A Two-Level Formal Model for Big Data Processing Programs

João Batista de Souza Neto, Anamaria Martins Moreira, Genoveva Vargas-Solar, Martin A Musicante

Highlights

A Two-Level Formal Model for Big Data Processing Programs

João Batista de Souza Neto, Anamaria Martins Moreira, Genoveva Vargas-Solar, Martin A. Musicante

- This is an extended version of *Modeling Big Data Processing Programs*, by João Batista de Souza Neto, Anamaria Martins Moreira, Genoveva Vargas-Solar and Martin A. Musicante. SBMF 2020.
- This extended version contains the following improvements, in relation to the SBMF 2020 paper:
  - Extension of the modeling primitives to support iterative programs. This is the main contribution of the extended version. Section 3.3 present the *iterate* and *iterateWithCondition* primitives and their semantics in terms of Monoid Algebra. Iterations are represented by a loop on the Petri Net that defines the program. In order to have an acyclic graph to represent the program, these loops are *unfolded* to build a Petri Net without cycles. We give an example use of these operations, by modeling a Spark program taken from [2]. In this example, we show how the upper layer (Petri Net) of the model is unfolded to consider the new primitives. We conclude that the extension of the model provides more expressiveness to model both non-iterative and iterative Big Data processing algorithms. Particularly, iterative ones that come up in current analytics algorithms based on data mining, machine learning, graph analysis, and artificial intelligence techniques.
  - The inclusion of the description of Big Data Processing Frameworks, describing their characteristics and highlighting the strategies they propose concerning the implementation of iterative algorithms. Given the diversity of strategies adopted for addressing iteration, we discuss

the importance of providing an abstract model that can represent issues related to iteration. The description includes, for each framework, a list of primitives they provide and their correspondence to the operations in our model (see Table 1).

– Besides the new technical content, the paper was revised and expanded as follows: The new contents include:

(i) A better motivation and more clear explanation of the application of the model;

(ii) Improved description of Petri Nets and Monoid Algebra. We particularly extended the description of the Monoid Algebra including the *repeat* operation.

(iii) Better and more complete description of the model. In particular, we include the definition of some primitive operations that do not appear in the SBMF paper. In this sense, the extended version provides a full description of the proposed model.

(iv) Overall improvement of the Related Work and Conclusions. In the related work, we refer to very recent works addressing iterative algorithms in Big Data processing frameworks. Our model is complementary to these approaches since it attempts to model iteration independently of the technical characteristics of concrete target frameworks.

A Two-Level Formal Model for Big Data Processing Programs

João Batista de Souza Neto[a,b,*], Anamaria Martins Moreira[c], Genoveva Vargas-Solar[d], Martin A. Musicante[a]

[a] *Department of Informatics and Applied Mathematics (DIMAp)*
Federal University of Rio Grande do Norte, Natal, Brazil.
[b] *Department of Informatics, Management and Design (DIGD-DV)*
Federal Center for Technological Education of Minas Gerais, Divinópolis, Brazil.
[c] *Institute of Computing (IC)*
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.
[d] *French Council of Scientific Research (CNRS), LIRIS, Lyon, France.*

Abstract

This paper proposes a model for specifying data flow-based parallel data processing programs agnostic of target Big Data processing frameworks. The paper focuses on the formal abstract specification of non-iterative and iterative programs, generalizing the strategies adopted by data flow Big Data processing frameworks. The proposed model relies on *Monoid Algebra* and *Petri Nets* to abstract Big Data processing programs in two levels: a higher level representing the program data flow and a lower level representing data transformation operations (e.g., filtering, aggregation, join). We extend the *model for data processing programs* proposed in [1], for modeling iterative data processing programs. The general specification of these programs implemented by data flow-based parallel programming models is essential given the democratization of iterative and greedy Big Data analytics algorithms. Indeed, these algorithms call for revisiting parallel programming models to express iterations. The paper gives

*This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

**This paper is an extended version of [1].

*Corresponding author

Email addresses: jbsneto@ppgsc.ufrn.br,jbsneto@cefetmg.br (João Batista de Souza Neto), anamaria@ic.ufrj.br (Anamaria Martins Moreira), genoveva.vargas-solar@liris.cnrs.fr (Genoveva Vargas-Solar), mam@dimap.ufrn.br (Martin A. Musicante)

a comparative analysis of the iteration strategies proposed by Apache Spark, DryadLINQ, Apache Beam, and Apache Flink. It discusses how the model achieves to generalize these strategies.

Keywords: Big Data processing, Data flow programming models, Petri Nets, Monoid Algebra

---

1. Introduction

The intensive processing and analytics of datasets with significant volume, variety, and velocity scales, namely Big Data, calls for parallel programming models adapted to exploiting their potential. Parallel programming models are based on the principle that significant problems can often be divided into smaller ones, which can then be solved simultaneously. Large-scale data processing frameworks have emerged as solutions to process and analyze Big Data through parallel programming models. An execution engine provided by the framework manages the parallel and distributed processing of datasets transparently. This facility allows developers to avoid dealing with low-level details inherent to the use of distributed and parallel environments.

Big Data processing frameworks can be general-purpose, SQL-based, graph processing, and stream processing [3]. General-purpose frameworks implement parallel programming models adopting either control flow-based or data flow-based approaches. Under the control flow strategy, a single system node controls the entire program execution (master, orchestrator). In the data flow parallel model, the processes that execute the program trigger the execution of other program components. An example of control flow-based systems is *Apache Hadoop* [4] and examples of data flow-based systems are *Dryad/DryadLINQ* [5, 6], *Apache Flink* [7], *Apache Beam* [8] and *Apache Spark* [9]. Data flow-based models have gained popularity for implementing Big Data processing and analytics parallel programs. Therefore we address the formal modeling of data flow-based parallel programs models.

In data flow-based frameworks, a program is built from individual processing

blocks. These processing blocks implement operations that perform *transformations* on the data. The interaction between these blocks defines the *data flow* that specifies the order to perform operations. Datasets exchanged among the blocks are modeled by data structures such as key-value tuples or tables. The sequence of operations applied in a data flow-based program is modeled by a DAG (Directed Acyclic Graph) representing the program's execution plan.

Depending on the dataset properties (velocity, volume), performance expectations, and computing infrastructure characteristics (cluster, cloud, HPC nodes), it is often a critical programmer's decision to choose a well-adapted target system used for running data processing programs. This diversity suggests that programs have different performance scores depending on their context and available resources. Programs performance does not depend entirely on efficient code but on infrastructure configuration tuning often done empirically (i.e., trial and error) and embedded in the code (e.g., caching and data sharing strategies). Therefore, this close dependence of programs, data and infrastructure configuration leads to solutions challenging to reuse, maintain and enhance.

We believe that abstract programs can promote the design of platform-agnostic code that would be deployed in a variety of frameworks. Having formal models of the parallel execution implemented by frameworks of the same family can be used for comparing infrastructures, defining pipelines to test parallel data processing programs, and verifying programs properties (such as correctness, completeness, or concurrent access to data).

To the extent of our knowledge, most works addressing Big Data processing programs have worked on technical and engineering challenging aspects. Few works, such as [10], [11], and [12] have worked on formal specifications to reason about their execution abstractly. A first version [1] of the work presented here introduced a model for non-iterative Big Data processing programs. The main goal of our approach is to have an abstract representation common to data-centric programs. This representation can be used to compare different frameworks and as (intermediate) representation to translate, refine, or optimize programs. We have used the model to define mutation operators that can

be instantiated for different systems. In particular, specifications in our model have been used as an intermediate representation of programs in a mutation testing tool of Apache Spark programs [13].

Our model has two levels: a higher level, where we use *Petri Nets* [14] to represent the program data flow, and a lower level, where we use *Monoid Algebra* [15, 16], a formal system to describe processing of distributed data, to represent data transformation operations. Combining these formalisms allows the program logic to be described independently of the target Big Data processing system, by: *(i)* representation of programs execution through directed acyclic graphs (DAGs) where vertices represent operations and datasets, and edges represent data communication, and *(ii)* operations applied on data (e.g., filtering, aggregation, join). Throughout the paper, we use the notion of *data flow* to refer to the representation of a program's data flow graph and *transformations* to the operations over datasets that compose the program.

Big Data processing and analytics tasks combine unary non-iterative operations like filtering and aggregations (e.g., max, min) and binary operations like joins that are addressed by the first version of our model proposed in [1]. However, there are iterative tasks that are not addressed by the first version of our model. Examples of these are machine learning algorithms.

Therefore, this paper extends the model for data processing programs proposed in [1], to also model iterative programs. Besides the introduction of iterative processing primitives, this paper extends [1] by *(i)* providing a complete description of our model, including more comprehensive use of the resources provided by Petri Nets; *(ii)* giving a more detailed comparison of data flow-based systems to show how our proposal can model them; and *(iii)* showing a potential application of the extended model to derive iteration, specific mutation operators.

The remainder of the paper is organized as follows. Section 2 introduces the main characteristics of data flow-based Big Data processing frameworks and presents the formalisms used in the model, namely, Petri Nets and Monoid

Algebra. Section 3 presents the model for formally expressing Big Data processing programs. Section 4 discusses how our proposal can model operations and iteration strategies of data flow-based Big Data processing frameworks. Section 5 describes the general lines of the way the model can be used in a concrete program testing application. Section 6 introduces related work addressing approaches for generalizing control and data flow parallel programming models. Finally, Section 7 concludes the paper and discusses future work.

2. Background

This section introduces big data processing frameworks to summarize their main characteristics that were considered for the construction of the model presented in this work. Afterward, we make a brief presentation of Petri Nets and Monoid Algebra, the two formalisms used to build the model.

2.1. Big Data Processing Frameworks

Big Data processing frameworks adopt *control flow* or *data flow* based parallel programming models for implementing programs. Dependence analysis is a formal theory in compilation theory for determining ordering constraints between computations [17]. The theory distinguishes between control and data dependencies. Control flow models focus on sequential (imperative) programming [18]. Thus the data follows the control, and computations are executed explicitly based on the sequence programmed. Data flow models focus on data dependencies and allow avoiding spurious control dependencies like accidental locking [18], which simplifies the definition of concurrent and independent computations.

Apache Hadoop [4] is an open-source control flow system for the processing of distributed data that implements the *MapReduce* programming model [19]. MapReduce is a parallel computing model that divides processing into two operations: *map* and *reduce*. The *map* operation applies the same function to all the elements of a list of key/value records. The result of *map* is feed to

the *reduce* operation which processes key/value data aggregated by the key. Other systems such as Apache Spark [9], Apache Flink [7], Apache Beam [8], and Dryad/DryadLINQ [5, 6], adopt data flow models that show better performance.

With increasing interest in running these algorithms on massive datasets, there is a need to execute iterations in a massively parallel fashion. Therefore, existing systems propose different strategies for implementing iterative operations. The following lines analyze and compare these strategies.

Apache Spark. [9] is a general-purpose system for in-memory parallel data processing. Spark is centered on the concept of RDDs (*Resilient Distributed Datasets*), which are distributed datasets that can be processed in parallel in a processing cluster. Spark programs are represented through a DAG that defines the program's data flow, where RDDs are processed by applying operations to them. Spark offers two types of operations. *Transformations*, which process the data in an RDD and generate a new RDD as output, and *actions*, which save the contents of the RDD or generate a different result from an RDD. Spark adopts a lazy evaluation strategy, where actions trigger the processing of data, possibly applying transformations.

The in-memory processing of Spark proved to be more efficient than that of Apache Hadoop, making it more suitable for iterative programs since intermediate data does not need to be stored on disk [9], as occurs in Hadoop. However, Spark does not have a native solution for defining iterative programs, making it necessary to use resources from the underlying programming language, like *while* and *for* loops, so that iterations can be defined. Since Spark adopts a lazy evaluation strategy, the data flow definition through the call of successive transformations forms an execution plan. This plan is optimized in a DAG and executed in parallel when an action is called. The definition of iterative programs follows the same principle. In this way, transformations called within an iteration form a step in the execution plan, making these transformations to be repeated in the DAG as many times as the number of iterations programmed

in the loop.

Apache Beam. [8] is a unified model for defining both batch and streaming data-parallel processing pipelines. Beam is well adapted for implementing parallel data processing tasks, where the problem can be decomposed into many smaller bundles of data that can be processed independently and in parallel. A pipeline can be executed by one of Beam's supported distributed processing back-ends, which include *Apache Flink*, *Apache Spark*, and *Google Cloud Dataflow*.

Apache Beam programs are defined as data pipelines (Pipeline) that encapsulate its data flow with distributed data collections (PCollection) and data processing operations (PTransform). Thus, a program is defined by reading an input dataset, applying operations to datasets, and writing an output dataset. This pipeline is optimized in a DAG and submitted for execution in a back-end engine. Similar to Apache Spark, Beam does not provide a definitive solution for implementing iterative programs. Thus, the definition of iterative programs is based on the use of resources from the underlying programming language, relying on external control to the pipeline to control iterations.

Dryad/DryadLINQ. [5] is a system and model for parallel and distributed programming that was proposed by *Microsoft*. *Dryad* offered a flexible programming model by representing a program through a DAG where the vertices are processing operations, and the edges are communication channels through which data is transferred. With this model, a program is not limited to just two operations as in MapReduce. *Dryad* was expanded through *DryadLINQ* [6], a high-level interface that introduces an abstraction for representing distributed datasets (*DryadTable*) and offered a comprehensive set of operations. A program in *DryadLINQ* is represented by a data stream defined as a DAG, in which datasets are processed by applying operations in sequence. The definition of iterative programs in Dryad/DryadLINQ also follows the approach of Apache Spark and Apache Beam, *i.e.*, there is no native operation to control iterations, but they can be defined using loops from the underlying programming language.

Apache Flink. is a framework and distributed processing engine for batch and streaming data processing [7]. The system processes arbitrary data flow programs in a distributed runtime environment. The data flow is organized as a DAG with one or more entry or exit points in other frameworks. Flink implements a lightweight fault-tolerant model based on the use of *checkpoints* that can be manually placed in the program, or the system can add that. Flink offers the DataSet API for batch processing and the DataStream API for streaming processing. Both offer a comprehensive set of operations for data processing, with mapping, filtering, and aggregation operations, in addition to other types of operations.

From the Big Data processing frameworks analyzed in this work, Flink is the only one that offers a native solution for iterative programs. For the definition of iterative programs, Flink offers the *iterate* operation. This operation takes as an argument a high-order function, called step function, which encapsulates the iterative data flow that consumes an input dataset and produces an output dataset, which in turn is the input for the next iteration. The iterate operator implements a simple form of iterations: in each iteration, the step function consumes the entire input (the result of the previous iteration or the initial dataset) and computes the next version of the partial solution. There are two options to specify termination conditions for an iteration specifying: (i) the maximum number of iterations; the iteration will be executed this many times; (ii) custom convergence function that implements a convergence criterion to end iterations. Flink also offers the delta iterate operator (*iterateDelta*) to address the case of incremental iterations that selectively modify elements of their solution and evolve the solution rather than fully recompute it. This strategy leads to more efficient algorithms because not every element in the solution set changes in each iteration.

2.2. Petri Nets

Petri Nets [20] are a formal tool to model and analyze the behavior of distributed, concurrent, asynchronous, and/or non-deterministic systems [14]. A

Petri Net is defined as a directed bipartite graph that contains two types of nodes: *places* and *transitions*. Places represent the system's state variables, while transitions represent the actions performed by the system. These two components are connected through directed edges that connect places to transitions and transitions to places. With these components, it is possible to represent (i) the different states of a system; (ii) the actions taken by the system to move from one state to another (transitions) (iii) and how the state changes due to actions (edges). This modeling is done by using *tokens* to decorate places of the net. The distribution of the tokens among places indicates that the system is in a given state. The execution of an action (transition) takes tokens from one place to another, leading to the system's state evolution.

Formally, a Petri net is a quintuple $PN = (P, T, F, W, M_0)$ where $P \cap T = \emptyset$, $P \cup T \neq \emptyset$ and:

$P = \{p_1, p_2, \ldots, p_m\}$	is a finite set of *places*,
$T = \{t_1, t_2, \ldots, t_n\}$	is a finite set of *transitions*,
$F \subseteq (P \times T) \cup (T \times P)$	is a finite set of *edges*,
$W : F \to \{1, 2, 3, \ldots\}$	is function associating positive weights to edges,
$M_0 : P \to \{0, 1, 2, 3, \ldots\}$	is a function defining the initial marking of a net.

The execution of a system is defined by *firing* transitions. Firing a transition t consumes $W(s, t)$ tokens from all its input places s, and produces $W(t, s')$ tokens to each of its output places s'. The transition t can only be fired (it is said to be *enabled*) if there are at least $W(s, t)$ tokens on all its input places s. The semantics of a given process is then given by the evolution of markings produced by firing enabled transitions.

2.3. Monoid Algebra

Monoid Algebra was proposed in [15] as an algebraic formalism for data-centric distributed computing operations based on monoids and monoid homomorphisms. A monoid is an algebraic structure $(S, \oplus, e_\oplus)$ formed by a set S,

an associative operation $\oplus$ in S and a neutral element $e_{\oplus}$. The function $\oplus$ is usually used to identify the monoid. A *monoid homomorphism* is a function H over two monoids, say $\otimes = (S, \otimes, e_{\otimes})$ to $\oplus = (T, \oplus, e_{\oplus})$, such that:

$$H(X \otimes Y) = H(X) \oplus H(Y) \quad \text{for all } X \text{ and } Y \text{ of type } S$$
$$H(e_{\otimes}) = e_{\oplus}$$

Monoid algebra uses monoid and monoid homomorphism concepts to define operations on distributed datasets, which are represented as *monoid collections*. The associative nature of monoid homomorphisms makes it possible to model parallel operations over distributed data.

One type of monoid collection is *bag*, an unordered data collection of elements of type α (denoted as $Bag[\alpha]$). The elements of $Bag[\alpha]$ are formed by using the unit injection function $\mathbb{U}_{\uplus}$, which generates the unitary bag $\{\!\{x\}\!\}$ from an element x ($\mathbb{U}_{\uplus}(x) = \{\!\{x\}\!\}$), the associative operation $\uplus$, which unites two bags ($\{\!\{x\}\!\} \uplus \{\!\{y\}\!\} = \{\!\{x, y\}\!\}$), and the neutral element $\{\!\{\}\!\}$, which is an empty bag. Another monoid collection is the one formed by lists. It can be defined as an ordered bag. It can be defined from the set $List[\alpha]$ containing lists of elements of type α, and using $\mathbb{U}_{++}$ as the unit injection function, the list concatenation $++$ as the associative operation and the empty list $[\,]$ as the neutral element of the monoid.

Monoid algebra defines distributed operations as monoid homomorphisms over monoid collections (which represent distributed datasets). These homomorphisms are defined to abstractly describe the basic blocks of distributed data processing systems such as map/reduce or data flow systems. The key idea behind monoid algebra is to use the associativity property of the monoid operations and the homomorphism between monoids to represent the processing of partitioned data and the combination of the results, independently from how data is partitioned.

Let us now define the most common operations used in monoid algebra. The **flatmap** operation receives a function f of type $\alpha \rightarrow Bag[\beta]$ and a collection X

of type $Bag[\alpha]$ as input and returns a collection $Bag[\beta]$ resulting from the union $\uplus$ of the results of applying f to each element of X. This operation captures the essence of parallel processing since f can be executed in parallel on top of different data partitions in a distributed dataset. Notice that **flatmap** f is a monoid homomorphism since it is a function that preserves the structure of bags.

The operations **groupby** and **cogroup** capture the data shuffling process by representing the reorganization and grouping of data. The **groupby** operation groups the elements of $Bag[\kappa \times \alpha]$ using the first component of type κ as a key, resulting in a collection $Bag[\kappa \times Bag[\alpha]]$, where the second component is a collection containing all elements of type α that were associated with the same key k in the initial collection. The **cogroup** operation works similarly to **groupby**, but it operates on two collections that have a key of the same type κ. In this way, the result of **cogroup**, when applied to two collections of type $Bag[\kappa \times \alpha]$ and $Bag[\kappa \times \beta]$ is a collection of type $Bag[\kappa \times Bag[\alpha \times \beta]]$.

The **reduce** operation represents the aggregation of the elements of $Bag[\alpha]$ into a single element of type α from the application of an associative function f of type $\alpha \to \alpha \to \alpha$.

The operation **orderby** represents the transformation of a bag $Bag[\kappa \times \alpha]$ into a list $List[\kappa \times \alpha]$ ordered by the key of type κ which supports the total order $\leq$.

These operations are monoid homomorphisms, as proved in [15]. This property makes it possible to make transparent to the model how data has been distributed when parallelizing tasks. However, they are not enough to model applications where iteration is needed. For this, Monoid algebra, as presented in [15], includes the **repeat** operation.

The **repeat** operation provided by Monoid Algebra is used to allow the representation of iterative algorithms [15], such as machine learning and graphs processing algorithms. The **repeat** operation receives a function f of type $Bag[\alpha] \to Bag[\alpha]$, a predicate p of type $Bag[\alpha] \to boolean$, a count number n, and a collection X of type $Bag[\alpha]$ as input and returns a collection of type

$Bag[\alpha]$ as output. The definition of **repeat** is given below [16]:

$$\begin{aligned}\textbf{repeat}(f, p, n, X) \triangleq\ & if\ n \leq 0\ \vee\ \neg p(X) \\ & then\ X \\ & else\ \textbf{repeat}(f, p, n-1, f(X))\end{aligned}$$

The **repeat** operation stops when the counter n is zero or the condition p in X is false. While these conditions are not met, the operation computes $f(X)$ and decrements n recursively. Intuitively, in each iteration, the collection resulting from the previous iteration is processed by f which produces a new collection for the next iteration (or for the output when **repeat** stops).

In addition, monoid algebra also supports the use of lambda expressions $(\lambda x.e)$, conditionals (**if-then-else**).

Our proposal combines Petri Nets with Monoid Algebra to build abstract versions of the primitives present in Big Data processing applications.

3. Modeling Big Data Processing Programs

This section introduces the proposed formal model for Big Data processing programs. The model is organized in two levels: *data flow* and *transformations*. Data flow in our model is defined using Petri Nets, and the semantics of the transformations applied to the data are modeled as monoid homomorphisms on datasets.

3.1. Data Flow

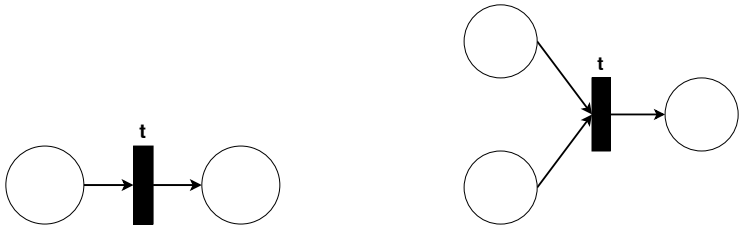
For the upper level of our two-level modelization, we define a graph representing the data flow of a data processing program. We rely on the data flow graph model presented in [21], which was formalized using Petri Nets [14].

A program P is defined as a bipartite directed graph where places stand for the program's distributed datasets (D), and transitions stand for its transformations (T). Datasets and transformations are connected by edges (E):

$$P = \langle D \cup T, E \rangle$$

This graph can be seen as a Petri Net, as defined in Section 2. Datasets correspond to the places of the net, and transformations correspond to the net transitions. The initial marking (M_0) of the Petri Net represents the availability of the input datasets for the computation to begin. There will be as many tokens in an input dataset as the number of its uses in the program. The weight function W is defined as 1 for every edge leaving a place and as k, for every edge arriving at a place, where k is the number of times the same dataset is used in the program. That is, for each edge $(t, d) \in E,\ W(t, d) = |O(d)|$ and for each edge $(t, d) \in E,\ W(d, t) = 1$, where $O(d)$ represents the set of transformations that receive d as input, *i.e.*, the number o edges coming out of d.

To construct the data flow model, the available transformations on the modeled frameworks fall into two categories: basic transformations (without cycles) and iterative transformations. We first present the more common case of acyclic programs. The extension of our model to deal with iterations is presented in Section 3.3. All basic transformations in our model can have their data flow modeled by either a single transition with one input and one output edge (see Figure 1a) or a single transition with two input and one output edges (see Figure 1b). We call *unary transformations* those that receive only one dataset as input and *binary transformations* those that receive two datasets as input. The transitions must be sequenced by matching the corresponding input and output datasets to construct the graph (actually a DAG).



(a) Unary Transformation. (b) Binary Transformation.

Figure 1: Types of transformations in the data flow.

```
1  def unionLogsExample(firstLogs: RDD[String], secondLogs: RDD[
       String]): RDD[String] = {
2   val aggregatedLogLines = firstLogs.union(secondLogs)
3   val uniqueLogLines = aggregatedLogLines.distinct()
4   val cleanLogLines = uniqueLogLines.filter((line: String) => !(line.
        startsWith("host") && line.contains("bytes")))
5   return cleanLogLines
6  }
```

Figure 2: Sample log union program in Spark.

To illustrate the model, let us consider the Spark program shown in Figure 2. This program receives as input two datasets (RDDs) containing log messages (line 1). It makes the union of these two datasets (line 2), removes duplicate logs (line 3), and ends by filtering headers, removing logs that match a specific pattern (line 4). The program ends by returning the filtered RDD (line 5).

In this program, we can identify five RDDs, that will be referred to using short names for conciseness. So, $D = \{d_1, d_2, d_3, d_4, d_5\}$, where $d_1 =$ `firstLogs`, $d_2 =$ `secondLogs`, $d_3 =$ `aggregatedLogLines`, $d_4 =$ `uniqueLogLines`, and $d_5 =$ `cleanLogLines`. For simplicity, each RDD in the code was given a unique name. It makes it easier to reference them in the text. However, the model considers that each RDD is uniquely identified, independently of the concrete name given to it in the code.

We can also identify the application of three transformations in P; thus the set T in our example is defined as $T = \{t_1, t_2, t_3\}$, where $t_1 =$ `union`(d_1, d_2), $t_2 =$ `distinct`(d_3), and $t_3 =$ `filter((line: String) => !(line.startsWith("host") && line.contains("bytes"))`$, d_4)$.

Each transformation in T receives one or two datasets belonging to D as input and produces a dataset in D as output. Besides, the sets D and T are disjoint and finite.

Edges connect datasets with transformations. An edge can be a pair in $D \times T$, representing the input dataset of a transformation, or it can be a pair

in $T \times D$, representing the output dataset of a transformation. In this way, the set of edges of P is defined as $E \subseteq (D \times T) \cup (T \times D)$.

The set E in our example program is, then:

$$E = \{(d_1, t_1), (d_2, t_1), (t_1, d_3), (d_3, t_2), (t_2, d_4), (d_4, t_3), (t_3, d_5)\}$$

Using these sets, we can define a graph representing the Spark program in Figure 2. This graph is depicted in Figure 3. The distributed datasets in D are represented as circle nodes, and the transformations in T are represented as thick bar nodes of the graph, as is usual in representing Petri Nets. The edges are represented by arrows that connect the datasets and transformations. The token marking in d_1 and d_2 indicate that the program is ready to be executed (initial marking). For simplicity, we only indicate the weight of edges of the Petri Net when they are different from 1.

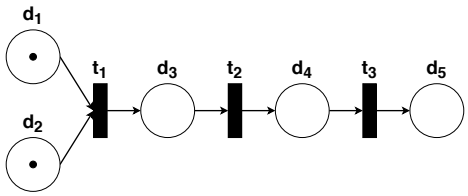


Figure 3: Data flow representation of the program in Figure 2.

3.2. Data Sets and Transformations

The data flow model defined above represents (i) the datasets and transformations of a program P; (ii) the order in which transformations are processed when the program P is executed. These representations are abstract from their actual contents or semantics.

To define the contents of datasets in D and the semantics of transformations in T, we make use of *Monoid Algebra* [15, 16]. Datasets are represented as monoid collections, and transformations are defined as operations supported by monoid algebra. These representations are detailed in the following.

3.2.1. Distributed Datasets

A distributed dataset in D can either be represented by a *bag* ($Bag[\alpha]$) or a *list* ($List[\alpha]$). Both structures represent collections of distributed data [16], capturing the essence of the concepts of *RDD* in Apache Spark, *PCollection* in Apache Beam, *DataSet* in Apache Flink and *DryadTable* in DryadLINQ. These structures provide an abstraction of the actual distributed data in a cluster in the form of a simple collection of items.

We define most of the transformations of our model in terms of bags. We consider lists only for transformations implementing sorts, which are the only ones in which the order of the elements in the dataset is relevant.

In monoid algebra, bags and lists can either represent distributed or local collections. Monoid homomorphisms treat these two kinds of collections in a unified way [16]. In this way, we will not distinguish between distributed and local collections when defining our transformations.

3.2.2. Transformations

In our model, transformations take one or two datasets as input and produce one dataset as an output. Transformations may also receive other parameters such as functions, which represent data processing operations defined by the developer, and literals such as boolean constants. A transformation t in the transformation set T of a program P is characterized by (i) the operation it implements, (ii) the types of its input and output datasets, (iii) and its input parameters.

We define the transformations of our model in terms of the operations of monoid algebra defined in Section 2. We group transformations into categories according to the types of operations we identified in the data processing systems we studied.

Mapping Transformations. transform values of an input dataset into values of an output dataset by applying a mapping function. Our model provides two mapping transformations: *flatMap* and *map*. Both transformations apply a

given function f to every element of the input dataset to generate the output dataset, the only difference being the requirements on the type of f and its relation with the type of the generated dataset. Given an input dataset of type $Bag[\alpha]$, the *map* transformation accepts any $f : \alpha \to \beta$ and generates an output dataset of type $Bag[\beta]$, while the *flatMap* transformation requires $f : \alpha \to Bag[\beta]$ to produce a dataset of type $Bag[\beta]$ as output.

The definition of *flatMap* in our model is just the monoid algebra operation defined in Section 2:

$$\mathit{flatMap} :: (\alpha \to Bag[\beta]) \to Bag[\alpha] \to Bag[\beta]$$
$$\mathit{flatMap}(f, D) = \mathbf{flatmap}(f, D)$$

The *map* transformation derives data of type $Bag[\beta]$ when given a function $f : \alpha \to \beta$. For that to be modeled with the **flatmap** from monoid algebra, we create a lambda expression that receives an element x from the input dataset and results in a $Bag[\beta]$ collection containing only the result of applying f to x ($\lambda x.\{\!\{f(x)\}\!\}$). Thus, *map* is defined as:

$$\mathit{map} :: (\alpha \to \beta) \to Bag[\alpha] \to Bag[\beta]$$
$$\mathit{map}(f, D) = \mathbf{flatmap}(\lambda x.\{\!\{f(x)\}\!\}, D)$$

Filter Transformation. uses a boolean function to determine whether a data item should be mapped to the output dataset. As in the case of *map*, we use a lambda expression to build a singleton bag:

$$\mathit{filter} :: (\alpha \to \mathit{boolean}) \to Bag[\alpha] \to Bag[\alpha]$$
$$\mathit{filter}(p, D) = \mathbf{flatmap}(\lambda x.\ \mathbf{if}\ p(x)\ \mathbf{then}\ \{\!\{x\}\!\}\ \mathbf{else}\ \{\!\{\}\!\}, D)$$

For each element x of the input dataset D, the *filter* transformation checks the condition $p(x)$. It forms the singleton bag $\{\!\{x\}\!\}$ or the empty bag ($\{\!\{\}\!\}$), depending on the result of that test. This lambda expression is then applied to the input dataset using the **flatmap** operation.

For instance, consider the boolean function $p(x) = x \geq 3$ and a bag $D = \{\!\{1, 2, 3, 4, 5\}\!\}$. then, $\mathit{filter}(p, D) = \{\!\{3, 4, 5\}\!\}$.

Grouping Transformations. group the elements of a dataset with respect to a key. We define two grouping transformations in our model: *groupByKey* and *groupBy*. The *groupByKey* transformation is defined as the **groupby** operation of Monoid Algebra. It maps a key-value dataset into a dataset associating each key to a bag. Our *groupBy* transformation uses a function k to map elements of the collection to a key *before* grouping the elements with respect to that key:

$$groupBy :: (\alpha \to \kappa) \to Bag[\alpha] \to Bag[\kappa \times Bag[\alpha]]$$
$$groupBy(k, D) = \mathbf{groupby}(\mathbf{flatmap}(\lambda x.\{\!\{(k(x), x)\}\!\}, D))$$

$$groupByKey :: Bag[\kappa \times \alpha] \to Bag[\kappa \times Bag[\alpha]]$$
$$groupByKey(D) = \mathbf{groupby}(D)$$

For example, let us consider the identity function to define each key, and the datasets $D_1 = \{\!\{1, 2, 3, 2, 3, 3\}\!\}$, and $D_2 = \{\!\{(1, a), (2, b), (3, c), (1, e), (2, f)\}\!\}$. Applying *groupBy* and *groupByKey* to these sets results in:

$$groupBy(\lambda k.k, D_1) = \{\!\{(1, \{\!\{1\}\!\}), (2, \{\!\{2, 2\}\!\}), (3, \{\!\{3, 3, 3\}\!\})\}\!\}$$
$$groupByKey(D_2) = \{\!\{(1, \{\!\{a, e\}\!\}), (2, \{\!\{b, f\}\!\}), (3, \{\!\{c\}\!\})\}\!\}$$

Set-like Transformations. correspond to binary mathematical operations in distributed collections such as those defined in set theory. They operate on two datasets of the same type and result in a new dataset of the same type. The definition of these transformations is based on the definitions in [16].

The *union* transformation represents the union of elements from two datasets into a single dataset. This operation is represented in a simple way using the *bags* union operator ($\uplus$):

$$union :: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha]$$
$$union(D_x, D_y) = D_x \uplus D_y$$

We also define the *intersection* and *subtract* transformations. To define these transformations, we first define auxiliary operations *some* and *all* that represent the existential ($\exists$) and universal ($\forall$) quantifiers, respectively. These operations

receive a predicate function p and reduce the dataset to a logical value:

$$\begin{aligned}
some &:: (\alpha \to boolean) \to Bag[\alpha] \to boolean \\
some(p, D) &= \textbf{reduce}(\vee, t_1(p, D)) \\
t_1(p, D) &= \textbf{flatmap}(\lambda x.\{\!\{p(x)\}\!\}, D) \\
all &:: (\alpha \to boolean) \to Bag[\alpha] \to boolean \\
all(p, D) &= \textbf{reduce}(\wedge, t_1(p, D)) \\
t_1(p, D) &= \textbf{flatmap}(\lambda x.\{\!\{p(x)\}\!\}, D)
\end{aligned}$$

Using *some* and *all*, we can define the transformations *intersection* and *subtract* as follows:

$$\begin{aligned}
intersection &:: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha] \\
intersection(D_x, D_y) &= \textbf{flatmap}(\lambda x.\ \textbf{if}\ some(\lambda y.x = y, D_y) \\
&\qquad\qquad \textbf{then}\ \{\!\{x\}\!\}\ \textbf{else}\ \{\!\{\}\!\}), D_x) \\
subtract &:: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha] \\
subtract(D_x, D_y) &= \textbf{flatmap}(\lambda x.\ \textbf{if}\ all(\lambda y.x \neq y, D_y) \\
&\qquad\qquad \textbf{then}\ \{\!\{x\}\!\}\ \textbf{else}\ \{\!\{\}\!\}), D_x)
\end{aligned}$$

The *intersection* of bags D_x and D_y selects all elements of D_x appearing at least once in D_y. Subtracting D_y from D_x selects all the elements of D_x that differ from every element of D_y.

Unlike the union operation in mathematical sets, the *union* transformation defined in our model maintains repeated elements from the two input datasets. To allow the removal of these repeated elements, we define the *distinct* transformation. To define *distinct*, we first map each element of the dataset to a key/value tuple containing the element itself as a key. After, we group this key/value dataset, which will result in a dataset in which the group is the repeated key itself. Last, we map the key/value elements only to the key, resulting in a dataset with no repetitions. The *distinct* transformation is defined as fol-

lows:

$$distinct :: Bag[\alpha] \to Bag[\alpha]$$
$$distinct(D) = \textbf{flatmap}(\lambda(k, g).\{\!\{k\}\!\}, t_2(D))$$
$$t_1(D) = \textbf{flatmap}(\lambda x.\{\!\{(x, x)\}\!\}, D)$$
$$t_2(D) = \textbf{groupby}(t_1(D))$$

Aggregation Transformations. collapses elements of a dataset into a single element. The most common aggregations apply binary operations on the elements of a dataset to generate a single element, resulting in a single value or on groups of values associated with a key. We represent these aggregations with the transformations *reduce*, which operates on the whole set, and *reduceByKey*, which operates on values grouped by key. The *reduce* transformation has the same behavior as the **reduce** operation of monoid algebra. The definition of *reduceByKey* is also defined in terms of **reduce**, but since its result is the aggregation of elements associated with each key rather than the aggregation of all elements of the set, we first need to group the elements of the dataset by their keys:

$$reduce :: (\alpha \to \alpha \to \alpha) \to Bag[\alpha] \to \alpha$$
$$reduce(f, D) = \textbf{reduce}(f, D)$$

$$reduceByKey :: (\alpha \to \alpha \to \alpha) \to Bag[\kappa \times \alpha] \to Bag[\kappa \times \alpha]$$
$$reduceByKey(f, D) = \textbf{flatmap}(\lambda(k, g).\{\!\{(k, \textbf{reduce}(f, g))\}\!\}, \textbf{groupby}(D))$$

Examples of this kind of transformation are sums, products, or the arithmetic mean of a set of values given as a column of a dataset. For instance, the sum of the values of a single-column dataset may be defined by using *reduce*. Let us consider the function $f(x, y) = x + y$ and the dataset $D_1 = \{\!\{1, 2, 3, 4, 5\}\!\}$. The expression $reduce(f, D_1)$ aggregates the values in D_1 using the sum operation f, resulting in the value 15.

Join Transformations. implement relational join operations between two datasets. We define four join operations, which correspond to well-known operations in

relational databases: *innerJoin*, *leftOuterJoin*, *rightOuterJoin*, and *fullOuterJoin*. The *innerJoin* operation combines the elements of two datasets based on a join-predicate expressed as a relationship, such as the same key. *LeftOuterJoin* and *rightOuterJoin* combine the elements of two sets like an *innerJoin* adding to the result all values in the left (right) set that do not match to the right (left) set. The *fullOuterJoin* of two sets forms a new relation containing all the information present in both sets.

See below the definition of the *innerJoin* transformation, which was based on the definition presented in [22]:

$$
\begin{aligned}
innerJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (\alpha \times \beta)] \\
innerJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_3(k, x, d_y), d_x) \\
t_3(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (x, y))\}\!\}, d_y)
\end{aligned}
$$

The definition of the other joins follows a similar logic, but conditionals are included to verify the different relationships. In cases where one side does not have pairs with a certain key, the result of the join is an empty bag on that side and the element that has the key on the other side. The definitions of *leftOuterJoin*, *rightOuterJoin*, and *fullOuterJoin* are as follows:

$$
\begin{aligned}
leftOuterJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (\alpha \times Bag[\beta])] \\
leftOuterJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if}\ d_y = \{\!\{\}\!\}\ \mathbf{then}\ t_3(k, d_x)\ \mathbf{else}\ t_4(k, d_x, d_y) \\
t_3(k, d_x) &= \mathbf{flatmap}(\lambda x.\{\!\{(k, (x, \{\!\{\}\!\}))\}\!\}, d_x) \\
t_4(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_5(k, x, d_y), d_x) \\
t_5(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (x, \{\!\{y\}\!\}))\}\!\}, d_y)
\end{aligned}
$$

$$rightOuterJoin :: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (Bag[\alpha] \times \beta)]$$

$$rightOuterJoin(D_x, D_y) = \textbf{flatmap}(\lambda(k,(d_x,d_y)).t_2(k,d_x,d_y), t_1(D_x,D_y))$$

$$t_1(D_x, D_y) = \textbf{cogroup}(D_x, D_y)$$

$$t_2(k, d_x, d_y) = \textbf{if } d_x = \{\!\{\}\!\} \textbf{ then } t_3(k, d_y) \textbf{ else } t_4(k, d_x, d_y)$$

$$t_3(k, d_y) = \textbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{\}\!\}, y))\}\!\}, d_y)$$

$$t_4(k, d_x, d_y) = \textbf{flatmap}(\lambda x.t_5(k, x, d_y), d_x)$$

$$t_5(k, x, d_y) = \textbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{x\}\!\}, y))\}\!\}, d_y)$$

$$fullOuterJoin :: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (Bag[\alpha] \times Bag[\beta])]$$

$$fullOuterJoin(D_x, D_y) = \textbf{flatmap}(\lambda(k,(d_x,d_y)).t_2(k,d_x,d_y), t_1(D_x,D_y))$$

$$t_1(D_x, D_y) = \textbf{cogroup}(D_x, D_y)$$

$$t_2(k, d_x, d_y) = \textbf{if } d_x \neq \{\!\{\}\!\} \wedge d_y = \{\!\{\}\!\} \textbf{ then } t_3(k, d_x) \textbf{ else } t_4(k, d_x, d_y)$$

$$t_3(k, d_x) = \textbf{flatmap}(\lambda x.\{\!\{(k, (\{\!\{x\}\!\}, \{\!\{\}\!\}))\}\!\}, d_x)$$

$$t_4(k, d_x, d_y) = \textbf{if } d_x = \{\!\{\}\!\} \wedge d_y \neq \{\!\{\}\!\} \textbf{ then } t_5(k, d_y) \textbf{ else } t_6(k, d_x, d_y)$$

$$t_5(k, d_y) = \textbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{\}\!\}, \{\!\{y\}\!\}))\}\!\}, d_y)$$

$$t_6(k, d_x, d_y) = \textbf{flatmap}(\lambda x.t_7(k, x, d_y), d_x)$$

$$t_7(k, x, d_y) = \textbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{x\}\!\}, \{\!\{y\}\!\}))\}\!\}, d_y)$$

Sorting Transformations. Add the notion of *order* to a bag. In practical terms, these operations receive a bag and form a list, ordered according to some criteria. Sort transformations are defined in terms of the **orderby** operation of monoid algebra, which transforms a $Bag[\kappa \times \alpha]$ into a $List[\kappa \times \alpha]$ ordered by the key of type κ that supports the total order $\leq$ (we will also use the *inv* function, which reverses the total order of a list, thus using $\geq$ instead of $\leq$). We define two transformations, the *orderBy* transformation that sorts a dataset of type α, and the *orderByKey* transformation that sorts a key/value dataset by the key.

The definitions of our sorting transformations are as follows:

$$orderBy :: boolean \rightarrow Bag[\alpha] \rightarrow List[\alpha]$$
$$orderBy(desc, D) = \textbf{flatmap}(\lambda(k, v).[k], \textbf{orderby}(t_1(desc, D)))$$
$$t_1(desc, D) = \textbf{if } desc \textbf{ then } t_2(D) \textbf{ else } t_3(D)$$
$$t_2(D) = \textbf{flatmap}(\lambda x.\{\!\{(inv(x), x)\}\!\}, D)$$
$$t_3(D) = \textbf{flatmap}(\lambda x.\{\!\{(x, x)\}\!\}, D)$$

$$orderByKey :: boolean \rightarrow Bag[\kappa \times \alpha] \rightarrow List[\kappa \times \alpha]$$
$$orderByKey(desc, D) = \textbf{orderby}(t_1(desc, D))$$
$$t_1(desc, D) = \textbf{if } desc \textbf{ then } t_2(D) \textbf{ else } D$$
$$t_2(D) = \textbf{flatmap}(\lambda(k, x).\{\!\{(inv(k), x)\}\!\}, D)$$

The boolean value used as the first parameter defines if the direct order $\leq$ or its inverse is used.

To exemplify the use of sorting transformations let us consider $D_1 = \{\!\{1, 3, 2, 5, 4\}\!\}$ and $D_2 = \{\!\{(1, a), (3, c), (2, a), (5, e), (4, d)\}\!\}$. Then:

$$orderBy(false, D_1) = [1, 2, 3, 4, 5]$$
$$orderBy(true, D_1) = [5, 4, 3, 2, 1]$$
$$orderByKey(false, D_2) = [(1, a), (2, b), (3, c), (4, d), (5, e)]$$
$$orderByKey(true, D_2) = [(5, e), (4, d), (3, c), (2, b), (1, a)]$$

3.3. Modeling Iterative Programs

Iterative algorithms apply an operation repeatedly until a predetermined number of iterations or given conditions are reached. Common iterative algorithms are machine learning algorithms, such as *Logistic Regression* [23], and graph analysis algorithms, such as *PageRank* [24], which perform iterative optimizations and calculations.

Big Data processing systems like Apache Spark, Apache Flink, Apache Beam, and Dryad/DryadLINQ represent their programs as DAGs (Directed Acyclic Graphs). These systems apply a lazy evaluation strategy to execute

programs. Thus, the programs are first defined, then translated into an optimized DAG representing the execution plan, and, finally, sent to run in parallel. Due to this characteristic, iterative programs, characterized by cycles, must be translated into a DAG. Therefore, the operations executed iteratively in the program must be repeated n times in the DAG, where n is the number of iterations performed by the program.

In the systems Apache Spark, Apache Beam, and Dryad/DryadLINQ, iterative programs are defined with the aid of loop statements (such as *for* and *while*) of the underlying programming language to control iterations. Apache Flink, on the other hand, has a native operation (*iterate*) for that, where iterative operations must be encapsulated in a step function that is performed a predetermined number of times or until a specific condition, given by a convergence function, is reached.

Our model relies on the Apache Flink approach to represent the data flow of iterative programs. We define the transformations to be executed iteratively, encapsulating them in a step function that will be repeated as many times as specified in the program. The input and output of the step function must be datasets of the same type so that the output of an iteration is an input for the next one.

Iterative Data Flow. to represent the data flow of an iterative program, we use auxiliary transitions to represent the beginning of the iterations (t_{start}), the repetition of the step function through a cycle in the graph ($t_{iterative}$) and the end of the iterations (t_{end}). These transitions are identity transformations in practice since they do not change the data but only control the iterations. We assume that the iteration starts with an input dataset d_0 and that the step function will be executed n times, resulting in the dataset d_n as output. In the data flow model, we abstract the control of the number of iterations. Thus, the number of iterations in the data flow model is non-deterministic. We delegate this control for a specific transformation that will be presented later. Figure 4 shows how the data flow of an iterative program is represented in our model.

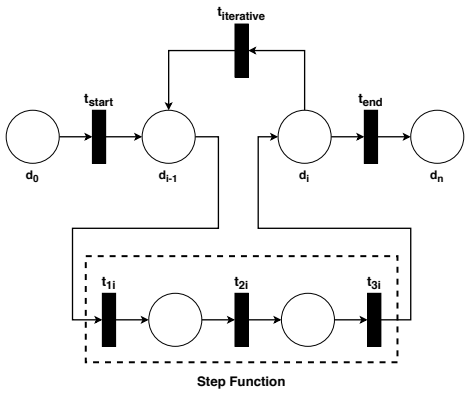


Figure 4: Iterative data flow.

We highlight the step function with dashed lines to represent the part repeated in each iteration.

Each iteration data flow is represented by such a sub-net and, to construct the complete Petri Net for a program, it must be composed with the other transformations as was the case with acyclic transformations. The place corresponding to its initial dataset (d_0) is the output from some previous transition, and the place corresponding to its final dataset (d_n) is the input for a transition in its sequence or a final (output) place.

This model can be reduced into a model without cycles. This reduction is valid because all of the studied systems require either an explicit limit of iterations (n) or the execution plan (which corresponds to the construction of the data flow model) is evaluated before the actual execution of the transformations. Consequently, the execution plan always contains the information on the number of required iterations, making it possible to unfold the iteration as many times as needed. For example, considering the iterative data flow shown in Figure 4, when unfolding this data flow for 3 iterations ($n = 3$), we obtain the data flow shown in Figure 5, in which the auxiliary transitions t_{start}, $t_{iterative}$ and t_{end} were removed and the transformations within the step function have were repeated 3 times.

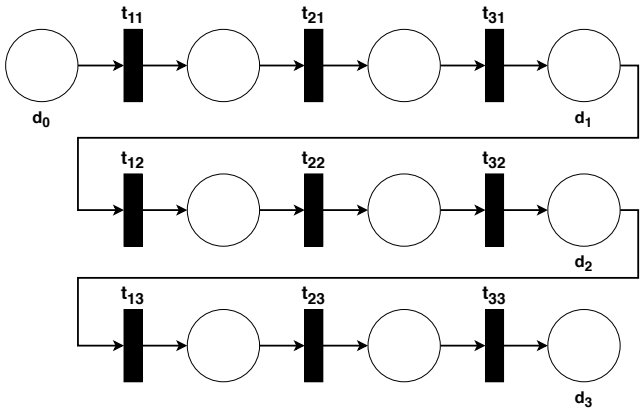


Figure 5: Expanded iterative data flow for 3 iterations.

Iterative Transformations. we define the semantics of iterative transformations in terms of the **repeat** operation of monoid algebra, which receives a step function f of type $Bag[\alpha] \rightarrow Bag[\alpha]$, a predicate function p of type $Bag[\alpha] \rightarrow boolean$, a counter n ($n \in \mathbb{Z}$) and a bag D of type $Bag[\alpha]$ as input and recursively applies the function f until the condition in p is reached or n iterations occur, returning the resulting collection as output.

We define two iterative transformations: *iterate* and *iterateWithCondition*. The *iterate* transformation takes a step function st, a counter n, and a collection D as input and applies st n times. The transformation *iterateWithCondition* is similar, but it receives an additional predicate function p, so it iterates n times or until the condition in p is false, whichever is reached first (n is necessary to avoid an infinite loop if p is never reached). The definitions of *iterate* and *iterateWithCondition* are as follows:

$$iterate :: (Bag[\alpha] \rightarrow Bag[\alpha]) \rightarrow \mathbb{Z} \rightarrow Bag[\alpha] \rightarrow Bag[\alpha]$$

$$iterate(st, n, D) = \mathbf{repeat}(st, \lambda x.true, n, D)$$

$$iterateWithCondition :: (Bag[\alpha] \rightarrow Bag[\alpha]) \rightarrow (Bag[\alpha] \rightarrow boolean) \rightarrow \mathbb{Z} \rightarrow Bag[\alpha] \rightarrow Bag[\alpha]$$

$$iterateWithCondition(st, p, n, D) = \textbf{repeat}(st, p, n, D)$$

Notice that our *iterateWithCondition* transformation corresponds exactly to the **repeat** primitive of Monoid Algebra. Choosing such a primitive is an important design decision for our framework since it provides iterations that have an upper limit in the number of times the body of an iteration may be executed. Despite the profound implications of our choice in terms of the theoretical expressiveness of our system, in practical terms, we believe that the impact of this decision is palliated by the fact that any natural value may be used to set the maximum number of iterations.

Example. to illustrate how an iterative program is represented in our model, let us consider the implementation of the *PageRank* algorithm [24] in Apache Spark presented in Figure 6. This version was based on the implementation presented in [2]. The *PageRank* algorithm calculates the importance (ranking) of a page based on the number of links from other pages to it. Rankings are calculated iteratively so that in each iteration, a page contributes to the ranking of the pages it links to and updates its ranking with the contribution it receives from the other pages that link to it.

The program shown in Figure 6 receives a key/value dataset of *links* as input, where the key is the address of a page, and the value is the collection of pages it links to (line 1). The program also receives the number of iterations (n) that will be made as input. The program starts by creating the initial *ranks* dataset, in which each page (key) of the links dataset receives an initial ranking of 1.0 (line 2). The iterative part is defined between lines 3 and 12, where the iterations are controlled through a *for* statement executed from 1 to n. We abstract the block inside the *for* statement (lines 4 to 11) as the step function that receives the

ranks dataset as input and produces, at the end of the iteration, a new version of the *ranks* dataset with the updated ranking of each page as output.

The step function starts with a join between *links* and *ranks* (line 4). Note that the dataset *links* is not changed in the step function, but is only used in the *join* with *ranks*. We have a dataset where each element is a tuple containing the page address, its ranking, and the list of pages it is linked to. Then we take only the part that contains the ranking and the list of links to other pages (line 5). After that, we calculate the contribution that each page sends to the ranking of the other pages it links to (lines 6 to 9). This contribution is equal to $\frac{r}{s}$, where r is the page ranking and s is the number of neighbors (pages it links to). Next, we aggregate the contributions with the *aggregateByKey* transformation (line 10). Since the *contribs* dataset has key/value pairs where the key is a page and the value is the contribution it receives from another page, the result of the aggregation is a key/value dataset with the page (key) and the sum of all contributions it received (value). At the end of the step function (line 11), we update the *ranks* dataset so that the ranking of each page is equal to

```
1  def pageRank(links: RDD[(String, Iterable[String])], n: Int) = {
2      var ranks = links.map( link => (link._1, 1.0) )
3      for(i <- 1 to n){
4          val linksRanks = links.join(ranks)
5          val values = linksRanks.map( lr => lr._2 )
6          val contribs = values.flatMap { v =>
7              val size = v._1.size
8              v._1.map( url => (url, v._2 / size) )
9          }
10         val aggregContribs = contribs.reduceByKey( (a, b) => a + b )
11         ranks = aggregContribs.map( rank => (rank._1, 0.15 + 0.85 * rank._2) )
12     }
13     ranks
14 }
```

Figure 6: *PageRank* implementation in Spark (based on [2]).

$0.15 + 0.85 \times c$, where c is the sum of all contributions received by the page. The program ends by returning the final *ranks* dataset with the ranking of each page calculated after n iterations (line 13).

To model the data flow of this program, we need to identify the datasets and transformations defined outside and inside the step function (iteration). Outside the step function, we have the input dataset $links$ of type $Bag[String \times Bag[String]]$ and the initial ranks dataset of type $Bag[String \times Double]$, defined before the iteration, which we call $ranks_0$. We also have the map transformation (t_1) that is applied to generate $ranks_0$.

Each iteration updates the datasets used within the step function (st). Within the step function, we denote the datasets and transformations with an i subscript, representing that a new version of the dataset or transformation will be created at each iteration.

In this example, we have the datasets $ranks_i : Bag[String \times Double]$ (note that is the same type of $ranks_0$), $linksRanks_i : Bag[String \times (Bag[String] \times Double)]$, $values_i : Bag[Bag[String] \times Double]$, $contribs_i : Bag[String \times Double]$ and $aggregContribs_i : Bag[String \times Double]$. To fit the iteration subnet pattern, we need to distinguish between the $ranks$ variable before and after the iteration. That gets us then the following set of places for our Petri Net:

$$D = \{links, ranks_0, ranks_{i-1}, linksRanks_i, values_i, contribs_i, \\ aggregContribs_i, ranks_i, ranks_n\}$$

We also have the *innerJoin* transformation t_{2i}, the *map* t_{3i}, the *flatMap* t_{4i}, the *reduceByKey* t_{5i} and the *map* t_{6i}.

The data flow graph representing the *PageRank* program is shown in Figure 7. In it, we can see the data sets and transformations defined and the edges that connect them. We can also see the t_{start}, $t_{iterative}$ and t_{end} transitions that represent the beginning, continuation and end of the iterations. In terms

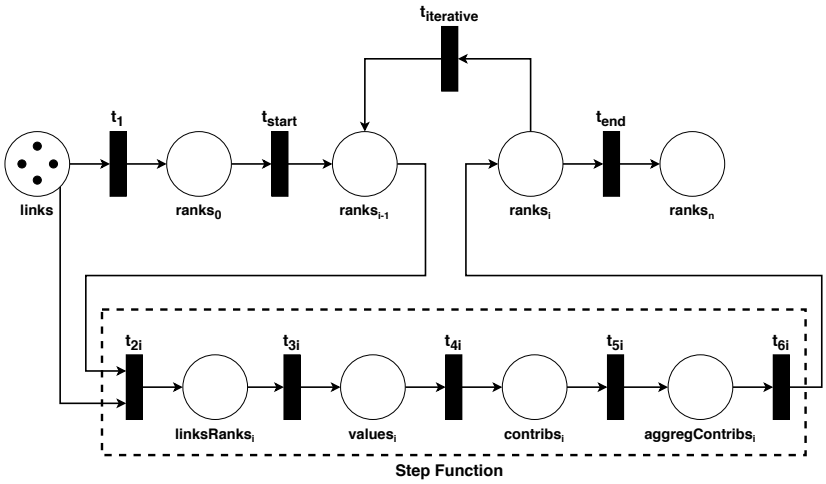


Figure 7: Data flow of the PageRank program.

of Monoid Algebra, the program is defined as follows:

$$t_1 = map(\texttt{link => } \ldots, links)$$
$$t_{2i}(ranks_i) = innerJoin(links, ranks_i)$$
$$t_{3i}(linksRanks_i) = map(\texttt{lr => } \ldots, linksRanks_i)$$
$$t_{4i}(values_i) = flatMap(\texttt{v => } \ldots, values_i)$$
$$t_{5i}(contribs_i) = reduceByKey(\texttt{(a,b) => } \ldots, contribs_i)$$
$$t_{6i}(aggregContribs_i) = map(\texttt{rank => } \ldots, aggregContribs_i)$$

where i ranges from 1 to n.

The iteration that begins at t_{start} and ends at t_{end} is defined as:

$$t_{iterate} = iterate(st, n, ranks_0), \text{ where}$$
$$st = t_{6i} \circ t_{5i} \circ t_{4i} \circ t_{3i} \circ t_{2i}$$

As we mentioned earlier, the data flow systems that we are modeling define their programs as DAGs, so the representation of iterative programs takes place through the repetition of operations n times where n is the number of iterations, having no cycles in the graph as we did in our model. Our iteration

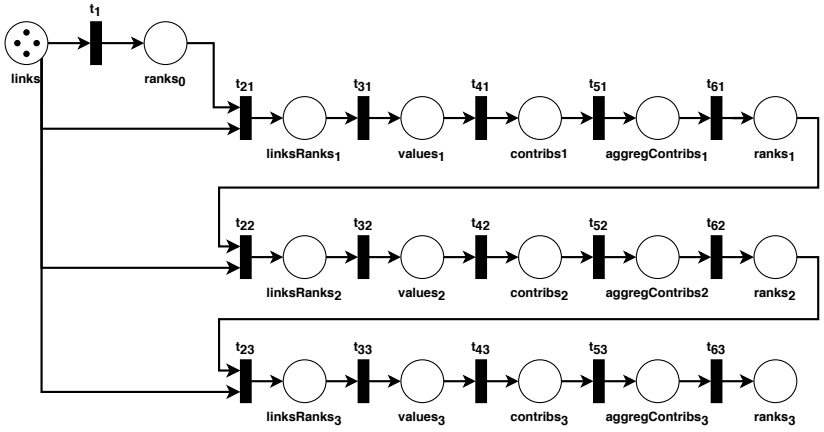


Figure 8: Expanded data flow (without cycle) of the *PageRank* program for 3 iterations.

representation is an abstraction for the expansion of the graph, but in fact, our model allows us to represent the DAG that would be created in the data flow systems. As an example, we can see the expanded representation of the data flow of the *PageRanks* program for 3 iterations in Figure 8. In it we can see that the iterations control transitions (t_{start}, $t_{iterative}$ and t_{end}) were removed and that the program is represented as a DAG.

The principles of the example given above apply to any structured iteration defined at the Petri Net level. It is easy to see that the transformation from an iterative Petri Net into an acyclic one, for a given n, can be defined using graph transformation/rewriting.

4. Comparing Parallel Big Data Processing Frameworks

The model proposed in this paper uses as reference the characteristics of the programming strategies implemented by most prominent data flow-based Big Data processing frameworks like *Apache Spark* [9], *Dryad/DryadLINQ* [5, 6], *Apache Flink* [7] and *Apache Beam* [8] that were presented in Section 2. These frameworks use a similar DAG-based model to represent the data processing programs workflow despite the adoption of different strategies for executing programs, optimizing, and processing data. DAGs are composed of data pro-

cessing operations that are connected through communication channels. The channels are places for intermediate data storage among operations.

Our model captures the Petri Net data flow component of DAGs (data processing operations and communication channels). The nodes for datasets represent the communication channels among operations. They represent at a high level, the abstractions used by Big Data processing frameworks for modeling distributed datasets, such as *RDD* in *Apache Spark* (see Figure 2 and Figure 3), *PCollection* in *Apache Beam*, *DataSet* in *Apache Flink* and *DryadTable* in *DryadLINQ*. Transformation nodes represent the processing operations that receive datasets and transmit the processing results to another dataset. The representations of the datasets and transformations in the data flow graph encompass the main abstractions of the DAGs in these systems and allow to represent and analyze a program independently of the system in which it will be executed.

The semantics of transformations and datasets are represented in the model using Monoid Algebra. The transformations included in the model were based on the main types of operations provided by the analyzed frameworks. These operations can be categorized into *mapping*, which are operations that transform values in a dataset; *filtering*, which are operations that remove values from a dataset based on some predicate; *grouping*, which are operations that group values based on a key; *aggregation*, which are operations that summarize a group of values into a single value; *Set-like* operations, which are inspired by operations in mathematical sets; *join*, which are operations based on the relational join between datasets; and *ordering*, which are operations that order the values of the dataset. This paper focuses on the abstract representation of both non-iterative and iterative Big Data processing programs.

Table 1 compares the transformations defined by the model and the operations implemented in the Big Data processing frameworks. Therefore, we grouped the transformations according to the types of processing that are done: *Mapping*, *Filtering*, *Grouping*, *Sets*, *Aggregation*, *Joins* and *Ordering*. In addition, we include *Iteration* in the table to show the support for iterative operations. We modeled the main types of operations provided by these frameworks.

Table 1: Comparing our model operations with operations in Big Data processing frameworks.

	Model	**Apache Spark**	**Apache Flink**	**Apache Beam**	**DryadLINQ**
Mapping	map, flatMap	map, flatMap	map, flatMap	ParDo, FlatMapElements, MapElements	Select, SelectMany
Filtering	filter	filter	filter	Filter	Where
Grouping	groupBy, groupByKey	groupBy, groupByKey	groupBy	GroupByKey	GroupBy
Sets	union, intersection, subtract, distinct	union, intersection, subtract, distinct	union, distinct	Flatten, Distinct	Union, Intersect, Except, Distinct
Aggregation	reduce, redubeByKey	reduce, reduceByKey, aggregateByKey	reduce, reduceGroup, aggregate	Combine	Aggregate
Joins	innerJoin, leftOuterJoin, rightOuterJoin, fullOuterJoin	join, leftOuterJoin, rightOuterJoin, fullOuterJoin	join, leftOuterJoin, rightOuterJoin, fullOuterJoin	CoGroupByKey	Join
Ordering	orderBy, orderByKey	sortBy, sortByKey	sortPartition, sortGroup		OrderBy
Iteration	iterate, iterateWithCondition	Support with external for and while loops	iterate, deltaIterate	Support with external for and while loops	Support with external for and while loops

In the table, we also indicate how the model and frameworks deal with iterative programs.

Some systems offer more specific operations that we do not define directly in our model. It is a work in progress to guarantee complete coverage of all the operations of the considered systems. However, most of the operations that are not directly represented in the model can easily be represented using the transformations provided by the model. For example, classic aggregation operations, like maximum, minimum, or the sum of the elements in a dataset.

We can easily represent these operations using the *reduce* operation of the model:

$$max(D) = reduce(\lambda(x, y).\ \textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y, D)$$
$$min(D) = reduce(\lambda(x, y).\ \textbf{if}\ x < y\ \textbf{then}\ x\ \textbf{else}\ y, D)$$
$$sum(D) = reduce(\lambda(x, y).x + y, D)$$

Ideally, Big Data processing frameworks should allow users to express data flow using simple imperative data flow statements while matching the performance of native data flow. Therefore, we believe it is necessary to propose formal models agnostic of the underlying programming models and their implementation to manipulate iterative and non-iterative data processing algorithms abstractly. The model proposed in the previous sections can be an abstraction of existing data flow-based programming models independently of their specific implementations by different frameworks. It provides abstractions of the data flow programming models that can be applied to specify parallel data processing programs independently of target systems.

An abstract representation of parallel data flow can be used to address program testing challenges, to compare Big Data processing tools when choosing one of them, as well as to help in migrating solutions from one framework to another. In particular, our model is used as a representation tool to apply mutation testing on data flow-based Big Data processing programs. In the next section, we briefly discuss how this is done in a testing tool we developed.

5. Applications of the model

The abstract and formal concepts provided by the model make it suitable for the automation of software development processes, such as those done by IDE tools. Consequently, we first applied the model to formalize the mutation operators presented in [13], where we explored the application of mutation testing in Spark programs, and in the tool TRANSMUT-SPARK[1] [25] that we devel-

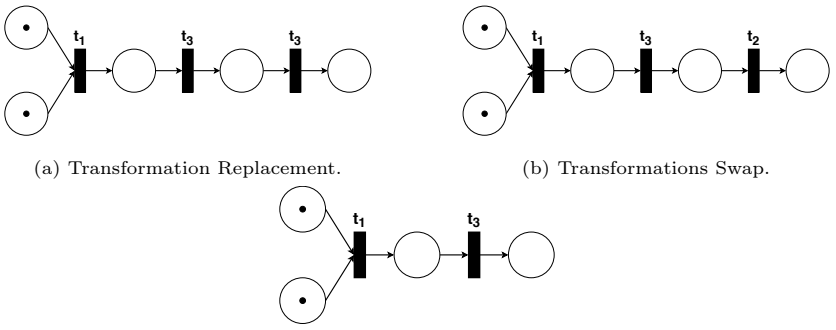
[1] TRANSMUT-SPARK is publicly available at `https://github.com/jbsneto-ppgsc-ufrn/transmut-spark`.

oped to automate this process. Mutation testing is a fault-based technique that simulates faults to design and evaluate test sets [26]. Faults are simulated by applying mutation operators, rules with modification patterns for programs (a modified program is called a *mutant*). In [13], we presented a set of mutation operators designed for Spark programs that are divided into two groups: *mutation operators for the data flow* and *mutation operators for transformations*.

These mutation operators were based on a taxonomy of faults found in Spark programs [25, 27] with the idea of mimicking them, and on common mutation operators for different languages, such as those proposed in [28]. This taxonomy was built using our experience as Spark programmers, as well as the observation of programs, published literature, and documentation about Apache Spark.

Although derived from a study of Spark faults, the definition of the operators is done at the model level, addressing each of its views (data flow and transformation semantics) so that they are also applicable to the other similar frameworks. Mutation operators for the data flow model change the DAG that defines the program. In general, we define three types of modifications in the data flow: replacing one transformation with another (both existing in the program), swapping the calling order of two transformations, and deleting the call of a transformation in the data flow. These modifications involve changes to the edges of the program. Besides, replacing a transformation with another must maintain the type consistency, i.e., the I/O datasets of both transformations must be of the same type. In Figure 9 we exemplify these mutations in the data flow that was presented in Figure 3.

Mutation operators associated with transformations model the changes done on specific transformations' types, such as aggregation transformations or set transformations. In general, we model two types of modifications: (1) replacement of the function passed as a parameter of the transformation and (2) replacement of a transformation by another of the same group. In the first type, we defined specific substitution functions for each group of transformations. For example, for a transformation of type aggregation, we define five substitution functions (f_m) to replace it. Considering the aggregation transformation



(a) Transformation Replacement.

(b) Transformations Swap.

(c) Transformation Deletion.

Figure 9: Examples of mutants created with mutation operators for data flow.

$t_1 = reduce(max(x, y), d)$, which receives as input a function that returns the greater of the two input parameters and an integer dataset, the mutation operator for aggregation transformation replacement will generate the following five mutants:

$$t_1 = reduce(f_m(x, y) = x, d)$$
$$t_1 = reduce(f_m(x, y) = y, d)$$
$$t_1 = reduce(f_m(x, y) = max(x, x), d)$$
$$t_1 = reduce(f_m(x, y) = max(y, y), d)$$
$$t_1 = reduce(f_m(x, y) = max(y, x), d)$$

In the second type of modification, we replace a transformation with others from the same group. For example, for set transformations (union, intersection, and subtract), we replace one transformation with the remaining two; besides, we replace the transformation for the identity of each of the two input datasets, and we also invert the order of the input datasets. Considering the set transformation $t_1 = subtract(d_1, d_2)$, which receives two integer datasets a input, the

set transformation replacement operator will generate the following mutants:

$$t_1 = union(d_1, d_2)$$

$$t_1 = intersection(d_1, d_2)$$

$$t_1 = identity(d_1)$$

$$t_1 = identity(d_2)$$

$$t_1 = subtract(d_2, d_1)$$

The mutation operators for the other transformations follow these two types of modifications, respecting each group's type consistency and particularities.

The tool TRANSMUT-Spark [25] uses the model as an intermediate representation. The tool reads a Spark program and translates it into an implementation of the model, so the mutation operators are applied to the model. TRANSMUT-Spark actual implementation is based on a previous version of the model ([1]) which only deals with non-iterative programs. Our next step in the evolution of the tool is to include mutation operators which can alter the iterative characteristics of the programs. In the following, we describe some potential mutation operators which may reflect common programming mistakes introduced by the iterative nature of the model. Their proposal is inspired by very well-known loop-related mistakes in any programming language, such as a wrong number of iterations or a stop predicate that is inverted, and on specifics of the proposed model.

5.1. Mutation Operators for Iterative Programs

Loops are a powerful tool, but also a source of many programming mistakes. The software testing community has been dedicating effort on how to best exercise iterative programs in order to gain confidence in their correction and quality for a long time ([29, 30, 31, 32]). Testing of iterative operations is traditionally done by applying graph coverage criteria [26] in which different execution paths can be tested in order to exercise loops and branches. Some recurring ideas are to exercise the program with test cases where the loop is not executed and exploring the upper bounds of the number of executions, when possible. It is

common to have some mandatory action occurring inside the loop that is not carried out when the loop is not executed. It is also very common to have a wrong stop condition that leads to a missing or to an exceeding execution of the loop. Other common mistakes are the use of a condition that reflects when the loop must be continued instead of a stop condition and vice versa, and, more generally, a wrong conditional control expression can lead to different undesired behaviors of the program. Test design must also look for these mistakes.

The mutation approach to dealing with specific kinds of programming issues is to simulate them with the mutation operators. With the extension of the model to represent iterative programs presented in Section 3.3, we can now formalize specific mutation operators for iterative operations.

Based on that, mutation operators that model faults in iterative transformations by modifying the number of times or conditions passed to the operation may be proposed. Those operators apply two strategies: change the number of iterations n passed to the transformation and change the conditional function p for the iterative transformation with condition. The mutation operators are formalized below:

Number of Iterations Replacement (NIR). Given an iterative transformation that takes a number of iterations n as parameter (*iterate* and *iterateWithCondition*), the NIR operator replaces n by: 0, 1, c, $n-1$ and $n+1$, where c is a constant number that can be chosen by the test engineer.

Negate Iteration Condition (NIC). Given an iterative transformation that takes a predicate function p as a parameter (*iterateWithCondition*), where p defines the condition to continue the iterations, the NIC operator replaces p with a predicate function p_m that negates the result of p ($p_m(x) = \neg p(x)$).

Remove Iteration Condition (RIC). Given an iterative transformation that takes a predicate function p as a parameter (*iterateWithCondition*), where p defines the condition to continue the iterations, the NIC operator replaces p with a predicate function p_m that is a tautology ($p_m(x) = true$), which in practice

removes the condition and leaves the iterative transformation based only on the maximum number of iterations n that is passed as parameter to the transformation.

With the NIR operator, we can simulate common mistakes related to the definition of iterations, which are the cases where the iterative operation is not executed or executed only once, as well as the mistake of executing the iteration one time less or more than it should. This operator is similar to the operators proposed in [32] that simulate faults in *for* loops by modifying the loop's initialization and sentinels. With the NIC and RIC operators, we can simulate faults in condition-based iterations, in which mistakes can be made in defining the stopping condition of the loop. In these operators, mutations are made only in the control of iterations, *i.e.*, in n or p. Finally, the transformations performed iteratively are encapsulated in the step function (st). For these transformations, it is possible to apply the other mutation operators that were proposed in [13] and there is no need for additional mutation operators.

To exemplify the mutation operators for iterative transformations, let us consider the *PageRank* program presented in Section 3.3. For the sake of simplicity, we consider only the iterative transformation ($t_{iterate} = iterate(st, n, ranks_0)$) since for the other transformations applied in st we can apply the previously defined [13] mutation operators, as said above. Applying the NIR operator in this transformation, we generate the following five mutants:

$$t_{iterate} = iterate(st, 0, ranks_0)$$
$$t_{iterate} = iterate(st, 1, ranks_0)$$
$$t_{iterate} = iterate(st, c, ranks_0)$$
$$t_{iterate} = iterate(st, n - 1, ranks_0)$$
$$t_{iterate} = iterate(st, n + 1, ranks_0)$$

6. Related Work

Data flow processing that defines a pipeline of operations or tasks applied on datasets has been traditionally formalized using (colored) Petri Nets [33], which

are well adapted to model the organization (flow) of the processing tasks that receive and produce data. A number of proposals use Petri Nets to model control flow and use other formal tools for modeling the operations applied on data. For example, [34, 35] uses nested relational calculus for formalizing operations applied to non first normal form-compliant data. Our approach is similar, in the sense that we model the workflow using Petri Nets, but use Monoid Algebra to model operations on datasets, thus allowing for the (implicit) treatment of distribution and parallelism inside operations.

Let us now describe some initiatives to formalize data processing parallel programming models. Our presentation focuses on the tools and strategies used to formalize control/data flows and data processing operations.

The authors in [10] formalize MapReduce using *CSP* [36]. The system is formalized with respect to four components: *Master*, *Mapper*, *Reducer* and *FS* (file system). The Master manages the execution process and the interaction between the other components. The Mapper and Reducer components represent, respectively, the processes for executing the map and reduce operations. Finally, FS represents the file system that stores the data processed in the program. These components implement the data processing pipeline implemented by these systems, loading data from a file system, executing a map function (by several mappers), shuffling and sorting, and executing a function reduce by reducers. The model allows the analysis of properties and interaction between these processes, as implemented by MapReduce systems.

In [12] MapReduce applications are formalized within *Coq*, an interactive proof assistant. As in [10], the authors also formalized the components and execution process of MapReduce systems. The user-defined functions of the map and reduce operations are also formalized with Coq. Then, these formal definitions are used to prove the correctness of MapReduce programs. This approach is different from the work presented in [10] that formalizes only the MapReduce system, without specific consideration to the user-defined functions implemented by each component. Although related to our study, both works in [10, 12] are restricted to the description of a less general model than ours.

More recent work has proposed formal models for data flow programming models, particularly associated with Spark. The work in [11] introduces *PureSpark*, a functional and executable specification for Apache Spark written in Haskell. The purpose of *PureSpark* is to specify parallel aggregation operations of Spark. Based on this specification, necessary and sufficient conditions are extracted to verify whether the outputs of aggregations in a Spark program are deterministic. The work [37] presents a formal model for Spark applications based on temporal logic. The model considers the DAG that forms the program, information about the execution environment, such as the number of CPU cores available, the number of program tasks, and the average execution time of the tasks. Then, the model is used to check time constraints and make predictions about the program's execution time. Both works ([11] and [37]) aim to evaluate Spark programs for specific properties. The abstraction level of our model is higher than that of the work of Marconi *et al.*, so our model is not suited, as it is, to evaluate cluster behavior. But it can be used and was used in TRANSMUT-SPARK, in enforcing the development of test cases that verify the deterministic behavior of aggregations through mutation, the goal of the work by Chen *et al.*.

The research community has also paid attention to the problem of addressing iterative programs in data flow-based programming frameworks and has proposed several solutions [38, 39, 40]. For example, Emma [38] can translate imperative control flow to Flink's native iterations. The translation is limited to cases when there is only a single while-loop containing a sequential body, without any other control flow statement. This approach is unsuitable for data analytics tasks, such as hyper-parameter optimization, simulated annealing, and strongly connected components, which require a more complex control flow. AutoGraph [39] and Janus [40] compile imperative control flow to TensorFlow's native iterations [41]. However, they do not support general data analytics other than machine learning. Mitos [42] allows users to write imperative control flow constructs, such as regular while-loops and if statements.

7. Conclusions and Future Work

This paper presents a model for data flow processing programs. Our model combines two formal mathematical tools: Monoid Algebra and Petri Nets. Monoid Algebra is an abstract way to specify operations over partitioned datasets. Petri nets are widely used to specify parallel computation. Our proposal combines these two models by building two-level specifications. The lower level uses Monoid Algebra to specify individual transformations (*i.e.*, operations whose arguments and results are datasets). The upper level defines the program using a Petri Net, where places represent datasets and transitions represent operations over that data.

The paper is an extended version of [1]. The main technical difference to that paper is the addition of iterations to the model and the proposed use of our model to specify the operations available in several existing Big Data processing frameworks. In this sense, the paper specifies data processing operations (i.e., transformations) provided as built-in operations in Apache Spark, DryadLINQ, Apache Beam, and Apache Flink.

In the proposed model, iterations are represented by a loop on the Petri Net that defines the program. Loops are statically *unfolded* to build a Petri Net without cycles, to have a DAG representing the program. This technique is convenient and realistic. It is convenient since it preserves the distributive and associative properties of the operations over datasets, as well as the standard representation of programs as DAGs. It is realistic since it provides a general model of strategies used by most prominent Big Data processing frameworks to process loops [2, 7, 8, 6].

Beyond the interest of providing a formal model for data flow-based programs, our proposal can be used as a comparison tool of target systems or to define program testing pipelines. We also showed how operations could be combined into data flows to implement mutation operations in mutation testing approaches. The model was already used as an intermediary representation to specify mutation operators that were then implemented in TRANSMUT-

SPARK, a software engineering tool for mutation testing of Spark programs [13]. A natural extension to this work would be to instantiate the tool for other systems of the data flow family (*DryadLINQ*, *Apache Beam*, *Apache Flink*). This instantiation can be done by adapting TRANSMUT-SPARK's front and back ends so that a program originally written in any of these systems can be tested using the mutation testing approach proposed in [13]. This line of work, where the model is used as the internal format, is suited for the more practical users, not willing to see the formalism behind their tools. However, when exploring the similarities of different frameworks, our model may be used as a platform-agnostic form of formally specifying and analyzing the properties of a program before its implementation. In addition, the formalization of iterative transformations allowed us to propose specific mutation operators for iterative operations. Thus, we can now extend TRANSMUT-SPARK to support the testing of iterative programs. We plan to implement these features in future work.

Furthermore, we intend to study the extension of our model to use Colored Petri Nets (CPN) and CPN Tools [43] to specify the types for transformations over datasets explicitly and to manipulate, analyze and animate the specifications. This extension may be helpful to detect design problems at an early stage. Also, we plan to work on the use of specifications for code generation to target data flow systems similar to Apache Spark. A simple form of this code generation was implemented to generate test programs in TRANSMUT-SPARK back-end.

References

[1] J. B. de Souza Neto, A. M. Moreira, G. Vargas-Solar, M. A. Musicante, Modeling big data processing programs, in: G. Carvalho, V. Stolz (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2020, pp. 101–118.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 2–2.
URL http://dl.acm.org/citation.cfm?id=2228298.2228301

[3] F. Bajaber, R. Elshawi, O. Batarfi, A. Altalhi, A. Barnawi, S. Sakr, Big Data 2.0 Processing Systems: Taxonomy and Open Challenges, Journal of Grid Computing 14 (3) (2016) 379–405. doi:10.1007/s10723-016-9371-1.

[4] Hadoop, Apache Hadoop Documentation (2019).
URL https://hadoop.apache.org/docs/r2.7.3/

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed Data-parallel Programs from Sequential Building Blocks, in: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, ACM, New York, NY, USA, 2007, pp. 59–72. doi:10.1145/1272996.1273005.
URL http://doi.acm.org/10.1145/1272996.1273005

[6] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey, DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 1–14.
URL http://dl.acm.org/citation.cfm?id=1855741.1855742

[7] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, K. Tzoumas, Apache Flink: Stream and Batch Processing in a Single Engine, IEEE Data Engineering Bulletin 38 (4) (2015) 28–38.

[8] A. Beam, Apache Beam: An advanced unified programming model (2016).
URL https://beam.apache.org/

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 10–10.
URL http://dl.acm.org/citation.cfm?id=1863103.1863113

[10] F. Yang, W. Su, H. Zhu, Q. Li, Formalizing MapReduce with CSP, in: 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, 2010, pp. 358–367.

[11] Y.-F. Chen, C.-D. Hong, O. Lengál, S.-C. Mu, N. Sinha, B.-Y. Wang, An Executable Sequential Specification for Spark Aggregation, in: A. El Abbadi, B. Garbinato (Eds.), Networked Systems, Springer International Publishing, Cham, 2017, pp. 421–438.

[12] K. Ono, Y. Hirai, Y. Tanabe, N. Noda, M. Hagiya, Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications, in: G. Barthe, A. Pardo, G. Schneider (Eds.), Software Engineering and Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 350–365.

[13] J. B. Souza Neto, A. Martins Moreira, G. Vargas-Solar, M. A. Musicante, Mutation Operators for Large Scale Data Processing Programs in Spark, in: S. Dustdar, E. Yu, C. Salinesi, D. Rieu, V. Pant (Eds.), Advanced Information Systems Engineering, Springer International Publishing, Cham, 2020, pp. 482–497.

[14] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580. doi:10.1109/5.24143.

[15] L. Fegaras, An algebra for distributed Big Data analytics, Journal of Functional Programming 27 (2017) e27. doi:10.1017/S0956796817000193.

[16] L. Fegaras, Compile-Time Query Optimization for Big Data Analytics,

Open Journal of Big Data (OJBD) 5 (1) (2019) 35–61.
URL https://www.ronpub.com/ojbd/OJBD_2019v5i1n02_Fegaras.html

[17] K. Kennedy, J. R. Allen, Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc., 2001.

[18] R. Ivanovs, Concurrency Glossary (2018).
URL https://slikts.github.io/concurrency-glossary/

[19] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004, pp. 137–150.

[20] C. A. Petri, Kommunikation mit automaten, Ph.D. thesis, Universität Hamburg, (In German) (1962).

[21] K. M. Kavi, B. P. Buckles, N. Bhat, A Formal Definition of Data Flow Graph Models, IEEE Transactions on Computers C-35 (11) (1986) 940–948. doi:10.1109/TC.1986.1676696.

[22] S. Chlyah, N. Gesbert, P. Genevès, N. Layaïda, An Algebra with a Fixpoint Operator for Distributed Data Collections (Mar. 2019).
URL https://hal.inria.fr/hal-02066649

[23] T. Hastie, R. Tibshirani, J. Friedman, The elements of statistical learning: data mining, inference, and prediction, Springer Science & Business Media, 2009.

[24] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer networks and ISDN systems 30 (1-7) (1998) 107–117.

[25] J. B. Souza Neto, Transformation mutation for spark programs testing, Ph.D. thesis, Federal University of Rio Grande do Norte (UFRN), Natal/RN, Brazil, (In Portuguese) (2020).

[26] P. Ammann, J. Offutt, Introduction to Software Testing, second edition Edition, Cambridge University Press, New York, NY, 2017.

[27] J. B. de Souza Neto, A. M. Moreira, G. Vargas-Solar, M. A. Musicante, Transmut-spark: Transformation mutation for apache spark (https://arxiv.org/abs/2108.02589. 2021). arXiv:2108.02589.
URL https://arxiv.org/abs/2108.02589

[28] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, E. Spafford, Design of Mutant Operators for the C Programming Language, techreport SERC-TR-41-P, Purdue University, West Lafayette, Indiana (March 1989).

[29] T. Chow, Testing software design modeled by finite-state machines, IEEE Transactions on Software Engineering SE-4 (3) (1978) 178–187.

[30] L. J. White, B. Wiszniewski, Path testing of computer programs with loops using a tool for simple loop patterns, Software: Practice and Experience 21 (10) (1991) 1075–1102.

[31] N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage, in: 2009 International Conference on Software Testing, Verification, and Validation Workshops, 2009, pp. 220–229.

[32] M. Raunak, C. Murphy, B. O'Haver, An Empirical Study of Off-by-one Loop Mutation, Technical report, University of Pennsylvania (2015).

[33] E. Lee, D. Messerschmitt, Pipeline interleaved programmable DSP's: Synchronous data flow programming, IEEE Transactions on acoustics, speech, and signal processing 35 (9) (1987) 1334–1345.

[34] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, J. Van den Bussche, Petri net + nested relational calculus = dataflow, in: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer, 2005, pp. 220–237.

[35] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, J. Van den Bussche, DFL: A dataflow language based on Petri nets and nested relational calculus, Information Systems 33 (3) (2008) 261–284.

[36] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, A Theory of Communicating Sequential Processes, J. ACM 31 (3) (1984) 560–599. doi:10.1145/828.833.

[37] F. Marconi, G. Quattrocchi, L. Baresi, M. M. Bersani, M. Rossi, On the Timed Analysis of Big-Data Applications, in: A. Dutle, C. Muñoz, A. Narkawicz (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2018, pp. 315–332.

[38] A. Alexandrov, G. Krastev, V. Markl, Representations and optimizations for embedded parallel dataflow languages, ACM Transactions on Database Systems (TODS) 44 (1) (2019) 1–44.

[39] D. Moldovan, J. M. Decker, F. Wang, A. A. Johnson, B. K. Lee, Z. Nado, D. Sculley, T. Rompf, A. B. Wiltschko, Autograph: Imperative-style coding with graph-based performance, arXiv preprint arXiv:1810.08061 (2018).

[40] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, B.-G. Chun, {JANUS}: Fast and flexible deep learning via symbolic graph execution of imperative programs, in: 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), 2019, pp. 453–468.

[41] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, et al., Dynamic control flow in large-scale machine learning, in: Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 1–15.

[42] G. E. Gévay, T. Rabl, S. Breß, L. Madai-Tahy, J.-A. Quiané-Ruiz, V. Markl, Efficient control flow in dataflow systems: When ease-of-use meets high performance, in: IEEE 37th International Conference on Data Engineering (ICDE), 2021.

[43] K. Jensen, L. M. Kristensen, L. Wells, Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (3) (2007) 213–254. doi:10.1007/s10009-007-0038-x.