CloudTL: A New Transformation Language based on Big Data Tools and the Cloud

Jesús M. Perera Aracil and Diego Sevilla Ruiz
DITEC, University of Murcia, Campus Universitario de Espinardo, 30100, Espinardo, Murcia, Spain

Keywords: MDE, Model-to-model Transformation, Cloud, Ecore.

Abstract: Model Driven Engineering (MDE) faces new challenges as models increase in size. These so called Very Large Models (VLMs) introduce new challenges, as their size and complexity cause transformation languages to have long execution times or even not being able to handle them due to memory issues. A new approach should be proposed to solve these challenges, such as automatic parallelization or making use of big data technologies, all of which should be transparent to the transformation developer. In this paper we present CloudTL, a new transformation language whose engine is based on big data tools to deal with VLMs in an efficient and scalable way, benchmarking it against the `de facto` standard, ATL.

1 INTRODUCTION

Models are a digital representation of reality, and as such, their size is constantly increasing. Transformations are used for manipulating these models, but nowadays are currently becoming slow or even impossible to run because of the huge size of input and output models.

Big data (Manyika et al., 2011) tools have been proven to be a solution for manipulating huge amounts of data in considerable time, by using distribution across different compute nodes (both physical and/or virtual machines (VMs)). Thus, one can think that using them to perform model transformations might be the way to go for dealing with VLMs.

The challenges that are arising in the Model Driven Engineering (MDE) ecosystem due to the fact of models becoming huge and complex can be seen as a parallelism to what the data analytics is experimenting. In fact, it is a logical parallelism since information is being constantly generated and is getting more and more complex. Thus, we believe that by using big data tools, which are already solving the issue of manipulating inmense ammounts of information, we could help improve the problems that MDE, and model transformations in particular, are facing.

The Mondo Project (Kolovos et al., 2015) is a Specific Targeted Research Project (STREP) of the Seventh Framework Programme for research and technological development (FP7) aiming to tackle the increasingly important challenge of scalability in MDE in a comprehensive manner. There has been publications and tools developed through this project which have aided to get us closer to a solution of dealing with VLMs.

As this kind of initiatives and projects indicate, solving these problems is a real issue for MDE and it is essential that solutions, tools, frameworks and new languages are developed so that MDE can continue and adapt to the needs of Software Engineering when faced to problems in which the huge magnitude of data must be handled in an efficient way.

This paper is structured as follows: Section 2 introduces the concepts used in our proposal for building our distributed transformation language, CloudTL. Section 3 presents our implementation and benchmarks it agains ATL, the *de facto* standard. Section 4 discusses related work and Section 5 summarizes conclusions and future work.

2 BACKGROUND

2.1 Apache Storm

Apache Storm is a free and open source distributed realtime computation system (Apache, 2016c). It is commonly used as a big data tool to analyse streams of information and obtain analytics or metadata from them.

Perera Aracil J. and Sevilla Ruiz D.
CloudTL: A New Transformation Language based on Big Data Tools and the Cloud.
DOI: 10.5220/0006203101370146
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 137-146
ISBN: 978-989-758-210-3

It is based on the idea of flow of information, called topologies, so that there are producers of infromation, called `Spout`s, and nodes of computation, called `Bolt`s, which are interconnected. These interconnections are called `Stream`s and are an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. `Tuple`s are a collection of primitive types (strings, integers...), but also serializers can be defined to use other types.

Streams connect spouts and bolts, as well as bolts to other bolts, by defining the structure of the tuples that will be emitted through them. This interconnection of spouts and bolts is called a Storm topology, which defines the flow of data from the spouts to bolts. These streams are also aware of replication of Storm bolts, so different strategies can be given for emitting tupples (i.e., broadcast a tuple to all the replicas, emit the tuple to a local target replica if it exists, using consistent hashing to send tuples to the same replica...). Figure 1 shows a simple Storm topology from the Storm webpage.

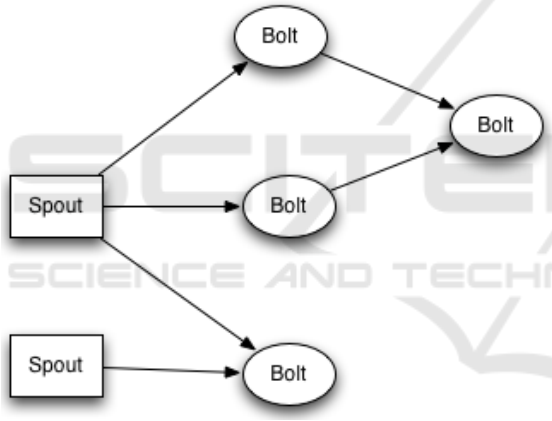

Figure 1: Simple Storm topology.

The execution of a Storm topology requires a Storm cluster, which is responsible of starting spouts and bolts, and the streams between them. There are two types of Storm clusters:

- local: A local cluster runs on a single machine, in which the Java main method is invoked as if it were a normal Java program. This is a useful option when developing and testing topologies.
- distributed: a cluster can be distributed between different physicial or virtual machines, so that the workload is shared between them. This kind of cluster makes use of to keep synchronization and keeping track of living nodes, such as Nimbus (Keahey and Freeman, 2016) and ZooKeeper (Apache, 2016d). This is the preferred option for executing Storm topologies in production mode.

2.2 Cloud Ecore

Cloud Ecore (Perera Aracil and Sevilla Ruiz, 2016) is a distributed Ecore implementation in JSON (JSON, 2016). The authors developed both a specification of Ecore in JSON and a implementation which served Ecore models and metamodels using a REST service. The main characteristic of Cloud Ecore is that it is URL based for the ids of elements that make a model (i.e. any model element is identified by a valid URL).

We are using an optimized version of this implementation, in which `EAttributess` are part of the JSON which is returned to the client when asked for a particular URL. Code Example 1 shows a comparison to clarify this modification.

```
// EClass (Original)
// URL: http://www.example.com/repo/0/
//      eClassifiers/0
{
"name":"http://www.example.com/repo/0/
          eClassifiers/0/name",
"eClass":"http://www.example.com/repo/0/
          eClassifiers/0/eClass",
...
}

// EClass (Optimized)
// URL: http://www.example.com/repo/0/
//      eClassifiers/0
{
"name":"EClass",
"eClass":"http://www.example.com/repo/0/
          eClassifiers/0/eClass",
...
}
```

Code Example 1: Partial EClass as JSON as both the original and the optimized version.

This modification helps reduce the number of HTTP petitions that are sent to the server, thus becoming a faster implementation as well as improving both the client and server performance.

Other optimization we have introduced is the pagination of lists of elements. Now, lists are paginated by default, each page containing a fixed number of elements. In this way, the parallelization of lists can be performed through pages instead of having a vast list and having to manually partition it.

3 CloudTL

CloudTL aims to be a model-to-model transformation language ready to handle VLMs in an efficient way, inspired by the syntax of ATL (Jouault and Kurtev,

2006), and using big data tools and distributed computation. Code Example 2 shows a simple transformation written in CloudTL, which implements a copy transformation (i.e., it generates as output the same model as it is given as input) for the metamodel shown in Figure 2; it consists of a `Super` metaclass containing a multi-valued `Sub EReference`.

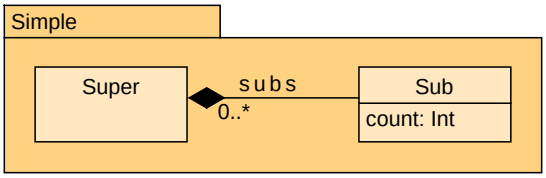

Figure 2: Simple metamodel.

CloudTL contains three main sections which define a transformation:

- Server section: The server serction of a CloudTL transformation defines the servers which contain the Cloud Ecore models and metamodels. These servers are defined by their IP address and port, as well as a unique ID to be identified throught the other sections wherever needed. In the example, we have defined a server named `local` which is at `http://<IP>:<Port>`.
- Namespace section: The namespace section defines the namespaces, both input and output, of the transformation. These are defined by a unique ID, a reference to a server defined in the server section, as well as a path (URL) to the Cloud Ecore metamodel. In the example, we define the `A` namespace as the input, which is contained in the local server at the `repo/1` path (i.e., the metamodel will be at `http://<IP>:<Port>/repo/1`). Namespace `B` is the output namespace of the transformation and, since this is a copy transformation, points to the same path as the input.
- Rule section: The rule section contains the different rules that compose a transformation, indicating both their input and output metaclass. There is a unique special rule which must be identified by the `init` keyword, marking it as the first rule to be executed whenever the transformation starts.

Rules have a collection of operations that set the target element `EStructuralFeatures` from the source element. There are, currently, three operations supported by CloudTL:

Initializators of Primitive Types. They set a target `EStructuralFeature` to a defined value.

```
tgt.count = 5;
```

This operation would set the `count` feature of `tgt` to `5` for every execution of the rule.

Copying of Values. If the source and target `EStructuralFeatures` are `EAttributes`, CloudTL will just copy the value from the source to the target without invoking other rule. An example would be

```
tgt.name = src.name;
```

This operation would copy the `name` feature from the `src` instance to the `tgt name` feature.

Transformation Invokations. If the source and target `EStructuralFeatures` are `EReferences`, CloudTL will find a suitable rule (i.e., a rule which transforms an instance from the source feature value to the target one) and invoke it. The operation

```
tgt.subs = src.subs;
```

would imply that CloudTL must find a rule which transforms a `Sub` to `Sub`, and invoke it, transforming all the instances of the source `subs` collection.

```
transformation SimpleCopy {
  servers {
    local@<IP>:<Port>
  }

  namespaces {
    in A as local@repo/1
    out B as local@repo/1
  }

  init super2super {
    from src is A::Super
    to tgt is B::Super {
      tgt.subs = src.subs;
    }
  }

  sub2sub {
    from src is A::Sub
    to tgt is B::Sub {
      tgt.count = src.count;
    }
  }
}
```

Code Example 2: CloudTL SimpleCopy transformation.

CloudTL is created as a Xtext framework (Foundation, 2016), and thus it compiles to standard Java code, which can be invoked or utilize any Java library available. It also generates an Eclipse editor and powerful aids when developing a transformation, such as auto-completion, by querying the Cloud Ecore metamodel.

3.1 Compilation to Java

We have used Apache Storm as the backend for the engine of our language, so what the Xtext compiler

does is create a customized Spout and several Bolts, as well as some helper classes which aid in the transformation process. The generation tries to extract as much information as possible from the Cloud Ecore input and output metamodels and making it static (i.e., not needing to query it again when running the transformation), so that there is as little network traffic as possible when the transformation executes. Cloud Ecore (Perera Aracil and Sevilla Ruiz, 2016) demonstrated that network traffic is one of the most important factors that degrade the performance of its distributed mechanism, and thus, we try to minimize it as much as possible.

The Storm topology is created by analizing the transformation rules and their data dependencies between them. For example, if the definition of rule A transforms a target EReference from a source one, this would be done by invoking rule B. In out copy transformation, this could be seen in the super2super rule: transforming the EReference subs for the target metamodel is done by invoking a rule which can transform a source Sub element to a target one (i.e., invoking the sub2sub rule over the subs collection of the source Super element).

Each rule of the transformation will generate a bolt, which will be responsible for generating its target model element form the source element. The bolt will be responsible for contacting the Cloud Ecore server and HTTP get the URL representing the source element, which will be a JSON Object containing all the EStructuralFeatures. Rules are composed of transformation operations, which will be compiled as emit statements to other bolts, depending on the source and target EStructuralFeatures that are transforming.

- Mono-valued EAttributes: will consist of adding the value to the target JSON Object as a new JSON pair.
- Multi-valued EAttributes: will consist of an emit statement to a helper bolt which will iterate over the collection and generate the target collection.
- Mono-valued EReferences: will consist of an emit statement to the bolt which is responsible for the transformation of the EClass of the source EReference to the target one.
- Multi-valued EReferences: will consist of an emit statement to a helper bot which will iterate over the collection and emit each element to the corresponding transformation bolt, as if it was a mono-valued EReference, and will generate it as a target collection.

The Storm topology created by the transformation SimpleCopy can be seen in Figure 3. As mentioned before, SimpleCopySpout is created, responsible for invoking the first rule of the transformation with both the input model URL and the output model URL.

Data dependencies are extracted and analized from the transformation text, so a valid topology can be created. If the language detects that it cannot solve a data dependencie (i.e., we have forgotten to include the sub2sub rule or we have used the incorrect input and output metaclasses), it will generate an editor error (i.e., marking the data dependency in red) so it can be fixed.

These data dependencies are EReferences to other model elements, so transforming these are done by invoking another transformation rule. This rule is found by matching the source and target elements EClasses to the input and output metaclasses of a rule. In code, this is generated as two different pieces of code in the generated Java code:

- A stream from a bolt to another (representing the dependency of a rule to another). This is generated in the Launcher class, where the Storm topology is configured.
- A emit statement in the execute method of a bolt to send information to another bolt (representing the execution of a particular model element).

If the EReference is multi-valued, it takes into account that it is paginated, so two additional helper bolts are generated in order to deal with the pagination and the invokation of each element of each page. In our example, since the subs EReference is multivalued, the generation will create two new bolts Super2superSubsPages, which is in charge of iterating over the pagination, and Super2superSubsElements, which will emit of each Sub element to the corresponding transformation bolt (i.e., in our transformation to Sub2sub). On the other hand, if the EReference is mono-valued, no helper bolts will be generated, and the emit statement will directly invoke another bolt (i.e., transformation rule).

A Launcher class is generated in which the bolts and spout are configured to build the Storm topology. This class contains the main method. A Spout will be generated which is in charge of invoking the initial rule of a transformation, sending it the root input element and defining where the root output element must be created.

Other helper bolts are also generated to aid in the transformation process, such as the TraceBolt, which is responsible for tracking target elements transformed from source elements (i.e., similar to a TraceLink in the ATL engine). The PosterBolt

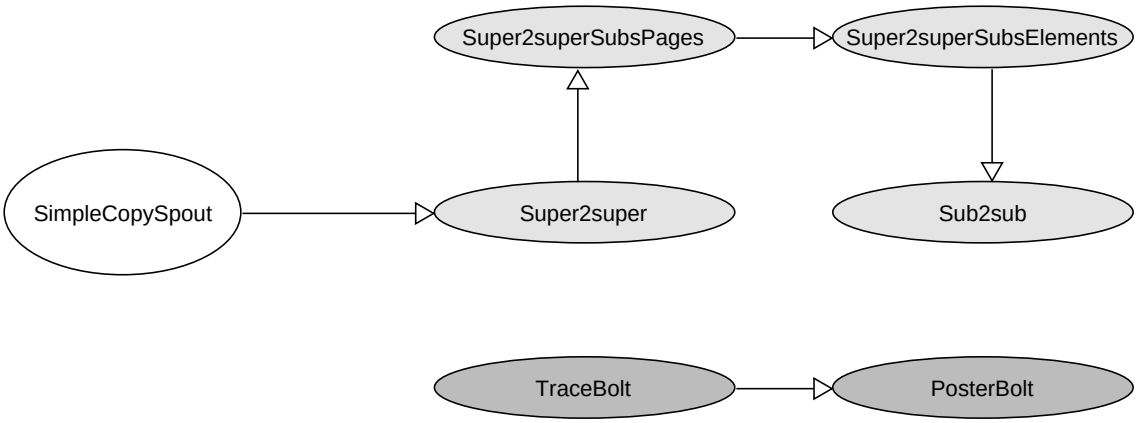

Figure 3: SimpleCopy Storm Topology.

is in charge of making the HTTP post calls in order to create the target model at a specified URL, sending the transformed JSON element. Each of the bolts generated from transformation rules will have a stream connecting them to both the `TraceBolt` and the `PosterBolt`, so that they can query elements already created or post the result of their execution (i.e., the target element created by executing it).

`TraceBolt` is special with respect to its generation, since elements will be sent to these bolts by hashing the source element, and so, this bolt is a distributed data structure. If it is queried for an element which has not yet been transformed, it will behave in a way inspired by Koan (Sánchez Cuadrado and Perera Aracil, 2014) and the use of continuations: the `TraceBolt` will keep the information of "pending" queries and serve them whenever it is available in the future.

Finally, the generator will create a SBT (Lightbend, 2016) build file so that the transformation can be compiled, imported to Eclipse or packaged into a single jar file to be uploaded and executed in a Storm cluster.

3.2 Execution

The execution of a CloudTL transformation is done by compiling and packaging the generated Java code to a jar file and uploading it to a Storm cluster, although executing it as a regular Java program is present, we recommend using the distributed cluster, so that parallelism and replication can be achieved.

The Storm cluster will then deploy this as a regular Storm topology, distributing the transformation rules (bolts) throughout the cluster. Then, when all the bolts are ready, Storm will execute the spout, which will trigger the start of the transformation process. This is simply emitting a tuple with the source URL from the root element of the input model and the target URL where the output element should be stored. Each rule will produce a JSON Object following our optimized Cloud Ecore specification and uploading it to its target URL. It will emit a message to `TraceBolt` indicating that a source element has been transformed to a target element in this rule, thus, providing this information to be queried in the future for other rules. Then, each element will be filled with the necessary information for it to be a valid Ecore JSON representation (i.e., filling in its `EClass` reference, which is extracted from the transformation definition, and all the other `EReferences` defined for `EObject`). Finally, the final JSON Object will be sent to a `PosterBolt` to be posted to the specified target URL.

The creation of the JSON Object depends on the type of `EStructuralFeature` of the target and source element. If a it is a mono-valued `EAttribute`, the source attribute will be directly copied into the resulting JSON Object, as well as if it is a initialization. A mono-valued `EReference` will be created as a pointer to the URL of the target feature of the rule that transforms those features. Code Example 3 illustrates with our running example how our language will transform some `EStructuralFeatures`.

The `TraceBolt` is needed whenever a rule needs to transform a mono-valued `EReference` whose `containment` feature is false. This means that CloudTL must not execute any rule to transform it, since it will be transformed through another rule. Thus, the `TraceBolt` must be queried in order to obtain the URL in which that element has been transformed. In case the rule in charge of transforming it has not been executed yet, it will be marked in the bolt so that whenever it enters the trace, it can be

```
init super2super {
    from src is IN::Super
    to tgt is OUT::Super {
      // This is a multi-valued
          EReference
      // Thus it will be transformed by
      // invoking a rule which can
          transform
      // a Sub source instance into a
          Sub
      // target instance (which is what
      // the sub2sub rule does)
      tgt.subs = src.subs;
    }
  }

  sub2sub {
    from src is IN::Sub
    to tgt is OUT::Sub {
      // This is a mono-valued
          EAttribute
      // Thus, it will be transformed
          by
      // copying the value of src.count
      // into the resulting JSON Object
      // in a new JSON pair.
      tgt.count = src.count;
    }
  }
```

Code Example 3: CloudTL Example.

3.3 Benchmarks

A series of benchmarks have been executed in order to test the performance of our implementation against ATL EMFTVM (Wagelaar et al., 2011) engine. We have implemented in ATL the same transformation shown in Section 3, which can be seen in Code Example 4. As it can be seen, both transformation seem identical, with some syntax differences. We have conducted ATL benchmarks by generating a Java launcher using the "ATL Plugin" wizard provided with the language in Eclipse and following the indications in (IBM, 2008a) and (IBM, 2008b). CloudTL benchmarks have been conducted by modifying manually the generated `Spout` to indicate the ellapsed seconds since the topology emitted the first tuple. Since there is no Storm benchmark solution (Apache, 2015), we have executed the `SimpleCopy` CloudTL transformation once, with one task and 1 hint parallelism as configuration for the Storm topology. We have used 3 virtual machines creating a Storm cluster:

- VM 1: responsible for the MongoDB (MongoDB, 2016) database, Cloud Ecore server and ZooKeeper, Nimbus and Storm UI servers.
- VM 2 and 3: Storm supervisors (working nodes).

```
-- @path MM=/SimpleCopyATL/
  models/Simple.ecore
-- @path MM1=/SimpleCopyATL/
  models/Simple.ecore

module SimpleCopy;
create OUT: MM1 from IN: MM;

rule super2super {
  from src: MM!Super
  to tgt: MM1!Super (
    subs <- src.subs
  )
}

rule sub2sub {
  from src: MM!Sub
  to tgt: MM1!Sub (
    count <- src.count
  )
}
```

Code Example 4: ATL SimpleCopy transformation.

Our client PC setup is the following:

- Intel i7 3770K 3.90GHz
- 16 GB RAM
- 10 Mbps downstream internet connection
- 600 Kbps upstream internet connection
- Windows 10 Pro 64 bit
- Eclipse 4.5.2 Mars
- JDK 1.8.0_111

This PC has been used for the execution of the ATL benchmark as well as the host for the VMs used for CloudTL.

Our VMs setup is the following:

- Single core virtualized Intel i7 3770K 3.50GHz
- 2 GB RAM
- 10 Mbps downstream internet connection
- 600 Kbps upstream internet connection
- Ubuntu Linux 16.04 LTS (Xenial Xerus)
- Kernel 4.4
- JDK 1.8.0_91

We will be usign the same metamodel shown Figure 2 as the input and output metamodel that will be used in this example. We have created 9 different models of different sizes, from 10000 to 90000 elements, in steps of 10000.

The results can be seen in Table 1 and graphically in Figure 5 for ATL execution times and Figure 4 for CloudTL execution times.

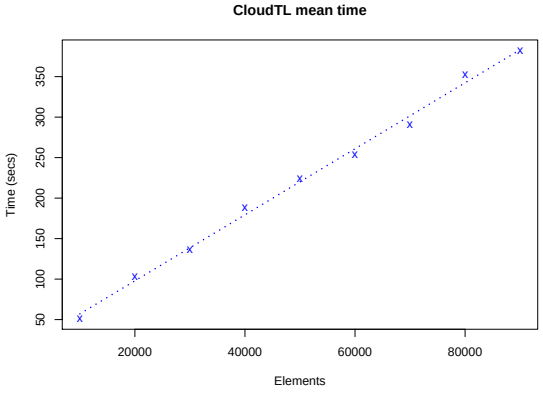

Figure 4: CloudTL Benchmarks.

The regression equation for the CloudTL benchmark is $f_{cloudtl}(x) = 4.076e^{-03}x + 16.34$ and $R^2 = 0.9983$. As shown, the algorithm complexity for CloudTL is linear ($O(n)$).

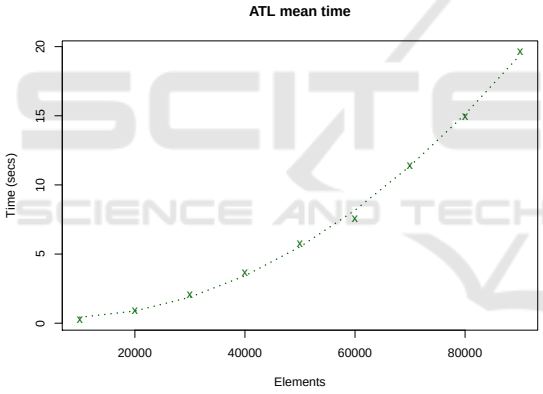

Figure 5: ATL Benchmarks.

The regression equation for the ATL benchmark is $f_{atl}(x) = 2.745e^{-09}x^2 - 3.738e^{-05}x + 0.5302$ and $R^2 = 0.9954$. As shown, the algorithm complexity is quadratic ($O(n^2)$), as it is expected due to the implementation of two passes of ATL (see (Jouault and Kurtev, 2006) Section 3.5).

If we equalize both regression functions, we can speculate that CloudTL is slower than ATL until the model size reaches around 706042.5 elements. Thus, we have created a second set of benchmarks in which the model have 10 times more elements, so that we can demonstrate that CloudTL is faster whenever the input model has a certain number of elements. The new 9 models used in this second benchmark range from 100000 to 900000 elements, by steps of 100000. The results of this second set of tests can be graphically seen in Figure 6 for ATL and Figure 7 for CloudTL, including the first set of benchmarks, demonstrating that all tests follow the same algorithm complexity.

Table 1: ATL and CloudTL execution times (in seconds).

Model Elements	ATL	CloudTL
10000	0.272	51.313
20000	0.877	102.795
30000	2.028	136.216
40000	3.672	188.648
50000	5.779	224.044
60000	7.579	253.035
70000	11.42	290.471
80000	14.906	352.694
90000	19.641	382.055

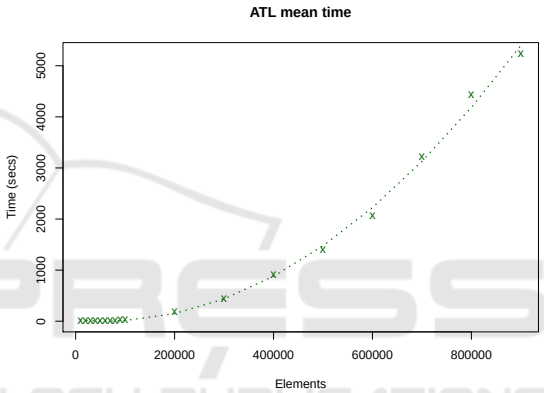

Figure 6: ATL Benchmarks (set 2).

The regression formula for ATL, taking into account the new benchmarks, is $f_{atl}(x) = 7.687e{-}09x^2 - 9.591e{-}04x + 32.28$ and $R^2 = 0.9973$.

The regression formula for CloudTL, taking into account the new benchmarks, is $f_{cloudtl}(x) = 3.220e{-}03x + 46.8$ and $R^2 = 0.9959$. Equalizing these new functions, we can deduce that CloudTL is faster than ATL when the input model has more than 547110.6 elements. The data collected in this benchmark can be seen in Table 2.

We have demonstrated that CloudTL is faster than ATL when dealing with models with a high number of elements, while it is slower when the number of elements is low due to the fact that we have to distribute and contact through the net the Storm cluster. Figure 8 shows all 4 benchmarks in a single graph, demonstrating how ATL outperforms CloudTL for input models with less than 500000 elements and how it quickly degrades for bigger input models, due to the fact that its transformation algorithm is quadratic in complexity.

It can be clearly seen how CloudTL will keep on

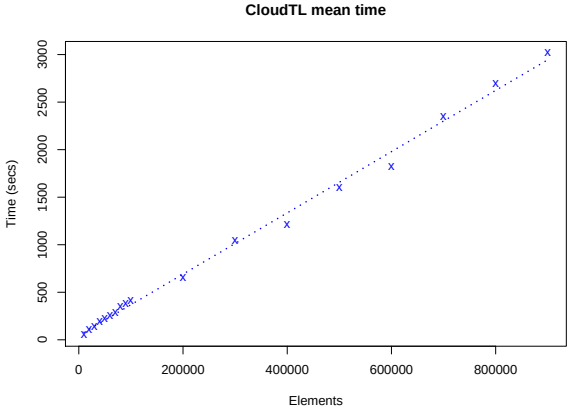

Figure 7: CloudTL Benchmarks (set 2).

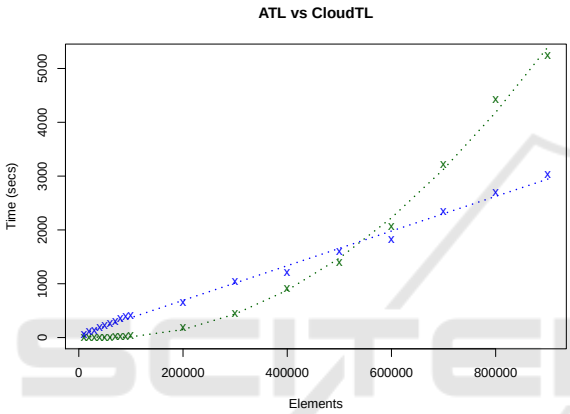

Figure 8: ATL vs CLoudTL Benchmarks.

Table 2: ATL and CloudTL execution times (in seconds).

Model Elements	ATL	CloudTL
100000	28.478	407.981
200000	176.084	651.932
300000	446.32	1047.269
400000	908.668	1216.150
500000	1396.361	1600.266
600000	2056.816	1821.210
700000	3213.287	2344.460
800000	4423.49	2694.124
900000	5243.931	3019.100

getting a better time execution than ATL for even bigger models, since the difference in execution time for two different algorithms ($O(n^2)$ and $O(n)$) gets bigger and bigger.

4 RELATED WORK

Mondo Project (Kolovos et al., 2015) aims to tackle the increasingly important challenge of scalability in MDE in a comprehensive manner. They have supported different projects and investigations which brings us closer to being able to handle VLMs in an efficient way.

Parallel ATL (Tisi et al., 2013) show an ATL engine implementation and compiler which parallelizes transformations using new opcodes for the ATL virtual machine. They have demonstrated a speedup of up to 2.5 in execution time with respect to the standard ATL engine using a CPU with 4 physical cores.

ATL-MR (Benelallam et al., 2015a) (Benelallam et al., 2015b), supported by the Mondo Project, show an implementation of ATL using Map Reduce and Hadoop using VMs. On some experiment, using up to 8 different VMs, the improvement shown has been up to 6 times faster than the default EMFTVM implementation. It is not mentioned whether they have achieved a better algorithm complexity or not, but we suppose the base ATL algorithm is used, and thus it is still $O(n^2)$.

Koan (Sánchez Cuadrado and Perera Aracil, 2014) is a transformation language that uses continuations in order to simulate the parallelism of execution of rules, and analyses data dependencies between rules to schedule automatically the execution of rules so that they can be resolved in a optimal way. If it detects a cycle, continuations will be used in order to keep running the transformation and try to create the needed element.

A roadmap (Clasen et al., 2012) has been proposed for the transformation of VLMs, in which they discuss the importance of distributing models and strategies for partitioning them.

CPU+GPU heterogeneous architectures (Fekete and Mezei, 2016) are being studied in order to build a transformation tool using OpenCL (Group, 2016).

5 CONCLUSION AND FUTURE WORK

In this paper we have presented a new model-to-model transformation language that uses Big Data technologies as its core building element. Our language demonstrates that these new technologies can be used in MDE in order to achieve faster execution times of transformations when input models are VLM. We have shown that our transformation language not only improves time execution when handling large models, but also improving the algorithm complexity of the transformation language.

Algorithm complexity is an important parameter to take into account when dealing with large inputs of data, such as when transforming VLMs. This can

actually make it difficult or even impossible to execute successfully a transformation or any handling of the model. Thus, improving it should be a priority for the next transformation languages, implementations and improvements to be able to keep using MDE for the future. We have empirically demonstrated that our implementation improves the `de facto` standard whenever a model of a defined size is used as input for a transformation.

CloudTL has demonstrated that whenever creating a new transformation language or tool, attention must be payed to the algorithm complexity of the engine that implements it, since it becomes an important issue when dealing with VLMs.

As for future work, we believe that there is still much work to do and by using big data tools, it opens up a whole new way of studying and creating model transformations.

We plan on studying using other big data tools such as Apache Spark (Apache, 2016b) or Apache Flink (Apache, 2016a) for the back-end of our the transformation language, as they are tools or have subprojects for big data streaming analysis which could be adapted to our language.

New types of rules could make the development of transformation in CloudTL easier, since they would allow for a more ways of interacting and scheduling the transformation. We plan on adding a type of rule based on `lazy rules` from ATL, which are rules that get executed only when explicitly invoked.

We are also studying the implementation of CloudTL cluster by using Amazon EC2. This way, the optimization of the virtual machines needed for a given trasnformation could be done elastically and efficiently, as well as this would help to reduce the network overhead when two bolts are in different virtual machines.

We plan to enrich the library with useful operations for the base types (`String`, `Int`...) so that we can have better statements in our transformation languages by enabling the programmer with better type support (i.e. concatenation of strings, addition of integers...).

We would like to expand the transformation language and add filters for the input elements, so that rules are executed if and only if the input element passes the filter. We are considering to incorporate a mechanism to auto-detect the init ruled based on the `EClass`es of the root metamodels from the input and ouput.

Having our language handle as input not only Cloud Ecore models, but also other types of structured data is interesting, since it would allow to bring into MDE and modeling databases and projects that have not considered it. This would require that CloudTL could infer the structure (i.e., build an internal metamodel from the structured data) using tools such as JSONDiscoverer (Cánovas Izquierdo and Cabot, 2016) as a previous step to the generation of Storm and Java code.

REFERENCES

Apache (2015). Jira for storm. https://issues.apache.org/jira/browse/STORM-642.

Apache (2016a). Flink. http://flink.apache.org/.

Apache (2016b). Spark. http://spark.apache.org/.

Apache (2016c). Storm. http://storm.apache.org/.

Apache (2016d). Zookeeper. https://zookeeper.apache.org/.

Benelallam, A., Gómez, A., and Tisi, M. (2015a). ATL-MR: model transformation on MapReduce. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems - SEPS 2015*. Association for Computing Machinery (ACM).

Benelallam, A., Gómez, A., Tisi, M., and Cabot, J. (2015b). Distributed Model-to-Model Transformation with ATL on MapReduce. In *Proceedings of 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*, Pittsburgh, United States.

Cánovas Izquierdo, J. L. and Cabot, J. (2016). JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowledge-Based Systems*, 103:52–55.

Clasen, C., Didonet Del Fabro, M., and Tisi, M. (2012). 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. Springer.

Fekete, T. and Mezei, G. (2016). Towards a model transformation tool on the top of the OpenCL framework. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, pages 355–360. Scitepress.

Foundation, E. (2016). Xtext. http://www.eclipse.org/Xtext/.

Group, K. (2016). Opencl. https://www.khronos.org/opencl/.

IBM (2008a). Robust java benchmarking, part 1: Issues. http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html.

IBM (2008b). Robust java benchmarking, part 2: Statistics and solutions. https://www.ibm.com/developerworks/java/library/j-benchmark2/.

Jouault, F. and Kurtev, I. (2006). Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer Science + Business Media.

JSON (2016). Json. http://json.org/.

Keahey, K. and Freeman, T. (2016). Nimbus. http://www.nimbusproject.org/.

Kolovos, D. S., Rose, L. M., Paige, R. F., Guerra, E., Cuadrado, J. S., de Lara, J., Ráth, I., Varró, D., Sunyé, G., and Tisi, M. (2015). MONDO: scalable modelling and model management on the cloud. In *Proceedings of the Projects Showcase, part of the Software Technologies: Applications and Foundations 2015 federation of conferences (STAF 2015), L'Aquila, Italy, July 22, 2015.*, pages 44–53.

Lightbend (2016). Sbt. http://www.scala-sbt.org/.

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., and Byers, A. H. (2011). Big data: The next frontier for innovation, competition, and productivity.

MongoDB (2016). Mongodb website. https://www.mongodb.org/.

Perera Aracil, J. M. and Sevilla Ruiz, D. (2016). Towards distributed ecore models. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, pages 209–216. Scitepress.

Sánchez Cuadrado, J. and Perera Aracil, J. M. (2014). Scheduling model-to-model transformations with continuations. *Softw., Pract. Exper.*, 44(11):1351–1378.

Tisi, M., Martinez, S., and Choura, H. (2013). Parallel Execution of ATL Transformation Rules. In *MoDELS*, pages 656–672, Miami, United States.

Wagelaar, D., Tisi, M., Cabot, J., and Jouault, F. (2011). Towards a general composition semantics for rule-based model transformation. In *Model Driven Engineering Languages and Systems*, pages 623–637. Springer Science + Business Media.