Applying a Data-centric framework for Developing Model Transformations

ABSTRACT

Data-centric (*Dc*) and declarative languages are being used for data processing in several application domains, such as distributed systems, natural language processing, and others. In Model Transformations (MT), recognized declarative and hybrid languages have been used to develop model transformation rules, such as ATL, QVT, or ETL. Considering the execution semantics, a large part use functional-like or graph-matching semantics. However, these new data processing frameworks have different execution capabilities and semantics, including functional, procedural or logic. In this context, it is important to take stock of existing approaches for providing efficient transformation engines, ease to use, though with alternative semantics and in a distributed way. In this paper, we present a detailed study on applying a data-centric language called *Bloom* to develop model transformations. We developed an extractor into its collection-based format and then processed them using its highly-declarative rules. The results show the feasibility of applying data centric approaches for MDE applications, which might be an attractive alternative for MT in a single-machine or distributed nodes.

CCS CONCEPTS

• **Software Engineering** → **MDE**;

KEYWORDS

Model Transformation Rules, Data-centric, Bloom Language

ACM Reference Format:

. 2019. Applying a Data-centric framework for Developing Model Transformations. In *Proceedings of ACM SAC Conference (SAC'19).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Model Transformations (MTs) are key artifacts for existing MDE (Model-Driven Engineering) approaches, since they implement operations between models [15]. ATL [20] (Atlanmod Transformation language) is one of the most used solutions for model transformations. There are several other approaches, such as QVT [26] based languages, VIATRA [19], Epsilon Transformation Language (ETL) [23]. There are also approaches based on logical programming, such as the Prolog Transformation Language (PTL) [1].

There are recent initiatives that aim to improve existing solutions by adapting computation models, for instance, using the MapReduce computation model. However, such model adds complexity of coding and it is tied to a set of initial assumptions. Besides, one important challenge is how to integrate model transformation approaches within the data-intensive computing models. Works such as Burgueño et al. [16], Pagán et al. [27], Benelallam et al. [11] and Tisi et al. [29] aim at providing solutions for this new scenario using frameworks as Linda and MapReduce. However, they do not highlight high-level, declarative and logic contexts in MT.

In a different context, new scenarios to *Dc* computing systems are arising. Most parts focus on data processing tasks, independent of their structure, i.e., without the aim to integrate with a modeling platform. One challenge is to find approaches that ease the development of such applications, beyond traditional SQL technology. Design requirements of *Dc* computing systems include a high-level programming language to the development of applications on parallel and distributed environments [2]. The MapReduce [18], Hadoop frameworks [6] have been allowing and driving the development of the distributed/parallel implicit applications. However, their coding can be not straightforward. The declarative and *Dc* languages have added different capabilities for the developers, such that focusing on domain-specific tasks [2, 9]. Moreover, these languages include built-in processing primitives to ease the specific manipulation and transformation of data. Therefore, we believe that the features of a *Dc* language may be an attractive alternative for the development of transformation rules.

For these reasons, in this paper we present a detailed study on the application of a *Dc* language design style for model transformations, particularly to extract the input models and to define transformation rules. We do not intend to create a new framework, but to take stock of existing *Dc* solutions and to apply it in a Model-Driven context. Among the existing approaches, we have chosen to investigate the Bloom language and its framework [4]. It has been proven effective in different use cases and possibly a good candidate for MT [2–4].

In order to achieve an integration, there are several challenges we addressed. First, we provide data translations between the MDE world and the Bloom format. Despite being implemented on top of the Ruby language, it has its own collection format, thus it is not possible to use directly object-like

SAC'19, April 8-12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/xx.xxx/xxx_x

structures. Second, we implement and validate the transformation rules. Their semantics and syntax need to be simple enough in order to motivate its use.

We develop a set of rules and we validate them through experiments, both in a local and a parallel setting. We present a detailed description of the language, its strong and weaker points, as well as indications that it can be a feasible alternative for model transformations.

This paper is structured in 5 sections. In Section 2, we introduce the context for this work with the MDE and the Bloom language; In Section 3, we present our specifications for centralized and parallel executions of Bloom rules and experiments them on a case study; In Section 4, we present the related works; In Section 5, we conclude with future work.

2 CONTEXT

In this section, we introduce model transformations and Bloom, a *Dc* language.

A model can be defined as an abstraction of phenomena in the real world formed by a set of model elements [25]. Operations on models are implemented as model transformations (MT) solving different tasks. According to [15], MT represent a crucial ingredient of MDE and allow the definition of mappings between different models. For instance, in a model-to-model (m2m) transformation one or more input models are processed generating one or more models as output. Indeed, a model transformation aims to provide a mean to specify and produce target models from one or more source models. Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target models. Thus, a transformation is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language [28].

There is a set of transformation rule characteristics such as domain of rule, directionality (unidirectional or multidirectional), application condition (conditions for executing the rule), parametrization (use of parameters), among others [17]. Unidirectional transformation consists in accepting one or more source models as input to produce one or more target models as output. When it is necessary to synchronize between source and target models, the kinds of transformations may vary, from being unique specifications, to source-to-target and target-to-source ones [24, 25].

Bloom is a declarative language developed by in the BOOM research project Hosted by Berkeley University[1] [14]. It is based on a formal temporal logic called Dedalus [5]. The three Bloom aspects that we will use for specifying the extraction and transformation rules [4, 14]: *Data Model*, the Bloom data model is based on collections. A collection is an unordered set of facts, which are manipulate by declarative rules (logic) on single node or more nodes (single-machine or network); *Execution Semantic*, Bloom was designed for providing a programming way, where an unordered set of declarative rules is formed by an unordered set of predicates, and it captures the object states in unordered collections; and *Concise Code*, Bloom is a high-level declarative language and Bloom programs tend to be smaller than equivalent programs which processing data in traditional imperative languages [2]. Its syntax is defined as: `<collection-variable> <op> <collection-expression>`, where the variable(s) in left-hand side (**lhs**) and the expression(s) in right-hand side (**rhs**) of the collections are related by an operator (`<op>`) [4].

In next section, we present our approach.

[1] http://boom.cs.berkeley.edu/

3 *DATA-CENTRIC* TRANSFORMATION RULES USING BLOOM

We present our approach for data-centric model transformations using the Bloom language. First, we extract the input model into the Bloom data format, since it comes from a different technical space. Second, we describe the implemented transformation rules for execution on a single node as well as on more nodes. We present forward source-to-target transformations and discuss the implications on the rule implementations.

3.1 Extracting Model Elements to Bloom

To allow the interoperability between technical spaces, we define a specific format that is used in the extraction operation, in a systematic way. It is inspired by the RDF format [31], which is based on the idea of making statements about resources in the form of *subject-predicate-object (s,p,o)* expressions, known as *triples*. The subject denotes the resource, the predicate denotes the aspects of the resource and it expresses a relationship between the subject and the object.

We define a triple (T) as a data structure formed by an element name eN, an element reference eR and an element type/value eT, structured as: T = (eN, eR, eT). These elements denote: the model element name; a value indicating the element reference; and the type/value of model element respectively. This format will be further used to specify the Bloom collections, e.g., `:input_model, [:in_model_elm, :in_model_ref, :in_model_type]`. We use this format for instantiating the output of the Extraction Operation. Figure 1 shows an overview of our approach containing the Extraction and Transformation operations.

The transformation execution flow is as follows: from the `Extractor` output (`triples`), the source model elements, conforms to (`c2`) the source meta-model are instantiated in `Ruby Classes` guided by Bloom Injector. The instances from `Ruby Classes` are used as input to Bloom `Rules`, which are transformed to target objects. The target objects also are instances of `Ruby Classes`, which we specify from the `Target Meta-model`. As a result of the transformation, they

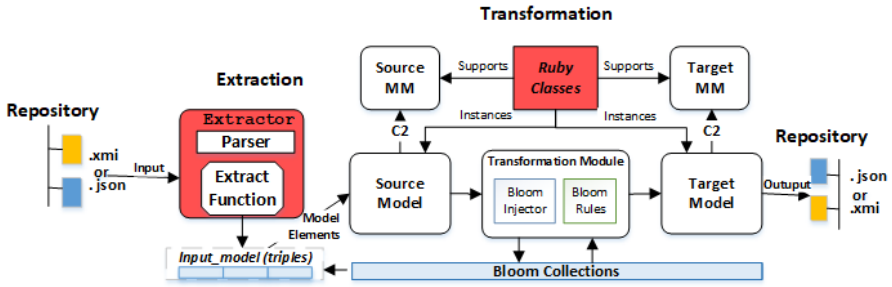

Figure 1: An Overview of Extraction and Transformation Operations

are stored in a repository with same input model format (`xmi or json`).

The source model written in JSON or XMI is assigned to a hash using a parser. Next, the hash is processed by an Extract Function, which instantiates the input model elements into Bloom collections. These collections follow the triples (eN, eR, eT) specification, and their content is used as input in a Transformation Operation. Furthermore, we use the triple format and the references among triple elements for generating the graphs for model partitioning (which will be explained later).

We use the Family model excerpt to illustrate how it is translated into the triple format and we then describe how they are mapped to Bloom elements.

Listing 1: XMI Family Model Scrap

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns="Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    ....
  </Family>
</xmi:XMI>
```

Listing 2: Family Model in Collections (triples)

```
1 [1, 0, ""]
2 [2, 1, "xmi:XMI"]
3 [3, 2, "Family"]
4 [4, 3, 0]
5 [5, 4, "father"]
6     ....
7 ["Jim", 5, "firstName"]
8 ["March", 4, "lastName"]
9 ["Families", 2, "xmlns"]
```

The reference element from each triple allows to relate them. For instance, the triple `[3, 2, "Family"]` can be linked with the triple `["March", 4, "lastName"]` by means of triple `[4, 3, 0]`. We can thus create a complete model. For instance, the `"lastName"` string is set to eN; the number `4` is set to eR; and the string "March" is set to eT. After the extraction, the Bloom collections containing the input model elements are used to instantiate the Ruby objects specified from the source meta-model.

3.2 Bloom Transformation Rules: case study

In this section, we present the implementation of model transformation using the Bloom framework. The input of the rules is a set of objects with elements encoded in the collections. We use two scenarios for assessing the feasibility of Bloom rules. First, we develop a set of rules for executing on single node. Second, we re-use the rules and injectors and we develop a prototype for running them on three nodes, in a parallel way. We implement a model partitioning clustering the elements using a clustering framework. In both scenarios, we have developed forward source-to-target transformation rules.

3.2.1 Transformation Specifications. The specification has four main steps. **First**, we take as illustrative example Family2Person transformation from the ATL zoo [21]. The transformation maps a Family representation to a Male or Female representation, according to the roles father, mother, sons and daughters. We use the source and target meta-models, shown in Figures 2 and 3 respectively, for mapping and implementing the meta-model elements as Ruby classes. For instance, the associative attributes `father, mother, sons`, and `daughters` were mapped to `role` attribute of the `class Member`. The meta-model below is implemented as Ruby classes.

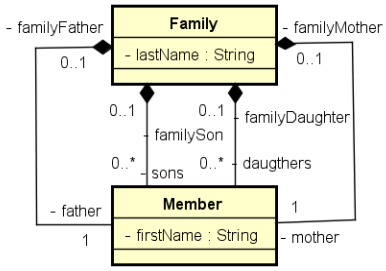

Figure 2: Families Meta-model

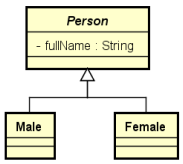

Figure 3: Person Meta-model

Second, we adopt injectors to process the source models (triples, output from Operation Extraction) and to inject them in auxiliary collections, easing the instantiation of the source objects. Listing 3 shows an Injector Implementation. This idea is similar to helpers of ATL [20] and PTL [1].

Listing 3: Injector Implementation

```
1  module Injector
2    state do
3      table :family_name, [:fn_name,:fn_ref,:fn_type]
4      table :family_members, [:fs_name,:fs_ref,:fs_type]
5      scratch :family_member, [:fm_name,:fm_ref,:fm_type]
6    end
7
8    bloom :bloom_injector do
9      family_name <= input_model_prop {|p| p if
10       p.in_elm_ptype == "lastName"}
11
12     family_member <= (family_name * input_model).pairs
13       (:fn_ref => :in_elm_ref){|f,i| i}
14
15     family_member <= (family_member * input_model)
16       .pairs(:fm_name => :in_elm_ref){|f,i|
17       [i.in_elm_name, f.fm_ref, f.fm_type] if
18       f.fm_type == "sons" or f.fm_type == "daughters"}
19
20     family_members <= (family_member*input_model_prop)
21       .pairs(:fm_name => :in_elm_pref)
22       {|f,p| [f.fm_type, f.fm_ref, p.in_elm_pname]}
23   end
24 end
```

We define the `family_name`, `family_member`, and `family_members` as auxiliary collections (lines 3 to 5). One of them is the type `scratch`, not persistent in memory after execution, and another is type `table`, persistent in memory. They are used for selecting family last names and their respective members. First, we select the `lastName` family elements (lines 9 and 10) from source model elements (`input_model_prop`). For this, we use the merge operator (`<=`) for assigning the collection-expression result (rhs) to the collection (lhs). We create the `|p|` block argument to access the elements of the `input_model_prop` collection (direct access it is not permitted), which are filtered by a conditional-expression. Next, we re-use the `family_name` collection in a join-expression for selecting family member elements (lines 12 and 13). Recursively, we create another join-expression with the `family_member` collection for selecting the son and daughter elements, when a family has more than a son or more than a daughter (lines 15 to 18). Finally, we implement the last expression at lines 20 to 22 for completing the names (`firstName`) of each family members assign them to the `family_members` collection. To create the join-expressions, we use the join operator (`*`) followed by the `pairs` method with a hash-pair `(:fn_ref => :in_elm_ref)`. A hash-pair is composed by the compatible elements of different collections contained in the join expression and by the *hashmap* operator (=>) relating them. This means that to form a hash-pair the collection elements must have matched values (referenced).

In `Family2Person` module, we specify the rules. They transform the `Family` object attributes contained in `families` collection to `male` and `female` collections. In this module, we implement the rules using two expressions: one for selecting the elements to `male` collection and another to select the elements to `female` collection. In both, we create a conditional-expression for filtering the elements from the `families` collection using `role` attribute. This means that the family roles such as father, mother, son, and daughter are translated to the gender (male or female), establishing implicit links among these elements.

Listing 4: Family2Person Rules

```
1  module Family2Person
2    bloom :family2person do
3      male <= families {|f| [[f.lastName, f.firsName]
4        .join(" ")] if (f.role == "father" or
5        f.role == "sons")}
6
7      female <= families {|f| [[f.lastName, f.firstName]
8        .join(" ")] if (f.role == "mother" or
9        f.role == "daughters")}
10   end
11 end
```

Third, we develop a monitor program to encapsulate the execution of the Extraction and Transformation Operations. It is necessary to instantiate the objects as collections, and to generate the target objects in a file; We do not show the monitor implementation since it composed by direct calls to existing components.

Last, we execute the implementations showed previously using as input the source model Family from Listing 1. The result is showed in Listing 5.

Listing 5: XMI Person Model

```
<?xmi version="1.1" encoding="US-ASCII"?>
  <Male fullName="Jim March"/>
  <Male fullName="Brandon March"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Brenda March"/>
</xmi:XMI>
```

3.2.2 Distributed Bloom Transformation. We re-use the modules from Figure 1 and we implement a prototype to execute the model transformations in a distributed/parallel way. Figure 4 shows an overview of Distributed Bloom Transformation Prototype (DBTP). We implement three additional modules: the `Graph` and `Partition` modules for allowing the partitioning models, and the `Distribution` module to distribute chunk models from the `Primary Node` (PN) to `Secondary Nodes` (SN) (Node 1, 2,..., n) [2]. We adopt the Primary/Secondary as a model of communication for the DBTP on a distributed setup. It allows us to simulate a distributed system for sending and receiving input model chunks among PN and SNs. It is necessary to partition the input models. Model partitioning is an operation that consists in extracting subsets of a model, that is splitting a model into chunks but preserving its consistency [12, 22]. `Graph Module`: it generates a directed graph ($G(V, E)$) from the extraction operation output. To generate a set of vertices (V) and edges (E) of the Graph (G), first we select the root elements from the input model and we add a sequential number for creating the vertices (lines 9 and 12 from Listing 6). Next, we create vertices to all the elements from the `input_model` collection related

[2]Primary and Secondary Nodes, following the Internet Systems Consortium (https://www.isc.org/) terminology.

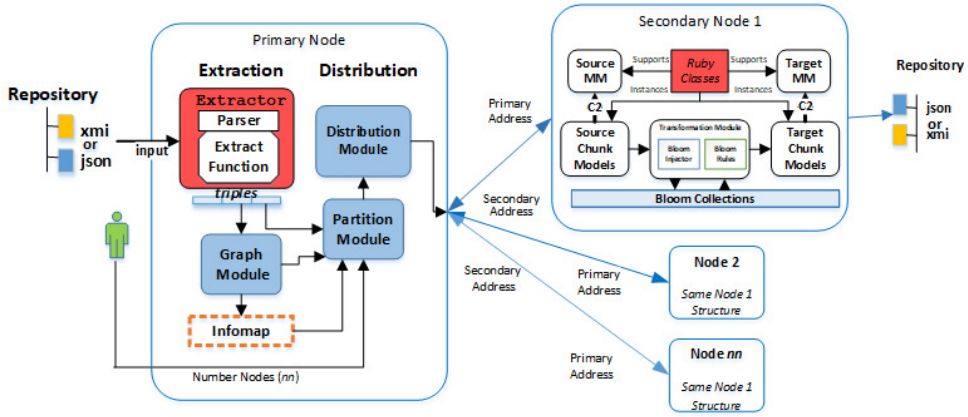

Figure 4: An Overview of Distributed Bloom Transformation Prototype

to the root elements previously selected in the `vertice` collection. To create the edges, we select all the references between the elements from `vertice` collection (lines 13 and 14).

Listing 6: Graph Module Implementation

```
1  module GenerateGraph
2    vertice = 0
3    state do
4      scratch :vertice, [:vert_elm,:vert_ref,:vert_typ,
5               :vert_nr]
6      scratch :edge, [:vert_i, :vert_j]
7    end
8    bloom :graph do
9      vertice <= input_model{|i| [i.in_elm_name,
10       i.in_elm_ref, vertice+=1]
11       if i.in_elm_type == "Family"}
12     vertice <= (vertice * input_model).pairs ...
13     edge <= (vertice * vertice).pairs(:node_elm =>
14       :node_ref){|v1, v2| [v1.vert_nr, v2.vert_nr]}
15   end
16 end
```

`Partition Module`: to guide the partitioning, we use the clustering of elements from `vertice` and `edge` collections. The development of a new clustering algorithm is not the scope of this work. However, it is necessary to ensure the consistency of the partitioned model by adopting a specific strategic. In our case, we use the `Infomap` tool and algorithms, from the MapEquation framework [13]. It is a fast stochastic and recursive search algorithm with a heuristic method based on the optimization of modularity. We create a file in the Pajek format [3] required by `Infomap`.

For instance: `*Vertices 27, 1 "node 1", ..., *Edges 33, 1 2 1, 1 3 1, ...` In Edges, we use default weight `1`. This means that all the links have the same importance degree, since we are interested in vertices clustering. Once the `.net` file is generated, we execute it with a call to `Infomap` from `Partition Module` and assign the output to the `cluster` collection:

```
imap = File.open('../Infomap/extractg.net', 'w')
imap.write ("*Vertices {prg.node.count}")
system("./Infomap extractg.net output/ -N 10 -directed -clu")
File.foreach( '../Infomap/output/extractg.clu' ) do |line|
```

In the Infomap call (`system`), we set the parameter `-N` to 10. This means that the `Infomap` algorithm iterates ten times over the vertices and edges generating a file with same name as the input file (`extractg.clu`) containing the clustering the vertices from input model. Figure 5 shows the content from the file `extractg.clu` on dispersion of Vertices for the Clusters. To achieve the result shown by Figure 5, we generate an input file (`extractg.net`) containing 1123 vertices and 2240 edges from a source model `Family` with 100 families. Whereas a family is composed by a father and a mother, it can have one to three sons or/and daughters elements. The result from the `Infomap` execution was the clustering of 1123 vertices in 100 clusters, which we use to describe the model partitioning.

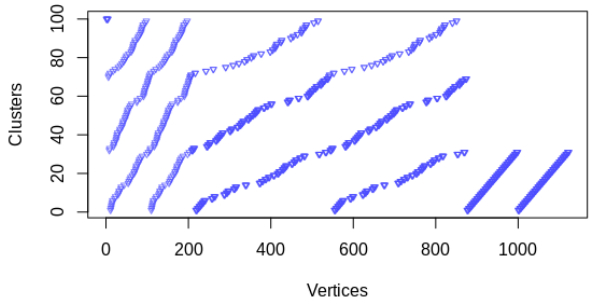

Figure 5: Clusters of Vertices Dispersion

Since the clusters are assigned to the `cluster` collection, we use the number of nodes (n_N), provided by the user, as the denominator in division of quantity of clusters (q_C). ($n_V = \frac{q_C}{n_N}$). The result is the number of vertices (n_V) used for the partitioning of the source model. For instance, given 100 clusters to divide between 5 SN (n_N), this means that each

[3] https://gephi.org/users/supported-graph-formats/pajek-net-format/

node has 20 clusters and their respective vertices. This allows us to select the input model elements for distributing among `SNs`. Nonetheless, there are constraints to the numerator (q_C) and denominator (N_n): $q_C \in (\mathbb{R}_{>0}) \wedge N_n \in (\mathbb{R}_{>1}) \wedge N_n \leq q_C$.

`Distribution Module`: in this module, we implement the partitioning and the sending of source model to SNs. Listing 7 shows code of this module.

Listing 7: Partitioning Source Model

```
1  module PartitionModel
2    state do
3      scratch range_vert,[:rg_ini, :rg_final, :rg_node]
4      table :part_model,[:pm_elm,:pm_ref,:pm_type,
5            :pm_node]
6    end
7    def range_vertices
8      puts "Enter Number of Secondary Nodes"
9      nn = gets.to_i
10     ...
11     range_vert <+ [[int_vert, fin_vert, nbr_node]]
12   end
13   bloom :partition do
14     part_model <= (range_vert * vertice).pairs{|r,v|
15       [v.vert_elm, v.vert_ref, v.vert_type, r.vert_nr]
16       if v.vert_nr >= r.rg_ini and
17         v.vert_nr <= r.rg_final}
18     part_model <= (part_model * input_model_prop).pairs
19       (:pm_elm => :in_elm_pref){|p,i| [i, p.pm_node]}
20   end
21   bloom :send_chunk do
22     pipe_chan <~ (sn_connected * part_model).pairs
23       (:id_sn =>:pm_node){|s,p| [s.addr_sn,
24       $id_msg+=1, p.pm_elm, p.pm_ref, p.pm_type]}
25   end
26 end
```

We implement the clusters division presented previously in the `range_vertices` method (lines 7 to 11) for instantiating a range of vertices to each SN into `range_vert` collection, which we use in the `bloom :partition` block. In this block (lines 13 to 20), we develop the partition model, where we specify two iterations (join-expressions) over the input model elements: first we select the elements from the `vertice` collection belonging to a range of vertices within `range_vert` collection creating a quadruple of collection elements. For instance, `[5, 4, "father", 1]`, these elements are assigned to the `part_model` collection and are sent to node 1, since number `1` establishes a relationship between them and the SN 1. Next, we implement the second iteration for selecting the input model elements from the `input_model_prop` collection related to elements contained into the `part_model` collection. Thus, the subsets of source model are instantiated to the `part_model` collection delimited by the SN id.

We present a code excerpt in Listing 8 that shows the utilization of the collections of type channel, as well as the connection among PN and the SNs. In the `ConnectProtocol` module, we define the `connect` collection to allow the connection among the PN and SN Nodes and the `pipe_chan` to support the sending of model elements from the PN to SNs. The `sn_connected` collection is used to instantiate the SN addresses (`:addr_sn`) compose by an IP (Internet Protocol) and a Port Number (endpoint of communication), and an id (`:id_sn`). In particular, we assign an IP (`127.0.0.1`) and Port (`12345`) to the PN (line 8) to ease the connection of SNs to PN. Therefore, each SN knows the PN address.

Listing 8: Primary and Second Nodes Connections

```
1  module ConnectProtocol
2    state do
3      channel :connect, [:@addr_pn, :addr_sn]
4      channel :pipe_chan, [:@dst, :idmsg] => [:pc_name,
5      :pc_ref, :pc_type]
6      table :sn_conneted, [:addr_sn, :id_sn]
7    end
8    PN_ADDR = "127.0.0.1:12345"
9  end
10
11 # On the PN side ========================
12 class PrimaryNode
13   ...
14 end
15 addrp = ARGV.first? ARGV.first : ConnectProtocol::PN_ADDR
16 ip, port = addrp.split(":")
17 pc = PrimaryNode.new(:ip => ip, :port => port.to_i)
18 pc.run_fg()
19
20 # On the SN side ======================
21 module ReceiveModel
22   include ConnectProtocol
23   state do
24     table :rcv_model, [:rc_name, :rc_ref, :rc_type]
25   end
26   bootstrap do
27     connect <~ [[PN_ADDR, ip_port]]
28   end
29   bloom :rcv_model do
30     rcv_model <= pipe_chan {|p| [p.pc_name, ...]}
31   end
32 end
33 class SecondaryNode
34   ...
35 end
36 sc = SecondaryNode.new
37 sc.run_fg()
```

On the PN side, we implement a monitor program (`primarynode.rb`) to encapsulate the execution of PN modules. We use the `pc.run_fg()` method for handling and evaluating network events. It allows the PN to wait for SN connections. This means that the PN has to be run before the SNs.

On the SN side, we specify the `rcv_model` collection to receive the model elements from the `pipe_chan` collection, which are processed by Injector and Bloom rules (specified in 3.2.1). Once PN is running, the connection of a SN is established by `bloom bootstrap` block (lines 26 to 28), whose statements are evaluated only once, before the other statements in the SN program. Thereby, the SN address is sent to the PN along with PN address. We obtain the SN address by means of the `ip_port` method. It returns the IP and Port from the Bud class instance. Although the developer may define an external IP and/or Port. As each SN connects to the PN, the SN addresses are instantiated into `sn_conneted` collection with a sequential number (starting with 1). We use it in the `bloom :send_chunk` block (Listing 7) for establishing a data pipeline among PN and SNs. Thus, each subset of the source model contained into `part_model`

collection is sent to respective SN. This is ensured by the `:id_sn => :pm_node` hash-pair in the join-expression. Furthermore, we assign an unique number, in a sequential way (`$id_msg+=1`), to each created message, since the `[:@dst, :idmsg]` pairs must be unique in the channel collection. We also highlight that for any operation involving a channel collection on the lhs, the asynchronous merge operator $<\sim$ has to be used (lhs $<\sim$ rhs). Finally, we implement a monitor program (`secondarynode.rb`) to run the SN program modules on different terminals.

3.3 Experiments

We use two scenarios to execute our approach: running on a single node (Figure 1); and running on the PN and SNs (Figure 4). In both, we utilize a single machine with the following configuration: VirtualBox 5 with Memory size 4256 MB; Ubuntu 17.10; Ruby 2.2; and Bud release 0.9.7. This virtual machine is hosted in a CPU Intel Core i5-4210U 1600 MHz Speed; Memory size 8096 MB; and 1 processor with two cores. We include in these scenarios the Class2Table (C2T) and Families2Persons (F2P) transformations. We aim to answer two questions: **Q1.** *How do the Bloom rules perform compared to another transformation language, ATL for instance on sequential execution?* **Q2.** *Is distributed model transformation in Bloom feasible?*

For all the executions, we take only into account the duration time of the transformation execution in seconds. In Bloom, we do not consider the time used for loading the models into memory or transmitting them to the SNs (distributing models in parallel executions). For running C2T in ATL, we take two of the cases from [8] Atenea Systems Modeling Group, MT Benchmark Class2Relational: case 0 (C0) consists of only one rule in ATL for transforming Class to Table (C2T); and case 5 (C5) with two ATL rules, C2T and Attribute to Column. We use the input models and the meta-models from Atenea for running on ATL and Bloom. The models size is defined in terms of the number of classes. For instance, at line 1 and column 1 from of Table 1 (10^1), there is the first input model with 10 classes, second 100, up to 1000000 classes. In this sense, we use Bloom for generating families models in xmi as input for transforming F2P. Thereby, each input model contains 10 to 1000000 families, and each family has one to three sons and daughters.

We run the ATL rules on the Eclipse Photon framework, using the option for measuring the execution time in seconds. In Bloom, we use in our implementations the `Time()` method from Ruby to measure the time in seconds. Following the implementation style presented in section 3.2.1, we implemented in Bloom the rules from cases C0 and C5. Every input model, except 10^6, is executed 7 times, having discarded (warm-up phase) the first two executions and computing the average value of five executions. Table 1 shows the result in a single node, in the attempt to answer **Q1**. Cells marked with dashes "–" indicate that the execution did not fit into memory. In the

Table 1: C2T in ATL and Bloom

size	C2T-c0		C2T-c5		F2P	
	ATL	Bloom	ATL	Bloom	ATL	Bloom
10^1	0.002	0.002	0.006	0,007	0,002	0.003
10^2	0.007	0.004	0.049	0,058	0,047	0.067
10^3	0.012	0.005	0.424	0,353	0,286	0.297
10^4	0.081	0.078	33.726	5,241	2.295	3.268
10^5	2.93	1.923	–	72.862	30.027	29.317
10^6	–	18.924	–	–	–	108.913

transformations using a single rule (C2T-c0), Bloom executes faster than ATL. However, when the transformations involve more than one rule (C2T-c5) and the number of input model elements is small, Bloom is slower than ATL. As the number of elements increases, Bloom performs slightly better that ATL. However, in both the speed-up is negatively influenced by model size and by amount of rules being executed. There are other frameworks that could be compared, though we chose a baseline simple ATL, since we assume it is the most used scenario.

To answer **Q2**, we first execute the transformations in a centralized setting (CS). Then we partition the same input models (described in Section 3.2.2) to run on three SNs. Table 2 shows execution results. Note that in all finished

Table 2: Local and Distributed Bloom Rules Executions

Size	CS-C2T	SN-C2T	CS-F2P	SN-F2P
10^3	**0.004**	sn1 = 0.002 sn2 = 0.002 sn3 = 0.001 **0.005**	**0,306**	sn1 = 0,178 sn2 = 0.159 sn3 = 0.148 **0.485**
10^4	**0.089**	sn1 = 0.038 **0.076**	**3.317**	sn1 = 0.938 **3.074**
10^5	**2.138**	sn1 = 0.672 **1.931**	**30.730**	sn1 = 10.149 **27.691**
10^6	**19.085**	sn1 = 6.537 **17.943**	**110.582**	sn1 = 34.603 **106.428**

executions, except of size 10^3, the total execution time on parallel setups such as SN-C2T and SN-F2P is smaller than the execution time on a single node, CS-C2T and CS-F2P. This means that the execution of parallel transformation in Bloom can be feasible. However, other aspects on distributed/parallel executions need to be better evaluated, such as exploring different partitioning models.

3.3.1 Discussions. The Bloom model is flexible and structured and any metamodel and model can be instantiated by the Extractor. The collected approach may ease the data modelling for distinct transformation scenarios (local and distributed/parallel). Syntactically and semantically the constructs of the Bloom language are relatively simple, though some join operation change the way of rules are developed if compared to existing rule-based frameworks. In addition to the collections, Bloom has a particular set of operators that act directly over data collections. The capability to process two different model formats (JSON and XMI) can

be considered a differential of our approach to those that accept only the XMI format. Another important aspect is the declarative style to encode the rules in rhs expressions. However, it requires a shift in thinking in order to change the context from a modeling technical space to the Bloom technical space. Furthermore, the inappropriate use of join-expressions can decrease the execution performance. In our experiments, the executions in Bloom were slightly faster than the executions on a plain ATL setting. Nevertheless, we still can not consider the Bloom transformations faster than any ATL transformations or setting. As the ATL, Bloom also presented problem with memory usage for loading and processing very large models.

Regarding Bloom language, it was embedded in a DSL, in this case, into the Ruby language. For this reason, it may be easier to integrate it on already existing programming environments, without the need for a separate editor/maintenance framework. The data representation in triples allowed to instantiate and to recover model elements from/to Bloom collections preserving its references and the connectivity of models, as well as generating graphs as input to model partitioning. Still, is an additional format that need to be taken into account. It would probably easier to be understood by developers with familiarity with RDF-based representations than the ones familiar with OO models. In the distributed/-parallel context, our approach can be alternative when the centralized transformation executions are not feasible. Nevertheless, aspects such as the sending of sub-models to SNs (throughput in data transmission), the model partitioning strategy (partitioning balanced), and performance improvements (memory usage) need to be better investigated. The linking between elements from different nodes is handled by making copy of connected elements in different clusters. This means that fully connected models may not be adequate for distribution. Furthermore, we did not explore yet the junction of target sub-models on a single node or on a decentralized persistence back-end. Also a benchmark involving a diversified set of transformation scenarios is necessary.

4 RELATED WORK

MDE approaches have already being reinterpreted under different views, for instance, Batory and Azenza [10] do a reinterpretation under the context of relational databases. To ease the understanding of MDE approaches they employ a *Dc* approach and a declarative language to model transformations. They map metamodels to relational tables and the Prolog language to write declarative constraints in m2m transformation. This approach has some aspects resembling our work such as: the mapping of models and metamodels to tuples; and use of *Dc* and declarative styles. However, it was elaborated for helping to explain the MDE concepts under relational perspectives, whereas we search to offer an approach for model transformation in *Dc* approach.

Based on logic programming, Almendros-Jiménez et al. [1] created a framework for model transformations using Prolog language for specification of MT constraints. To evaluate it, they create the PTL language. The approach uses declarative and logic styles for specifying transformation rules and constraint validations, and the use of the RDF library of SWI-Prolog for storing models as RDF triples. However, PTL is not evaluated for distributed or/and parallel environments.

Prolog was used by Varró [30] on top of VIATRA [19]. He presents an automated solution for generating transformation programs as implementations of a model transformation system. The models are stored as facts and modified at runtime. Graph pattern matching is accomplished by unification and VIATRA control structures are implemented as Prolog predicates. The representation model and declarative implementation are aspects that are closely to our approach.

Based on big data tools, Aracil and Ruiz [7] proposed CloudTL. They demonstrate that the big data technologies can be used in MDE to achieve faster execution times of transformations when input models are large. The CloudTL is inspired in ATL language and uses the Apache Storm as the backend for the engine. JSON objects are used for helping in distributing the models. Our approach is motivated by data processing frameworks and also aims at easing the development of MT using a new language.

Benelallam et al. [11, 12] present the ATL-MapReduce as a distributed MT engine. They embed the ATL on the MapReduce framework for obtaining an implicit distribution of ATL rules, achieving distributed execution. Using static analysis, they proposed a model partitioning for balancing and preserving the dependency among model elements by means of a greedy distribution algorithm. The strategy is relevant for applying an algorithm for balancing the partitioning.The ATL-MapReduce solution is dependent of the MapReduce framework and an implicit distribution of models, as well as the transformation executions on two phases (map and reduce).

5 CONCLUSION

In this paper, we applied a *Dc* framework for model transformations to assess if it is a valid alternative to the development of transformation rules. The main motivation was to experiment using a framework outside of the modeling world in order to identify its advantages or weaker points. We developed a set of transformation rules on the Bloom language and validated them with experiments using centralized and parallel executions. The obtained results indicate that the Bloom language can be an alternative for specification and execution of transformation rules on single node or distributed/parallel nodes. They also reveal that there is a set of open questions about the use of the *Dc* approach to MDE tasks. The syntax is simple, is integrated with Ruby, but many new operations have different execution semantics than the most known transformation languages. This means it would

need a change on the way of thinking for the developers. The distribution capabilities are useful, but need further studies to apply more sophisticated strategies, not a ready-to-use framework, As future work, we plan running Bloom rules on distributed environments as cloud computing aiming a benchmark with Very Large Models, to evaluate partition model strategies prioritizing the load balancing of sub-models, and to include a mechanism for jointing the target model chunks.

REFERENCES

[1] Jesús M. Almendros-Jiménez, Luis Iribarne, Jesús López-Fernández, and Ángel Mora-Segura. 2016. PTL: A model transformation language based on logic programming. *Journal of Logical and Algebraic Methods in Programming* 85, 2 (2016), 332 – 366.

[2] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. ellerstein, and Russell Sears. 2010. Boom Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/1755913.1755937

[3] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, Russell C Sears, Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2009. *Boom: Data-centric programming in the datacenter*. Technical Report UCB/EECS-2009-113. EECS Department, University of California, Berkeley.

[4] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011*. CIDRDB, CA, USA, 249–260.

[5] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. 2009. *Dedalus: Datalog in Time and Space*. Technical Report UCB/EECS-2009-173. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-173.html

[6] Hadoop / 04 August 2017 / Release 2.7.4 Apache. 2017. Apache Software Foundation. Retrieved 2018-07 from http://hadoop.apache.org/

[7] Jesús M. Perera Aracil and Ruiz Diego Sevilla. 2017. CloudTL: A New Transformation Language based on Big Data Tools and the Cloud. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD, Prague, Czech Republic, 137–146.

[8] Systems Modeling Group Atenea. 2018. MT, Benchmark. Retrieved 2018/08 from http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTBenchmark

[9] David Basin, Manuel Clavel, Marina Egea, Miguel A. García de Dios, Carolina Dania, Gonzalo Ortiz, and Javier Valdazo. 2011. Model-driven Development of Security-aware GUIs for Data-centric Applications. In *Model-driven Development of Security-aware GUIs for Data-centric Applications*, Alessandro Aldini and Roberto Gorrieri (Eds.). Springer-Verlag, Berlin, Heidelberg, Chapter Foundations of Security Analysis and Design VI, 101–124.

[10] Don Batory and Maider Azanza. 2017. Teaching model-driven engineering from a relational database perspective. *Software & Systems Modeling* 16, 2 (May 2017), 443–467.

[11] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2015. Distributed Model-to-model Transformation with ATL on MapReduce. In *2015 ACM SIGPLAN Software Language Engineering (SLE 2015)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2814251.2814258

[12] Amine Benelallam, Massimo Tisi, Jesús Sánchez Cuadrado, Juan de Lara, and Jordi Cabot. 2016. Efficient Model Partitioning for Distributed Model Transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. ACM, New York, NY, USA, 226–238. https://doi.org/10.1145/2997364.2997385

[13] L. Bohlin, D. Edler, Lancichinetti A., and Rosvall M. 2014. MapEquation Framework. Retrieved 2018/08 from http://www.mapequation.org

[14] Project BOOM, Berkeley Orders Of Magnitude. 2011. Bloom, Bloom Language. Retrieved 2018/07 from http://boom.cs.berkeley.edu/

[15] M. Brambilla, J. Cabot, and M. Wimmer. 2012. *Model-Driven Software Engineering in Practice* (1 ed. ed.). Vol. 1. Morgan & Claypool, Williston, USA.

[16] Loli Burgueno, Manuel Wimmer, and Antonio Vallecillo. 2016. A Linda-based Platform for the Parallel Execution of Out-place Model Transformations. *Inf. Software Technology* 79 (Nov. 2016), 17–35. https://doi.org/10.1016/j.infsof.2016.06.001

[17] K. Czarnecki and S. Helsen. 2006. Feature-based Survey of Model Transformation Approaches. *IBM System Journal* 45, 3 (jul 2006), 621–645.

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113.

[19] Foundation. Eclipse. 2017. Eclipse 2017/08/02 1.6.1. http://www.eclipse.org/viatra/. Accessed in 2018/1.

[20] Foundation Eclipse. 2018. ATL - A Model Transformation Technology. Retrieved 2018/08 from https://projects.eclipse.org/projects/modeling.mmt.atl

[21] Foundation Eclipse. 2018. ATL - Transformations list. Retrieved 2018/07 from http://www.eclipse.org/atl/atlTransformations/

[22] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, and Y. Le Traon. 2015. Stream my models: Reactive peer-to-peer distributed models@run.time. In *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, MODELS-2015, Ottawa, Canada, 80–89.

[23] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2008. *The Epsilon Transformation Language*. Springer Berlin Heidelberg, Berlin, Heidelberg, 46–60.

[24] N. F. M. Macedo. 2014. *A Relational Approach to Bidirectional Transformation*. Ph.D. Dissertation. Universidade do Minho, Minho, POR.

[25] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.* 152 (mar 2006), 125 – 142.

[26] OMG. 2016. QVT Query View Transformation, formal/2016-06-03 v1.3. http://www.omg.org/spec/QVT. Accessed in 2018/06.

[27] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. 2015. A repository for scalable model management. *Software & Systems Modeling* 14, 1 (2015), 219–239. https://doi.org/10.1007/s10270-013-0326-8

[28] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. 2012. A formal approach to the specification and transformation of constraints in MDE. *The Journal of Logic and Algebraic Programming* 81, 4 (2012), 422 – 457.

[29] Massimo Tisi, Salvador Martínez, and Hassene Choura. 2013. Parallel Execution of ATL Transformation Rules. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*. Springer-Verlag New York, Inc., New York, NY, USA, 656–672. https://doi.org/10.1007/978-3-642-41533-3_40

[30] Daniel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15, 3 (Jul 2016), 609–629.

[31] W3C. 2014. RDF 1.1 Concepts and Abstract Syntax. Retrieved 2018/07 from https://www.w3.org/TR/rdf11-concepts/