information

Article

DAG-Based Formal Modeling of Spark Applications with MSVL

Kaixuan Fan and Meng Wang *

Cyberspace Security and Computer College, Hebei University, Baoding 071000, China; fankaixuan@stumail.hbu.edu.cn
* Correspondence: wangmenghbu@hbu.edu.cn

Abstract: Apache Spark is a high-speed computing engine for processing massive data. With its widespread adoption, there is a growing need to analyze its correctness and temporal properties. However, there is scarce research focused on the verification of temporal properties in Spark programs. To address this gap, we employ the code-level runtime verification tool UMC4M based on the Modeling, Simulation, and Verification Language (MSVL). To this end, a Spark program S has to be translated into an MSVL program M, and the negation of the property P specified by a Propositional Projection Temporal Logic (PPTL) formula that needs to be verified is also translated to an MSVL program M1; then, a new MSVL program "M and M1" can be compiled and executed. Whether program S violates the property P is determined by the existence of an acceptable execution of "M and M1". Thus, the key issue lies in how to formalize model Spark programs using MSVL programs. We previously proposed a solution to this problem—using the MSVL functions to perform Resilient Distributed Datasets (RDD) operations and converting the Spark program into an MSVL program based on the Directed Acyclic Graph (DAG) of the Spark program. However, we only proposed this idea. Building upon this foundation, we implement the conversion from RDD operations to MSVL functions and propose, as well as implement, the rules for translating Spark programs to MSVL programs based on DAG. We confirm the feasibility of this approach and provide a viable method for verifying the temporal properties of Spark programs. Additionally, an automatic translation tool, S2M, is developed. Finally, a case study is presented to demonstrate this conversion process.

Keywords: Spark; MSVL; translation; verification; formalization

Citation: Fan, K.; Wang, M. DAG-Based Formal Modeling of Spark Applications with MSVL. *Information* **2023**, *14*, 658. https://doi.org/10.3390/info14120658

Academic Editor: Vincenzo Moscato

Received: 10 November 2023
Revised: 5 December 2023
Accepted: 8 December 2023
Published: 12 December 2023

1. Introduction

With the continuous development of the Internet, the amount of data we can collect is also expanding. How to deal with these increasing data has aroused extensive research and attention. Among them, Hadoop [1,2] and Spark [3,4] computing engines show their unique advantages in data processing.

MapReduce [5,6] is a distributed parallel computing technology. The cores of Hadoop are the MapReduce computing framework and the distributed file system HDFS [7]. In terms of processing tasks, MapReduce segments input data, constructs Map tasks for each segment, and submits them for processing. Data in the partition are transferred to the Map function as key–value pairs, and processed pairs are saved as intermediate results on disk. This characteristic results in Hadoop spending considerable time on complex data processing. Spark is a fast, universal and scalable big data analysis and computing engine based on memory. It is developed on the basis of Hadoop. One of its design goals is to avoid frequent calls to disk I/O operations. For this purpose, a basic data processing model called Resilient Distributed Datasets (RDD) is specially designed. In the process of calculation, data are cached in memory. Only when the memory capacity is insufficient are data are cached on a disk. Spark provides a series of operations for RDD, including not only Map() and Reduce() operations but also filter(), flatMap(), groupBy(), sortBy() and

other operations, which can provide developers with rich choices. It also supports APIs in Java, Python, Scala, R and other languages, enabling developers to quickly build different applications.

In the era of information explosion, the widespread application of big data has become a crucial force shaping the future [8]. With the continuous growth of big data volumes, ensuring the correctness and temporal properties of big data has become particularly critical. Big data not only encompasses vital information across various industries, from finance to healthcare, but also involves the processing of these data often tied to crucial business decisions and personal privacy. Therefore, the precise validation of big data, especially the verification of its temporal properties such as timeliness, reliability, and consistency, becomes paramount. This paper aims to delve into the correctness validation and temporal property verification of Spark applications, which may be the most widely used solution among big data processing applications.

The runtime verification tool UMC4M [9,10] uses a program M written in the Modeling, Simulation, and Verification Language (MSVL) [11–14] with the expected property P specified by the Propositional Projection Temporal Logic (PPTL) [15,16] formula as input. Whether M violates the property P is determined by whether there is an acceptable execution of a new MSVL program "M and M1", where the M1 is the negation of the property P [17,18]. In order to use the tool UMC4M to verify the temporal properties of Spark programs, it is necessary to use MSVL to formalize Spark programs. MSVL is a parallel programming language, which can be easily used for parallel computing. In [19], we merely proposed a formalization method using MSVL for Spark applications but did not implement it. Here, we formalize Spark programs by writing MSVL functions corresponding to RDD operations and extracting the execution relations of these functions based on the Directed Acyclic Graph (DAG) of Spark programs. This transformation converts a Spark program into a logically equivalent MSVL program, shifting the verification of properties in the Spark program to the verification of the MSVL program.

The contributions of this paper are three-fold:

(1) For RDD operations, we write corresponding MSVL functions to realize its processing of data. In addition, for those RDD operations with wide dependencies, we also write specialized Shuffle functions to complete the exchange of data between the two stages.
(2) We propose an algorithm for converting Spark programs to MSVL programs based on the DAG of Spark programs. The algorithm extracts formal relations within stages and formal relations between stages, respectively, thus completing the transformation of a Spark program into an MSVL program.
(3) Based on the algorithm mentioned above, we develop an automatic translation tool S2M to convert Spark programs to MSVL programs and give a case study to show the conversion process.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to research related to big data security and formalization and illustrates the advantages of using UMC4M to verify timing properties. Section 3 briefly introduces MSVL and the Spark framework. In Section 4, we describe the implementation of program translation system. In Section 5, a case study is given. Section 6 is a technical discussion of the methodology. Section 7 concludes the paper.

2. Related Work

The research on the validation correctness of big data applications is very important, especially for Spark applications, but there is little relevant research on this aspect.

Luo et al. [20] introduced an approach to predict the performance of Spark based on its configuration parameters using machine learning, specifically Support Vector Machine (SVM). This research focuses on understanding the impact of Spark's numerous configuration parameters and their interactions on performance. Additionally, Artificial Neural Network (ANN) is employed to model Spark's performance, and the paper reports that the SVM outperforms ANN.

Grossman et al. [21] proposed an SMT-based technique to verify the equivalence of interesting classes in Spark programs. By modeling Spark programs, they demonstrated the undecidability of checking equivalence even for Spark programs with a single aggregation operator. They also proved the completeness of the technique under certain restrictions.

Beckert et al. [22] proposed a method by dividing equivalence proof into equivalence proof sequences between intermediate programs with small differences for equivalence proof between imperative and MapReduce algorithms, and they verified the feasibility of the method on k-means and PageRank algorithms.

Yin et al. [23] formalized the main components of Spark on YARN using Communicating Sequential Processes (CSP). Then, they input the model into model checker Failures Divergence Refinement (FDR) to verify their main properties, such as Divergence Freedom, Load-Balancing, Deadlock Freedom and Robustness.

Baresi et al. [24] proposed a method based on model checking for verifying the execution time of Spark applications, and the method has been validated on some realistic cases.

de Souza Neto et al. [25] introduced TRANSMUT-SPARK, a tool designed to automate mutation testing for Big Data processing code within Spark programs. The complexity of Spark code makes it susceptible to false statements that require thorough testing. They explored the application of mutation testing, a fault-based technique, in Spark programs to automatically evaluate and design test sets.

However, these efforts have not verified the temporal properties that Spark applications should meet. At present, there are some tools for verifying temporal properties. Compared with the most relevant software model detection tools LTLAutomizer [26] and T2 [27–29], UMC4M has the following advantages: (1) the input of the UMC4M tool is a program written in the MSVL function, and the PPTL formula describes the properties to be verified. MSVL and PPTL are subsets of the Projection Temporal Logic (PTL), which leads to higher efficiency in program validation. (2) PPTL is capable of expressing fully regular properties, demonstrating greater expressive power than Linear Temporary Logic (LTL) and Computing Tree Logic (CTL).

3. Preliminaries

This section provides a brief introduction to the fundamental principles of MSVL and Spark.

3.1. MSVL

Modeling, Simulation and Verification Language (MSVL) is developed from the Framed Tempura [30] and is an executable subset of the Projection Temporal Logic (PTL) [12]. The arithmetic and Boolean expressions of MSVL can be summarized and defined as shown in Equations (1) and (2), which are used in MSVL statements.

$$e ::= c|x|\bigcirc e|\ominus e|g(e_1,\cdots,e_m)|ext\,f(e_1,\cdots,e_n) \tag{1}$$

$$b ::= true|false|\neg b|b_0 \wedge b_1|e_0 = e_1|e_0 < e_1 \tag{2}$$

where c is a constant, and x is a variable. $\bigcirc e$ represents the next state of variable e, and $\ominus e$ represents the previous state of variable e. $g(e_1,\cdots,e_m)$ is the call of the state function g. Each arithmetic operation(+|-|*|/) can be regarded as a function call $g(e_1,e_2)$. $extf(e_1,\cdots,e_n)$ is a call to an external function. b is a boolean expression that can be either true ($true$) or false ($false$). It may also be the negation ($\neg b$) of another boolean expression, the logical AND ($b_0 \wedge b_1$) of two boolean expressions, or the equality comparison ($e_0 = e_1$) or less-than comparison ($e_0 < e_1$) between two arithmetic expressions. The MSVL language includes the fundamental constructs as shown in Table 1, and by combining them according to certain rules, one can generate MSVL programs.

Table 1. The statements of MSVL

Name	Syntax	Semantics
1. Termination	empty	$\stackrel{\text{def}}{=} \varepsilon$
2. Assignment	$x := e$	$\stackrel{\text{def}}{=} \bigcirc x = e \wedge \text{len}(1)$
3. Positive immediate assignment	$x <== e$	$\stackrel{\text{def}}{=} x = e \wedge p_x$
4. State frame	$\text{lbf}(x)$	$\stackrel{\text{def}}{=} \neg\, \text{af}(x) \rightarrow \exists b\colon (\ominus x = b \wedge x = b)$
5. Interval frame	$\text{frame}(x)$	$\stackrel{\text{def}}{=} \Box\,(\text{more} \rightarrow \bigcirc\, \text{lbf}(x))$
6. Next	next ϕ	$\stackrel{\text{def}}{=} \bigcirc \phi$
7. Always	always ϕ	$\stackrel{\text{def}}{=} \Box \phi$
8. Conditional	if b then ϕ_0 else ϕ_1	$\stackrel{\text{def}}{=} (b \rightarrow \phi_0) \wedge (\neg b \rightarrow \phi_1)$
9. Local variable	exist $x : \phi(x)$	$\stackrel{\text{def}}{=} \exists x : \phi(x)$
10. Sequential	$\phi_0\,;\phi_1$	$\stackrel{\text{def}}{=} \phi_0; \phi_1$
11. Conjunction	ϕ_0 and ϕ_1	$\stackrel{\text{def}}{=} \phi_0 \wedge \phi_1$
12. While	while $b\{\phi\}$	$\stackrel{\text{def}}{=} (b \wedge \phi)^* \wedge \Box(\varepsilon \rightarrow \neg b)$
13. Selection	ϕ_0 or ϕ_1	$\stackrel{\text{def}}{=} \phi_0 \vee \phi_1$
14. Parallel	$\phi_0 \parallel \phi_1$	$\stackrel{\text{def}}{=} \phi_0 \wedge (\phi_1;\text{true}) \vee (\phi_0;\text{true}) \wedge \phi_1 \vee \phi_0 \wedge \phi_1$
15. Projection	$(\phi_1, \cdots, \phi_m)$ prj ϕ	$\stackrel{\text{def}}{=} (\phi_1, \cdots, \phi_m) prj \phi$
16. Await	await(b)	$\stackrel{\text{def}}{=} \text{frame}(x_1, x_2, x_3, \cdots, x_n) \wedge \Box(\varepsilon \leftrightarrow b)$

MSVL supports not only common statements in the imperative language such as assignment loop conditional statements, sequential statements and while statements, but also non-deterministic and concurrent statements. For example, $x <== 0$ indicates that 0 is immediately assigned to variable x, while $x := 0$ indicates that 0 is assigned to in the next state and the length of interval is one unit. $\phi_0\,;\phi_1$ represents the statement that ϕ_1 can only be executed after the completion of ϕ_0 statement. $await(b)$ means that the execution can only be continued when condition b is satisfied [31]. In addition, in order to improve the efficiency of programming, MSVL supports calling the library functions of C language, such as fopen(), fgets() and other functions. The function structure of MSVL is function $f()\{\cdots\}$. The structure also allows us to write functions that are logically equal to RDD operations [14]. At the same time, we also have the model checker UMC4M [9] for verifying MSVL programs.

3.2. Spark Framework

The core of the Spark framework is a computing engine. On the whole, it adopts the standard master–slave structure, in which the master node is responsible for managing the scheduling of tasks in the whole cluster, and the slave nodes are responsible for the actual execution of tasks. As the core model of data processing in the Spark framework, RDD supports two types of operations: one is transformation operations, such as map(), flatMap(), groupBy(), filter() and reduceByKey(), and the other is action operations, such as reduce(), collect(), foreach() and take(). There are dependency relationships between RDDs, which can be divided into narrow dependencies and wide dependencies. Spark forms a DAG based on the dependencies between RDDs. The DAG is submitted to the DAGScheduler, which divides the DAG into multiple stages of interdependence. When encountering a wide dependency, a stage is generated, and each stage contains one or

more tasks. Then, these tasks are submitted to the TaskScheduler for running in the form of taskSet.

The running framework of Spark, as shown in Figure 1, includes the Cluster Manager, Worker Nodes running multiple job tasks, Driver Nodes for the task control of each application, and Executor processes on each Worker Node responsible for the specific task execution.

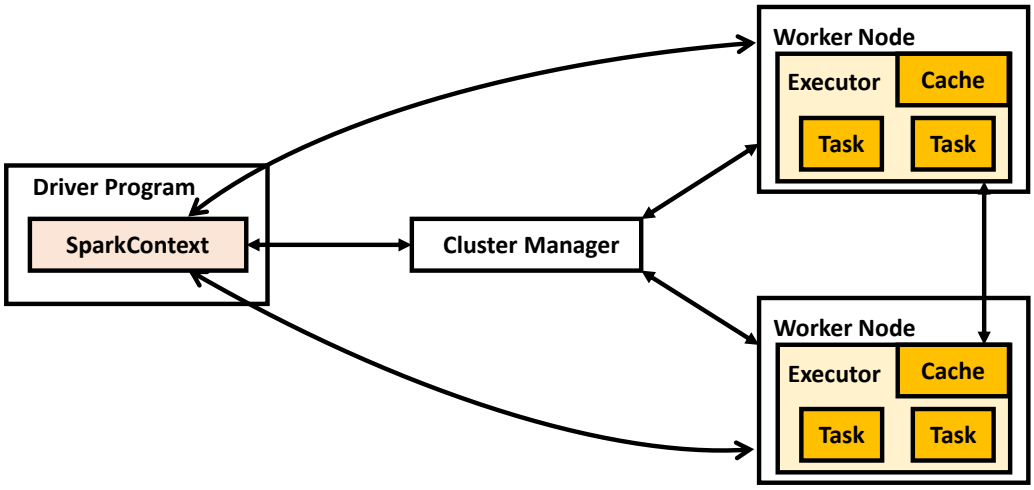

Figure 1. Spark running framework.

The Driver runs the main function of the Spark application and creates the SparkContext. The SparkContext is responsible for communication with the Cluster Manager, handling tasks such as resource acquisition, task assignment, and monitoring.The Cluster Manager is responsible for requesting and managing the resources needed to run the application on Worker Nodes. The Executor is a process running on a Worker Node where the application is executed. It is responsible for running tasks and storing data in memory or on disk.

A Spark application is divided into jobs, stages and tasks. Their definitions are given as follows:

(1) Application: a Spark application written by the user contains one or more jobs.
(2) Job: a set of stages executed as a result of an action operation. During the execution of a Spark application, each action operation triggers the creation and submission of a job.
(3) Stage: a set of tasks that perform the same computation in parallel based on partitions of input data.
(4) Task: unit of execution in a stage. A task is a single data processing on a data partition.

In different cluster environments, the process of Spark submission is basically the same. The driver starts to execute the main function after all the executors are registered. When it executes an action operation, it triggers a job and starts to divide the stages according to the wide dependencies. Each stage generates the corresponding taskSet, and then the driver distributes tasks to the specified executors for execution.

There are two kinds of RDD operations in Spark: namely, transformation operations and action operations. The transformation operations refer to the operation of creating a new RDD from the existing RDD and returning the new RDD. The transformation operations are lazily evaluated. They do not immediately trigger the execution of the actual conversion and only records the conversion relationship between RDDs. Only when an action operation is triggered can the transformation operations be truly executed and the calculation result be returned. Spark submits the operator graph to the DAGScheduler when an action operation is called. DAGScheduler divides DAG into stages according to wide dependencies. The dependencies between RDDs can be divided into narrow dependencies and wide dependencies, as shown in Figure 2. Narrow dependencies mean

that a partition of each parent RDD is used by at most one partition of the child RDD. Wide dependencies mean that each partition of the parent RDD can be used by multiple partitions of the child RDD, which are also known as shuffle dependencies.

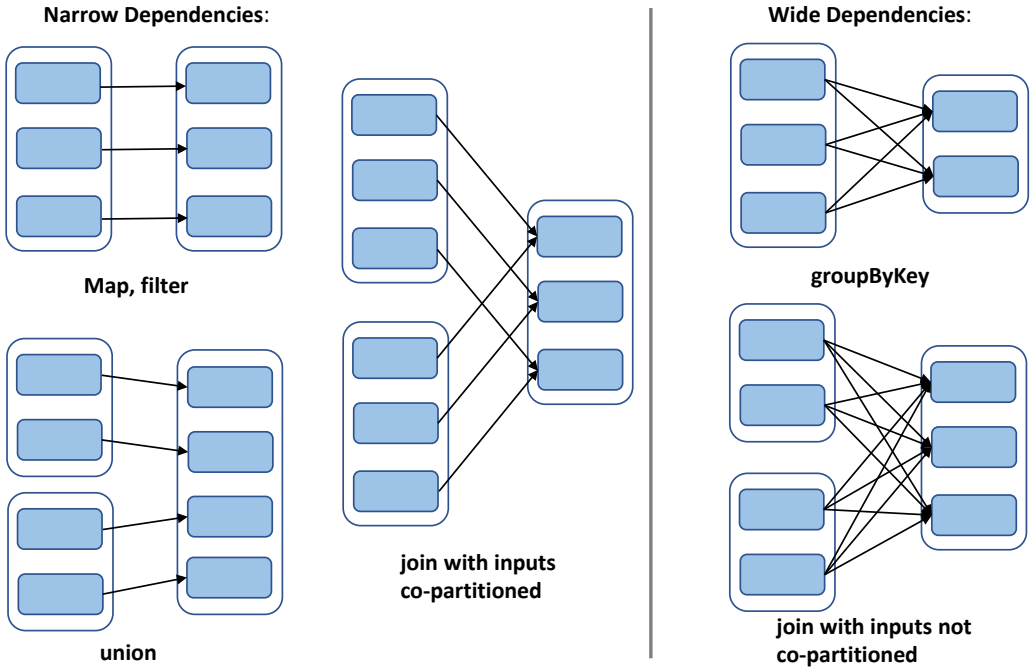

Figure 2. Dependency types.

4. Translation from Spark to MSVL

In this section, we realize the translation from Spark programs to MSVL programs and introduce it from the aspects of data storage, function translation and DAG-based formalization.

4.1. Data Storage Structure

RDD is the most basic data processing model in Spark. Spark provides rich transformation and action operations for RDD. The data in the RDD can only be updated through the transformation operations to generate a new RDD based on the original RDD. The data set in the RDD is logically and physically divided into multiple partitions. In a stage, one partition of the parent RDD is used by at most one partition of the child RDD, and the partitions in an RDD execute a series of transformation operations in parallel. The data in each partition can be executed in a separate task. The number of partitions and the number of tasks correspond one to one.

MSVL supports multiple data types to build data structures. Based on the Spark RDD data feature, we can use the MSVL-type struct to build special data structures for storing data. We define two structures, **ReadData** and **KeyValue**, to process data in the read stage and non-read stage, respectively, where the read stage refers to the stage containing the textFile operation.

```
struct ReadData
{
        char *data[10] and
        int value[10] and
        int order
 };
struct KeyValue
{
        char *data and
        int value and
        int order
};
```

In the struct **ReadData**, **data[0]** is used to store a line of data read from the data file. When **data[0]** is handled by the split function, **data[0]** is split, and the split data are saved in **data[i] (0** $\leq$ **i** $<$ **10)**. Here, **data[0]** is used to avoid memory waste. For the key–value pairs type in Spark, we define the array **value** to store the number of data occurrences. In a stage, data are processed in a pipeline way. To this end, we define an **order** to represent which function processes the data. For the non-read stage, the main task is to complete the data operation. To achieve that, we define the structure **KeyValue**, where **data** is used to store data, **value** represents the number of data occurrences, and **order** represents which function processes the data. Partitions in an RDD are executed in parallel. To this end, we define two arrays **struct ReadData lineData[N]** and **struct KeyValue finaldata[M]** for the read stage and non-read stage, respectively. Here, **N** and **M** are both integer constants, which are typically chosen to be slightly larger than the parallelism of the function. This choice aims to prevent any impact on the parallelism of the function and the size of the transformed program.

Stages are divided according to wide dependencies. That is to say, a shuffle occurs in two directly connected stages. Shuffling can cause data disruption and reorganization. For the conversion of data between directly connected stages A and B, we define structures **CollectData** and **PrepData**, where **CollectData** is used to collect the processed data in stage A and **PrepData** is used to preprocess the data for the next stage B.

```
struct CollectData
{
        char *data and
        int value and
        int idle
 };
struct PrepData
{
        char *data and
        int value[100] and
        int idle
};
```

The struct **CollectData** is used to complete data collection. In the struct **CollectData**, **data** is used to store data, **value** represents the number of **data** occurrences and **idle** represents the status of memory, where **idle = 0** means the memory is free, and **idle = 1** means that the data have been stored but not used. The struct **PrepData** is used to complete data preprocessing. In the struct **PrepData**, **data** is used to store data, and array **value** is used to store the number of the respective occurrences of the same data. **idle** also indicates

the status of the memory. We define two arrays, **struct CollectData end_Data1[L]** and **struct PrepData end_Data[L]**, to complete data collection and preprocessing separately, where L is a constant and is usually set to the largest possible number to avoid some data not being collected.

4.2. Translation from Spark Operations to MSVL Functions

The RDD operations are divided into two types: transformation operations and action operations. The action operations actually calculate the dataset, which are the basis for Spark to divide jobs. Stages are divided according to wide dependencies. In a stage, the partitions in an RDD are processed in parallel. MSVL not only supports function definitions and statements that are similar to imperative languages, such as assignment, condition, and loop statements, but it also supports non-deterministic and concurrent programming, such as selection and parallelism statements. This means that corresponding MSVL functions can be written for Spark RDD operations. Most operational translation ideas are the same. First, the execution condition is evaluated, and then the internal logic is executed when the conditions are met. Due to space constraints, we briefly introduce the translations of several operations here.

4.2.1. Translation of textFile Operation

During the execution of an MSVL program, if it encounters C language library function calls, the MSVL interpreter automatically calls the corresponding C functions and returns the execution result. Thus, when writing the function corresponding to textFile operation, we can call C functions such as fopen(), fclose(), fgets(), etc. The *textFile* operation is a transformation operation that reads data from a file as a collection of lines. Function 1 shows the MSVL function corresponding to textFile operation.

MSVL Function 1: ReadWords(char ****datafile*)**

```
1  frame(fp, i, brk) and
2  (
3    FILE*fp <== fopen(datafile, "r") and skip;
4    int i <== 0, brk <== 0 and skip;
5    while (feof(fp) = 0){
6      i := 0; brk := 0;
7      await(lineData[0].order = 1 OR ··· OR lineData[N − 1].order = 1);
8      while (i < N AND brk = 0){
9        if(lineData[i].order = 1) then {brk := 1}else {i := i + 1}
10     };
11     fgets(lineData[i].data[0], bufferSize, fp) and skip;
12     lineData[i].order := lineData[i].order+1
13   };
14   readOver := 1
15 )
```

The MSVL function *ReadWords* is the first function executed. The function reads data from file *datafile*. When there is an idle memory, i.e., $lineData[i].order = 1$ ($0 \leq i < \mathrm{N}$), at most one line of *bufferSize* characters is read from the file *datafile* and stored in $lineData[i].data[0]$, where *bufferSize* is a global constant, and then the $lineData[i].order$ is added by 1, which indicates that the data $lineData[i]$ are ready to be executed by the second function. This process is repeated until the file *datafile* is read completely (Lines 5–13). After reading all data, the global variable *readOver* is set to 1, indicating that the *datafile* has been completely read (Line 14).

4.2.2. Translation of Shuffle

A job consists of multiple stages, and each stage has multiple transformation operations. The transformation operations are categorized into two types based on their dependencies: narrow transformations and wide transformations. Wide transformations are the basis of stage division. It means that each partition of the parent RDD can be used by multiple child RDD partitions, and a partition of each child RDD usually uses all parent RDD partitions. This property of wide transformations results in shuffling, which refers to shuffling the data between stages. For this reason, we define the shuffle function of MSVL based on its properties to handle data shuffling between two stages. Although the data types processed in different stages are different, the principle remains the same. We present the *shuffle* function in the read stage in Function 2.

MSVL Function 2: shuffle(int $tNum$**, char** $*Op$**)**

```
1  frame(i, brk) and
2  (
3    int i <== 0, brk <== 0 and skip;
4    while(lineData[0].order! =1 OR··· OR lineData[N − 1].order! = 1 OR
readOver! =1){
5      await(lineData[0].order = tNum OR ··· OR lineData[N − 1].order = tNum);
6      i := 0; brk := 0;
7      while (i < N AND brk = 0){
8        if(lineData[i].order = tNum) then {brk := 1} else {i := i + 1}
9      };
10      Collect(i);
11      lineData[i].order := 1;
12   };
13   Pretreat(Op);
14   readOver := 0;
15   read_num := 0
16 )
```

In Function 2, $tNum$ indicates the number of operations in the current stage plus 1, and Op represents the name of the first operation in the next stage. The data in $lineData[i]$ ($0 \leq i <$ N) in the current stage are collected by the *Collect* function after processing, and then $lineData[i].order$ is set to 1. This process is repeated until all data in the current stage are processed (Lines 4–12). After collection, the collected data are preprocessd by the function *Pretreat* (Line 13). The data preprocessing function *Pretreat* differs for different wide transformations. For example, for the *reduceByKey* operation, the data are aggregated, and these data with the same key are stored in $end\_Data[j]$ ($0 \leq j <$ L) . The values corresponding to the same key are saved in array *value* of array $end\_Data[j]$. Finally, we set both *readOver* and *read_num* to 0 (Lines 14–15), where the global variable *read_num* indicates the reading position of array *end_Data* in the next stage.

4.2.3. Translation of reduceByKey Operation

The *reduceByKey* is a wide transformation operation, and its function signature is $def\ reduceByKey\ (func:\ (V,\ V)\ =>\ V):\ RDD\ [(K,\ V)]$. The operation aggregates values with the same key based on the anonymous function $func$ passed in and returns a new RDD. We use MSVL statements to write the corresponding *ReduceByKey* function based on the characteristics of the *reduceByKey* operation. The function is not executed until the *shuffle* function completes data preprocessing.

MSVL Function 3: ReduceByKey(int n)

```
1  frame(i, brk) and
2  (
3    int i <== 0, brk <== 0 and skip;
4    while (readOver = 0){
5      Bakery(n, number, choose);
6      await(finaldata[0].order = 1 OR ··· OR finaldata[M − 1].order = 1 OR
readOver=1);
7      i := 0; brk := 0;
8      while (i < M AND brk = 0){
9        if(finaldata[i].order = 1 AND parallel[i] = 0) then {brk := 1}else {i := i + 1}
10     };
11     parallel[i] := 1;
12     reduceImplicit(i, n);
13     finaldata[i].order:=finaldata[i].order+1;
14     parallel[i] := 0
15   };
16 )
```

The code for the function is shown in Function 3. As this operation has wide dependencies, the function is executed first in the current stage. When the *reduceBykey* functions are executed in parallel, in order to prevent the same functions from performing the same processing on $finaldata[i]$ ($0 \leq i <$ M) and obtain mutually exclusive access to global variable *read_num*, we use the *Bakery* algorithm and array *parallel* to implement it. The $Bakery(n, number, choose)$ is used for locking, where n represents the function ID, array *number* is a global variable that represents the queue number, and array *choose* is also the global variable that represents whether the number is being retrieved (Line 5). When there is a free memory block ($finaldata[i].order$=1 ($0 \leq i <$ M)) and the data block is not locked by a function ($parallel[i] = 0$), the data block is locked by the function (Line 11) and data processing is performed through the function *reduceImplicit*. After processing is completed, we add 1 to $finaldata[i].order$ and unlock the locked data $finaldata[i]$. This process is repeated until array *end_Data* is fully read (Lines 4–15).

MSVL Function 4: reduceImplicit(int $sign$, int n)

```
1  frame(k, num) and
2  (
3    int k <== 0, num <==0 and skip;
4    num :== read_num;
5    read_num := read_num + 1;
6    number[n] :=0;
7    if(end_Data[num].idle=1)then{
8      finaldata[sign].data:=end_Data[num].data;
9      while(end_Data[num].value[k]! = 0){
10       finaldata[sign].value := finaldata[sign].value + end_Data[num].value[k];
11       k := k + 1
12     };
13     end_Data[num].idle := 0
14   }else{
15     readOver :=1
16   }
17 )
```

For different anonymous functions, we automatically generate corresponding reduceImplicit functions. Taking the anonymous function (_+_) as an example, we generate *reduceImplicit* as shown in Function 4. In Function 4, the parameter *sign* represents which data are processed, and the parameter *n* represents the ID of the function. We use *num* to save the value of *read_num*, where *read_num* is a global variable representing which element in array *end_Data* is read (Lines 4–5). Then, we unlock these resources (Line 6). If the data to be read from array *end_Data* exist, the data in $end\_Data[num].data$ are saved in $finaldata[sign].data$, and values are aggregated and saved in $finaldata[sign].value$. Then, the *idle* of the data is set to 0 (Lines 7–13). If the data to be read from array *end_Data* do not exist, the global variable *readOver* is set to 1, indicating that array *end_Data* has been completely read (Lines 14–16).

4.3. DAG-Based Formalization

When an action operation is performed, a new DAG is generated. Each operation generates a new RDD. The original RDD forms a DAG after a series of transformations. According to the different dependency relationships between RDDs, the DAG can be divided into different stages. To formalize Spark programs based on the DAG, we have developed a tool called S2M, whose architecture is shown in Figure 3. The formalization process for Spark programs based on the DAG consists of four main steps: extraction of required information, formalization of operations within stages, formalization between stages, and function replacement and supplementation.

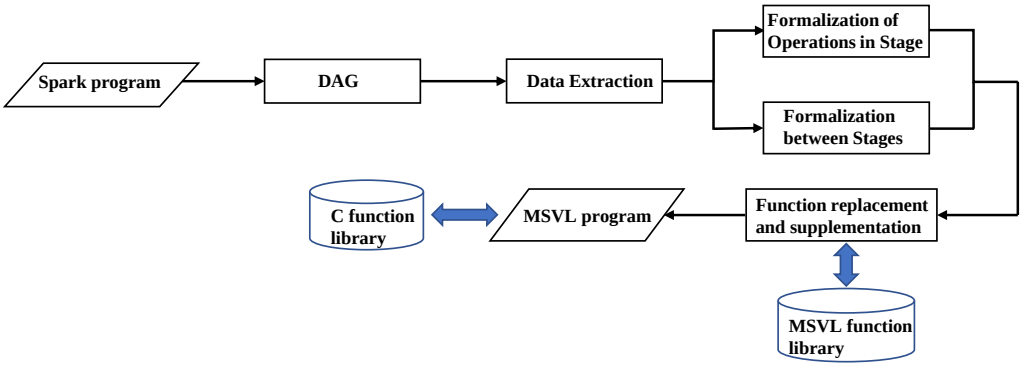

Figure 3. Architecture of S2M.

4.3.1. Data Extraction

This section is mainly about extracting the information between RDD operations in stages and the information between stages from the DAG of a Spark application. We use Python to implement the extraction for obtaining the required information from the SVG tag in the DAG UI interface of the application.

In a DAG, each blue rectangle corresponds to a Spark operation, and the nodes in each rectangle represent the RDDs created under the corresponding operation. The RDDs are connected by arrows. For each stage in a DAG, we need to extract the name of operation and nodes in each blue rectangle as well as the nodes connected by the arrows connecting blue rectangles (operations). Each blue rectangle may have one or more nodes. For each stage, we define an array *Opandnode* where each element holds a blue rectangle information, and the element is stored in tuple $(name, \{node1, node2, \cdots\})$, where the first element in the tuple is the name of the operation and the second element called *nodeset* is the set of nodes. For arrows in a stage, we construct an array *instage_arrow* to store the nodes connected by the arrows connecting blue rectangles. The elements are stored in tuples $(startnode, endnode)$. For information extraction between stages, we construct an array *stage_stage* to store the connecting nodes of the arrows connected between stages. The elements are also stored in the format of $(startnode, endnode)$. In addition, we create an array *Op_position* to save the number of lines in the code file for each operation in the DAG. This allows us to extract the anonymous functions in each incoming operation.

4.3.2. Formalization of Operations in Stages

To extract the formalization of operations within a stage, we first obtain the execution paths in the stage, where a path refers to the order of operations execution directly connected by arrows. Then, we process these paths to complete the formalization of RDD operations in the stage.

After we obtain the name and nodes of each operation (blue rectangle) in a stage, as well as the nodes connected by the arrows connecting RDDs, we determine the two operations that are directly connected by determining which operation the *startnode* and *endnode* of an arrow are in. The acquisition of execution paths during a stage is presented in Algorithm 1, where each element in *instage_path* is an execution path in a stage. If there is only one operation in a stage, this operation is added to a path (Lines 1–3). Then, we access each arrow of connection operations (Line 4) and look for the operations where the *startnode* and *endnode* of the arrows are located (Lines 5–8). There are three situations at this time: the first is if the operation (operation1) where *startnode* is located is the last element in a path; then, the operation (operation2) where *endnode* is located is directly added to that path (Lines 9–11). The second is if *operation*1 is not the last element in a path; then, we copy the path from the beginning to *operation*1 to a new path and add *operation*2 to the new path (Lines 12–16). The third is if *operation*1 is not in any paths; then, a new path is added and we add both *operation*1 and *operation*2 to this path (Lines 17–21).

Algorithm 1: Execution paths of RDD operations in a stage

```
   Input: instage_arrow, Opandnode
   Output: instage_path
 1 if Opandnode has only an element then
 2 |   instage_path[0][0] ← Opandnode[0];
 3 end
 4 for arrow in instage_arrow do
 5 |   for operation1 in Opandnode do
 6 |   |   if arrow.startnode ∈ operation1.nodeset then
 7 |   |   |   for operation2 in Opandnode do
 8 |   |   |   |   if arrow.endnode ∈ operation2.nodeset then
 9 |   |   |   |   |   if operation1 is the last element of one path: instage_path[i][k] then
10 |   |   |   |   |   |   instage_path[i][k + 1] ← operation2;
11 |   |   |   |   |   end
12 |   |   |   |   |   if operation1 is non-last element of one path:instage_path[i][l] then
13 |   |   |   |   |   |   add a new path:instage_path[j] ← instage_path[i][0tol];
14 |   |   |   |   |   |   insatge_path[j][l + 1] ← operation2;
15 |   |   |   |   |   end
16 |   |   |   |   |   if operation1 is not the elements of all paths then
17 |   |   |   |   |   |   add a new path: instage_path[m];
18 |   |   |   |   |   |   instage_path[m][0 − 1] ← operation1 and operation2;
19 |   |   |   |   |   end
20 |   |   |   |   end
21 |   |   |   end
22 |   |   end
23 |   end
24 end
```

To formalize operations within a stage, we process the execution paths in that stage. Algorithm 2 outlines this process, where *path* is the first path of *instage_path*, *instage_path* stores all paths obtained through Algorithm 1, and stage stores the formalization of operations in a stage. Firstly, we assign the name of the first operation in the first path(*path*) to the stage, and if there is only one operation in *path*, we exit the function directly (Lines 1–4).

Then, we sequentially access the remaining operations (*operation*1) in *path* (Line 5). For each *operation*1 accessed, we access other paths to find the path (*path*1) containing this *operation*1, and these two paths are different in the part before *operation*1. If such a path exists, we remove it from all paths (*instage_path*) and perform the same processing on the parts of the path before *operation*1. After processing is completed, these two paths are parallel in the parts before *operation*1 (Lines 7–14), and *operation*1 is executed only after these paths are executed (Lines 15–17). If such a path does not exist, *operation*1 and the previous operations are executed in parallel (Lines 18–20). After visiting the first path, we obtain the formal relationships within the stage.

Algorithm 2: FormInStage(*path*, *insatge_path*)

```
   Input: path,instage_path
   Output: stage
1  stage ← path[0].name;
2  if path has only an operation then
3  |  return stage;
4  end
   // n is is the length of path
5  for operation1 ← path[1] to path[n − 1] do
6  |  sign ← 0;
   |  // m is the length of instage_path
7  |  for path1 ← instage_path[1] to instage_path[m − 1] do
8  |  |  if path1 contains operation1 and the operations before operation1 in path and
   |  |    path1 are different then
9  |  |  |  sign ← 1;
10 |  |  |  delete path1 from instage_path;
11 |  |  |  path1 ← delete the last operation from path1;
12 |  |  |  stage ← stage + "||" + FormInStage(path1, instage_path);
13 |  |  end
14 |  end
15 |  if sign == 1 then
16 |  |  stage ← "(" + stage + ")" + ";" + operation1.name;
17 |  end
18 |  else
19 |  |  stage ← "(" + stage + ")" + "||" + operation1.name;
20 |  end
21 end
22 Delete excess "(" and ")";
```

4.3.3. Formalization between Stages

The formalization between stages is similar to the formalization of operations in a stage. First, we need to obtain the paths between directly connected stages and then process these paths to achieve formalization between stages. For obtaining paths between stages, we first define an array *stage_path* to hold the execution paths between stages, where each element in *stage_path* is an array representing a sequential execution path between stages. Then, we process the connecting arrows between stages stored in *stage_stage*, where each element connects two directly connected stages. The process is similar to Algorithm 1, which sequentially traverses each arrow in *stage_stage*. For the two stages of arrow connection (*stage*1, *stage*2), if there is a path in *stage_path* that contains *stage*1 and *stage*1 is the last element of the path, *stage*2 is added to the end of the path. If there is a path in *stage_path* that contains *stage*2 and *stage*2 is the first element of the path, *stage*1 is added to the first element of the path. For other situations, we add a new path and sequentially

add *stage*1 and *stage*2. After processing all arrows between stages, we obtain the sequential execution paths between stages and store it in *stage_path*.

After we obtain the execution paths between stages, we process them to achieve formalization between stages, as shown in Algorithm 3, where *path* is the first path in *stage_path*. Firstly, we assign the first stage in *path* to the output *M_stage* (Line 1). Then, we sequentially access each stage (*stage*1) in *path* (Line 2). For each *stage*1, we search for other paths that also contain *stage*1. If such paths exist, paths before *stage*1 are parallel, and we also perform recursive operations on paths (Lines 3–9). After processing paths containing *stage*1, the execution relationship between *stage*1 and the previous one is sequential (Line 10). Finally, we remove parentheses that do not affect the execution order (Line 12).

Algorithm 3: FormStage(*path*, *stage_path*)

```
   Input: path, stage_path
   Output: M_stage
1  M_stage ← path[0];
   // n is the length of path
2  for stage1 ← path[1] to path[n − 1] do
       // m is the length of stage_path
3      for path1 ← stage_path[1] to stage_path[m − 1] do
4          if path1 contains stage1 then
5              delete path1 from stage_path;
6              path1 ← delete the last stage from path1;
7              M_stage ← M_stage + "||" + FormStage(path1, satge_path);
8          end
9      end
10     M_stage ← "(" + "(" + M_stage + ")"+ ";" + stage1 + ")";
11 end
12 Delete excess "(" and ")";
```

4.3.4. Function Replacement and Supplementation

Because the same operation may occur multiple times, the internal processing logic may also differ. Therefore, we need to differentiate them by assigning different names to operations that may occur multiple times. In addition, operations in the same stage are executed in parallel, so we need to parallelize the transformed MSVL function and set its ID. One example is the MSVL function Map() corresponding to the map operation: we process it as Map(1) $\| \cdots \|$ Map(n), where n represents the ID of the function. Additionally, for each wide transformation operation, we add a $shuffle$ function before the MSVL function to complete data collection and preprocessing, as shown in Algorithm 4.

In Algorithm 4, *stage_num* is the number of stages. *Seq_Op* stores the operation names and anonymous functions for each stage after formalization. *instage_Op* stores the formalization of operations in each stage. *M_stage* stores formalization between stages. The array para stores the parallelism of all stages. The format method is used to format strings. We process the formalized operations in each stage (Lines 1–2). If this operation is *textFile*, we add it to the array *visit_Op*, which records the processed operations. Then, we check the number of times the operation has already occurred, change the MSVL function corresponding to the *textFile* operation based on the number of occurrences, and replace the *textFile* operation in the formalization of operations of this stage (Lines 3–8). If this operation is *reduceByKey*, the *reduceByKey* is added to the array *visit_Op* and the number of occurrences is recorded. Then, we perform parallel processing on the corresponding MSVL function based on the parallelism of this stage, replacing the *reduceBykey* operation with parallelized MSVL function in the formalization of operations of this stage (Lines 9–21). After formalizing a stage, we add the corresponding shuffle function based on the wide dependency operation of the next stage connected to it (Lines 24–26), and we replace the

corresponding stage in the formalization of the stage with the formal operations of this stage after processing (Line 27).

Algorithm 4: Formalization of Spark programs

```
   Input: stage_num, Seq_Op, instage_Op, M_stage, para
   Output: M_stage
1  for i ← 0 to stage_num - 1 do
2      for OpandImplicit in Seq_Op[i] do
3          if OpandImplicit[0] is "textFile" then
4              add "textFile" to array visitOp;
5              Op_num ← the number of occurrences of "textFile" in visitOp;
6              MFunction ← "ReadWords()".format(Op_num);
7              replace the first occurrence of "textFile" in instage_Op[i] with
                MFunction;
8          end
9          if OpandImplicit[0] is "reduceByKey" then
10             add "reduceByKey" to array visisOp;
11             Op_num ← the number of occurrences of "reduceByKey" in visitOp;
12             for k ← 1 to para[i] do
13                 if k == 1 then
14                     MFunction ← "ReduceBykey{0}({1})".format(Op_num, k);
15                 end
16                 else
17                     MFunction ← MFunction + "||" +
                        "ReduceByKey{0}({1})".format(Op_num, k);
18                 end
19             end
20             replace the first occurrence of "reduceByKey" in instage_Op[i] with
                MFunction;
21         end
22         ...
23     end
24     if i ≠ stage_num - 1 then
25         instage_Op[i] ← instage_Op[i] + "||" + shuffle function;
26     end
27     replace the "stage{}".format(i) in M_stage with instage_Op[i];
28 end
```

When an action operation is encountered, a job submission is triggered, and the Driver program submits the job to the DAGScheduler, which constructs the job into a DAG. However, in a program, two action operations may occur in succession, and we need to specifically handle such cases. To address this, we need to determine where the action operations in the DAG are located in the program. For subsequent action operations, we use sequential construction to formalize them, as shown in Algorithm 5. The variable *moreOp* stores action operations that have not yet been added to DAGs, where each element stores the name and anonymous function of an action operation. $Take(num)$ and $Foreach()$ are MSVL functions corresponding to the action operations take and foreach, respectively.

Algorithm 5: Supplement to the MSVL

```
Input: moreOp, stage_seq[0]
Output: stage_seq[0]
stage_seq[0] ← "(" + stage_seq[0] + ")";
for actionOp in moreOp do
    if actionOp[0] is "take" then
        num ← actionOp[1];
        stage_seq[0] ← stage_seq[0] + ";" + "Take(num)";
    end
    if actionOp[0] is "foreach" then
        stage_seq[0] ← stage_seq[0] + ";" + "Foreach()";
    end
    ...
end
```

5. Case Study: TopN

In this section, we demonstrate our work by implementing the analysis of click stream log data using Spark. Subsequent sections provide a detailed description of the translation process. We established a Spark cluster environment with five cores using virtual machines. By executing a TopN case, we obtained its Directed Acyclic Graph (DAG). The SVG label information of the DAG was input into the S2M tool to formalize the TopN program.

5.1. Application Programs

The aim of this case study is to analyze clickstream log data using Spark and find the top five URLs with the most visits. Figure 4 displays the Scala code for the TopN application.

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object TopN {
 def main(args: Array[String]): Unit = {
  val sparkConf: SparkConf = new SparkConf().setAppName("TopN")
  val sc = new SparkContext(sparkConf)
  val dataRDD = sc.textFile("hdfs://hadoop102:8020/input/words.txt")
  val filterRDD = dataRDD.filter(x =>x.split(" ").length >10)
  val urlsRDD = filterRDD.map(x=>x.split(" ")(10))
  val urlAndOneRDD = urlsRDD.map(x=>(x,1))
  val result = urlAndOneRDD.reduceByKey(+)
  val sortRDD = result.sortBy(.2,false)
  val top5 = sortRDD.take(5)
  top5.foreach(println)
  sc.stop()
 }
}
```

Figure 4. Code of TopN application.

The application begins by reading data from the hdfs text file. The variable dataRDD stores multiple lines of text content after the $textFile$ operation is executed. Next, we filter out any dirty data in $dataRDD$ using $dataRDD.filter(x => x.split(" ").length > 10)$, where $x => x.split(" ").length > 10$ is a lambda expression where the left side is the parameter and the right side is the function to be executed by the lambda. $x.split(" ").length > 10$ means that the data are divided according to spaces, and when the length of the divided data is greater than 10, it is retained in the $filterRDD$. We then use $filterRDD.map(x => x.split(" ")(10))$ to traverse every data item in $filterRDD$ and retain only the eleventh data

item because it represents the URL address. Further, we map the data to the form of a tuple (x, 1) using the lambda expression $x => (x, 1)$, where x is the key and 1 indicates that x appears once. Next, we perform the $reduceByKey(\_ + \_)$ operation on the resulting RDD, which aggregates the value of the same key. Finally, we sort the RDD in descending order using $sortBy(\_.2, false)$ and take the top five items using the $take(5)$ operation.

5.2. Formalization

We first extract the formalization of RDD operations within stages and the formalization between stages according to the DAG. To execute the application, we built a Spark cluster environment and obtained the DAG of the application, which is shown in Figure 5. This figure illustrates how Spark executes the application, where each blue rectangle in the stage represents an RDD generated by an association operation and arrows indicate the relationships between RDDs.

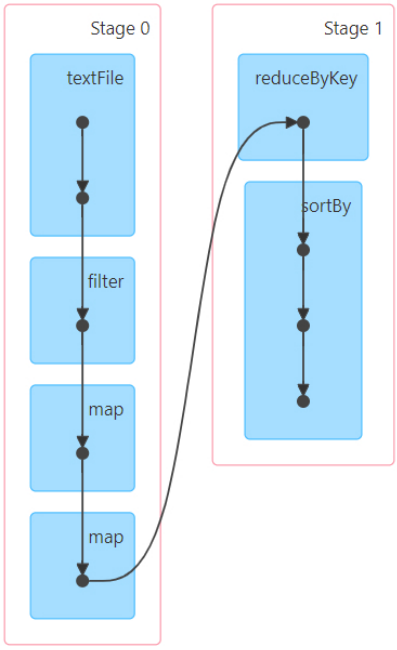

Figure 5. DAG of TopN application.

Based on the steps described in Section 3, we extract the name of the operation and nodes from each blue rectangle in each stage. Then, we extract the location of each RDD operation in the code file to facilitate the extraction of anonymous function in each RDD operation. Next, we process the execution order of RDD operations within each stage. The formalizations of RDD operations within $Stage0$ and $Stage1$ are as follows:

$$Stage0 = textFile||filter||map||map$$
$$Stage1 = reduceByKey||sortBy$$

After processing the formalizations of RDD operations within stages, we begin to process the execution relationship between stages. We extract the start and end nodes of the connecting arrows between stages and then obtain the formalization between $Stage0$ and $Stage1$ based on the node relationship in stage_stage. The formalization between stages is shown below:

$$TopN = Stage0; Stage1$$

Once we obtain the formalization between RDD operations within stages and between stages, we formalize the entire application using Algorithm 4 to replace function names and

Algorithm 5 to add action operations that are not displayed in the DAG. The formalized result is shown in Figure 6.

```
MSVL functions:
(ReadWords1("TopN.txt")||Filter1(2,1)||Filter1(2,2)||Filter1(2,3)||Filter1(2,4)||Filter
1(2,5)||Map1(3,1)||Map1(3,2)||Map1(3,3)||Map1(3,4)||Map1(3,5)||Map2(4,1)||
Map2(4,2)||Map2(4,3)||Map2(4,4)||Map2(4,5)||shuffle1(5,"reduceByKey"));(Reduc
eByKey1(1)||ReduceByKey(2)||ReduceByKey1(3)||ReduceByKey1(4)||ReduceByKey1
(5)||SortBy(2));Take(5);Foreach()
```

Figure 6. The formalized result.

The parallelism depends on the default configuration of the Spark framework. In our simulated environment, the number of cores is set to five. Therefore, we configure the parallelism of the functions Filter, Map1, Map2 and ReduceByKey1 to be 5, meaning that each of them can process five data pieces in parallel. Map1 and Map2 distinguish between two different execution logics. For instance, Map1 is used to process (_*2), while Map2 is used to process (_2). Since the take and foreach operations are not displayed in the DAG, they are handled specifically through Algorithm 5 and added to the translated MSVL program.

6. Technical Discussion

MSVL is a framework temporal logic language, constituting an executable subset of PTL. Programs written in MSVL can be utilized for modeling, simulation, and property verification. PPTL, the propositional part of PTL, is capable of expressing fully regular properties, demonstrating greater expressive power than LTL and CTL. The code-level runtime verification tool UMC4M is employed to verify systems written in MSVL and to perform verification based on properties described in PPTL. Due to the unified framework logic of MSVL and PPTL, property verification significantly reduces time overhead.

In previous research on the validation of Spark programs, there has been a greater emphasis on the correctness verification of Spark programs with limited attention given to the verification of their temporal properties. To employ UMC4M for formal verification of Spark programs, particularly focusing on verifying the temporal properties of Spark programs, the key lies in the conversion of Spark programs into MSVL programs. This paper, based on the DAG of Spark programs, transforms them into MSVL programs, presenting a novel approach for the verification of temporal properties in Spark programs.

7. Conclusions

Spark, as a pivotal technology in big data processing, necessitates the validation of its temporal properties. To facilitate the verification of the temporal characteristics of Spark programs using the runtime verification tool UMC4M, the formal modeling of Spark programs through MSVL programs becomes imperative. In this paper, we present a translator, S2M, designed to convert Spark programs into MSVL programs. Consequently, by translating Spark programs into MSVL programs, existing tools can be employed to validate the temporal properties of Spark programs. This transformation effectively shifts the problem of checking the satisfiability of a Spark program to the problem of checking the satisfiability of an MSVL program. However, our current implementation addresses the fundamental transformation from Spark programs to MSVL programs. Further research is required to handle intricate transformations. In the future, we plan to refine the S2M tool for implementing complex transformations and validating the temporal properties of Spark programs using the verification tool UMC4M.

Author Contributions: Conceptualization, K.F. and M.W.; methodology, K.F.; software, K.F.; validation, K.F. and M.W.; investigation, K.F.; resources, K.F.; writing—original draft preparation, K.F.; writing—review and editing, M.W.; supervision, M.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Hebei Natural Science Foundation under grant No. F2020201018, Science and Technology Research Project of Higher Education in Hebei Province under grant No. QN2021020 and Advanced Talents Incubation Program of the Hebei University under grant No. 521000981346.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The list of abbreviations and symbols is shown below.

Symbols	**Definition**
c	Constant
x	Variable
e	Arithmetic expressions
$\ominus e$	The previous state of variable e
$\bigcirc e$	The next state of variable e
$g(e_1, \cdots, e_m)$	The call of the state function g
$extf(e_1, \cdots, e_n)$	The call to an external function
b	Boolean expression
$<==$	Positive immediate assignment
$:=$	Next-state assignment
and	Conjunction
$skip$	One unit of time over an interval
;	Sequential execution
$\|\|$	Parallel execution
$await(b)$	Execution upon satisfaction of boolean expression b
N	Constant
M	Constant
L	Constant
$bufferSize$	Constant
$readOver$	Global variable
Acronyms	**Full Form**
MSVL	Modeling, Simulation and Verification Language
PPTL	Propositional Projection Temporal Logic
RDD	Resilient Distributed Datasets
DAG	Directed Acyclic Graph
PTL	Projection Temporal Logic
CTL	Computing Tree Logic
LTL	Linear-time Temporal Logic

References

1. Ketu, S.; Mishra, P.K.; Agarwal, S. Performance analysis of distributed computing frameworks for big data analytics: Hadoop vs. spark. *Comput. Sist.* **2020**, *24*, 669–686. [CrossRef]
2. Zhang, J.; Lin, M. A comprehensive bibliometric analysis of Apache Hadoop from 2008 to 2020. *Int. J. Intell. Comput. Cybern.* **2023**, *16*, 99–120. [CrossRef]
3. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
4. Chambers, B.; Zaharia, M. *Spark: The Definitive Guide: Big Data Processing MADE Simple*; O'Reilly Media: Sebastopol, CA, USA, 2018.
5. Kalia, K.; Gupta, N. Analysis of hadoop MapReduce scheduling in heterogeneous environment. *Ain Shams Eng. J.* **2021**, *12*, 1101–1110. [CrossRef]
6. Hedayati, S.; Maleki, N.; Olsson, T.; Ahlgren, F.; Seyednezhad, M.; Berahmand, K. MapReduce scheduling algorithms in Hadoop: A systematic study. *J. Cloud Comput.* **2023**, *12*, 143. [CrossRef]

7. Ghazi, M.R.; Gangodkar, D. Hadoop, MapReduce and HDFS: A developers perspective. *Procedia Comput. Sci.* **2015**, *48*, 45–50. [CrossRef]
8. Liu, J.; Li, J.; Li, W.; Wu, J. Rethinking big data: A review on the data quality and usage issues. *Isprs J. Photogramm. Remote. Sens.* **2016**, *115*, 134–142. [CrossRef]
9. Wang, M.; Tian, C.; Duan, Z. Full regular temporal property verification as dynamic program execution. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 226–228.
10. Wang, M.; Tian, C.; Zhang, N.; Duan, Z. Verifying full regular temporal properties of programs via dynamic program execution. *IEEE Trans. Reliab.* **2018**, *68*, 1101–1116. [CrossRef]
11. Duan, Z. An Extended Interval Temporal Logic and a Framing Technique for Temporal Logic Programming. Ph.D. Thesis, Newcastle University, Newcastle, UK, 1996.
12. Duan, Z. *Temporal Logic and Temporal Logic Programming*; Science Press: Sydney, Australia, 2005.
13. Yang, K.; Duan, Z.; Tian, C.; Zhang, N. A compiler for MSVL and its applications. *Theor. Comput. Sci.* **2018**, *749*, 2–16. [CrossRef]
14. Zhang, N.; Duan, Z.; Tian, C. A mechanism of function calls in MSVL. *Theor. Comput. Sci.* **2016**, *654*, 11–25. [CrossRef]
15. Duan, Z.; Tian, C.; Zhang, L. A decision procedure for propositional projection temporal logic with infinite models. *Acta Inform.* **2008**, *45*, 43–78. [CrossRef]
16. Duan, Z.; Tian, C. A practical decision procedure for propositional projection temporal logic with infinite models. *Theor. Comput. Sci.* **2014**, *554*, 169–190. [CrossRef]
17. Yang, X.; Wang, X. A Practical Method based on MSVL for Verification of Social Network. In Proceedings of the 2021 8th International Conference on Dependable Systems and Their Applications (DSA), Yinchuan, China, 5–6 August 2021; pp. 383–389.
18. Zhao, L.; Wu, L.; Gao, Y.; Wang, X.; Yu, B. Formal Modeling and Verification of Convolutional Neural Networks based on MSVL. In Proceedings of the 2022 9th International Conference on Dependable Systems and Their Applications (DSA), Wulumuqi, China, 4–5 August 2022; pp. 280–289.
19. Wang, M.; Li, S. Formalizing Spark Applications with MSVL. In Proceedings of the International Workshop on Structured Object-Oriented Formal Language and Method, Singapore, 1 March 2021; Springer: Cham, Switzerland, 2020; pp. 193–204.
20. Luo, N.; Yu, Z.; Bei, Z.; Xu, C.; Jiang, C.; Lin, L. Performance modeling for spark using svm. In Proceedings of the 2016 7th International Conference on Cloud Computing and Big Data (CCBD), Macau, China, 16–18 November 2016; pp. 127–131.
21. Grossman, S.; Cohen, S.; Itzhaky, S.; Rinetzky, N.; Sagiv, M. Verifying equivalence of spark programs. In Proceedings of the Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, 24–28 July 2017; Part II 30; Springer: Berlin/Heidelberg, Germany; 2017; pp. 282–300.
22. Beckert, B.; Bingmann, T.; Kiefer, M.; Sanders, P.; Ulbrich, M.; Weigl, A. Relational equivalence proofs between imperative and MapReduce algorithms. In Proceedings of the Verified Software. Theories, Tools, and Experiments: 10th International Conference, VSTTE 2018, Oxford, UK, 18–19 July 2018; Revised Selected Papers 10; Springer: Berlin/Heidelberg, Germany; 2018; pp. 248–266.
23. Yin, J.; Zhu, H.; Fei, Y.; Fang, Y. Modeling and Verifying Spark on YARN Using Process Algebra. In Proceedings of the 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE), Hangzhou, China, 3–5 January 2019; pp. 208–215.
24. Baresi, L.; Bersani, M.M.; Marconi, F.; Quattrocchi, G.; Rossi, M. Using formal verification to evaluate the execution time of Spark applications. *Form. Asp. Comput.* **2020**, *32*, 33–70. [CrossRef]
25. de Souza Neto, J.B.; Martins Moreira, A.; Vargas-Solar, G.; Musicante, M.A. TRANSMUT-Spark: Transformation mutation for Apache Spark. *Softw. Testing, Verif. Reliab.* **2022**, *32*, e1809. [CrossRef]
26. Dietsch, D.; Heizmann, M.; Langenfeld, V.; Podelski, A. Fairness modulo theory: A new approach to LTL software model checking. In Proceedings of the Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, 18–24 July 2015; Proceedings Part I 27; Springer: Berlin/Heidelberg, Germany, 2015; pp. 49–66.
27. Brockschmidt, M.; Cook, B.; Ishtiaq, S.; Khlaaf, H.; Piterman, N. T2: Temporal property verification. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, 2–8 April 2016; Proceedings 22; Springer: Berlin/Heidelberg, Germany, 2016; pp. 387–393.
28. Cook, B.; Khlaaf, H.; Piterman, N. On automation of CTL* verification for infinite-state systems. In Proceedings of the International Conference on Computer Aided Verification, San Francisco, CA, USA, 18–24 July 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 13–29.
29. Cook, B.; Koskinen, E. Making prophecies with decision predicates. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Austin, TX, USA, 26–28 January 2011; pp. 399–410.
30. Duan, Z.; Yang, X.; Koutny, M. Framed temporal logic programming. *Sci. Comput. Program.* **2008**, *70*, 31–61. [CrossRef]
31. Wang, X.; Tian, C.; Duan, Z.; Zhao, L. MSVL: A typed language for temporal logic programming. *Front. Comput. Sci.* **2017**, *11*, 762–785. [CrossRef]