SimpleStmt* that defines or uses at least a variable. Output patterns, delimited by the clause '*to*' (lines 7-11) 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 9 copies the textual representation of the instruction, the binding at line 10 fills the *dfNext* link with values computed by the *computeNextDataFlows* OCL *helper*.

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. At this stage, only the output pattern elements type is considered, bindings evaluation is left to the apply 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. Features initialization is performed in two steps: 1) first the corresponding binding expression is computed, resulting in a collection of elements of the input model; 2) it is then passed to a resolution algorithm that, if needed, navigates the trace links produced in the match phase to determine the corresponding target elements.

2.2 MapReduce

MapReduce is a programming model and software framework developed at Google in 2004 [5]. It allows easy and transparent distributed processing of big data sets while concealing the complex distribution details a developer might cross. 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*. Every *split* comprises a set of logical *Records*.

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 records and stores the results locally. When *Map workers* finish, the *Master* forwards these locations to the *Reduce workers* that sort the records by key so that all occurrences of the same key

[3] A detailed introduction can be found in the Dragonbook [11], Chapter 9

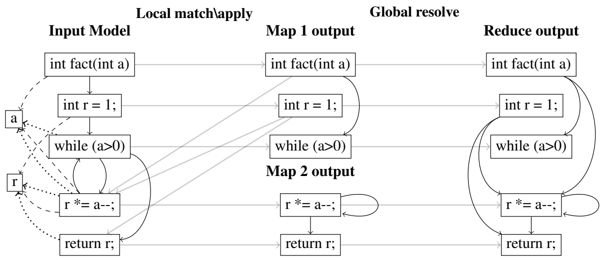


Figure 2: ControlFlow2DataFlow example on MapReduce

are grouped together. The *mapper* then passes the key and list of values to the user-defined reduce function for processing.

3. ATL-MR

Our ATL-MR transformation engine 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. Its source code is available at the paper's website. The engine is built on top of the ATL Virtual Machine (EMFTVM [15]) and Apache Hadoop [2].

Fig. 2 shows how the ATL transformation of our running example is executed on top of three nodes, two map and one reduce workers. The input model is equally split over map workers. In the map phase, each worker runs independently the full transformation code but applies it only to 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 worker that performs the reduce phase, together with trace information. These trace links (grey arrows in Fig. 2) 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*. 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.

At the **map** phase, input splits are assigned to map workers. Each one of these splits contains a subset of the input model for processing. However, each worker has a full view of the input models in case it needs additional data for bindings computation. For example, executing the binding *dfNext* over the method *fact(int a)*, results in {*while(a)>0*, *r*=a--;*} (grey arrows in Fig. 2). Since *while(a)* and *fact(int a)* reside in the same node, a *dfNext* reference between them is created in the target model during this map phase. 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. For each trace link, the reducer iterates over the unresolved elements of its property traces, resolves their corresponding element in the output model, and updates the target element with the final references. In the right-hand side of Fig. 2 all the trace properties have been substituted with final references to the output model elements.

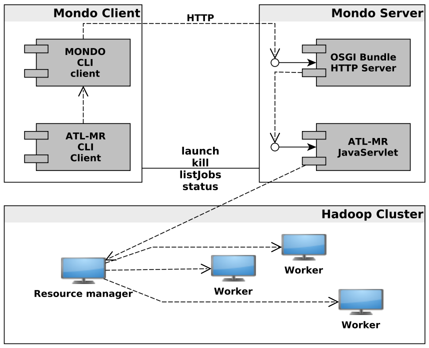


Figure 3: ATL-MR's integration within MONDO platform

The standalone and distributed ATL engines share most of the code. In the standard ATL VM, the transformation engine iterates over the matched rules, and looks for the elements that match its application condition (guard). Instead, our distributed VM iterates over each input model element, and checks if it is matched by an existing rule. In this perspective, we extended the ATL VM with a minimal set of functions 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 transformation of single model elements. More information about the tool usage and deployment can be found at the tool's website[2].

3.1 Platform Integration

Besides running independently, ATL-MR can be used as a component of MONDO, an integrated platform to support scalable MDE [12]. MONDO consists of a distributed platform and an Eclipse-based tool workbench counterpart through which developers can interact with the facilities provided by the platform. As depicted in Fig. 3 ATL-MR client is implemented as OSGI bundle integrated with the MONDO client. It comes with a command line-based API enabling four primary operations, namely *launch* and *kill* for launching and stopping a transformation respectively, whereas, *status* returns the status of a transformation job. Finally, *listJobs*, returns the list of running jobs. On the server side, ATL-MR is implemented as part of the MONDO server by means of a web servlet communicating with the *Resource Manager* of the Hadoop cluster.

3.2 Limitations

Currently, our ATL VM supports only the default EMF serialization format XMI (XML Metadata Interchange). This

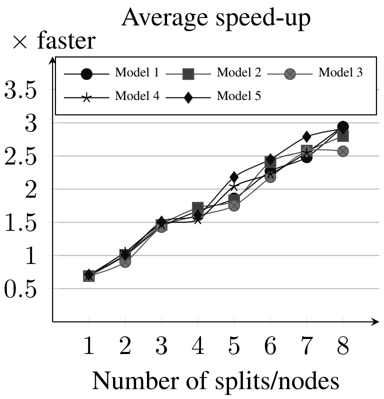


Figure 4: Speed-up obtained in Experiment I

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 support 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 not 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 [7] with support for concurrent read/write and on-demand loading [6] on top of Apache HBase [14]. In our current work, we are coupling it with our VM to solve these two particular issues.

4. Experimentation

In this section we experimentally evaluate the scalability of our approach by conducting two different but complementary experiments. We run our experiments in our running example and compare how it performs in two different test environments (clusters). The transformation covers a sufficient set of declarative ATL constructs enabling the specification of a large group of MTs. We use as input different sets of models of various sizes, reverse-engineered from a set of automatically generated Java programs. While the first one shows a quasi-linear speed-up w.r.t. the cluster size for input models with similar size, the second one illustrates that the speed-up grows with increasing model size.

For the first 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*. 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]). Fig. 4 summarizes the speed-up results.

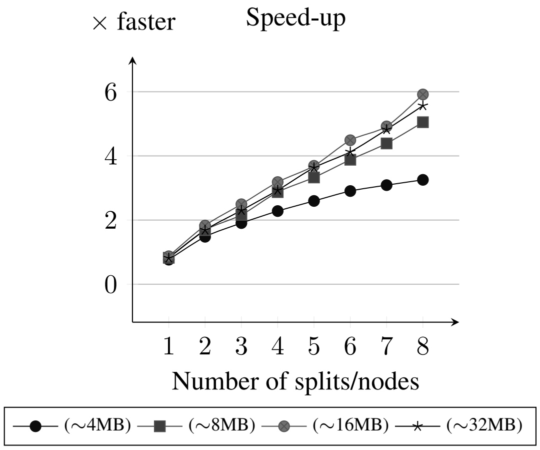


Figure 5: Execution times and speed-up on Experiment II

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). This time the experiments have been executed in a virtual cluster composed by 12 instances built on top of OpenVZ containers running Hadoop 2.5.1.

As shown in Fig. 5, 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.

5. Conclusion and Future Work

In this paper we introduce ATL-MR a tool to enable the distribution of ATL transformation in a distributed environment. We show its interoperability by integrating it with a distributed MDE platform, MONDO. Also, 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 tool, by addressing related research aspects. We aim to investigate: (i) coupling with the transformation engine a distributed model-persistence backend supporting concurrent read/write, and on-demand loading, (ii) efficiently distributing the input model over map workers with the aim to optimize load balancing and minimize workload.

Acknowledgments

This work is partially supported by the MONDO (EU ICT-611125) project. Many thanks to Antonio García-Domínguez for his valuable help while integrating ATL-MR with the MONDO platform.

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

[4] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *CloudMDE*, Copenhagen, Denmark, 2012. Springer.

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

[6] A. Gómez, A. Benelallam, and M. Tisi. Decentralized Model Persistence for Distributed Computing. In *Proc. of 3rd BigMDE Workshop*. CEUR Workshop, July 2015. urn:nbn:de:0074-1406-4.

[7] A. Gómez, M. Tisi, G. Sunyé, and J. Cabot. Map-based transparent persistence for very large models. In A. Egyed and I. Schaefer, editors, *Proc. of FASE'15*, volume 9033 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2015.

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

[9] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[10] 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.

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

[12] MONDO Project. MONDO Platform, June, 2015. URL: `https://github.com/mondo-project/`.

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

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

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