Modeling Big Data Processing Programs

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

Modeling Big Data Processing Programs*

João Batista de Souza Neto[1][0000−0002−8142−2525], Anamaria Martins Moreira[2][0000−0002−7707−8469], Genoveva Vargas-Solar[3][0000−0001−9545−1821], and Martin A. Musicante[1][0000−0001−5589−3895]

[1] Department of Informatics and Applied Mathematics (DIMAp)
Federal University of Rio Grande do Norte, Natal, Brazil.
jbsneto@ppgsc.ufrn.br, mam@dimap.ufrn.br
[2] Computer Science Department (DCC)
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.
anamaria@dcc.ufrj.br
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG-LAFMIA, Grenoble, France.
genoveva.vargas@imag.fr

Abstract. We propose a new model for data processing programs. Our model generalizes the data flow programming style implemented by systems such as Apache Spark, DryadLINQ, Apache Beam and Apache Flink. The model uses directed acyclic graphs (DAGs) to represent the main aspects of data flow-based systems, namely Operations over data (filtering, aggregation, join) and Program execution defined by data dependence between operations. We use *Monoid Algebra* to model operations over distributed, partitioned datasets and *Petri Nets* to represent the data/control flow. This allows the specification of a data processing program to be agnostic of the target Big Data processing system. Our model has been used to design mutation test operators for big data processing programs. These operators have been implemented by the testing environment TRANSMUT-Spark.

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

1 Introduction

The access to datasets with consequent volume, variety and velocity scales, namely Big Data, calls for alternative parallel programming models. These models fit well to the implementation of data analytic tasks that can exploit the potential of those datasets. These programming models have been implemented by large scale data processing systems that provide execution infrastructures for large scale computing and memory resources.

Large scale data processing systems can be classified according to their purpose into general-purpose, SQL, graph processing, and stream processing systems [2]. These systems adopt different approaches to represent and process

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

data. Examples of general-purpose systems are Apache Hadoop [9], Dryad [12], Apache Flink [5], Apache Beam [3] and Apache Spark [23]. According to the programming model adopted for processing data, general-purpose systems can be based on control flow (like Apache Hadoop) or data flow (like Apache Spark). In these systems, a program devoted to processing distributed datasets is written by composing individual processing blocks. These processing blocks are defined as operations that perform *transformations* on the data. The interaction between these blocks defines the *data flow* that specifies the order in which operations are performed. Datasets exchanged among the blocks are modeled by data structures such as key-value tuples or tables. The system infrastructure manages the parallel and distributed processing of datasets transparently. This allows developers to avoid having to deal with low-level details inherent to the distributed and parallel environments.

Yet, according to the dataset properties (velocity, volume), performance expectations and computing infrastructure characteristics (cluster, cloud, HPC nodes), it is often an important programmers' decision to choose a well-adapted target system to be used for running data processing programs. Indeed, each hardware/software facility has its particularities concerning the infrastructure and optimizations made to run the program in a parallel and distributed way. This means that the systems can have different performance scores depending on their context or available resources. The choice of the specific resources depends on non-functional requirements of the project, available infrastructure and even preferences of the team that develops and execute the program. In order to tackle these subjective criteria, we believe that data processing programs design must be agnostic of the target execution platform.

This paper proposes a **model for data processing programs** that abstracts the main aspects of data flow-based data processing systems: *(i)* operations applied on data (e.g., filtering, aggregation, join); *(ii)* representation of programs execution through directed acyclic graphs (DAGs) where vertices represent operations on datasets and edges the I/O of data. According to the proposed model, a program is defined as a DAG composed of successive transformations (i.e., operations) on the dataset that is being processed.

Our model represents the DAG of the data flow and transformations (i.e., operations) separately. We use *Petri Nets* [17] to represent the data flow, and *Monoid Algebra* [7], an algebra for processing distributed data, to model transformations. This allows the same program to be implemented independently of target Big Data processing systems, requiring adjustments about the programming language and API when deployed on a target system (*Apache Spark*, *DryadLINQ*, *Apache Beam* and *Apache Flink*.).

The originality of our model is to provide a formal and infrastructure agnostic specification of data processing programs implemented according to data flow-based programming models. For the time being works addressing Big Data processing, programming have merely concentrated efforts on technical and engineering challenging aspects. Yet, few approaches have addressed on formal specifications that can reason about their execution abstractly. This can be important

for comparing infrastructures, for pipelines for testing parallel data processing programs, and eventually verifying programs properties (like correctness, completeness, concurrent access to data). This paper shows how we used our formal specification for comparing data processing systems and developing a testing method and an associated tool.

The remainder of the paper is organised as follows. Section 2 introduces the background concepts of the model, namely, Petri Nets and Monoids Algebra. Section 3 introduces the model for formally expressing big data processing programs. Section 4 describes experiments we conducted where we applied the model. Section 5 introduces related work addressing approaches that generalise control and data flow parallel programming models. Section 6 concludes the paper and discusses future work.

2 Background

This section briefly presents Petri Nets and Monoid Algebra, upon which our model is built. For a more detailed presentation, the reader can refer to to [17, 7].

Petri Nets [19] is a formal tool that allows to model and analyze the behavior of distributed, concurrent, asynchronous, and/or non-deterministic systems [17]. 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 three components it is possible to represent the different states of a system (places), the actions that take the system from one state to another (transitions) and how the change of state is made due to actions (edges). This 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 state (place) to another.

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:

$$
\begin{aligned}
P = \{p_1, p_2, \ldots, p_m\} &\quad \text{is a finite set of } \textit{places},\\
T = \{t_1, t_2, \ldots, t_n\} &\quad \text{is a finite set of } \textit{transitions},\\
F \subseteq (P \times T) \cup (T \times P) &\quad \text{is a finite set of } \textit{edges},\\
W : F \to \{1, 2, 3, \ldots\} &\quad \text{is a set of weights associated to edges},\\
M_0 : P \to \{0, 1, 2, 3, \ldots\} &\quad \text{is a function defining the initial marking of a net},
\end{aligned}
$$

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'.

Monoid Algebra was proposed in [7] as an algebraic formalism for data-centric distributed computing operations based on monoids and monoid homomorphisms. A monoid is a triad $(S, \oplus, e_{\oplus})$ formed by a set S, an associative operation $\oplus$ in S and a neutral element $e_{\oplus}$ (consider that $\oplus$ identifies the monoid). A monoid homomorphism is a function H over two monoids, from $\otimes$ to $\oplus$, which respects:

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

Monoid algebra uses the concepts of monoid and monoid homomorphism to define operations on distributed datasets, which are represented as monoid collections. One type of monoid collection is the *bag*, an unordered data collection of type α (denoted as $Bag[\alpha]$) that has the unit injection function $\mathbb{U}_{\uplus}$, which generates the bag $\{\!\{x\}\!\}$ from the unitary 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 type of monoid collection is the list, which can be considered an ordered bag ($List[\alpha]$ denotes a list of type α), with $\mathbb{U}_{++}$ as the unit injection function, $++$ as the associative operation and $[\,]$ as the neutral element.

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 **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 of the results of applying f to the elements of X. This operation captures the essence of parallel processing since f can be executed in parallel on 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 $Bag[\kappa \times \alpha]$ through the first component (key) of type κ and results in a collection $Bag[\kappa \times Bag[\alpha]]$, where the second component is a collection containing all elements of type α to which 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.

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 $\alpha \rightarrow \alpha \rightarrow \alpha$. 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 $\leq$.

In addition, monoid algebra also supports the use of lambda expressions ($\lambda x.e$), **if-then-else**, and the union operation on bags ($\uplus$).

In our work, we combine Petri Nets with Monoid Algebra to build abstract versions of the primitives present distributed in Big Data processing applications. The main goal of our approach is to have a common representation of data-centric programs. This representation may be used to compare different frameworks, as well as to translate or optimize programs.

3 Modeling Big Data Processing Programs

In this section we present a formal model for big data processing. Our model is organized in two levels: *data flow*, and *transformations*. Data flow in our model is defined by means of Directed Acyclic Graphs (DAG), while transformations are modeled as monoid homomorphisms on datasets.

3.1 Data Flow

To define the DAG that represents the data flow of a data processing program, we rely on the data flow graph model presented in [14], which was formalized using Petri Nets [17].

A program P is defined as a bipartite directed graph where places stand for the distributed datasets (D) of the program, and transitions stand for its transformations (T). Datasets and transformations are connected by edges (E):

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

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

```
1 def unionLogsProblem(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 }
```

Fig. 1: Sample log union program in Spark.

In this program we can identify five RDDs, that will be referred to by 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`.

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 the transformations in T are $t_1 = \texttt{union}(d_1, d_2)$, $t_2 = \texttt{distinct}(d_3)$, and $t_3 = \texttt{filter}(\texttt{(line: String) => !(line.startsWith(“host”) \&\& line.contains(“bytes”))}, d_4)$.

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

Datasets and transformations are connected through edges. An edge may either be a pair in $D \times T$, representing an input dataset of a transformation, or a pair in $T \times D$, representing an 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, the DAG representing the Spark program in Figure 1 can be seen in Figure 2. The distributed datasets in D are represented as circle nodes and the transformations in T are represented as thick bar nodes of the graph. 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).

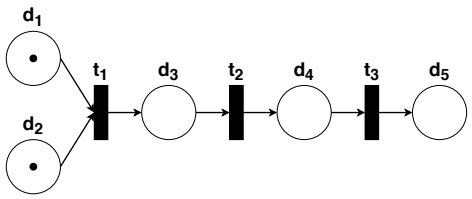


Fig. 2: Data flow representation of the program in Figure 1.

3.2 Data Sets and Transformations

The data flow model presented above represents the datasets and transformations of a program P, as well as the order in which these transformations are processed when P is executed. To define the contents of datasets in D and the semantics of transformations in T, we make use of *Monoid Algebra* [7, 8]. Datasets are represented through monoid collections and transformations are defined as operations supported by monoid algebra. These representations are detailed in the following.

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 [8], capturing the essence of the concepts of *RDD* in Apache Spark, *PCollection* in Apache Beam, *DataSet* in Apache Flink and *DryadTable* in DryadLINQ.

We define most of the transformations of our model in terms of bags, considering lists only in the sorting transformations, 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. These two kind of collections are treated by monoid homomorphisms in a unified way [8]. In this way, we will not distinguish distributed and local collections when defining our transformations.

Transformations In our model, transformations on datasets taking one or two datasets as input and producing one dataset as output. Transformations may also receive other types of parameters such as functions, which represent data processing operations defined by the developer, as well as literals such as boolean constants. A transformation t in the transformation set T of a program P is characterized by the operation it carries out, the types of its input and output datasets, 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, following the types of operations that were observed in the systems of processing of large volumes of data studied.

Mapping Transformations Mapping transformations transform values of an input dataset into values of an output dataset through the application of 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. The *map* transformation accepts any $f : \alpha \to Bag[\beta]$ and generates an output dataset of type $Bag[\beta]$, while the *flatMap* transformation requires f to directly produce $Bag[\beta]$ elements.

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

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

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:

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

Filter Transformation. This transformation uses a boolean function to define whether or not 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:

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

For each element x of a Bag, the *filter* function 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, $filter(p, D) = \{\!\{3, 4, 5\}\!\}$.

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:

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

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

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 Transformations corresponds to binary mathematical set operations on Bags. They operate on two sets of data of the same type and result in a new set of the same type. The definition of these transformations is based on the definitions in [8].

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 bag intersection and subtract operations:

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

where the auxiliary functions *some* and *all* are defined in [20].

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.

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 set by their keys:

$$\begin{aligned} reduce &:: (\alpha \to \alpha \to \alpha) \to Bag[\alpha] \to \alpha \\ reduce(f, D) &= \mathbf{reduce}(f, D) \end{aligned}$$

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

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:

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

The definition of the other joins follows a similar logic, but conditionals are included to verify the different relationships.

The types for our other join transformations are:

$$\begin{aligned} leftOuterJoin :: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (\alpha \times Bag[\beta])] \\ rightOuterJoin :: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (Bag[\alpha] \times \beta)] \\ fullOuterJoin :: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (Bag[\alpha] \times Bag[\beta])] \end{aligned}$$

The full definition of our join operations is not included here due to lack of space. It can be found in [20].

Sorting Transformations add the notion of *order* to a bag. In practical terms, these operations receive a bag and form a list, ordered in accordance of 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 $\leq$). The types for our sorting transformations are:

$$\begin{aligned} orderBy :: boolean \rightarrow Bag[\alpha] \rightarrow List[\alpha] \\ orderByKey :: boolean \rightarrow Bag[\kappa \times \alpha] \rightarrow List[\kappa \times \alpha] \end{aligned}$$

The boolean value used as first parameter defines if the direct order $\leq$ or its inverse is used. The full definition of these operations can be found in [20].

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:

$$\begin{aligned} orderBy(false, D_1) = [1, 2, 3, 4, 5] \\ orderBy(true, D_1) = [5, 4, 3, 2, 1] \\ orderByKey(false, D_2) = [(1, a), (2, b), (3, c), (4, d), (5, e)] \\ orderByKey(true, D_2) = [(5, e), (4, d), (3, c), (2, b), (1, a)] \end{aligned}$$

This model is used as a common representation for defining mutation test operators for big data processing program. In the next section we briefly present how this is done in a test tool we are developing.

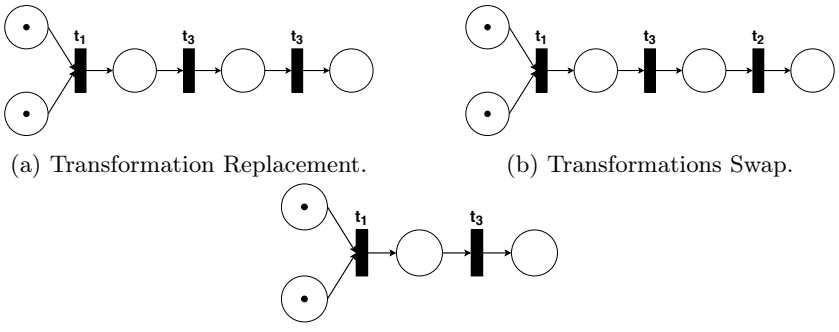
4 Applications of the model

The model proposed in the previous section can be as an abstraction of existing data flow programming models used by processing systems. It provides abstractions of the data flow programming models that can be applied to specify parallel data processing programs independently of target systems. Finally, the abstract and formal concepts provided by the model, make it quite suitable for the automation of software development process, such as it is done by IDE tools.

We first applied the model to formalize the mutation operators presented in [21], where we explored the application of mutation testing in Spark programs, and in the tool TRANSMUT-SPARK[4] that we developed to automate this process.

Mutation testing is a fault-based testing technique that relies on simulating faults to design and evaluate test sets [1]. Faults are simulated by applying mutation operators, which are rules with modification patterns for programs (a modified program is called a mutant). In [21], 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*.

Mutation operators for the data flow model changes in the DAG that defines the program. In general, we define three types of modifications in the data flow: replacement of one transformation with another (both existing in the program); swap the calling order of two transformations, and delete the call of a transformation in the data flow. These modifications involve changes to the edges of the program. In addition, the type consistency of the program must be maintained, i.e., if one transformation is replaced by another, then both must receive and return datasets of the same type. In Figure 3 we exemplify these mutations in the data flow that was presented in Figure 2.



(a) Transformation Replacement.

(b) Transformations Swap.

(c) Transformation Deletion.

Fig. 3: Examples of mutants created with mutation operators for data flow.

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

Mutation operators for transformations model changes in specific groups of transformations, such as operators for aggregation transformations or set transformations. In general, we model two types of modifications: replacement of the function passed as a parameter for the transformation; and 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 aggregation transformation we define five substitution functions (f_m) to replace an original function. Considering the aggregation transformation $t_1 = reduceByKey(max(x, y), d)$, 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 mutants:

$$
\begin{aligned}
&t_1 = reduce(f_m(x, y) = x, d)\\
&t_1 = reduce(f_m(x, y) = y, d)\\
&t_1 = reduce(f_m(x, y) = max(x, x), d)\\
&t_1 = reduce(f_m(x, y) = max(y, y), d)\\
&t_1 = reduce(f_m(x, y) = max(y, x), d)
\end{aligned}
$$

In the other 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, in addition we also 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:

$$
\begin{aligned}
&t_1 = union(d_1, d_2)\\
&t_1 = intersection(d_1, d_2)\\
&t_1 = identity(d_1)\\
&t_1 = identity(d_2)\\
&t_1 = subtract(d_2, d_1)
\end{aligned}
$$

The mutation operators for the other groups of transformations follow these two types of modifications, respecting the type consistency and the particularities of each group. The model was used in the tool TRANSMUT-Spark 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. We use the model as an intermediate representation in the tool in order to expand it in the future to apply the mutation test to programs in *Apache Flink*, *Apache Beam* and *DryadLINQ*.

5 Related Work

Data flow processing that defines a pipeline of operations or tasks applied on datasets, where tasks exchange data, has been traditionally formalised using (coloured) Petri Nets [15]. They seem well adapted for modeling the organization (flow) of the processing tasks that receive and produce data. In the case of data processing programs based on data flow models, in general, proposals use Petri Nets to model the flow and they use other formal tools for modeling the operations applied on data. For example, [10, 11] uses nested relational calculus for formalizing operations applied to non first normal form compliant data. Next we describe works that have addressed the formalization of data processing parallel programming models. The analysis focuses the kind of tools and strategies used for formalizing either control/data flows and data processing operations.

The authors in [22] formalize MapReduce using *CSP* [4]. The objective is to formalize the behaviour of a parallel system that implements the MapReduce programming model. 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, the FS represents the file system that stores the data processed in the program. These components implement the data processing pipeline implemented by these systems which loading data from an FS, executing a map function (by a number of mappers), shuffling and sorting, and then executing a reduce function by reducers. The model allows the analysis of properties and interaction between these processes implemented by MapReduce systems.

In [18] MapReduce applications are formalized with *Coq*, an interactive theorem proving systems. As in [22], 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 is different from the work presented in [22] (described above) that formalizes only the MapReduce system.

More recent work have proposed formal models for data flow programming models, particularly associated to Spark. The work in [6] 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 [16] presents a formal model for Spark applications based on temporal logic. The model takes into account the DAG that forms the program, information about the execution environment, such as the number of CPU cores available, the number of tasks of the program 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.

6 Conclusions and Future Work

This paper introduced a model of data flow processing programs that formally specifies the data flow using Petri Nets and operations performed on data using Monoid Algebra. The paper gave the specification of data processing operations (i.e., transformations) provided as built-in functions in Apache Spark, DryadLINQ, Apache Beam and Apache Flink.

The model combines existing proposals. Monoid Algebra is an abstract way to specify operations over partitioned datasets and Petri nets are widely used to specify parallel computation. Our proposal simply combines these to models to have an intermediate representation of data flow based programs so that we can safely manipulate them.

This paper showed how these operations can be combined into data flows for implementing data mutation operations in mutation testing approaches. Beyond the interest of providing a formal model for data flow based programs, the model can be a comparison tool of target systems and a way of defining programs testing pipelines.

The model was used for specifying mutation operators that were then implemented in TRANSMUT-Spark, a software engineering tool for mutation testing. 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 can be done by adapting TRANSMUT-Spark's front and back ends so that a program originally written in any of them can be tested with the mutation testing proposed in [21].

This line of work, where the model is only used as the internal format, is suited for users not willing to see the formality behind their tools. However, still exploring the similarities between these systems, the model may be used as a platform-agnostic form of formally specifying and analyzing properties of the program before implementation in one of those systems. Coloured Petri Nets and tools such as CPN Tools [13], can be powerful allies in the rigorous development process of such programs. As future work, we intend to study the mapping of our model into the input notation of CPN Tools for model simulation and analysis. As a further goal, we can also envision the use of this specification for code generation in those similar systems.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, New York, NY, second edition edn. (2017)
2. Bajaber, F., Elshawi, R., Batarfi, O., Altalhi, A., Barnawi, A., Sakr, S.: Big Data 2.0 Processing Systems: Taxonomy and Open Challenges. Journal of Grid Computing **14**(3), 379–405 (2016). https://doi.org/10.1007/s10723-016-9371-1
3. Beam, A.: Apache Beam: An advanced unified programming model (2016), `https://beam.apache.org/`
4. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. J. ACM **31**(3), 560–599 (Jun 1984). https://doi.org/10.1145/828.833

5. Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., Tzoumas, K.: Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin **38**(4), 28–38 (2015)
6. Chen, Y.F., Hong, C.D., Lengál, O., Mu, S.C., Sinha, N., Wang, B.Y.: An Executable Sequential Specification for Spark Aggregation. In: El Abbadi, A., Garbinato, B. (eds.) Networked Systems. pp. 421–438. Springer International Publishing, Cham (2017)
7. Fegaras, L.: An algebra for distributed Big Data analytics. Journal of Functional Programming **27**, e27 (2017). https://doi.org/10.1017/S0956796817000193
8. Fegaras, L.: Compile-Time Query Optimization for Big Data Analytics. Open Journal of Big Data (OJBD) **5**(1), 35–61 (2019), https://www.ronpub.com/ojbd/OJBD_2019v5i1n02_Fegaras.html
9. Hadoop: Apache Hadoop Documentation (2019), https://hadoop.apache.org/docs/r2.7.3/
10. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: Petri net + nested relational calculus = dataflow. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 220–237. Springer (2005)
11. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: DFL: A dataflow language based on Petri nets and nested relational calculus. Information Systems **33**(3), 261–284 (2008)
12. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 59–72. EuroSys '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1272996.1273005, http://doi.acm.org/10.1145/1272996.1273005
13. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer **9**(3), 213–254 (2007). https://doi.org/10.1007/s10009-007-0038-x
14. Kavi, K.M., Buckles, B.P., Bhat, N.: A Formal Definition of Data Flow Graph Models. IEEE Transactions on Computers **C-35**(11), 940–948 (Nov 1986). https://doi.org/10.1109/TC.1986.1676696
15. Lee, E., Messerschmitt, D.: Pipeline interleaved programmable DSP's: Synchronous data flow programming. IEEE Transactions on acoustics, speech, and signal processing **35**(9), 1334–1345 (1987)
16. Marconi, F., Quattrocchi, G., Baresi, L., Bersani, M.M., Rossi, M.: On the Timed Analysis of Big-Data Applications. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NASA Formal Methods. pp. 315–332. Springer International Publishing, Cham (2018)
17. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (April 1989). https://doi.org/10.1109/5.24143
18. Ono, K., Hirai, Y., Tanabe, Y., Noda, N., Hagiya, M.: Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications. In: Barthe, G., Pardo, A., Schneider, G. (eds.) Software Engineering and Formal Methods. pp. 350–365. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
19. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Universität Hamburg (1962), (In German)

20. Souza Neto, J.B.: Transformation Mutation for Spark Programs Testing. Ph.D. thesis, Federal University of Rio Grande do Norte (UFRN), Natal/RN, Brazil (2020), (In Portuguese)
21. Souza Neto, J.B., Martins Moreira, A., Vargas-Solar, G., Musicante, M.A.: Mutation Operators for Large Scale Data Processing Programs in Spark. In: Dustdar, S., Yu, E., Salinesi, C., Rieu, D., Pant, V. (eds.) Advanced Information Systems Engineering. pp. 482–497. Springer International Publishing, Cham (2020)
22. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing MapReduce with CSP. In: 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems. pp. 358–367 (2010)
23. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1863103.1863113