Journal of Software Engineering Research and Development, 2021, 9:10, doi: 10.5753/jserd.2021.477

A Data-centric Model Transformation Approach using Model2GraphFrame Transformations

Luiz Carlos Camargo [**Universidade Federal do Paraná C3SL Labs** | ***lccamargo@inf.ufpr.br***]
Marcos Didonet Del Fabro [**Universidade Federal do Paraná, C3SL Labs** | ***didonet@inf.ufpr.br***]

Abstract

Data-centric (Dc) approaches are being used for data processing in several application domains, such as distributed systems, natural language processing, and others. There are different data processing frameworks that ease the task of parallel and distributed data processing. However, there are few research approaches studying on how to execute model manipulation operations, as model transformations models on such frameworks. In addition, it is often necessary to provide extraction of XMI-based formats into possibly distributed models. In this paper, we present a *Model2GraphFrame* operation to extract a model in a modeling technical space into the Apache Spark framework and its *GraphFrame* supported format. It generates *GraphFrame* from the input models, which can be used for partitioning and processing model operations. We used two model partitioning strategies: based on sub-graphs, and clustering. The approach allows to perform model analysis applying operations on the generated graphs, as well as Model Transformations (MT). The proof of concept results such as *model2GraphFrame*, *GraphFrame* partitioning, *GraphFrame* connectivity, and GraphFrame model transformations indicate that our Model Extraction can be used in various application domains, since it enables the specification of analytical expressions on graphs. Furthermore, its model graph elements are used in model transformations on a scalable platform.

Keywords: *Model Extractor, Data-centric approach, Spark GraphFrames, Model Transformations*

1 Introduction

Model Transformations (MTs) are key artifacts for existing MDE (Model-Driven Engineering) approaches, since they implement operations between models (Brambilla et al., 2012). Nevertheless, the transformation of models via parallel and/or distributed processing is still a challenging question in MDE platforms. There are recent initiatives that aim to improve existing solutions by adapting the computation models, for instance, using MapReduce (Dean and Ghemawat, 2008) to integrate model transformation approaches within the data-intensive computing models. Works such as Burgueno et al. (2016), Pagán et al. (2015), Benelallam et al. (2015) and Tisi et al. (2013) aim at providing solutions for this new scenario using frameworks such as Linda and MapReduce. Even when adopting these frameworks, the model processing is not a straightforward task, since the models are semi-structured, which can have self-contained or inter-contained elements, different of flat data structures on linear space usage, such as logs, text files, and others.

The need for performing complex processing on large volumes of data has led to the re-evaluation of the utilization of different kinds of data structures (Raman, 2015). Very Large Models (VLMs) are composed of millions of elements. VLMs are present in specific domains such as the automotive industry, civil engineering, Software Product Lines, and modernization of legacy systems (Gómez et al., 2015). Furthermore, new applications are emerging involving domains, such as Internet of Things (IoT), open data repositories, social networks, among others, demanding intensive and scalable computing for manipulating their artifacts (Ahlgren et al., 2016).

There is a wide range of approaches of model transformations (Kahani et al., 2018), such as QVT (OMG, 2016), ATL, ETL (Kolovos et al., 2008), VIATRA (Varró et al., 2016), among others. However, most of these approaches adopt as strategy the local and sequential execution for the transformation of models, conditioning the processing of models with large amounts of elements (VLMs) to the capacity of the execution environment.

Given the nature of models and meta-models, they can have elements that are densely interconnected. This hardens the processing of transformation rules, mainly when executing a pattern matching step (Jouault et al., 2008). Moreover, distributed Model Transformation (MT) requires strategies for partitioning and distributing the model elements on distinct nodes, while at the same time, ensuring the consistency among their elements (Benelallam et al., 2018).

A large part of model-based tools uses a graph-oriented data model. These tools have been designed to help users in specifying and executing model-graph manipulation operations efficiently in a variety of domains (Xin et al., 2013; Szárnyas et al., 2014; Junghanns et al., 2016; Shkapsky et al., 2016; Li et al., 2017; Benelallam et al., 2018; Tomaszek et al., 2018; Azzi et al., 2018). The extraction of large semi-structured data under a graph perspective can be useful in choosing a strategy to design distributed/parallel MTs, graph-data processing, model partitioning, and to analyze model inter-connectivity, as well as to offer graph-structured information to different contexts. Even though, the graph processing in the MT context requires more research, involving implicit parallelism, parallel/distributed environments, lazy-evaluation, and other mechanisms for model processing.

For these reasons, in this paper, we present an evaluation study on the application of a Data-centric (Dc) approach for model extraction and MT in the Spark framework, based on GraphFrames (Apache, 2019). Therefore, we consider that the mechanisms, such as implicit parallelism, lazy-

evaluation, model partitioning, and scalable framework, can compose an approach for MT.

First, we inject the input model into a DataFrame, which is a format supported by Apache Spark. Second, we implement in Scala a model extraction with graph generation from the DataFrame and its schema. It translates the input models into GraphFrame from a DataFrame, through a Model2GraphFrame transformation, which allows us to process them. We evaluate how to query the graph elements using its native query language, and also, how to specify different kinds of operations over GraphFrames. We focus on the partition of graphs from GraphFrames into sub-graphs, as well as the clustering of its vertices, which are used in Model Transformations. We provide the following contributions:

- We produce an automated mechanism for data translations between the MDE technical space and the DataFrame and GraphFrame formats, which allows the execution of different operations (including MT) over the models from the GraphFrame;
- We use two partitioning strategies of models on GraphFrame (semi-automated), one based on the Motif algorithm and another on clustering using the Infomap framework. The model partitioning result is used on MT, aiming to improve the execution performance;
- To validate our approach, We implemented a proof of concept, in which we compared the partitioning strategies in MT executions on top of the Spark, a scalable framework.

This paper is organized into 6 sections. In Section 2, we introduce the context for this work with the DataFrame and GraphFrames APIs and their data formats, as well as Model Transformations using Graphs; In Section 3, we present the specifications of our approach, including extracting, translating, partitioning, and model transformations; In Section 4, we describe the proof of concepts for validating our approach; In Section 5, we present related work; In Section 6, we conclude with future work.

2 Context

In this section, we present DataFrame, a distributed collection of data organized into named columns, and GraphFrames, a graph processing library based on DataFrames, both for Apache Spark. We also introduce: the MT, the key artifact for existing MDE approaches; Model Extractor (ME) for extracting model elements from different technical spaces; and Graph, a data structure composed of vertices and edges, which may be used in MT.

2.1 Data Structures on GraphFrame

Apache Spark (Apache, 2019) is a general-purpose data processing engine providing a set of APIs that allow the implementation of several types of computations, such as interactive queries, data and stream processing, and graph processing. The DataFrame Spark API uses distributed Datasets. A Dataset is a strongly-typed data structure organized in collections. The Dataset API allows the definition of a distributed collection of structured data from JVM objects, and its manipulation using functional transformations such as `map`, `flatMap`, `filter`, and others.

Structurally, a DataFrame is a two-dimensional labeled data structure with columns of potentially different types. Each row in a DataFrame is a single record, which is represented by Spark as an object of type `Row`. Each DataFrame contains data grouped into named columns, and keeps track of its own schema. Summarizing, a DataFrame is similar to a table in a relational database, but with a difference, their columns allow the manipulation of multivalued attributes. A DataFrame can be transformed into new DataFrames using various relational operators available in its API and expressions based on SQL-like functions. DataFrames and Datasets are (distributed) table-like collections with well defined rows and columns. Each column must have the same number of rows and each column has type information that must be consistent for every row in the collection. DataFrames and Datasets represent immutable and lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output (Chambers and Zaharia, 2018). Figure 1 shows an example of a DataFrame. It is formed by three rows and five columns, and contains data extracted from model Families (Rows with `March, Sailor, and Camargo families`. A Row can have Columns with different types, such as `String, Integer, Date, Boolean`, and `Array`.

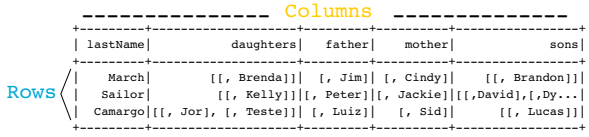

Figure 1. DataFrame Families

Another possible way to describe elements and their relationships is the creation of graphs, due to their high expressiveness. Spark provides the GraphX and GraphFrames APIs to process data in graph formats. In the GraphFrames API, the GraphFrame class is used for instantiating graphs. In Figure 2, we present a simple illustrative example of a Family model, using the March family elements into a GraphFrame instance. It can be created from vertex (`nameVerticesDF`) and edge (`roleEdgesDF`) DataFrames. A vertex DataFrame has to contain a special column named `"id"`, which specifies a unique ID for each vertex in the graph. An edge DataFrame should contains two special columns: `"src"` (as the source vertex ID of the edge) and `"dst"` (as the destination vertex ID of the edge) (Chambers and Zaharia, 2018; Apache, 2019).

The GraphFrame model supports user-defined attributes within each vertex and edge. The GraphFrames API provides the same operations of the DataFrame API, such as `map, select, filter, join`, and others. It has a set of built-in graph algorithms, such as breadth-first search (BFS), label propagation, PageRank, and others. The GraphFrames and DataFrame APIs are based on the concept of a Resilient Distributed Dataset (RDD), which is an immutable collection of records partitioned across a number of computers or nodes. To provide fault tolerance, each RDD is logged to construct

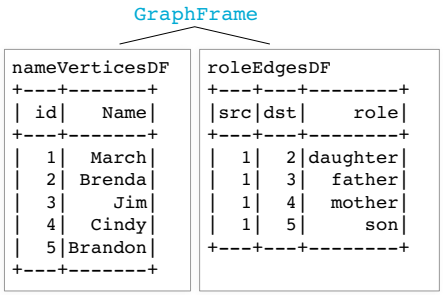

Figure 2. March Family GraphFrame

a lineage Dataset (Data lineage (Tang et al., 2019)). When a data partition of a RDD is lost due to the node failure, the RDD can recompute that partition with the full information on how it was generated from other RDD partitions (Apache, 2019).

2.2 Model Transformations using Graphs

A directed graph may be represented by $(G(V, E))$, where V represents a set of vertices and E the set of edges of the graph G. A sub-graph S of a graph G is a graph whose vertices $V(S)$ are a sub-set of the set of vertices $V(G)$, where $V(S) \subseteq V(G)$, and the set of edges $E(S)$ is a sub-set of the edges $E(G)$, that is, $E(S) \subseteq E(G)$. Extensions of this basic representation have been proposed to define the graph as a data model (Junghanns et al., 2016; Barquero et al., 2018).

Graphs are useful for modeling computational problems. They can be adopted to model relationships among objects. A graph can be used, such as a representation format for models, enabling abstract features of a model. In model transformation processes, graphs can be used to translate instances from one modeling language to another, since the structures of that language can be represented by a type of graph. The Triple Graph Grammars approach (Schürr, 1995) is a way to specify translators of data structures and to check their consistency. In addition to model transformation, there is a variety of based-graph algorithms used for processing graph models in different domains, such as complex network structures, network analysis, business intelligence, and others (Junghanns et al., 2016; Löwe, 2018).

Graph transformation has been widely used for expressing model transformations, since graphs are well suited to describe the underlying structures of models and meta-models. Operations are implemented as model transformations solving different tasks. A transformation is a set of rules that describe how a model in the source language can be transformed into a model in the target language (Rutle et al., 2012). The extraction is a process that transcribes model/meta-model elements from the native source platform to the target platform (Jia and Jones, 2015). This is necessary mainly when the input model comes from a different technical space (e.g., input model is in the XMI format and the transformation platform works on data collections).

3 A Data-centric Approach for MT

In a previous work (Camargo and Fabro, 2019), we presented a study on applying a data-centric language called Bloom (Alvaro et al., 2011) to develop model transformations. There are three major differences from the previous study to this paper: a) We define a specific format based on RDF (W3C, 2014), and we used it in the injection/extraction operations for translating source model in new modeling domain; b) We implement the RDF models in data collections and specify transformation rules, mapping the source and target meta-models and models elements as Ruby classes; and c) We choose the Bloom language, a Data-centric declarative language, since it is based in collections (unordered sets of facts) and provides implicit-parallelism. On the other hand, the use of the Data-centric approach, and parallel model transformations are the main similarities between these works.

The proposed approach in this work is built on top of the Apache Spark framework, using Dc aspects such as high-level programming, parallel/distributed environments, and considering that a model element is a set of data. It allows the extraction of models and meta-models in different formats and transforming them to a directed-graph, which is assigned to a GraphFrame. The transformation output is the input to process graph operations and model transformations. In order to improve the performance of transformation executions, we use two different strategies for partitioning models from GraphFrame. Figure 3 shows an overview of our approach. There are arrows between Spark components, mainly in Spark Context. It is the responsible for managing all executions on the Spark framework. The arrows among the approach modules (2, 3, and 4) represent the interaction between them and their outputs, forming a workflow. All the steps of the workflow are automated, except for the Operation on Graph to the partitioning of models (semiautomated). We describe these steps in the next sections.

The Driver Node controls the execution of a Spark Application and maintains all states of a Spark cluster. It exchanges messages with the Cluster Manager in order to obtain physical resources and launch executors (Worker Nodes). The Executor is the process that performs the tasks assigned by the Spark driver. The Executors have the responsibility to receive the tasks (Task) assigned by the driver, run them, and report back their state and results. The interaction between the Work Nodes and Spark Context is supported by a Cluster Manager, which is responsible for maintaining a cluster of machines (nodes) that will run one or more Spark Applications (Chambers and Zaharia, 2018; Apache, 2019). In our approach, the modules 2 and 3 are executed on the Driver Node. The Injector module is responsible for extracting the input model to the DataFrame, which is transformed into a GraphFrame by the Model Translator module. The Model Transformation (module 4) is executed on Worker Node(s).

For the Module 3, we create a meta-model to instantiate the result of the translation of the input model to a graph model. It is necessary for assuring the conformance and consistency of translation output. Such meta-model is based on the GraphDB meta-model proposed by (Daniel et al., 2016), which focuses on NoSQL graph databases. Figure 4, depicts our Graph Meta-model, where GraphElement represents all elements of a graph. Their sub-types, Graph Vertex and Graph Edge, express the vertices and edges, respectively. A GraphVertex has an `Id` attribute, meaning that each vertex is unique. Also, there are `type` and `value` attributes to

represent the model element properties, forming a triple. In contrast, the GraphEdge type has a string attribute `key` for identifying the elements from `src` and `dst` links, which are represented by `src` (source) and `dst` (destination) associations between GraphVertex and GraphEdge Classes.

We use the Graph Meta-model as a schema to instantiate model elements and their relationships by means of the GraphVertex and GraphEdge classes. Their properties, such as attributes and associations indicate the model element structures. GraphVertex and GraphEdge classes are instantiated into a GraphFrame, and from the GraphFrame it is possible to specify operations and queries to manipulate them. An instance of the Graph Meta-model is shown in SubFigures 5a and 5b.

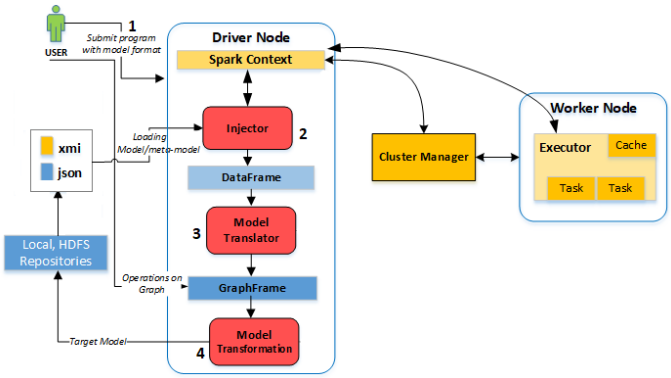

Figure 3. An Overview of Data-centric Approach for MT

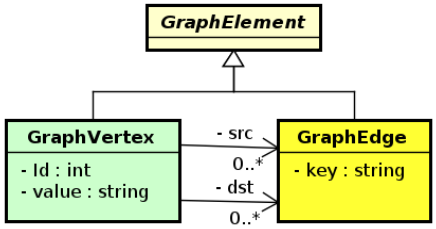

Figure 4. Graph Meta-model

A set of operations over graph elements of GraphFrame can be executed, such as the Motif algorithm to split graph in sub-graphs, graph degree to compute the valency of a vertex in a graph, queries, and others[1]. In addition to such executions, the Model2GraphFrame (M2G) output is also used as input by the Model Transformation module, which transforms the input model elements in a directed-graph format to the target model.

In the next sections, we present the steps to extract and transform models, as well as two alternatives for model partitioning.

3.1 Extracting model elements into a DataFrame

The initial step consists of the extraction of the input model elements into a DataFrame model. It starts when the user submits (**1** in Figure 3) the input model with its name, and location (path) (Figure 3) to the Driver Node. The Injector Module (**2** in Figure 3) assigns the input model in formats such as XMI or JSON to a variable (`modelPath`) which is read for loading the input model. Next, the input model is parsed (DataFrame API) and its elements are assigned to a DataFrame (`modelDF`). All DataFrame has a schema for describing the data structures, such as the input model. Thus, a schema is formed according to the input data structures. Listing 2 shows an example of a DataFrame schema. We choose to use the DataFrame in this step due to their schema. It preserves the input data structures, easing the translation of the input models to the GraphFrame through the reuse of these structures. Furthermore, it is not necessary to implement a parser for loading the input model to DataFrame.

We use the Family model excerpt from the ATL Zoo (Eclipse, 2019) to illustrate the extraction into the DataFrame and we then describe how model elements are represented in a DataFrame. In Spark, the operations on data are made by means of Transformations and Actions. A Transformation is formed by a set of instructions to manipulate data and an Action is specified to trigger the computation on data. When it is called, it notifies the Spark Engine to compute a result from a series of transformations (Chambers and Zaharia, 2018). Listing 3 illustrates the extraction result from the model Family (excerpt) in XMI format (Listing 1) to a DataFrame, where its structure is supported by DataFrame Schema shown in Listing 2.

Listing 1: Model Families Excerpt

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

Listing 2: Family Schema Excerpt

```
root
 |-- Family: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- lastName: string (nullable = true)
 |    |    |-- daughters: struct (nullable = true)
 |    |    |     |-- firstName: string (nullable = true)
       ...
```

Listing 3: DataFrame Family Excerpt

```
+---------+--------------------+---------+----------+---+
| lastName| daughters| father| mother| sons|
+---------+--------------------+---------+----------+---+
| March| [[, Brenda]]| [, Jim]| [, Cindy]| [[, Brandon]]|
   ...          ...      ...        ...         ....
```

According to Figure 3, the model elements are structured in a set of columns with an unspecified number of rows, since a schema defines the column names and types of a DataFrame. The rows are unspecified because the reading of the model elements is a lazily-evaluated operation (lazy evaluation (Michael l., 2016)). The schema does not require the rows to be identified explicitly.

Although a DataFrame Schema can be specified manually, we opt for the Schema generated by the parser by the read operation of the input model (Extraction step). In this schema, the structures of input model elements are preserved in a

[1]The valency of a vertex of a graph is the number of edges that are incident to the vertex

tree format by the translation process. Listing 2 has a translation example, where the DataFrame Schema is structured by element `root` and their rows are represented by `Family` element. The multivalued elements are represented by arrays (`array`) and their elements are represented by `structs` that may have one or more elements, including null values (`containsNull`). These elements represent the leaves (i.e., `lastName`) and have a type (i.e., `string`). All elements represented on DataFrame Schema have the (`nullable`) attribute assigned as `true` by default. This is for fitting the Spark framework for handling the Dataframe columns, with the `nullable` attribute `true` or `false`. Their columns are logical constructions that represent a value computed by means of programmatic expressions. Thus, to have a real value for a column, we need to have a row, and consequently to have a DataFrame. Therefore, since the input model was translated to a DataFrame, it can be transformed according to the transformation domains of the user.

3.2 Translating the input DataFrame to GraphFrame

In a second step, the Model Translator Module (**3** in Figure 3) translates the input model, which was assigned to a DataFrame, into a GraphFrame. We use the model elements in the DataFrame as input to the Model Translator. In addition to elements, the schema associated with the DataFrame that describes the model element structures is essential for our Model Translator, since we use it for reproducing these element structures in a graph, assigning them to the GraphFrame. We create an algorithm for translating a DataFrame to a GraphFrame, conforming to the meta-model of Figure 4. Algorithm 1 is responsible for such translation. As input, the Algorithm receives a DataFrame, which is processed by combining its content and Schema. Algorithm 1 contains the functions `model2GraphFrame` and `model2GraphSchema`. The source code of the functions is available on[2]. Since the modelDF DataFrame contains all model elements, it is assigned as a parameter to the `model2GraphFrame` function. It is responsible for starting the transformation process called. For simplicity's sake, we omit the specification of the `model2GraphSchema` function in Algorithm 1 (line 2), the `model2GraphSchema` function with the model elements and the DataFrame Schema as parameters. It performs the processing of model elements and their structures together with the respective schema columns of DataFrame in a recursive way, assigning its result into the `verticesDF` and `edgesDF` DataFrames. (lines 3 and 4). We use the wildcard parameters (`_1` and `_2`) and the `toDF` function with its parameters, and the respective DataFrame columns (`"id","value"`). Thus, the first elements are separated to the `verticesDF` DataFrame and the remaining elements are to the `edgesDF` DataFrame. Both DataFrames shape the vertices and edges and are assigned into the GraphFrame (`GF`, line 7) by `model2GraphFrame` function.

[2] `https://github.com/lzcamargo/extracSpk`

```
+-------+---------+
|id     |value    |
+-------+---------+
|1048592|sons     |
|1048590|sons     |
|1048582|Brandon  |
|1      |Family   |
|1048576|March    |
|0      |Family   |
|1048585|Sailor   |
|1048589|Jackie   |
|1048595|Kelly    |
|1048587|Peter    |
|1048591|David    |
+-------+---------+
only showing top 20 rows
```

(a) GraphFrame Vertices

```
+-------+-------+---------+
|src    |dst    |key      |
+-------+-------+---------+
|1      |1048585|lastName |
|0      |1048576|lastName |
|1      |1048590|2        |
|1048583|1048584|firstName|
|1048592|1048593|firstName|
|1048594|1048595|firstName|
|1048577|1048578|firstName|
|1      |1048586|0        |
|1      |1048594|4        |
|1      |1048592|3        |
|1048590|1048591|firstName|
+-------+-------+---------+
only showing top 20 rows
```

(b) GraphFrame Edges

Figure 5. Family Model Elements Translating to GraphFrame

Algorithm 1 M2G Translation Algorithm

Input: $modelDF : DataFrame$
Output: $GF : GraphFrame$

1: **function** model2GraphFrame($modelDF$)
2: $graphData \leftarrow model2GraphSchema(modelDF.collect, modelDF.schema, 0)$
3: $verticesDF \leftarrow graphData.\_1.toDF(\text{"}id\text{"}, \text{"}value\text{"})$
4: $edgesDF \leftarrow graphData.\_2.toDF(\text{"}src\text{"}, \text{"}dst\text{"}, \text{"}key\text{"})$
5: **return** $(verticesDF, edgesDF)$
6: **end function**
7: $GF \leftarrow model2GraphFrame(modelDF)$

We use some Family model elements (Listing 2) as input to present a translation example (an Algorithm 1 execution). To access the vertex and edge contents, we execute the commands: `GF.vertices.show()` and `GF.edges.show()`. Its outputs are represented in Figures 5a and 5b. The values of Family model elements from the DataFrame are instantiated into graph vertices. The model element names are assigned to graph edges as keys. The links (`src` and `dst`) among vertices and edges establish the relationship of the model elements. In Figure 5 we use circles and rectangles for illustrating the model element structures and their relationships. For example, the vertices and edges marked in red demonstrate the structure of the `lastName Sailor` element, and the blue ones denote the `firstName David` element. The relationship between these two elements is marked on edge (Figure 5b), where the `src` column value is noted in red, and the value of the `dst` column is noted in blue. The join of these structures (the match between `id`, `src`, and `dst` columns) allows to identify that `David` is a son (`sons`), and belongs to `Sailor` Family. Thus, the model elements are structured into GraphFrames so that they can be queried and processed for different purposes.

In the first two steps, we obtain the extraction of the input model to the `modelDF` DataFrame and its translation to the GraphFrame `GF`. We consider the result of these operations as the transformation of the input model to a graph, in particular the Model2GraphFrame transformation. In the next steps, we use the GraphFrame contents for Model Partitioning and Model Transformations.

3.3 Model Partitioning

In this step, we present two strategies that we use for partitioning models from GraphFrame: one based on the model key-element names with the Motif Algorithm, and another using clustering. First we present their implementation. In the next section, we present a proof of concept on using these strategies. We choose the first strategy because it allows us to use the transformation rule names with an algorithm implemented on the GraphFrames API itself, in this case the Motif algorithm. Regarding clustering, we choose it to link the model elements on clusters by means of the related vertices (`src` to `dst`) in edges contained on the GraphFrame. We use the clusters as parameters for the Spark framework partitions in the processing of the Model Transformations. In a graph, a motif can be defined as a pattern of interconnections of edges that occurs in a graph (Milo et al., 2002). We are interested in finding patterns in a graph for a given purpose, forming sub-graphs as such partitions from this graph. Thus, we consider the following definition, where a Graph G' is a sub-graph of graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E \cup (V' * V')$. If $G' \subseteq G$ and G' contains all of the edges $\langle u, v\rangle \in E$ with $u, v \in V'$, then G' is an induced sub-graph of G.

In our context, consider a scenario with the following transformation rule names: *Package2Schema*, *Class2Table*, *Att2Col*, and *Family2Person*. From each rule name, we use its prefix (i.e., *Package*, *Class*, *Att*, and *Family*) as a parameter (key-element) in graph partitioning using Motif algorithm, particularly for the `key` column of the edges. This means that these prefixes are interest points in the graph.

In a GraphFrame, the Motif Finding is implemented in a Domain-Specific Language (DSL) for expressing structural queries. For example, `graph.find("(a)-[e]->(b); (b)-[e2]->(a)")` will search for pairs of vertices a,b connected by edges in both directions. It will return a DataFrame of all the structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. The returned columns will be the vertices a, b, and edges e, e2 (Apache, 2019).

We specify the sub-graphs extraction combining Motif Finding and a filter. This means that depending on the input model it is necessary to adjust of Motif algorithm parameters and/or filter, characterizing the model partitioning semi-automated. Listing 4 shows the implementation in Spark Scala for the Class elements through the tag `"classes"`, which were mapped to column `key` of the `edgesDF` DataFrame. Graph motifs are patterns that occur repeatedly in the graphs and represent the relationships among the vertices. In a GraphFrame, Motif Finding uses a declarative DSL for expressing structural queries for finding patterns among edges and vertices by means of the `find()` function. Therefore, we choose it for easing the sub-graph extractions. We believe that its characteristics can generate consistent sub-graphs from key model elements (prefix name rules). Line 3 of Listing 4 is the specification of a query for searching for pairs of vertices between `(a,b)`, `(b,c)`, and `(c,d)`, which are respectively connected by edges `e`, `ea`, and `eb`. We also use a filter for delimiting the vertex pairs, starting from an edge, whose `key` property element is equal to the tag `"classes"`. This means that the execution of this expression will return as `motifsDF` all the structures (vertices and edges) related to the filtered property (`classes`) on the graph, which are arranged in `a, e, b, ea, c, eb`, and `d` columns. We select the edges contained in `motifsDF` and assign them to the `subE` immutable variable (line 5). We use it as edges for composing the `subG` sub-graph, whose vertices are the same as in the `GF` graph. We apply the *dropIsolatedVertices()* function to exclude the isolated vertices (i.e., vertices with degree zero, if there are any.) for ensuring that the links among vertices and edges in `subG` sub-graph. In this case, Listing 4 allows us to get all the Class elements and their associated elements from the GraphFrame that represent a Class model, producing a sub-graph.

Listings 11 and 12 show an example of the edges and vertexes of a sub-graph (S-G), such as a result from Listing 4. This example and the results from of the other Motif specifications for the model key-elements, such as `Package`, `Att`, `Female`, and `Male` are presented in Section 4.

Listing 4: Motifs Sub-Graph Extraction

```
1 object SubGraph {
2   def main(args: Array[String]): Unit = {
3     val motifsDF = GF.find("(a)-[e]->(b); (b)-[ea]->(c);
4         (c)-[eb]->(d)").filter("e.key = 'classes'")
5     val subE =  motifsDF.select("eb.src", "eb.dst", "eb.key")
6     val subG = GraphFrame (GF.vertices, subE)
7       .dropIsolatedVertices()
8   }
9 }
```

Now we present the utilization of clustering as a strategy, by implementing it using the `Infomap` from the MapEquation framework (Bohlin et al., 2014). There are other alternatives for such implementation, such as the utilization of the k-means algorithm (MacQueen, 1967), one of the most commonly used clustering algorithms. We could also adapt the Apache Spark MLlib, machine learning (ML) library. It provides various operations based in ML, including clustering. Infomap is a fast stochastic and recursive search algorithm with a heuristic method Louvain (Blondel et al., 2008) based on the optimization of modularity. When it is executed with vertices and edges of a graph, the neighbor nodes are joined into modules, which are subsequently joined into super-modules and so on, clustering tightly interconnected nodes into modules. Infomap has been used in community partition problems (Aslak et al., 2018; Edler et al., 2017), for detecting communities in large networks, and to help in the analysis of complex systems. In addition, Infomap operates on graph-structures in the Pajeck format (`file.net`)[3], which can be easily extracted from the GraphFrame as input to Infomap. For example, Listing 5 shows a excerpt of the File.net extracted from Class-0 model, and Listing 6 shows the `.clu` output file, the clustering result, where the nodes are gathered in the respective clusters (`node` and `cluster` columns). Column `flow` contains cluster indices for each node, but they are discarded when the `.clu` file is injected into DataFrame by a loading operation and used in clustering model elements. However, the clustering from GraphFrame using the Infomap framework is a semi-automated operation, since we do not implement integration between our approach and the Infomap framework (Operations on Graph, Figure 3)

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

Listing 5: Class-0 File.net

```
*Vertices 50031
0 0
1 1
2 2
...
*Arcs 50030
1 2
4 5
4 6
..
```

Listing 6: Clustering Nodes

```
# node cluster flow:
8 1 0.0457141
7 1 0.00261991
10 1 0.00261991
6 1 0.00222776
9 1 0.00222776
5 1 0.00195755
11 1 0.027326
46 1 0.00233907
    ...
```

Later, we present the use of Infomap and the model partitioning in Section 4.

3.4 MT using GraphFrame

In the last step, we specify a set of operations and transformation rules to transform the source model in GraphFrame into a target model. They are executed as parallel tasks on Worker Nodes of the Spark framework, through the Model Transformation module (**4** in Figure 3). The source code of the operations and transformation rules are available on[4]. Listing 7 shows the `Family2Person` rule written in Scala as a singleton object (`object Family2Person`). We separate the male and female elements in the `maleEdgesDF` and `femaleEdgesDF` DataFrames. They contain the target values (`dstm, dstf, dst`) that link each last name with its first names. We use the `select`, `join`, and `filter` functions to select the last and first names from of `maleEdgesDF`. For each join operation, we use the `filter` function (lines 4, 6, 12, and 14) to ensure the accurate selection of model elements, since they are formed by relationships among edges and vertices (`"dstm" === "id"`). In lines 7 and 15, we use the `select` and `concat` functions to assign the last name (`lastName`) and the respective first names (`value`) as the full name (`fullName` column) to the `maleFullNamesDF` DataFrame.

Listing 7: F2P Rule

```
1  object Family2Person {
2    val maleFullNamesDF = maleEdgesDF
3      .select($"dstm", $"dst").join(GF.vertices)
4      .filter($"dstm" === $"id")
5      .select($"value".alias("lastName"), $"dst")
6      .join(GF.vertices).filter($"dst"===$"id")
7      .select(concat($"lastName", lit(" "), $"value")
8        as "fullName")
9
10   val femaleFullNamesDF = femaleEdgesDF
11     .select($"dstf", $"dst").join(GF.vertices)
12     .filter($"dstf" === $"id")
13     .select($"value".alias("lastName"), $"dst")
14     .join(GF.vertices).filter($"dst"===$"id")
15     .select(concat($"lastName", lit(" "), $"value")
16       as "fullName")
17 }
```

For the `femaleFullNamesDF` DataFrame (lines 9 to 14), we use the same idea applied to the `maleFullNamesDF` Dataframe. These DataFrames are merged (`union` function) in the `personDF` DataFrame, each one with a new column Gender (`withColumn("Gender")`) to ensure the gender distinction among persons.

[4] `https://github.com/lzcamargo/transformSpk`

Next, we specify an operation, using `coalesce(1)` method to instantiate the transformation output in a single partition (1). This means that output tasks will be reduced in a single partition (distinct output) as the final result of the transformation. The example in Listing 8 is obtained with the `write` function, and the tags (root and row) of the `databricks:spark-xml` library, indicating that the format was assigned as xml. We separate these commands (write operations in the target model) from the loading rules for better code legibility. Since the target model was stored in a repository, it enables to load the output in xml/xmi format and instantiate it back in GraphFrame.
Listing 8 shows a portion of the `persons.xml` file content. It represents the Family2Person transformation result, using the Family model presented in Listing 1 as the source model.

Listing 8: Persons Model Excerpt

```
<Persons>
    <gender> Male </gender>
    <fullName> March Jim </fullName>
    <gender> Male </gender>
    <fullName> Sailor Dylan </fullName>
    <gender> Female </gender>
    <fullName> March Cindy </fullName>
    <gender> Female </gender>
    <fullName> March Brenda </fullName>
        ...
</Persons>
```

In this section we described our approach. In the next section, we perform the proof of concepts in order to validate its feasibility.

4 Implementation

We implemented a Proof of Concept (PoC) (Kendig, 2016) using GraphFrames to demonstrate the feasibility of our approach and to show its usefulness under following aspects: the processing of Model2GraphFrame outputs, the partitioning of graphs contained in the GraphFrame, connectivity among model elements in a set of GraphFrames, and the execution of model transformation using the GraphFrames.

We run the PoC in a single machine with the following software stack: Ubuntu 18.04; Spark 2.4; and Scala 2.3. It is hosted by an Intel Core i5-4210U 1600 CPU with 8096 MB of RAM; and the processor has two cores. As input, we use the both Class and Family models in XMI format. There are four models with the following specifications:

- Class-0, class model with no attributes or methods, only Package and Class elements. This kind of model is used in Domain Modeling, useful to understand the ideas and concepts of the domain (Larman, 2004);
- Class-3, class model with Package and Class elements, each Class contains from 1 to 3 methods and attributes;
- Class-6, as the previous item, but each class contains from 1 to 6 methods and attributes;
- Family model with 0 to 3 sons and daughters. Its elements are self-contained in `LastName` elements and their attributes.

We get the Class models from, each one with 10000 classes[5]. They were created to be used as a benchmark for the Class2Relational transformation case studies in parallel transformations using Lintra (Burgueno et al., 2015)[6]. These models have references among their elements established by attributes. For instance, the Class-0 model has 10 Package elements and each Package has 1000 Class elements. The Family model has 10000 LastName elements, which we created for this proof of concept. In this case, we consider these elements as self-contained (Class-0 and Family). However, there are models (Class-3 and Class-6) that besides self-contained elements, also contain inter-connected elements, where Class elements are referenced by one or more Class elements, which are contained in other Packages. Attributes such as super, and type establish such references.

The models used on PoC have a different density (Class-0, Family, and Class-6) and interconnectivity (Class-3 and Class-6) among their elements. This means that we will validate our approach in relation to these model aspects. To measure the execution times in seconds, we use the `System.currentTimeMillis()` function from the Scala language, in a dedicated machine with no UI interactions. The input model elements once extracted to a GraphFrame, they must be available. Each model element in the GraphFrame vertices has to be linked to its properties through GraphFrame edges.

We have defined three research questions to validate the PoC implementation and its main aspects.

Q1: *How to check if the Model2GraphFrame output is available for processing?*
To address this question, we use the directed-graph property (DGP) to check the total of Edges and Vertices in a directed-graph G, $\sum_{v=0}^{V(G)} -1 = \sum_{e=0}^{E(G)}$, where the $V(G)$ total minus 1 is equal to $E(G)$ total. When this property is true to a directed-graph it is considered as a simple directed graph (Hochbaum, 2008). A directed-graph is no longer simple if there are multiple edges or loops. Hence, the V(G) total is less than to the E(G) total ($\sum_{v=0}^{V(G)} < \sum_{e=0}^{E(G)}$). In addition, we execute a set of queries on the GraphFrame to validate the contents of vertices and edges, whose input models contain 100 classes and 100 families. This means that we take a set of model elements contained into GraphFrame and we compare it with its input model elements.

Although the M2G outputs are directed-graphs into GraphFrame, we need to know whether it is achievable to use them in model transformations. To address this issue, we define question Q2.

Q2: *Is it possible to perform MT using GraphFrame?*
We address this question in order to use GraphFrame in Model Transformations. Our goal is to verify how the source models into GraphFrames can be transformed to target models. We specify operations and rules using methods and functions in Scala for manipulating vertices and edges in GraphFrame (e.g., Listing 7). They are similar to transformation specifications in ATL - ATLAS Transformation Language (Jouault et al., 2008), where Helpers and Transformation Rules are the constructs used to specify the transformation functionality.

Finally, the last question is about performance of MT executions using clusters.
Q3: *Does executing model transformations using model partitioning improve performance?*
We address this question in order to verify whether the executions of model transformations using model partitioning improve performance, since we adopted two partitioning strategies for this approach: partitioning of input model into GraphFrame in sub-graphs, and generating of clusters from GraphFrame vertices. In the following Sections, we present the proof of concepts, results and the answers for the above questions, as well as further discussions.

4.1 Processing Model2GraphFrame Outputs

To check the GraphFrame outputs with respect to the input models, we obtain the total of vertices and edges and we use the DGP to check their amount. Columns V(G) and E(G) of Table 2 show the total of vertices and edges from the input models (Model column). The amount of vertices V(G) - 1 is equal to the amount of edges E(G) for the Class-0 and Family models, demonstrating that they are simple directed-graphs. However, the total of vertices and edges from the Class-3 and Class-6 models indicate that they are not simple directed-graphs (V(G) < E(G)). In addition, we execute queries as shown below, and their results are compared to input model elements to validate the M2G consistency. It returns the values of class properties such as `name`, `isAbstract`, and `visibility` from the GraphFrame vertices. It does not return Attributes and Methods, because the key-element (`key`) is assigned the `"classes"` value.

```
gf.edges.where($"key"==="classes")
  .select($"dst".as("dstv")).join(gf.edges)
  .filter($"dstv"===$"src").select($"dst")
  .join(gf.vertices).filter($"dst"===$"id").show()
```

Listings 9 and 10 show excerpts of Class-0 model elements and the query output. They represent an example of our validation. In this case, the relation among classes and their properties are established by the GraphFrame edges (`gf.edges` `src` and `dst`), whereas the value of each property is assigned to the GraphFrame vertices (`gf.vertices`).

Listing 9: Class-0 Model

```
<classes name="Class14"
isAbstract="true"
visibility="public">
<classes name="Class15"
isAbstract="true"
        ...
```

Listing 10: Query Output

```
+---+---------+----------+
| id|    value| valueType|
+---+---------+----------+
| 23|   public|    string|
| 22|  Class14|    string|
| 21|     true|   boolean|
        ...
```

Table 2 (the first four columns) and the query outputs (example in Listing 10) show that the M2G results from the input models seem correct. This comparison complements the quantitative checking through the DPG. For example, when using the DPG for the Class models, we identify that the Class-0 model is a simple directed-graph, since it only

[5] http://atenea.lcc.uma.es/Descargas/MTBenchmark/classModels
[6] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra

has Package and Class elements. Furthermore, there is a single relationship between Package and Class elements that link them. In this case, 1000 Classes in each Package, since the Class-0 model has 10 Package elements. On the other hand, the Class-3 and Class-6 models have Package, Class, Attribute, and Method elements, and relationships among them include Data-type elements. This means that these models, when transformed to graphs have a total of edges larger than the total of vertices. Therefore, we answer the question **Q1** by validating the total of vertices and edges for each input model, as well as the respective contents. We follow with the proof of concepts executing the Motif Algorithm for all input models (Class and Families) using the strategy shown in Listing 4. Next, we measuring the connectivity of models assigned to the GraphFrames.

4.2 Measuring GraphFrame Connectivity

The idea of measuring the GraphFrame connectivity is a strategy to reveal how complex the input models that we use are, with respect to the graph elements (vertices and edges). It can help the choice of the strategy to be adopted for the partitioning and for operations over models with GraphFrame. Furthermore, a set of functions can be executed from the GraphFrame, as for example the `outDegrees` and `inDegrees` functions.

We execute the `outDegrees` and `inDegrees` functions for all vertices of the GraphFrame models (e.g., `outDeg = gf.outDegrees`). These functions determine the amount of outward-directed (`outDegrees`) and inward-directed (`inDegrees`) graph edges from GraphFrame vertices. Once the degree is calculated for all vertices of the graph, they are grouped and summed (`outDeg.groupBy("vertices").sum()`). Table 1 shows the execution results. The amount of Degrees for each GraphFrame Model is in descending order (only the first four or six amounts are shown) in the Out-Degrees and In-Degrees Columns, and the total of vertices of each calculated degree is in the Total Vertices column. It is worth mentioning that no sink vertex was found (vertex with out-degree equal to 0) in the GraphFrame models, but one source vertex was found (vertex with in-degree equal to 0), in this case the vertex with `id` equal to 0, that represents the root vertex.

In the In-Degrees column (Table 1), we can see that the Class-0 and Family models in GraphFrame Models column have Degrees equal to 1 for most of its vertices (except for the vertex 0). The degree calculated of outgoing edges for the vertices (Out-Degrees column) for these models show their characteristics. For instance, a degree equal to 1000 and a total of vertices equal to 10, mean that there are 10 vertices with 1000 out-going edges. In particular, they represent the directed-links between the Package and Class elements of the Class-0, Class-3, and Class-6 models, since there are 10 Packages and 10000 Classes into each Class Model. Therefore, the results obtained from the Class-0 and Family models indicate that they are simple directed-graphs and weakly connected. On the other hand, the results in Out-Degrees and In-Degress columns show that the Class-3 and Class-6 models are directed-graphs and strongly connected, since there are directed-links among Package, Class, and Attribute elements. In addition, the in-going edge from GraphFrame vertex elements, such as Datatype and Type are also represented in the In-Degrees Columns for these models. The result from the `inDegrees` and `outDegrees` functions is useful to evaluate how complex models are.

In the next section, we present the results of our two partitioning strategies over the GraphFrames. Furthermore, in the following sections, we discuss the influence of these strategies in model transformation executions, as well as we describe the distribution of model elements over the executor processes (Worker nodes) on Spark framework in local mode executions.

4.3 Partitioning M2G Outputs

Our approach provides model partitioning with two different strategies (Section 3.3): the Motif Find algorithm available in GraphFrames API, and Infomap framework. From the GraphFrame the Motif algorithm finds patterns among edges and vertices for producing sub-graphs. Using the vertices and edges from GraphFrame, the Infomap framework generates clusters of vertices from format files (`.net`). We use them for partitioning the operations on the GraphFrame in the process of model transformations.

In Section 3.3, we showed a specification of how the Motif algorithm may be used in graph partitioning, where a sub-graph (G') is formed from the graph edges that contain a key element extracted from the rule name (`object Package2Schema`). For example, for the Key element "Package" (k-element column at Table 2) there are 30 vertices (V(G') column) and 20 edges (E(G') column) of the sub-graphs from the input (Class-0,Class-3, and Class-6) model transformation (M2G) outputs. In addition to Package names, their nearest neighbor elements are partitioned together in the respective edges (in this case, Class elements). Listings 11 and 12 show edge and vertex samples of a sub-graph (S-G), where vertex 17 contains the value `Pck0`. Listing 11 has this property, and the nearest neighbor linked in two edges (16,17; 16,18). In this manner, for each Package element, the partition contains three vertices and two edges. All Class models have 10 Package elements; this justifies the amount of 30 vertices and 20 edges in the sub-graphs for the key-element Package. Every Class element in the Class-0 model has 10000 Class elements, composed of `name`, `isAbstract`, and `visibility` attributes. For each attribute an edge is created, whose source (`src`) vertex is a `StructType`. Thus, each Class element has four vertices and three edges. Keeping the quantifiable amount of model elements in mind, we execute the Motif algorithm for the key elements such as Package, Classes, and Attributes (methods were not partitioned).

Table 1. Input GraphFrame Model Degrees

GraphFrame Models	Out-Degrees		In-Degrees	
	Degrees	Total Vertices	Degrees	Total Vertices
Class-0	1000	10	1	40030
	10	1		
	5	1		
	4	10000		
Class-3	1000	10	15011	1
	10	1	9919	1
	7	10000	4902	1
	5	14990	3363	1
	3	50737	3289	1
	2	13748	1	318800
Class-6	1000	10	48142	1
	10	1	32069	1
	7	10000	16179	1
	5	37179	9074	1
	4	7098	9053	1
	3	143346	1	740376
Family	10000	1	1	160480
	5	10000		
	2	46866		
	1	6748		

Listing 11: S-G Edges

```
+----+----+--------+
|src |dst |    key|
+----+----+--------+
| 16 | 17 |   name|
| 16 | 18 | classes|
| 21 | 22 |   name|
| 21 | 23 | classes|
  ..   ...  ...
```

Listing 12: S-G Vertices

```
+----+-----------+------+
| id |      Type| value|
+----+-----------+------+
| 16 | StructType|     |
| 17 |    string| Pck0|
| 18 |  ArrayType|     |
| 21 | StructType|     |
  ..       ...
```

As seen in the previous section, the Class-3 and Class-6 models are directed-graphs and strongly connected. Their elements such Package, Class, and Attribute elements have links to each other through `type` and *super* attributes. There are links among Attribute and DataType elements such as `float`, `string`, `integer`, and others. For Class elements there are edges containing attributes, such as `(19,22,name)`, `(19,23,super)`, `(19,24,visibility)`, `...(99,101,name)`. These attributes are also connecting structures between the class elements, when the types of attributes are classes. For instance, to establish links among Class elements using attributes, edges such as the `{(23,101,lnk)` are formed to link the `super` attribute of a Class element to the `name` attribute of another Class element. These edges are joined with their vertices, forming the sub-graph Classes. The Attributes sub-graph is formed in same way, and its links to other elements (i.g.,DataType) are established via `type` attributes. Consequently, the amount of edges (E(G')) is larger than the vertices (V(G')) for the sub-graphs (partitions) of Classes and Attributes.

The Class element has 6 attributes, which are assigned to vertices. For the type of structure (`StrucType`) of a Class element, more than one vertex is assigned. These are linked to the source (`src`) vertex of the class properties (18,19,"0"). Thus, for each Class element in a Class sub-graph there are 7 vertices, explaining the 70000 vertices. Regarding the Female and Male sub-graphs, they were partitioned from the Family model transformation (M2G) output. The `lastName` element and its structure are duplicated into these sub-graphs. Thus, the total of vertices (V(G')) and edges (E(G')) is more than the total of vertices (V(G)) and edges (E(G)) of the Family M2G output. Table 2 (the last four columns) shows the partitioning results for each Model translated into GraphFrame. Each model partition (V(G') and E(G')) sub-graph) is related to a key element (K-element). We extract the prefix from transformation rule names, such as Package, Classes, and Attribute, and we execute the Motif algorithm for each of them, except the key element Attribute for the Model Class-0. This partitioning strategy is dependent on the key attribute of GraphFrame edges. It requires that all links among model elements are instantiated into edges. Otherwise the partitioning will not be correct.

Now considering the graph clustering strategy, we extracted from the GraphFrame the vertices and edges in the Pajeck format (`.net`), as required by the Infomap framework[7]. Once the `.net` file is available for processing, we execute it with a call to `Infomap`. The runtime arguments are `-z, -N 10, --directed, --clu`, meaning respectively: start with vertex equal to zero; iterate ten times over the vertices and edges; the input is a directed graph; and the output will be a file containing clusters of vertices. The Infomap framework execution output is a text file (`.clu`) containing a list of pairs formed by vertices and clusters ((11,1),(12,1),(13,1),(21,1),(22,1)). This list is formed according to the incidence of each vertex in the edges. All links shaped from a vertex are grouped in a single cluster, and thus, a vertex belongs only to a single cluster.

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

Table 2. M2G Transformations and GraphFrame Partitioning

Model	$M2G_s$	V(G)	E(G)	K-element	V(G')	E(G')	$G2G'_s$
Class-0	3s	40031	40030	Package	30	20	8s
				Classes	40000	30000	12s
				Attribute	n/a	n/a	n/a
Class-3	11s	318789	350006	Package	30	20	55s
				Classes	70000	84400	54s
				Attribute	90934	129945	59s
Class-6	115s	740380	880776	Package	30	20	127s
				Classes	70000	88200	129s
				Attribute	179982	200982	149s
Family	118s	160481	160480	Female	109808	92276	35s
				Male	110698	92938	36s

We execute the Infomap framework for the four input models using `.net` files. These files contain vertices and edges extracted from GraphFrames. Table 3 displays the execution results of the Infomap framework using the `.clu` files. In the Infomap output (Clusters(G) column), we can note that the amount of clusters generated from the Class-0 (10012) and Family (160480) models is much larger than Class-3 (7) and Class-6 (13) models. This is related to the density of each model. The higher the number of interconnections among model elements (edges), the lower the number of clusters generated, because each vertex in an interconnection is associated to the same cluster. As we saw in the previous section, the Class-0 and Family GraphFrame models are simple directed-graphs and weakly connected. This corroborates the cluster granularity in both cases. In contrast, the Class-3 and Class-6 models, when translated to GraphFrames, are directed-graphs as well, but the graphs are not simple. They are in fact strongly connected. As a result, the number of clusters generated for these models is much smaller when compared to the Class-0 and Family models.

Once Infomap has generated the `.clu` Cluster file for each model, we load them into their respective DataFrames. As an example, for loading the Class-0 clusters:

```
val clusterPath = "/Infomap/output/class-0.clu"
val clusterInput = spark.read.option("header","true")
   .option("delimiter", " ").txt(clusterPath)
   .select($"node", $"cluster")
```

The graphs are used in the execution of model transformations to direct the model element partitions on the parallel Spark execution (section 4.4). We can also use them on distributed Spark execution (cluster of machines in future executions). For example, the number of nodes provided by the user or obtained from machine clusters can be used as the denominator in a division of the amount of clusters from the partitions. On other hand, the partitioning and distribution of data done by the Spark partitioner can be improved in terms of data dependency among the environment nodes. Since the clustering and the sub-graphs tend to have the linked model elements closely. The partitioning and distribution of data in Spark is done at run-time using the RDD (*Resilient Distributed Datasets*) API.

In Table 3, we report again the amount of vertices and edges of models (Table 2) to show the relation of the number of clusters for each model, considering their total vertices and edges.

Table 3. Clustering Models

Model	**Vertices(G)**	**Edges(G)**	**Clusters(G)**
Class-0	40031	40030	10012
Class-3	318789	350006	7
Class-6	740365	880776	13
Family	160481	160480	3354

The results from the Motif algorithm executions showed that Motif Find can be used as a partitioning strategy for graphs represented in GraphFrames. Regarding Infomap, it uses the GraphFrame output as input for processing the clustering, and the result is injected back to Spark through a DataFrame. This process requires an integration between Infomap and Spark frameworks. There is a drawback in the way that we adopt both strategies of partitioning since we do not consider data balancing. Even though we know that it is difficult to treat the densely connected models, we believe that we may explore this challenge in a future work. In the next section we execute the model transformations using GraphFrames.

4.4 Executing Model Transformations using GraphFrame

We execute the Class to Relational (C2R) and the Family to Person (F2P) transformations using Class-0, Class-3, Class-6, and Family as source GraphFrame models. Once they are transformed to GraphFrames, their elements are used as input in filtering operations and transformation rules (as shown in Listing 7), which we submit to the Spark framework for execution. For each GraphFrame containing the source models, we execute the transformations considering:

- No Partition (NP) in these executions, without any partitioning strategy. We execute the model transformations for the whole model in the GraphFrame;
- Motif, running the model transformations using the subgraphs from the Motif partitioning strategy (Table 2);
- Infomap, executing the model transformations using the clusters of vertices from the Infomap framework.

An operation that we used in F2P transformation has the following specification:

`val lastNameFamilyDF = edgesDF.filter("key = 'lastName'").select($"src", $"dst", $"key")`. It selects the edges (`src` and `dst`) for each family last name (`lastName`) and assigns them to the `lastNameFamilyDF` DataFrame. Table 4 shows the model transformation results, which include the times in seconds. These times were computed as the average of 7 transformation executions for every input GraphFrame model, having discarded the first 3 executions. They were considered as warm-up phases for the virtual machine of the Spark framework.

Before performing model transformations using the Clustering strategy, we validate the cluster partitions using the same procedure that we apply to the Motif partitioning (Section 4.3). We identify that the clustering of GraphFrame is consistent only when Infomap is executed without the level parameter (visualizing the cluster output in modules), particularly for the Class-3 and Class-6 models. Otherwise, vertices of the same model elements were in different clusters due to the cluster modularity aspect of the Infomap algorithm. We consider consistency as the main requirement of the partitioning results, but at the same time recognize that the partitioning obtained through Infomap is conservative when evaluating the total of vertex clusters in Table 3. The vertices of the Class-3 and Class-6 GraphFrame models were clustered to 7 and 13 clusters respectively. It means the densely interconnected models require a more efficient partitioning strategy with regards to the balancing and consistency of model elements.

We run model transformations on the Spark framework in local mode using four nodes, which were used for executing the parallel tasks in memory. A task is the smallest unit of schedulable work in a Spark program. A stage is a set of tasks that can be run together (Apache, 2019). In this manner, the requests for data manipulation operations (name of transformation) and actions (requests for output, for instance) are coordinated by the Spark framework. Thus, we establish a traceability among source and target elements, assigning the links among model elements from the GraphFrame to a DataFrame (write the target model output). For instance, the following expression selects all links (vertices of last and first names) of `sons` elements.

```
val sonsNamesLinksDF = maleFamilyDF
   .where($"key"==="sons")
   .select($"dstm",$"dst".alias("dstt")).join(edgesDF)
   .filter($"dstt"===$"src").select($"dstm",$"dst")
```

These links are inserted into the `sonsNamesLinksDF` DataFrame, and we can use it to obtain the complete names of sons on a parallel Spark framework execution. For each source model (Table 4) in the GraphFrame, we submit all the operation expressions and transformation rules to the `Spark context` needed by the transformation of the model in question.

According to Table 4, the partitioning model strategy with the Motif algorithm penalized the performance of sub-graphs transformation executions (Motif), due to the memory consumption and the possible negative effect on mechanisms of the Spark framework that minimize the data exchange among the executors (data shuffles). The execution results with no partitioning strategy (NP) have shown better performance when compared with the sub-graphs executions (it does not interfere on Spark partitioner). In the cluster partition executions (Infomap), we interfered in the Spark partitioner, submitting the partitions from cluster model elements to the Spark framework to improve the performance. This strategy showed the best performance when compared to the other executions (NP and Motif). The results on using these strategies are present in this section.

Table 4. Execution Times for Model Transformations Using GraphFrame

C2R and F2P	Class-0	Class-3	Class-6	Family
No Partitions	11s	54s	147s	39s
Motif	25s	107s	341s	78s
Infomap	8s	52s	141s	34s

Regarding execution times of model transformations, we observe that the models which we consider weakly connected, have the lowest execution times (Class-0 and Family columns) when compared to the execution times of the Class-3 and Class-6 models. These models have their transformations times increased as the amount of interconnected elements grow. In the transformations using Infomap, we use the clusters to re-partition in run-time the default Spark partitioning. For each expression, we add the `clusterInput` DataFrame (`node` and `cluster` columns) and invoke the `repartition` function with the `cluster` column as parameter, in order to interfere in Spark partitioner. This is necessary because the input model clusters were generated before by Infomap framework. The expression below shows an example of interference in the partitioning of the Spark through the `repartition()` function.

```
val lastNameFamily = clusterInput
   .select($"node",$"cluster")
   .join(edgesDF).where($"node" === $"dst" &&
      $"key"==="lastName")
   .select($"src", $"dst", $"key",$"cluster")
   .repartition($"cluster")
```

We create the partitions of the model elements from the clusters. This means that for the strongly connected models the number of partitions in run-time is smaller. Consequently, the amount of shuffling (operation in Spark to distribute data across multiple partitions) also diminishes. However, the execution times for the Motif is higher than in the other executions for all input models used in this PoC. This is due to memory consumption used to process the Motif partitioning and model transformations, since all the steps of our approach were executed in memory. In addition, when the action is invoked by the program all the operations in lazy evaluations are triggered.

A spark applications consist of a driver process (Driver node) and executor processes (Worker nodes). The driver runs, analyzes, and distributes work across the executors. A partition is a logical chunk of a large distributed data set (Apache, 2019). In our case, when the models are submitted to execution on the Spark framework, they are partitioned automatically when there is no interference via code (`repartition()` or `coalesce()`). For example, when a class element requires one or more model elements that are

in other nodes, these elements are shuffled and distributed between nodes, and processed. These results are available to the Worker nodes, which can then be used in a subsequent operation.

Concerning the influence of partitioning strategies in model transformations, we note that the generation of sub-graphs with Motif execution penalizes the performance of model transformation executions. Since they are running together in local mode, the sub-graphs were in the same JVM (Java Virtual Machine) as the transformation code. We believe that the fact of model elements required by operations and transformation rules being together in sub-graphs (GraphFrames) may diminish the amount of shuffles during the model transformation executions. However, this strategy can be better explored and be made more efficient on distributed executions with priority to data locality (data and the code stored together on the same Worker Node). As for the clustering strategy, we see that the use of clusters as a parameter for re-partitioning (Spark manages data using partitions) of the source models for model transformation processing is favorable to the performance of model transformations, since they help parallelize data processing by minimizing shuffles between executors (Worker nodes). This is a consequence of each cluster and its vertices being distributed as a single partition in run-time. However, this strategy is susceptible to data skew when the data is unbalanced. In distributed execution, where each cluster of model elements is in a partition and localization, the model transformation processing can be minimized, with less network traffic overhead for sending data between executors (Worker Nodes). In both strategies there are open issues, such as data balancing (Le et al., 2014), data skew processing (Gao et al., 2017), and data locality (Jin et al., 2011) that need be contemplated in our approach.

Although there are open questions, we answer the question **Q2** admitting that it is feasible to use GraphFrame for model transformations. According to the executing times of model transformations in Table 4, the model partitioning based on clustering performed better when compared with the other times. On the other hand, the graph partitioning using the Motif Find algorithm presented the worst performance in this PoC. That means that we answer the question **Q3**. In the next section, we present further discussion about this work.

4.5 Discussion

DataFrame and GraphFrames are flexible, structured, and based in collections. These aspects allow us to extract metamodels and models from different formats, such as XMI and JSON. Moreover, their characteristics may ease the data modeling for distinct transformation scenarios (local and distributed/parallel). Syntactically and semantically, the functional constructs of the DataFrame and GraphFrames APIs are relatively simple, though proper usage of some constructs in Resilient Distributed Dataset (RDD) may require more skill in functional programming.

In addition to the APIs, DataFrame has a schema that allows interacting with its data structures and ease data operations. The capability to process different data formats, such as JSON and XMI can be considered a differential of our approach to those that accept only the XMI format. Another essential aspect is the transformation from a technical modeling space to the GraphFrame space, easing different operations over model graphs through the GraphFrames API. However, in our proof of concepts, we observe that the Model2GraphFrame transformations consume a considerable amount of memory, as the input model is loaded and held in memory during the recursive processing. Even when using a platform such as Spark, this problem with memory usage needs to be addressed, in particular when using recursive processing, or by avoiding it altogether.

Regarding the GraphFrames API, it has a set of functions and built-in algorithms that can be used by different languages. Its GraphFrame data representation (vertex and edge DataFrames) allowed the manipulation of model elements while preserving their references and the connectivity of models. The links between model elements are assigned to GraphFrame edges, and from them, it is possible to identify and process the elements and their links, such as when using Motif algorithm to generate sub-graphs, or via functions that measure the connectivity of GraphFrames, among other operations over GraphFrames. On the other hand, the model elements that are in GraphFrame vertices and edges can be joined in a single DataFrame using functions such as `join`, `union`, and `merge`. When there is a need for a single output from parallel/distributed processing, a reduce operation can be executed using functions such as `repartition(1)` or `coalesce(1)`.

The model partitioning strategies used in fully connected models need to be better investigated and integrated to model partitioning in the Spark framework, mainly the clustering. Issues of balancing the partitioning outputs, data skew, and data locality need to be treated under the distributed/parallel model transformations. Our approach can contribute to scalable MDE, since the Spark framework provides mechanisms for such context. Nevertheless, we did not yet explore distributed processing in our approach. Furthermore, more experiments involving a diversified set of transformation scenarios are necessary.

The results show that the parallel model transformations is feasible in our approach, but is necessary to explore other aspects such as the learning cost to use it, the semantic to specify the transformation rules, and to know how difficult it is to use our approach regarding the ATL-based approaches.

5 Related Work

Data extraction and operations on directed-graphs are used in most application domains, such as MT, Reverse Engineering, Software Evolution, and others. We report some works that highlight the Dc approach in MT, parallel/distributed MT, some extraction processes, as well as works that process graphs on the Spark framework.

MDE approaches have already been reinterpreted under different views, for instance, Batory and Azanza (2017) 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 use the Prolog language to write declarative constraints in

m2m transformation. This approach employs a Dc approach to declarative styles. However, it was elaborated for helping to explain the MDE concepts under relational perspectives, whereas we seek to offer an approach for model transformation in a Dc approach.

In Wischenbart et al. (2012) an approach is proposed to derive social network schemas from social network data. They derive schema information expressed in JSON data. For this, schema extraction strategies are provided for integration tools that build on different technical spaces. Moreover, they propose to apply techniques from MDE to transform schemas into instance data. The extraction strategy used in this work is similar to the one adopted in our ME, in which we use the DataFrame schema from the output Injector module for preserving the consistency of model elements during their translation into GraphFrame. However, they only use JSON as input to the extraction and apply it on three social networks. Our approach using DataFrame eases the extraction of data from more than one format, such as XMI and JSON, and translates it in a graph model for model transformations or other GraphFrame operations.

Triple Graph Grammars (TGG) is considered a standard framework for model transformation based on graphs (Tomaszek et al., 2018). Its expressiveness and its mathematical basis are relevant aspects in graph transformation, since a single set of triple rules is sufficient to generate the operational rules for the forward and backward model transformations (Hermann et al., 2014). Tools such as eMoflon, MoTE, TGG Interpreter, and EMorF are TGG-based (Kahani et al., 2018; Edgar et al., 2014). However, the optimization is still a challenge for applications based on TGG, which may be a trade-off between expressiveness and scalability (Anjorin et al., 2016). Our approach uses directed-graphs as a means of representing the model elements and easing the operations over them. In addition, the aspects of the platform that we use can be a differential for development of parallel/distributed model transformation.

Bollati et al. (2013) introduced the MeTAGeM, a methodological and technical framework for the development of model transformations, which bundles a set of Domain Specific Languages (DSL) for modeling model transformations with a set of meta-model transformations in order to bridge these languages in (semi-) automated model transformations development. Amongst the aspects of the MeTAGeM, we report two to our work: the concern with interoperability of different languages in the transformation process; and the Platform Dependent Transformation (PDT) model, that allows the use of Injectors/Extractors for model-to-text transformations. However, the injector/extractor are based on Textual Concrete Syntax (TCS), which provides a DSL for the specification of the correspondence between the meta-model of a given DSL and its textual representation. This means that, for each DSL it is necessary to recover its TCS correspondent, in case it already includes the DSL. Otherwise is needed to develop a TCS. Vara and Marcos (2012) developed a textual editor and model extractor for Oracle OR models using the TCS language to support textual editing of models and the extraction of models from legacy code, for validating a systematic study and a technical solution for MDE development of information systems. Our approach extracts model elements in XMI/JSON format and transforms them to the directed-graph format using the GraphFrames API from the Spark framework. The extraction output is used in model partitioning, as well as in model transformations. In addition, it may be used for general purposes in graph-oriented applications.

Distributed/parallel graph processing has been applied as a way to optimize the graph operations. Imre and Mezei (2012) introduced an algorithm to do graph transformations in a parallel way using threads on a GPU (Graphics Processing Unit). The transformation is executed on this algorithm in two phases: matching and modifier. Although, the algorithm can take the advantage of multi-core processors, the modifier phase executes the modifications sequentially on a single thread. In our approach, a graph can be processed on a distributed and/or parallel way, since we utilize the parallel implicit operations on a general-purpose cluster computing framework. When some operation and/or transformation is required by a program on the Spark framework, the model is automatically split in partitions (this step can be changed by the developer) and processed on nodes by a set of tasks in memory. Furthermore, the graph operations can be specified SQL-like declarative style and/or functional-like.

Benelallam et al.(Benelallam et al., 2015, 2016) 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. From static analysis of transformation rules, 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 on the MapReduce framework and an implicit distribution of models, as well as the transformation executions on two phases (map and reduce). Our partitioning strategies are based on directed-graph and search split model into in sub-graphs and clusters of vertices. Our approach uses the GraphFrame as a bridge between input models and model transformations. It depends on the Spark framework.

NeoEMF is a scalable model persistence framework based on a modular architecture enabling model storage into multiple data stores. This framework is proposed by Daniel et al. (2017), it provides model-to-database mappings for persistence solutions and enables to store models in graphs, key-value, and column databases. This framework provides an API compatible with the Eclipse Modeling Framework (EMF) API, meaning that the NeoEMF accepts only models from EMF. Furthermore, the NeoEMF focuses on scalable model persistence, whereas our approach aims at scalable transformation of models, supporting input models in XMI or JSON formats.

Junghanns et al. (2016) propose the Extended Property Graph Model (EPGM), a graph data model that supports flat and graph collections with heterogeneous vertices and edges. They implemented a set of analytic operators using a DSL on top of Apache Flink[8] to graph processing of single graph representations (i.e., in collection), and to provide

[8]https://flink.apache.org/

general-purpose operators (i.e., select, count,...) on graph. The graph representation in EPGM contains three object types (GraphHead, Vertex, and Edge), whereas ours has two object types: GraphNode and GraphEdge, which are subtypes of GraphElement (Figure 4) and are contained in the GraphFrames API. In EPGM, the input data format for processing is not informed. Our extractor processes different data formats (XMI and JSON). To process the graphs, we use the operators from of the GraphFrames API itself (specific for graphs) in a function-like style assembling a lazy evaluated pipeline of transformed data. The vertex and edge instances are the input for model transformations.

Szárnyas et al. (2014) handle MDE scalability issues, proposing an architecture (the IncQuery-D) for a distributed and incremental model query framework by adapting incremental graph pattern matching techniques to a distributed cloud based infrastructure. This architecture evaluates graph patterns over EMF models using Rete algorithm in a distributed environment. It focuses in distributed data store and distributed query evaluation network for model transformations. The Rete algorithm uses tuples to represent the vertices, edges and subgraphs in the graph. This graph representation is similar with our approach, which uses Dataframes for representing vertices and edges into a Graphframe (graph instance). Furthermore, the IncQuery-D uses incremental queries based on joins to specify rules transformation, similar to the specifications (transformation rules) of our approach.

The works (Szárnyas et al., 2014; Junghanns et al., 2016; Benelallam et al., 2018) address the scalability with model partitioning and or graph pattern techniques in MT. Our approach includes these aspects on Spark, a scalable framework. Furthermore, the lazy-evaluate in monotonic operations, the implicit parallelism, transformation rules in declarative specifications (SQL-like functions), data collections, and parallel/distributed environment establish the technical space of our approach.

6 Conclusion

We applied a Dc approach for model transformations through the GraphFrames API, including model extraction. We evaluate the API, together with an implemented extraction procedure, to assess if they are a valid alternative for directed-graph operations including model transformations. We developed a Model Extraction from technical modeling spaces to the Apache framework on its DataFrame and GraphFrame formats. From GraphFrame, we developed two partitioning strategies, one based on the Motif algorithm and another based on clustering using the Infomap framework. Both may be used for partitioning models, but their outputs are not balanced.

We also developed a set of operations and transformation rules on the Scala language and validated them with a proof of concept using the Spark framework on four nodes, in local mode. The results obtained indicate that the extraction of large semi-structured data under a directed-graph perspective can be useful in choosing a strategy to design model transformations in a scalable platform, such as the Spark framework. In addition, the model GraphFrame may be used for model partitioning, graph-data processing, and to analyze model inter-connectivity, as well as to offer graph-structured information to different contexts. However, there is a need for further studies to apply more sophisticated strategies in model partitioning and for improving the integration with the Spark framework.

As future work, we plan run transformation rules using GraphFrame on distributed environments such as cloud computing, aiming for a benchmark with Very Large Models on top scalable frameworks, to evaluate the scalability and model partition strategies, whilst prioritizing load balancing, minimizing data skew, and improving data locality of submodels. The benchmark can be based on works such as Varro et al. (2005); Szárnyas et al. (2018). Furthermore, it is also worth assessing whether our approach is practical, or too difficult for a typical developer.

References

Ahlgren, B., Hidell, M., and Ngai, E. C. (2016). Internet of things for smart cities: Interoperability and open data. *IEEE Internet Computing*, 20(6):52–56.

Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. (2011). Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011*, pages 249–260, CA, USA. CIDRDB.

Anjorin, A., Leblebici, E., and Schürr, A. (2016). 20 years of triple graph grammars: A roadmap for future research. *Electronic Communications of the EASST*, 73.

Apache, S. F. (2019). Apache spark, 2019 may, release 2.4.3. `https://spark.apache.org/`. Online, accessed 2019-08.

Aslak, U., Rosvall, M., and Lehmann, S. (2018). Constrained information flows in temporal networks reveal intermittent communities. *Phys. Rev. E 97, 062312 (2018)*, 97(6):062312.

Azzi, G. G., Bezerra, J. S., Ribeiro, L., Costa, A., Rodrigues, L. M., and Machado, R. (2018). The verigraph system for graph transformation. In Heckel, R. and Taentzer, G., editors, *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*, pages 160–178. Springer International Publishing.

Barquero, G., Burgueño, L., Troya, J., and Vallecillo, A. (2018). Extending complex event processing to graph-structured information. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 166–175, New York, NY, USA. ACM.

Batory, D. and Azanza, M. (2017). Teaching model-driven engineering from a relational database perspective. *Software & Systems Modeling*, 16(2):443–467.

Benelallam, A., Gómez, A., Tisi, M., and Cabot, J. (2015). Distributed model-to-model transformation with atl on mapreduce. In *2015 ACM SIGPLAN Software Language Engineering*, SLE 2015, pages 37–48, New York, NY, USA. ACM.

Benelallam, A., Gómez, A., Tisi, M., and Cabot, J. (2018).

Distributing relational model transformation on mapreduce. *Journal of Systems and Software*, 142:1 – 20.

Benelallam, A., Tisi, M., Cuadrado, J. S., de Lara, J., and Cabot, J. (2016). Efficient model partitioning for distributed model transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 226–238, New York, NY, USA. ACM.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):10008.

Bohlin, L., Edler, D., A., L., and M., R. (2014). Mapequation framework.

Bollati, V. A., Vara, J. M., Jiménez, A., and Marcos, E. (2013). Applying mde to the (semi-)automatic development of model transformations. *Inf. Softw. Technol.*, 55(4):699–718.

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

Burgueno, L., Troya, J., Wimmer, M., and Vallecillo, A. (2015). Parallel in-place model transformations with lintra. In *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering*, pages 52–62.

Burgueno, L., Wimmer, M., and Vallecillo, A. (2016). A linda-based platform for the parallel execution of out-place model transformations. *Inf. Software Technology*, 79:17–35.

Camargo, L. C. and Fabro, M. D. D. (2019). Applying a data-centric framework for developing model transformations. In *ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1570–1573, New York, NY, USA. Association for Computing Machinery.

Chambers, B. and Zaharia, M. (2018). *Spark: The Definitive Guide*, volume 1. Ó Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, USA, 1 ed. edition.

Daniel, G., Sunye, G., Benelallam, A., Tisi, M., Vernageau, Y., Gomez, A., and Cabot, J. (2017). Neoemf: A multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9 – 14. Special Issue on MODELS'16.

Daniel, G., Sunyé, G., and Cabot, J. (2016). Umltographdb: Mapping conceptual schemas to graph databases. In Comyn-Wattiau, I., Tanaka, K., Song, I.-Y., Yamamoto, S., and Saeki, M., editors, *Conceptual Modeling*, pages 430–444, Cham. Springer International Publishing.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Eclipse, F. (2019). Atl transformations list (zoo). `http://www.eclipse.org/atl/atlTransformations/`. Online, accessed 2019/02.

Edgar, J., Sebastian, B., Dennis, W., Li, D., Abel, H., Markus, H., Tassilo, H., Elina, K., Christian, K., Kevin, L., Markus, L., Arend, R., Louis, R., Sebastian, W., and Steffen, M. (2014). A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85:41 – 99. Special issue on Experimental Software Engineering in the Cloud(ESEiC).

Edler, D., Bohlin, L., and Rosvall, M. (2017). Mapping higher-order network flows in memory and multilayer networks with infomap. *CoRR*, abs/1706.04792.

Gao, Y., Zhou, Y., Zhou, B., Shi, L., and Zhang, J. (2017). Handling data skew in mapreduce cluster by using partition tuning. In *Journal of healthcare engineering*, pages 1–12.

Gómez, A., Tisi, M., Sunyé, G., and Cabot, J. (2015). Map-based transparent persistence for very large models. In *Fundamental Approaches to Software Engineering 18th International Conference, (FASE)*, pages 19–34.

Hermann, F., Ehrig, H., Golas, U., and Orejas, F. (2014). Formal analysis of model transformations based on triple graph grammars. *Mathematical Structures in Computer Science*, 24(4).

Hochbaum, D. S. (2008). The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Oper. Res.*, 56(4):992–1009.

Imre, G. and Mezei, G. (2012). Parallel graph transformations on multicore systems. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, MSEPT'12, pages 86–89, Berlin, Heidelberg. Springer-Verlag.

Jia, X. and Jones, C. (2015). Design of adaptive domain-specific modeling languages for model-driven mobile application development. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–6.

Jin, J., Luo, J., Song, A., Dong, F., and Xiong, R. (2011). Bar: An efficient data locality driven task scheduling algorithm for cloud computing. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 295–304.

Jouault, F., Allilaire, F., Bézivin, J., and I., K. (2008). Atl: A model transformation tool. *Science of Computer Programming*, 72(1):31 – 39. Special Issue on Second issue of experimental software and toolkits (EST).

Junghanns, M., Petermann, A., Teichmann, N., Gómez, K., and Rahm, E. (2016). Analyzing extended property graphs with apache flink. In *SIGMOD workshop on Network Data Analytics (NDA)*, pages 1–8.

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., and Varró, D. (2018). Survey and classification of model transformation tools. *Software & Systems Modeling*.

Kendig, C. E. (2016). What is proof of concept research and how does it generate epistemic and ethical categories for future scientific practice? In Nature, S., editor, *Science and Engineering Ethics*, pages 735–753. Springer International Publishing, Switzerland AG.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). *The Epsilon Transformation Language*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, volume 1. Prentice Hall, Upper Saddle River, United States, 3 ed. edition.

Le, Y., Liu, J., Ergün, F., and Wang, D. (2014). Online load balancing for mapreduce with skewed data input. In *IEEE*

INFOCOM 2014 - IEEE Conference on Computer Communications, pages 2004–2012.

Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. (2017). A simple yet effective balanced edge partition model for parallel computing. *SIGMETRICS Perform. Eval. Rev.*, 45(1):6–6.

Löwe, M. (2018). Model transformations as free constructions. In Heckel, R. and Taentzer, G., editors, *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*, pages 142–159. Springer International Publishing, Cham.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif. University of California Press.

Michael l., S. (2016). *Programming Language Pragmatics*. Morgan Kaufmann, 4 ed. edition.

Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). Network motifs: Simple building blocks of complex networks. *Science (New York, N.Y.)*, 298:824–7.

OMG (2016). Qvt query view transformation, formal/2016-06-03 v1.3. http://www.omg.org/spec/QVT. Accessed in 2018/06.

Pagán, J. E., Cuadrado, J. S., and Molina, J. G. (2015). A repository for scalable model management. *Software & Systems Modeling*, 14(1):219–239.

Raman, R. (2015). Encoding data structures. In Rahman, M. S. and Tomita, E., editors, *WALCOM: Algorithms and Computation*, pages 1–7, Cham. Springer International Publishing.

Rutle, A., Rossini, A., Lamo, Y., and Wolter, U. (2012). A formal approach to the specification and transformation of constraints in mde. *The Journal of Logic and Algebraic Programming*, 81(4):422 – 457.

Schürr, A. (1995). Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG 94, pages 151–163. Springer-Verlag.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., and Zaniolo, C. (2016). Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD16, pages 1135–1149.

Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., and Varró, D. (2014). Incquery-d: A distributed incremental model query framework in the cloud. In Dingel, J., Schulte, W., Ramos, I., Abrahão, S., and Insfran, E., editors, *Model-Driven Engineering Languages and Systems*, pages 653–669. Springer International Publishing.

Szárnyas, G., Izsó, B., Ráth, I., and Varró, D. (2018). The train benchmark: cross-technology performance evaluation of continuous model queries. *Software System Model, 17*, 4:28.

Tang, M., Shao, S., Yang, W., Liang, Y., Yu, Y., Saha, B., and Hyun, D. (2019). Sac: A system for big data lineage tracking. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1964–1967.

Tisi, M., Martínez, S., and Choura, H. (2013). Parallel execution of atl transformation rules. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, pages 656–672, New York, NY, USA. Springer-Verlag New York, Inc.

Tomaszek, S., Leblebici, E., Wang, L., and Schürr, A. (2018). Model-driven development of virtual network embedding algorithms with model transformation and linear optimization techniques. In Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D., and Seidl, C., editors, *Modellierung 2018*, pages 39–54, Bonn. Gesellschaft für Informatik e.V.

Vara, J. M. and Marcos, E. (2012). A framework for model-driven development of information systems. *Journal of Systems Software.*, 85(10):2368–2384.

Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., and Ujhelyi, Z. (2016). Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629.

Varro, G., Schurr, A., and Varro, D. (2005). Benchmarking for graph transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 79–88.

W3C (2014). Rdf 1.1 concepts and abstract syntax.

Wischenbart, M., Mitsch, S., Kapsammer, E., Kusel, A., Pröll, B., Retschitzegger, W., Schwinger, W., Schönböck, J., Wimmer, M., and Lechner, S. (2012). User profile integration made easy: Model-driven extraction and transformation of social network schemas. In *Proceedings of the 21st International Conference on World Wide Web*, pages 939–948.

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6.