<table>
<thead>
<tr>
<th><strong>Project Partner Contact Information</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARMINES</strong></td>
</tr>
<tr>
<td>Massimo Tisi</td>
</tr>
<tr>
<td>Rue Alfred Kastler 4</td>
</tr>
<tr>
<td>44070 Nantes Cedex, France</td>
</tr>
<tr>
<td>Tel: +33 2 51 85 82 09</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:massimo.tisi@mines-nantes.fr">massimo.tisi@mines-nantes.fr</a></td>
</tr>
<tr>
<td><strong>Autonomous University of Madrid</strong></td>
</tr>
<tr>
<td>Juan de Lara</td>
</tr>
<tr>
<td>Calle Einstein 3</td>
</tr>
<tr>
<td>28049 Madrid, Spain</td>
</tr>
<tr>
<td>Tel: +34 91 497 22 77</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:juan.delara@uam.es">juan.delara@uam.es</a></td>
</tr>
<tr>
<td><strong>BME</strong></td>
</tr>
<tr>
<td>Daniel Varro</td>
</tr>
<tr>
<td>Magyar Tudosok korutja 2</td>
</tr>
<tr>
<td>1117 Budapest, Hungary</td>
</tr>
<tr>
<td>Tel: +36 146 33598</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:varro@mit.bme.hu">varro@mit.bme.hu</a></td>
</tr>
<tr>
<td><strong>IKERLAN</strong></td>
</tr>
<tr>
<td>Salvador Trujillo</td>
</tr>
<tr>
<td>Paseo J.M. Arizmendiarieta 2</td>
</tr>
<tr>
<td>20500 Mondragon, Spain</td>
</tr>
<tr>
<td>Tel: +34 943 712 400</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:strujillo@ikerlan.es">strujillo@ikerlan.es</a></td>
</tr>
<tr>
<td><strong>Soft-Maint</strong></td>
</tr>
<tr>
<td>Vincent Hanniet</td>
</tr>
<tr>
<td>Rue du Chateau de L’Eraudiere 4</td>
</tr>
<tr>
<td>44300 Nantes, France</td>
</tr>
<tr>
<td>Tel: +33 149 931 345</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:vhanniet@sodifrance.fr">vhanniet@sodifrance.fr</a></td>
</tr>
<tr>
<td><strong>SOFTEAM</strong></td>
</tr>
<tr>
<td>Alessandra Bagnato</td>
</tr>
<tr>
<td>Avenue Victor Hugo 21</td>
</tr>
<tr>
<td>75016 Paris, France</td>
</tr>
<tr>
<td>Tel: +33 1 30 12 16 60</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:alessandra.bagnato@softeam.fr">alessandra.bagnato@softeam.fr</a></td>
</tr>
<tr>
<td><strong>The Open Group</strong></td>
</tr>
<tr>
<td>Scott Hansen</td>
</tr>
<tr>
<td>Avenue du Parc de Woluwe 56</td>
</tr>
<tr>
<td>1160 Brussels, Belgium</td>
</tr>
<tr>
<td>Tel: +32 2 675 1136</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
</tr>
<tr>
<td><strong>UNINOV A</strong></td>
</tr>
<tr>
<td>Pedro Maló</td>
</tr>
<tr>
<td>Campus da FCT/UNL, Monte de Caparica</td>
</tr>
<tr>
<td>2829-516 Caparica, Portugal</td>
</tr>
<tr>
<td>Tel: +351 212 947883</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:pmm@uninova.pt">pmm@uninova.pt</a></td>
</tr>
<tr>
<td><strong>University of York</strong></td>
</tr>
<tr>
<td>Dimitris Kolovos</td>
</tr>
<tr>
<td>Deramore Lane</td>
</tr>
<tr>
<td>York YO10 5GH, United Kingdom</td>
</tr>
<tr>
<td>Tel: +44 1904 32516</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:dimitris.kolovos@york.ac.uk">dimitris.kolovos@york.ac.uk</a></td>
</tr>
</tbody>
</table>
## Contents

1 Introduction ............................................. 2

2 Distributed ATL ......................................... 3

  2.1 Running Example: Model Transformation for Data-Flow Analysis  4

  2.2 ATL on MapReduce .................................... 7

    2.2.1 ATL and MapReduce Alignment ................. 8

  2.3 Tool Support ....................................... 13

    2.3.1 Distributed Transformation Engine ........... 13

    2.3.2 Data Distribution ............................. 13

    2.3.3 Tool Limitations ............................. 14

  2.4 Experimental Evaluation ......................... 14

    2.4.1 Experiment I: Speed-Up Curve ............... 16

    2.4.2 Experiment II: Size-Speed-Up Correlation . 16

3 INCQUERY-D ........................................... 18

  3.1 Running Example: a DSL for Railways System Design .......... 19

    3.1.1 Queries ...................................... 19

    3.1.2 Transformations .............................. 20

  3.2 A Distributed Incremental Model Query Framework .......... 21

    3.2.1 Architecture ................................ 21

    3.2.2 The Rete Algorithm in a Distributed Environment . 23

  3.3 Evaluation ......................................... 26

    3.3.1 Benchmark Scenario .......................... 26

    3.3.2 Benchmark Architecture ........................ 27

    3.3.3 Results ..................................... 28
4 Related Work

4.1 Distributed Batch Computation .............................................. 31
4.2 Distributed Incremental Query Evaluation ................................. 32

5 Summary and Future Work ..................................................... 34

5.1 Contributions .................................................................... 34
5.2 Future Work .................................................................... 34
# Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>First draft</td>
<td>2 Novembre 2015</td>
</tr>
<tr>
<td>0.6</td>
<td>Full draft</td>
<td>15 Novembre 2015</td>
</tr>
<tr>
<td>1.0</td>
<td>Final version</td>
<td>23 Decembre 2015</td>
</tr>
</tbody>
</table>
Executive Summary

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 Model Query and Transformation (MQT).

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 document, we propose an approach to automatically distribute the execution of model transformations written in a popular Model Transformation 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.

We also propose a novel architecture for distributed and incremental queries, and conduct experiments to demonstrate that IncQuery-D, a prototype system, can scale up from a single workstation to a cluster that can handle very large models and complex incremental queries efficiently.
Chapter 1

Introduction

Model-Driven Engineering (MDE) is gaining ground in industrial environments, thanks to its promise of lowering software development and maintenance effort [32]. It has been adopted with success in producing software for several domains like civil engineering [62], car manufacturing [35] and modernization of legacy software systems [9]. 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 [66], e.g. in reverse engineering [9]. The AtlanMod Transformation Language (ATL) [30] 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 [34], that comes in MDE in the form of Very Large Models (VLMs) [13]. For example, the Building Information Modeling language (BIM) [62] 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.

This work Chapter 2 presents a distributed version of ATL and Chapter 3 INCQUERY-D, a distributed incremental query engine.

---

Chapter 2

Distributed ATL

One way to overcome scalability issues in Model-to-Model transformation is by 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 [15] or Pregel [39]. 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 with respect to sequential programming, linked to task synchronization and shared data access, (iii) entails complex analysis for performance optimization.

In this work we show that ATL, thanks to its high 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 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[1] 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 [45] and ETL [33]) 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 [55] [46][12]. In the MapReduce context, they allow for independent representation of queries with respect to the program logic, automatic query optimization and maxi-
m \rightarrow_{dfNext} n \iff def(m) \cap use(n) \neq \emptyset \land \exists \text{Path } m = n_0 \rightarrow_{cfNext} \ldots \rightarrow_{cfNext} n_k = n : \\
(def(m) \cap use(n)) \setminus \left( \bigcup_{0<i<k} def(n_i) \right) \neq \emptyset 

(2.1)
Figure 2.2 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 chapter we refer to an ATL implementation of the transformation named ControlFlow2DataFlow and available at the article website.

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.

Listing 2.1: ControlFlow2DataFlow - ATL transformation rules (excerpt)
However, rules can be augmented with imperative sections to simplify the expression of complex algorithms. Hereafter, 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 2.1 line 1). The query language used in ATL is the OMG’s Object Constraint Language (OCL) [44]. A significant subset of OCL data types and operations is supported in ATL. Listing 2.1 shows a subset of the rules in the ControlFlow2DataFlow transformation and Listing 2.2 an excerpt of its OCL (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 combinations of input elements satisfying that guard are matched. In ATL, a source pattern lays within the body of the clause 'from' (Listing 2.1 line 13). For instance, in the rule SimpleStmt, the source pattern (Listing 2.1 line 15) matches an element of type SimpleStmt that defines or uses at least a variable. Output patterns, delimited by the clause 'to' (Listing 2.1 line 17) 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 populate each of the features (attributes and references) of the produced elements. The binding at line 19 copies the textual representation of the instruction, and the one at line 20 fills the dfNext link with values computed by the computeNextDataFlows OCL helper. The rule for transforming methods is analogous (Listing 2.1 lines 3-11).

OCL helpers enable the definition of reusable OCL expressions. An OCL helper must be attached to a context, that can be a type or the global context. Since target models are not navigable, only source types are allowed. Listing 2.2 shows our implementation of the computeNextDataFlows helper.

```
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)
5     -=excluding( self)))
6   ->flatten();
7 helper def :isDefinedBy(start : ControlFlow!FlowInstr, input : Sequence(ControlFlow!FlowInstr),
8    end : ControlFlow!FlowInstr, visited :Sequence(ControlFlow!FlowInstr), forbidden : Sequence
9    (ControlFlow!FlowInstr) : Boolean =
10   if input->exists(i | i != end) then true
11   else let newInput : Sequence(ControlFlow!FlowInstr) = input ->collect(i | i.cfPrev) ->
12     flatten() ->reject(i | i.visited) ->exists(v | v = i) or forbidden, ->exists(f | f = i))
13     in
14   if newInput ->isEmpty() then false
15   else thisModule.isDefinedBy(start, newInput, end, visited->union(newInput)->asSet() ->
16     asSequence(), forbidden)
17   endif
18   endif;
```

Listing 2.2: ControlFlow2DataFlow - OCL helpers (excerpt)
D3.3 – Cloud-enabled query and transformation engine

derived by the direct translation in OCL of the data-flow definition we gave in Equation 2.1. It has as context \textit{FlowInstr} and returns a sequence of same type (Listing 2.2, line 1).

ATL matched rules are executed in two phases, a \textit{match phase} and an \textit{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 types are considered, bindings evaluation is left to the next phase.

The \textit{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 \textit{resolve algorithm}) for final update into the output model. The \textit{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 section 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 2.1\textsuperscript{2}). 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 \cite{9}. In our experimental evaluation we will show that already for medium sized code bases (100,000 lines of code), the processing time of the full \textit{ControlFlow2DataFlow} transformation might take several hours (more than 4 hours for our code base).

2.2 ATL on MapReduce

MapReduce is a programming model and software framework developed at Google in 2004 \cite{15}. 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 \textit{Map} and \textit{Reduce} invocations are distributed across cluster nodes, thanks to the \textit{Master} that orchestrates jobs assignment.

Input data is partitioned into a set of chunks called \textit{Splits} as illustrated in Figure 2.3. The partitioning might be monitored by the user throughout a set of parameters. If not, \textit{splits} are automatically and evenly partitioned. Every \textit{split} comprises a set of logical \textit{Records}, each containing a pair of \langle key, value \rangle.

Given the number of \textit{Splits} and idle nodes, the Master node decides the number of workers (slave machines) for the assignment of \textit{Map} jobs. Each \textit{Map worker} reads one or many \textit{Splits}, iterates over

\footnote{\textsuperscript{2}An algorithm with better efficiency, is described e.g. in the Dragonbook \cite{36}, Chapter 9.1, and is implemented with ATL in \cite{14}}.
the *Records*, processes the \(\langle \text{key}, \text{value} \rangle\) pairs and stores locally the intermediate \(\langle \text{key}, \text{value} \rangle\) 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 Figure 2.3. 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 \(\langle \text{key}, \text{value} \rangle\) 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 \(\langle \text{key}, \text{value} \rangle\) 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 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.

### 2.2.1 ATL and MapReduce Alignment

Transformations in ATL could be regarded as the union of elementary transformation tasks, where each takes charge of transforming a pattern of model elements. 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...
D3.3 – Cloud-enabled query and transformation engine

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

Figure 2.4: ControlFlow2DataFlow example on MapReduce

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 is 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 one 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. This differentiates ATL from typically recursive graph-transformation languages. This should not be confused with recursion in OCL helpers, that are responsible for 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 below.

As an example, Figure 2.4 shows how the ATL transformation of our running example could be executed on top of a MapReduce architecture comprising three nodes, two map and 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 Figure 2.4) 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**

**Algorithm 1:** Map function

```plaintext
input : Long key, ModelElement elmt

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

Figure 2.5: Local resolve of dfNext for \{int fact (int a)\}
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 the current approach we perform a random assignment. Intelligent assignment strategies for model elements, especially based on static analysis of the transformation code, are left for future work.

The pseudo-code for the processing in this phase is given in Algorithm 1. For every model element in the split, 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. 2.1). In the case of rules that match multiple elements, the map function would consider the elements of the split as the first element of the matched pattern, and look for combinations of other elements satisfying the guard. For instance, in Figure 2.2 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 the 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 the 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 Figure 2.5(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 (Figure 2.5(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 Figure 2.4 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 \in links do
2   foreach prop \in getTraceProperties(link) do // unresolved properties
3       foreach elmt \in getSourceElements(prop) do
4           trgElmt ← resolveTarget(elmt);
5           updateUnresolvedElement(prop, trgElmt);

Trace Metamodel

MT languages like ATL need to keep track during execution of the mapping between source and target elements \cite{64}. We define a metamodel for transformation trace information in a distributed setting (see Figure 2.6).

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.

![Figure 2.6: Extended Trace metamodel](image-url)
### Table 2.1: API extension

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>OPERATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecEnvironment</td>
<td>matchSingle(EObject)</td>
<td>Matching a single object</td>
</tr>
<tr>
<td></td>
<td>localApplySingle(EObject)</td>
<td>Matching and Applying if possible</td>
</tr>
<tr>
<td></td>
<td>globalResolve()</td>
<td>Resolving unresolved bindings and assignments in the global scope</td>
</tr>
<tr>
<td>TraceLinkSet</td>
<td>mergeTrace(TraceLink)</td>
<td>Add traceLink if does not exist and resolve input and output cross references</td>
</tr>
</tbody>
</table>

#### 2.3 Tool Support

##### 2.3.1 Distributed Transformation Engine

We implemented our approach as a Distributed ATL engine, whose source code is available at the tool’s website. The engine is built on top of the ATL Virtual Machine (EMFTVM \[52\]) and Apache Hadoop \[4\]. 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 set of 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 2.1). In this perspective we extended the ATL VM with a minimal set of functions (see Table 2.1) 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 \[3\].

##### 2.3.2 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

[https://github.com/atlanmod/ATL_MR/](https://github.com/atlanmod/ATL_MR/)
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 split 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 \( \frac{\text{model size}}{\text{available nodes}} \).

### 2.3.3 Tool Limitations

Currently, our ATL VM supports only the default EMF serialization format XMI. 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 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 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 [21] with support for concurrent read/write [22] on top of Apache HBase [51]. NeoEMF is also part of the WP5 (High-Performance Model Persistence Format), and presented in details in D5.6. 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.

### 2.4 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 [26] 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...
Table 2.2: Execution times and speed-up (between parentheses) per model

<table>
<thead>
<tr>
<th>#</th>
<th>SIZE</th>
<th>ELTS</th>
<th>VM</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>∼4MB</td>
<td>20706</td>
<td>244s</td>
<td>319s</td>
<td>165s</td>
<td>128s</td>
<td>107s</td>
<td>94s</td>
<td>84s</td>
<td>79s</td>
<td>75s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(×0.8)×1.5</td>
<td>(×1.9)×2.3</td>
<td>(×2.6)×2.9</td>
<td>(×3.1)×3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>∼8MB</td>
<td>41406</td>
<td>1005s</td>
<td>1219s</td>
<td>596s</td>
<td>465s</td>
<td>350s</td>
<td>302s</td>
<td>259s</td>
<td>229s</td>
<td>199s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(×0.8)×1.7</td>
<td>(×2.2)×2.9</td>
<td>(×3.3)×3.9</td>
<td>(×4.4)×5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>∼16MB</td>
<td>82806</td>
<td>4241s</td>
<td>4864s</td>
<td>2318s</td>
<td>1701s</td>
<td>1332s</td>
<td>1149s</td>
<td>945s</td>
<td>862s</td>
<td>717s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(×0.9)×1.8</td>
<td>(×2.5)×3.2</td>
<td>(×3.7)×4.5</td>
<td>(×4.9)×5.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>∼32MB</td>
<td>161006</td>
<td>14705s</td>
<td>17998s</td>
<td>8712s</td>
<td>6389s</td>
<td>5016s</td>
<td>4048s</td>
<td>3564s</td>
<td>3050s</td>
<td>2642s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(×0.8)×1.7</td>
<td>(×2.3)×2.9</td>
<td>(×3.6)×4.1</td>
<td>(×4.8)×5.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 experimentations. The first one shows a quasi-linear speed-up with respect to the cluster size for input models with similar size, while the second one illustrates that the speed-up grows with increasing model size.
2.4.1 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 [3]). 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.

Figure 2.7 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 with respect to cluster size. We naturally expect the speed-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.

2.4.2 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 Figure 2.8 and Table 2.2, 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 2.8: Execution times and speed-up on Experiment II
Chapter 3

IncQUery-D

In the previous chapter we proposed a solution for distributing batch transformations, that compute a full target model at each execution. However in several practical scenarios the source model of a query/transformation evolves in time. In these cases, recomputing a full result at each change of the source would not be judicious. Incremental evaluation of model queries aims to reduce query execution time by limiting the impact of model modifications to query result calculation. Such algorithms work by either (i) building a cache of interim query results and keeping it up-to-date as models change (e.g. EMF-IncQUERY [8]) or (ii) applying impact analysis techniques and reevaluating queries only in contexts that are affected by a change [20,47]. This technique has been proven to improve performance dramatically in several scenarios (e.g. on-the-fly well-formedness validation or model synchronization), at the cost of increasing memory consumption. Unfortunately, this overhead is combined with the increase in model sizes due to in-memory representation (found in state-of-the-art frameworks such as EMF [53]). Since single-computer heaps cannot grow arbitrarily (as execution times degrade drastically due to garbage collection problems), memory consumption is the most significant scalability limitation.

An alternative approach to tackling MDE scalability issues is to make use of advances in persistence technology. As the majority of model-based tools uses a graph-oriented data model, recent results of the NoSQL and Linked Data movement [43,12] are straightforward candidates for adaptation to MDE purposes (as experimented e.g. in Morsa [17] or Neo4EMF [5]). Unfortunately, this idea poses difficult conceptual and technological challenges as property graph databases lack strong metamodeling support and their query features are simplistic compared to MDE needs [29]. Additionally, the underlying data representation format of semantic databases (RDF [23]) has crucial conceptual and technological differences to traditional metamodeling languages such as Ecore [53]. Additionally, while there are initial efforts to overcome the mapping issues between the MDE and Linked Data worlds [25], even the most sophisticated NoSQL storage technologies lack efficient and mature support for executing expressive queries incrementally.

We aim to address these challenges by proposing a novel architecture for a distributed and incremental model query framework by adapting incremental graph pattern matching techniques to a distributed cloud based infrastructure. A main contribution of our novel architecture is that the distributed storage of data is completely separated from the distributed handling of indexing and query evaluation. Therefore, caching the result sets of queries in a distributed fashion provides a way to scale out the
memory intensive components of incremental query evaluation, while still providing instantaneous execution time for complex queries.

3.1 Running Example: a DSL for Railways System Design

In this work, we use the Train Benchmark \cite{29, 58} to present our core ideas and evaluate the feasibility of the approach. The Train Benchmark is used in the MONDO project to compare query evaluation performance of various MDE tools and it is publicly available\cite{https://github.com/FTSRG/trainbenchmark}. It is built around the railroad system defined in the MOGENTES EU FP7 \cite{54} project. The system defines a network composed of typical railroad items, including signals, segments, switches and sensors. The complete EMF metamodel is shown in Figure 3.1.

Figure 3.1: The metamodel of the Train Benchmark.

3.1.1 Queries

The Train Benchmark defines four queries which have similar characteristics to the workload of a typical MDE application. The queries look for violations of well-formedness constraints in the model. The violations are defined by graph patterns. The graphical representation of the patterns is shown in Figure 3.2. Opaque blue rectangles and solid arrows mark positive constraints, while red rectangles and dashed arrows represent negative application conditions (NACs). The result of the query (also referred as the match set) is marked with transparent blue rectangles. Additional constraints (e.g. arithmetic comparisons) are shown in the figure in text.

```
1 pattern routeSensor(Sen : Sensor) = {
2   Route(R);
3   SwitchPosition(Sp);
4   Switch(Sw);
5   Route.switchPosition(R, Sp);
6   SwitchPosition.switch(Sp, Sw);
7   TrackElement.sensor(Sw, Sen);
8   neg find noRouteDefinition(Sen, R);
```

\cite{https://github.com/FTSRG/trainbenchmark}
The queries contain a mix of join, antijoin and filtering operations. The two simpler queries involve at most 2 objects (PosLength and SwitchSensor), while the other two queries involve 4–8 objects and multiple join operations (RouteSensor and SignalNeighbor).

For the sake of conciseness, we only discuss the RouteSensor query in detail. The RouteSensor constraint requires that all sensors that are associated with a switch that belongs to a route must also be associated directly with the same route. Therefore, the query (Figure 3.2c) looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route. This query checks for the absence of circles, so the efficiency of both the join and the antijoin operations is tested.

The textual representation of the RouteSensor query, defined in INCQUERY Pattern Language, is shown in [Listing 3.1]. This query binds each variable (Sen, Sw, Sp, R) to the appropriate type. It defines the three edges as relationships between the variables and defines the negative application condition as a negative pattern (neg find).

### 3.1.2 Transformations

The Train Benchmark defines a quick fix model transformation for each query. The graphical representation of the transformations is shown in Figure 3.3. The insertions are shown in green with a «new» caption, while deletions are marked with a red cross and a «del» caption. In general, the goal of these transformations is to remove a subset of the invalid elements from the model. For example, in the case of the RouteSensor query, randomly selected invalid sensors are disconnected from their switch, which means that the constraint is no longer violated (Figure 3.3c).
### 3.2 A Distributed Incremental Model Query Framework

The queries and transformations introduced in [Section 3.1](#) represent a typical workload profile for state-of-the-art modeling tools [29]. With current MDE technologies, such workloads can be acceptably executed for models up to several hundred thousand model elements [58], however when using larger models consisting of multiple million elements (a commonplace in complex domains such as AUTOSAR [8]), the performance of current tools is often not acceptable [34]. Incremental techniques can provide a solution, however they require additional resources (memory).

The primary goal of our approach is to provide an architecture that can make use of the distributed cloud infrastructure to scale out memory-intensive incremental query evaluation techniques. As a core contribution, we propose a three-tiered architecture. To maximize the flexibility and performance of the system, model persistence, indexing and incremental query evaluation are delegated to three independently distributable asynchronous components. Consistency is ensured by synchronized construction, change propagation and termination protocols.

#### 3.2.1 Architecture

In the following, we introduce the architecture of INCQUERY-D (see Figure 3.4), a scalable distributed incremental graph pattern matcher. The architecture consists of three layers: (i) the storage layer, (ii) the distributed indexer with the model access adapter and (iii) the distributed query evaluation network.

**Storage**

For the storage layer, the most important issue from an incremental query evaluation perspective is that the indexers of the system should be filled as quickly as possible. This favors database technologies where model sharding can be performed appropriately (i.e. with balanced shards in terms of type-instance relationships), and elementary queries can be executed efficiently. Our framework can be
adapted to fundamentally different storage back-ends, including triple stores, graph databases and relational database managements systems.

**Model Access Adapter**

In contrast to a traditional setup where the distributed model repository is accessed on a per-node basis by a model manipulation transaction, INCQUERY-D provides a model access adapter that offers three core services:

1. The primary task is to provide a *surrogate key mechanism* so that each model element in the entire distributed repository can be uniquely identified and located within storage shards.

2. The model access adapter provides a *graph-like data manipulation API* (① in Figure 3.4) to the user. The model access adapter translates the operations issued by the user to the query language of the backend and forwards it to the underlying data storage.

3. *Change notifications* are required by incremental query evaluation, thus model changes are captured and their effects are propagated in the form of *notification objects* (③ in Figure 3.4). The notifications generate *update messages* that keep the state of the query evaluation network consistent with the model. While relational databases usually provide *triggers* for generating notifications, most triplestores and graph databases lack this feature. Due to the lack of general support, notifications are controlled by the model access adapter by providing a façade for all model manipulation operations.
Distributed Indexer

Indexing is a common technique for decreasing the execution time of database queries. In MDE, model indexing has a key role in high performance model queries. As MDE primarily uses a metamodeling infrastructure, all queries utilize some sort of type attribute. Typical elementary queries include retrieving all vertices of a certain type (e.g. get all vertices of the type Route), or retrieving all edges of a certain type/label (e.g. get all edges of label sensor).

To support efficient query processing, INCQUERY-D maintains type-instance indexes so that all instances of a given type (both vertices and edges) can be enumerated quickly. These indexers form the bottom layer of the distributed query evaluation network. During initialization, these indexers are filled from the database backend (2 in Figure 3.4).

The architecture of INCQUERY-D facilitates the use of a distributed indexer which stores the index on multiple servers. A distributed indexer inherently provides some protection from exceeding memory limits.

Distributed Query Evaluation Network

INCQUERY-D constructs a distributed and asynchronous network of communicating nodes that are capable of producing the results set of the defined queries (4 in Figure 3.4). Our prime candidate for this layer is the Rete algorithm, however, the architecture is capable of incorporating other incremental (e.g. TREAT [41]) and search-based query evaluation algorithms as well. In the upcoming section, we provide further details on this critical component of the architecture.

3.2.2 The Rete Algorithm in a Distributed Environment

Numerous algorithms were proposed for the purpose of incremental query evaluation. The Rete algorithm was originally proposed for rule-based expert systems [18] and later improved and adapted for EMF models in [6]. Our current work discusses how to adapt the Rete algorithm in a distributed environment.

Data Representation and Structure

The Rete algorithm uses tuples to represent the vertices (along with their properties), edges and subgraphs in the graph. The algorithm defines an asynchronous network of communicating nodes (see Figure 3.6).

The network consists of three types of nodes. Input nodes are responsible for indexing the model by type, i.e. they store the appropriate tuples for the vertices and edges. They are also responsible for producing the update messages and propagating them to the worker nodes. Worker nodes perform a transformation on the output of their parent node(s) and propagate the results. Partial query results are represented in tuples and stored in the memory of the worker node thus allowing for incremental query reevaluation. Production nodes are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation.
Construction

The system constructs the Rete network from the layout derived from the query specification. The construction algorithm may apply various optimization techniques, e.g. reusing existing Rete nodes, known as node sharing [6]. An efficient Rete construction algorithm is discussed in detail in [60].

In a distributed environment, the construction of the Rete network introduces additional challenges. First, the system must keep track of the resources available in the server cluster and maintain the mapping between the Rete nodes and the servers accordingly. Second, the Rete nodes need to be aware of the current infrastructure mapping so they can send their messages to the appropriate servers. In our system, the Rete nodes are remotely instantiated by the coordinator node. The coordinator node then sends the infrastructure mapping of the Rete network to all nodes. This way, each node is capable of subscribing to the update messages of its parent node(s). The coordinator also starts the operations in the network, such as loading the model, initiating transformations and retrieving the query results.

Operation

The operational workflow of INCQUERY-D is shown in Figure 3.5. Based on the metamodel and the query specification, INCQUERY-D first constructs a Rete network (1) and deploys it (2). In the next step, it loads the model (3) and traverses it to initialize the indexers of the Rete network. The Rete network evaluates the query by processing the incoming tuples (4). Because both the Rete indexers and the database shards are distributed across the cluster, loading the model and initializing the Rete network needs network communication. The client is able to retrieve the results (5)–(6), modify the model and reevaluate the query (7)–(9).

The modifications are propagated in the form of update messages (also known as deltas). Creating new graph elements (vertices or edges) results in positive update messages, while removing graph elements results in negative update messages. The operation of the network is illustrated on the instance graph depicted in the lower left corner of Figure 3.6. This graph violates the well-formedness constraint defined by the RouteSensor query, hence the tuple ⟨3, 4, 2, 1⟩ appears in the result set of the query. The figure also shows the Rete network containing partial matches of the original graph.

To resolve the violation, we apply the quick fix transformation defined in the Train Benchmark and delete the sensor edge between vertices 4 and 1. When the edge is deleted, the sensor type indexer (an input node) receives a notification from the model access adapter (1) and sends a negative
Figure 3.6: A transformation sequence on a distributed Rete network.

update ② with the tuple ⟨4, 1⟩. The subsequent join node processes the update messages and propagates a negative update ③ with the tuple ⟨3, 4, 2, 1⟩. The antijoin node also propagates a negative update message with the same tuple ④. This is received by the production node, which initiates the termination protocol ⑤–⑦. After the termination protocol finishes, the indexer signals the client about the successful update. The client is now able to retrieve the results from the production node. The client may choose to retrieve only the change set, i.e. only the tuples that have been added or deleted since the last modification.

Termination Protocol

Due to the asynchronous propagation of changes in Rete, the system must also implement a termination protocol to ensure that the query results can be retrieved consistently with the model state after a given transaction (i.e. by signaling when the update propagation has been terminated).

The protocol works by adding a stack to the update message propagated through the network. The stack registers each Rete node the message passes through. After the message reaches a production node, the termination protocol starts. Based on the content of the stack, acknowledgement messages (Ready) are propagated back along the network. When all relevant input nodes (where the original
update message(s) started from) receive the acknowledge messages, the termination protocol finishes. The operation of the termination protocol can be observed in Figure 3.6 (messages 5–7).

### 3.3 Evaluation

To evaluate the feasibility and performance of the INCQUERY-D approach, we created a distributed benchmark environment. We implemented a prototype of INCQUERY-D and compared its performance to a state-of-the-art non-incremental SPARQL query engine of a (distributed) RDF store.

#### 3.3.1 Benchmark Scenario

In order to measure the efficiency of model queries and manipulation operations over the distributed architecture, we adapted the Train Benchmark [29, 58] (d introduced in Section 3.1) to a distributed environment. The main goal of the Train Benchmark is to measure the query reevaluation times in systems operating on a graph-like data set. The benchmark targets a “real-world” MDE workload by running a specific set of queries (Section 3.1.1) and transformations on the model (Section 3.1.2). In this workload profile, the system runs either a single query or a single transformation at a time, as quickly as possible.

To assess scalability, the benchmark uses instance models of growing sizes, each model containing twice as many model elements as the previous one. Scalability is also evaluated against queries of different complexity. For a successful run, the tested tool is expected to evaluate the query and return the identifiers of the model elements in the result set.

#### Execution Phases

The benchmark transaction sequence consists of four distinct phases. The serialization of the model is loaded into the database (load); a well-formedness query is executed on the model (initial validation); some elements are programmatically modified (transformation) and the query is reevaluated (revalidation).

#### Instance Models

We developed a generator that creates instance models. The instance models are generated pseudorandomly, with pre-defined structural constraints and a regular fan-out structure (i.e. the in-degree and out-degree of the vertices follow a uniform distribution) [29].

#### Transformations

In the transformation phase, the benchmark runs quick fix transformations (Section 3.1.2) on 10% of the invalid elements (the result set of the initial validation phase), except for the SignalNeighbor query, where 7/8 of the invalid elements are modified. The transformations run in a single logical transaction, implemented with multiple physical transactions.
Metrics

To quantify the complexity of the benchmark test cases, we use a set of metrics that have been shown to correspond well to performance [29]. The values for the test cases are shown in Figure 3.7. The problem size numbers take the values of $2^n$ in the range from 1 to 4096. For space considerations, only every other problem size is listed. The complexity of an instance model is best described by the number of its triples, equal to the sum of its nodes and edges. The queries are quantified by the number of their variables (shown in parentheses) and their result set size (RSS). The transformations are characterized by the number of model elements modified (modification size, MS).

3.3.2 Benchmark Architecture

Benchmark Executor

The benchmark is controlled by a distinguished node of the system, called the executor. The executor delegates the operations (e.g. loading the model) to the distributed system. The queries and the model manipulation operations are handled by the underlying database management system which runs them distributedly and waits for the distributed operation to finish, effectively creating a synchronization point after each transaction.

Methodology

We defined two benchmark setups. (1) As a non-incremental baseline, we used an open-source distributed triplestore and SPARQL query system, 4store. (2) We deployed INCQUERY-D with 4store as a backend database. It is important to mention that the benchmark is strongly centralized: the coordinator node of INCQUERY-D runs on the same server as the benchmark executor.

The benchmark executor software used the framework of the Train Benchmark to collect data about the results of the benchmark. These were not only used for performance benchmarking but also to ensure the functional equivalence of the systems under benchmark.

The precise execution semantics for each phase are defined as follows. (1) The load phase includes loading the model from the disk (serialized as RDF/XML), persisting it in the database backend, and, in the case of INCQUERY-D, initializing the Rete network. (2) The execution time of the...
initial validation phase is the time required for the first complete evaluation of the query. (3) The transformation phase starts with the selection of the invalid model elements and is finished after the modifications are persisted in the database backend. In the case of INQUERY-D, the transformation is only finished after the Rete network has processed the changes and is in a consistent state. (4) The revalidation phase re-runs the query of the initial validation phase, and retrieves the updated results.

The execution time includes the time required for the defined operation, the computation and I/O operations of the servers in the cluster and the network communication (to both directions). The execution times were determined using the System.nanoTime() Java method.

Environment

We used 4store [24] (version 1.1.5) as our storage backend. The servers ran the Ubuntu 12.10 64-bit operating system with Oracle Java 7. For the implementation of the distributed Rete network, we used Akka [57] (version 2.1.4), a distributed, asynchronous messaging system.

The system was deployed on the private cloud that runs on the Apache VCL (Virtual Computing Lab) platform. We reserved four virtual machines on separate host machines, with each using a quad-core Intel Xeon L5420 CPU running at 2.5 GHz and having 16 GB of RAM. The host machines were connected to a dedicated gigabit Ethernet network.

3.3.3 Results

The benchmark results of our experiments are shown in Figure 3.8. On each plot, the x axis shows the problem size, i.e. the size of the instance model, while the y axis shows the execution time of a certain phase, measured in seconds. Both axes use logarithmic scale.

First, we discuss the results for RouteSensor, a query of medium complexity. Figure 3.8a presents the combined execution time for the load and initial validation phases. The execution time is a low order polynomial of the model size for both the standalone 4store and the INQUERY-D system. The results show that despite the initial overhead of the Rete network initialization, INQUERY-D has a significant advantage starting from medium-sized models (with approximately 1 million triples). Figure 3.8b shows the execution time for the sum of the transformation and revalidation phases. The results show that the Rete maintenance overhead imposed by INQUERY-D on model manipulation operations is low, and overall the model transformation phase when using INQUERY-D is considerably faster for models larger than a few hundred thousand triples. Figure 3.8c focuses on the revalidation phase. The performance of INQUERY-D is characteristically different from that of the SPARQL engine of 4store. Even for models with tens of millions of tuples, INQUERY-D provides close to instantaneous query re-evaluation.

Figure 3.8d–3.8f are presented to compare the results for the PosLength, the SignalNeighbor and the SwitchSensor queries, respectively. The PosLength query uses only a few variables but has a large result set. The SignalNeighbor query includes many variables but has a small match set. The SwitchSensor query uses a few variables and has a medium-sized result set.

The large result set of the PosLength query (Figure 3.8d) is a challenge for incremental query evaluation systems, however, INQUERY-D still provides reasonably fast load, transformation and
query evaluation times, while outperforming 4store on the revalidation time. The results for the SignalNeighbor query (Figure 3.8e) show INCQUERY-D has a characteristic advantage on both the transformation and the revalidation times. The SwitchSensor query also shows a clear advantage of INCQUERY-D for transformation and revalidation.

**Summary of Observations**

Based on the results, we can make the following observations. As expected, due to the overhead of the Rete construction, the non-incremental approach is often faster for small models. However, even for medium-sized models (with a couple of million triples), the Rete construction overhead already pays off for the initial validation. After the Rete network is initialized, INCQUERY-D provides significantly improved transformation and revalidation times, with the revalidation times being consistently orders of magnitude faster due to the different characteristics of their execution time.

In summary, these observations show that INCQUERY-D is not just capable of processing models with over 10 million elements (pushing the limits well beyond the capabilities of single-workstation modeling tools), but also, it provides close to instantaneous query evaluation times even for very complex queries.
Threats to Validity

To minimize internal threats to validity, we turned off the caching mechanisms of the operating system to force rereading the serialized model from the disk. Additionally, to avoid the propagation of the warmup effect of the Java Virtual Machine between the runs, each test case was started independently in a separate JVM.

As our cloud infrastructure was subject to minimal concurrent load during the measurements, we aimed to minimize the distortion due to load transients by running the benchmark three times and taking the minimum value for each phase into consideration. We did experience a certain deviation of execution times for smaller models (Figure 3.8f). However, for larger models (our most important target), the transient effects do not influence validity of the benchmark results.

Regarding external validity, we used a benchmark that is a faithful representation of a workload profile of a modeling tool for large-scale models [29,58]. The queries both for 4store and INQUERY-D were validated by domain experts. We aimed to minimize the potential bias introduced by the additional degrees of freedom inherent in distributed systems, e.g. by a randomized manual allocation of the processing nodes of Rete network in the cloud. We plan to conduct a more detailed investigation of these effects as future work.
Chapter 4

Related Work

This chapter presents the related work for distributed batch computation (Section 4.1) and distributed incremental query evaluation (Section 4.2).

4.1 Distributed Batch Computation

To our knowledge, the distributed ATL engine presented in Chapter 2 is the first application of the MapReduce programming model to model transformation. The only other proposal addressing MT distribution is Lintra, by Burgueño et al. [10], 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 [11]. 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 recent one is Mezei et al. [40]. 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).

Shared-memory parallelization is a closely related problem to distribution. For model transformation, Tisi et al. [56] 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 [27] 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 [7]. 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 \[12\], Pig Latin \[46\], and HiveQL \[55\] 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.

### 4.2 Distributed Incremental Query Evaluation

Up to our best knowledge, INCQUERY-D is the first approach to support distributed incremental query evaluation in an MDE context. Its conceptual foundations of INCQUERY-D are based on EMF-INQUERY \[8\], a tool that evaluates graph patterns over EMF models using Rete. With respect to an earlier prototype \[28\], the main contributions of the current paper are (i) a novel architecture that introduces a separate distributed indexer component in addition to the distributed data store and distributed query evaluation network (which is key distinguishing feature compared to similar tools \[42, 48, 65\]) and (ii) the detailed performance evaluation and analysis of the system with respect to a state-of-the-art distributed RDF/SPARQL engine.

A wide range of special languages have been developed to support graph-based querying over EMF \[53\] for a single-machine environment. OCL is a declarative constraint and query language that can be evaluated with the local-search based \[16\] engine. To address scalability issues, impact analysis tools \[20, 47\] have been developed as extensions.

Outside the Eclipse ecosystem, the Resource Description Framework (RDF \[23\]) is developed to support the description of instances of the semantic web, assuming sparse, ever-growing and incomplete data stored as triples and queried using the SPARQL \[63\] graph pattern language. Property graphs \[49\] provide a more general way to describe graphs by annotating vertices and edges with key-value properties. They can be stored in graph databases like Neo4j \[43\] which provides the Cypher \[50\] query language.

Even though big data storages (like document databases, column family stores or MapReduce based databases) provide fast object persistence and retrieval, query engines realized directly on these data structures do not provide dedicated support for incremental query evaluation or efficient evaluation of query primitives (like join). This inspired Morsa \[17\] and Neo4EMF \[5\] to use MongoDB and Neo4j, respectively, as a scalable NoSQL persistence backend for EMF persistence, extended with caching.
and dynamic loading capabilities. The commercial Virtuoso binds relational and RDF domains into one universal database, supporting SQL and SPARQL querying, and distributed query evaluation. While Morsa and Virtuoso use disk-based backend, Trinity.RDF is a closed source, pure in-memory solution, which executes a highly optimized local-search based algorithm on top of the Trinity distributed key-value store with low response time. However, the effect of data updating on query performance is currently not investigated.

Rete-based caching approaches have been proposed to process Linked Data (bearing the closest similarity of our approach). INSTANS uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors. Diamond evaluates SPARQL queries on Linked Data, where the main challenge is the efficient traversal of data, but our distributed indexing technique is still unique wrt. these approaches.

The Train Benchmark framework was introduced in, where the domain and scenario were defined together with four queries, and an instance model generator. In, we extended the approach by characterizing models and queries with metrics, and introducing 30 new queries, and a new instance model generator. There are numerous graph and model transformation benchmarks presented also at GRABATS and TTC tool contests, but only focuses specifically on query performance for large models.
Chapter 5

Summary and Future Work

In this work we presented a distributed ATL engine and INCQUERY-D, a distributed incremental query engine.

5.1 Contributions

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.

INCQUERY-D is a novel approach to adapt distributed incremental query techniques to large and complex model-driven software engineering scenarios. Our proposal is based on a distributed Rete network that is decoupled from a sharded graph database by a distributed model indexer and model access adapter. We presented a detailed performance evaluation in the context of quick-fix software design model transformations combined with on-the-fly well-formedness validation. The results are promising as they show nearly instantaneous complex query re-evaluation well beyond $10^7$ model elements.

5.2 Future Work

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

- I/O optimization of model processing in MapReduce by coupling with the transformation engine our 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.

For INQUERY-D, we plan to investigate the effects of various strategies for allocating the processing nodes of Rete network in the cloud [38].
Bibliography


