Formal Aspects of Computing

Using Formal Verification to Evaluate the Execution Time of Spark Applications

L. Baresi, M. M. Bersani, F. Marconi, G. Quattrocchi and M. Rossi

{luciano.baresi—marcellomaria.bersani—francesco.marconi—giovanni.quattrocchi—matteo.rossi}@polimi.it
Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano,
Via Golgi 42, 20133 Milano, Italy

Abstract. Apache Spark is probably the most widely adopted framework for developing big-data batch applications and for executing them on a cluster of (virtual) machines. In general, the more resources (machines) one uses, the faster applications execute, but there is currently no adequate means to determine the proper size of a Spark cluster given time constraints, or to foresee execution times given the number of employed machines. One can only run these applications and use her/his experience to size the cluster and predict expected execution times. Wrong estimation of execution times can lead to costly overruns and overly long executions, thus calling for analytic sizing/prediction techniques that provide precise time guarantees. This paper addresses this problem by proposing a solution based on model-checking. The approach exploits a Directed Acyclic Graph (DAG) to abstract the structure of the execution flows of Spark programs, annotates each node (Spark stage) with execution-related data, and formulates the identification of the global execution time as a reachability problem. To avoid the well-known state space explosion problem, the paper also proposes a technique to reduce the size of generated abstract models. This results in a significant decrease in used memory and/or verification time making our approach feasible for predicting the execution time of Spark applications given the resources available. The benefits of the proposed reduction technique are evaluated by using both Timed Automata and Constraint LTL over clocks (CLTLoc) logic to formally encode and analyze generated models. The approach is also successfully validated on some realistic case studies. Since the optimization is not Spark-specific, we claim that it can be applied to a wide range of applications whose underlying model can be abstracted as a DAG.

Keywords: Formal Verification; Big Data; Metric Temporal Logic; Timed Automata

Correspondence and offprint requests to: L. Baresi, M.M. Bersani, F. Marconi, G.Quattrocchi and M.Rossi

1. Introduction

Big-data applications are becoming more widespread, and thus frameworks dedicated to the processing of large datasets, like Hadoop[1], Spark[2], and Flink[3], are becoming popular. These frameworks allow users to process massive amounts of data over clusters of (virtual) servers: the business logic and input dataset are partitioned on different servers to carry out the computation concurrently by applying the same operations in parallel. While the different computation models are clear, the use of resources still needs attention. These frameworks do not provide (sophisticated) self-adaptive capabilities to allow for the optimal use of resources: they often adopt a greedy approach and tend to offer the best qualities of service (e.g., execution time) given the resources at hand. To find a compromise between execution times and foreseen costs, there exists solution that extends these systems with dynamic resource management capabilities [BQ18]. Nonetheless, solutions that can help estimate qualities of service given the size of the cluster —or that can help size the cluster given foreseen qualities of service— would be greatly appreciated. Some solutions already exist (e.g., [GRB+17, LHW+14]), but often the estimations are only based on experience and empirical assessments.

While the problem is general, this paper concentrates on Spark, probably the most widely used solution for big-data batch applications, and proposes an analytical approach that is based on the use of model checking as rigorous analytic method. Being batch applications, one can only consider their execution time as quality of service, and thus address the interplay between used resources and execution speed (or degree of parallelism). The actual size of supplied data and the number of machines used to run the applications define the execution time required by the framework to provide the results. In this context, the maximum allowed execution time is defined in terms of *deadline*s [KA10]. Since the actual execution times are only known at the end of computations, the proper amounts of resources needed to fulfill deadlines are often "guessed" using experience and domain knowledge.

The execution model of Spark applications is based on *stages* and *tasks*. A task is an atomic unit of computation executed on a partition of the dataset by means of a single CPU core. A stage comprises different tasks that work on diverse data chunks and produces a transformed dataset when all its tasks complete their execution. Different stages can execute at the same time if they do not depend on each other, and thus tasks that belong to different stages can be executed concurrently on different CPU cores. The more resources (CPU cores) one allocates to an application, the higher its parallelism (and the execution speed) can be.

The proposed modeling approach abstracts each specific execution of a Spark application as a direct acyclic graph (DAG), where executed stages act as nodes, the arcs between them identify data dependencies, meaning that the target stage uses what is produced by the source, and branches indicate the parallelism among the different stages. Each node is annotated with its profiled/inferred execution time, the ratio between the sizes of inputs and outputs and other execution-specific data. Obtained graphs are translated into both Timed Automata (TA) [AD94] and the Constraint LTL over clocks (CLTLoc) logic [BRS16]. TA are well-known in the area of formal verification and the analysis of their temporal properties is supported by many consolidated tools, such as Uppaal [BDL+06]. CLTLoc is an extension of LTL whose atomic formulae can be arithmetical constraints over clock variables. It is semantically equivalent to TA [BRS17], and it can be efficiently solved by a bounded decision procedure based on SMT solvers [BRS16].

Some preliminary experimental results obtained through the presented approach are shown in [MQB+18], where a limited number of realistic Spark applications are analyzed through the verification of their associated CLTLoc models. This paper extends the initial results with: (a) the use of TA as additional formal machinery, with the intent of developing a multi-formalism estimation toolkit for Spark executions; (b) an optimization technique to reduce the size of generated formal models, and thus extend the family of analyzable applications; and (c) a more complete and thorough assessment of a wider set of Spark applications.

The multi-formalism estimation toolkit has been developed to address the main drawback observed in [MQB+18], that is, the high computational cost in terms of execution time required by the verification procedure. The proposed solution allows one to select either CLTLoc or TA given the chosen verification approach. The implemented decision procedure for CLTLoc is based on the bounded satisfiability of formulae, whereas the most common and standard tools for the analysis of TA implement verification algorithms that are based on the exhaustive exploration of the state space of the system. Bounded satisfiability stems from the

[1] hadoop.apache.org
[2] spark.apache.org
[3] flink.apache.org

well-known Bounded Model-Checking [BCC+03] (BMC) approach and requires that the model of interest (e.g., the one that violates a given property) be of a limited size s to restrict the search to only those models that are smaller than s. In many cases, BMC reduces the cost of the state space exploration, but often the size of the candidate model (if it exists) is unlikely to be known in advance, before running the verification. Sometimes, state space exploration is still too large to allow for the analysis of all candidate models (completeness). The guarantee of completeness is instead inherent in algorithms that exhaustively explore the state space of the system, such as the one implemented in Uppaal. On the other hand, such algorithms tend to suffer from the state explosion that leads to memory saturation even for models of relatively small size. For this reason, famous verification toolkits, such as NuSMV [CCG+02], often provide users with several tools that implement different verification approaches.

The decision procedures for TA and CLTLoc have a PSPACE complexity and, even if they include optimizations that speed up the resolution of the verification problem, analyzing relatively small models might require hours. This evidence motivates the second novel contribution of this paper: a technique that reduces the overall time and memory consumption of the verification procedure when ordered computations are considered —such as, for instance, those realized by Spark applications. The technique works on the DAGs that abstract Spark computations and not on the underlying decision procedures dedicated to solve CLTLoc formulae or to verify TA. The optimization is not Spark-specific, and we claim that it can be applied to a wide range of applications whose underlying model can be abstracted as a DAG. The technique exploits a property of partially ordered sets (posets), proven in Dilworth's partition theorem: *a finite poset can be partitioned into n independent posets (called* chains*), where n is the cardinality of the largest set of incomparable elements.* Intuitively, in the context of this work, n is the maximum number of stages that can be active simultaneously. The following two facts motivate the adoption of the property in the verification procedure. Firstly, all Spark stages behave in the same way: the tasks are executed in batches whose size depends on the number of available CPU cores, until all tasks are processed. Hence, the computation of every stage can be represented by means of the same *model template*, which is then instantiated properly with different values for its parameters. Secondly, every chain of the DAG can be described by means of a unique set of CLTLoc formulae (or a single TA) that describes all the stages in the chain, instead of many similar copies of the same template, each one modeling a single stage.

Our experimental evidence is promising and demonstrates that obtained estimated durations are close to the actual durations when applications are executed on real clusters. The new experiments, with respect to those presented in [MQB+18], are based on well-known benchmark applications such as the machine learning algorithm *SVM* [Bur98] and some benchmark applications from the TPC-H suite[4]. The experiments also include an implementation of a community detection algorithm for large graphs called Louvain[5]. The accuracy of our models leads to an error that is less than 11% with respect to the a-posteriori measured execution time. We have also compared the optimized and non-optimized versions of the models and the results show that the verification of the optimized models is up to 10 times faster than the verification of the corresponding non-optimized ones. Used memory follows a similar pattern. Experimental results witness that the precision of TA models and CLTLoc ones is the same.

The paper is organized as follows. Section 2 provides an overview of Apache Spark and recalls the definition of CLTLoc and TA. Section 3 presents the way we abstract Spark computations determined and shows CLTLoc and TA models. Section 4 outlines the theoretical foundations of Dilworth's theorem and the optimized models. Section 5 reports on the experimental activities carried out to determine the accuracy of the models and to evaluate the impact of the optimization. Section 6 surveys related approaches, and Section 7 concludes the work.

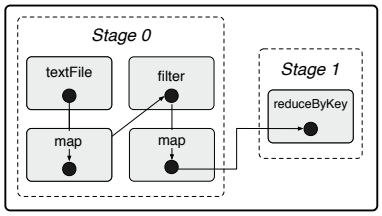
2. Background

To make the paper self-contained, this section summarizes the key characteristics of Apache Spark, Constrained LTL over clocks (CLTLoc) and Timed Automata (TA).

[4] `http://www.tpc.org/tpch/`
[5] `https://sourceforge.net/projects/louvain/`

```
spark.textFile("path/to/file")
.map(x => x.split(":"))
.filter(x => x(2) != "false")
.map(x => (x(0), x(1).toInt))
.reduceByKey(_ + _)
.collect()
```

(a) Code



(b) DAG

textFile	map	filter	map	reduceByKey
'a:2:true'	['a','2','true']	['a','2','true']	('a', 2)	('a', 7)
'e:3:true'	['e','3','true']	['e','3','true']	('e', 3)	('e', 3)
'a:3:false'	['a','3','false']	['u','1','true']	('u', 1)	('u', 6)
'u:1:true'	['u','1','true']	['u','5','true']	('u', 5)	
'o:4:false'	['o','4','false']	['a','5','true']	('a', 5)	
'u:5:true'	['u','5','true']			
'a:5:true'	['a','5','true']			

(c) Data transformed from strings, to arrays and finally to key-value pairs.

Fig. 1. Example of Spark application.

2.1. Apache Spark

Spark is usually deployed on a cluster of servers and exploits a master/worker architecture. The *master* schedules operations for execution in the cluster by assigning parts of the computation to each *worker*. The main abstraction in Spark is the *RDD* (resilient distributed dataset), i.e., an immutable and fault-tolerant collection of records. To foster parallel computations, RDDs are split into chunks, called *partitions*, and stored in a distributed way. RDDs can be saved and processed in memory to improve performance through reuse, making Spark particularly efficient for the execution of iterative algorithms (e.g., machine learning and graph computations).

Spark uses DAGs to organize the operations that compose a Spark application/job[6]. Note that the DAG generated by Spark is created incrementally at runtime and it is not the control flow graph of the application. A node in a Spark DAG is called *stage*. A stage is an abstraction that represents a sequence of operations that starts with at least one *wide transformation*, possibly followed by a finite sequence of *narrow transformations*. A *transformation* is an operation that reads at least one RDD and produces an output RDD. Narrow transformations (e.g., *map, filter*) fully exploit data locality and can be aggregated together (chaining) if they occur in a sequence, as they only operate on the local partitions. Conversely, the data required to compute wide transformations (e.g., sortByKey, reduceByKey) may reside in different, distributed partitions. For this reason, wide transformations always generate data shuffling: the workers that execute a wide transformation exchange some data partitions among them.

The DAG defines the precedence relation among the stages of an application: two stages are connected if one uses the data produced by the other to progress. Therefore, *a stage can only be executed if, and only if, all of its predecessors are completed.*

Once a stage is scheduled by the master, Spark defines the set of parallel *tasks* that need to be executed. A task executes all the transformations that compose a stage over a single partition of the input RDD. The operations that belong to a stage are executed by tasks that are executed on the worker machines. Therefore, a task is an atomic unit of computation executed by a single core and it is scheduled only when a CPU core of a worker becomes available (i.e., completes the execution of a previously scheduled task).

Finally, narrow and wide transformations are executed lazily. The actual distributed processing starts when an *action* (e.g., *take, collect*) is executed. Actions are used for returning a result to the Spark program that can be reused by other Spark computations.

[6] A Spark application is composed of one or more jobs. Each job has its own DAG, and jobs are executed sequentially. In this paper, without any loss of generality, we refer to the application DAG as the sequence of the DAGs of the jobs that constitute the application.

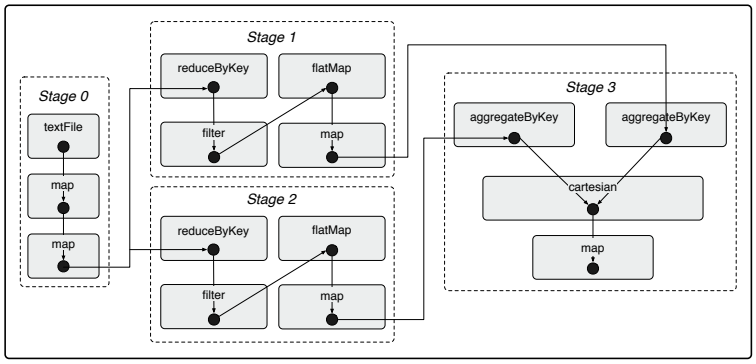


Fig. 2. A more complex application.

To exemplify the Spark processing model, Fig. 1a shows Scala code of an example Spark application that performs a simple aggregation over a dataset read from a text file where each line contains a vowel, a number, and a Boolean separated by colons. The program sums the numbers that correspond to the same vowels and which are not *false*. To this end, it chains different operations whose effect is shown in Fig. 1c: i) a *map* transforms each line into an array of strings by splitting it at each colon; ii) a *filter* discards the unnecessary arrays (those labeled with *false*); iii) a second *map* converts the remaining arrays into key-value pairs, each one composed of a vowel (the key) and a number (the value); iv) transformation *reduceByKey* is used to sum the numbers that share the same key; finally, v) action *collect* returns the dataset. Figure 1c shows how an example dataset is transformed at each step.

Figure 1b shows how the example program of Fig. 1a is executed by Spark. Each grey rectangle inside a stage is an RDD that is produced by the associated operation; the arrows define the ordering relation between the transformations in *Stage 0* and *Stage 1* and the precedence between the two stages: a sequential DAG of two nodes. Due to the lazy evaluation of transformations, nothing happens until *collect* is executed. At that moment, Spark creates the DAG of Fig. 1b. Since *map* and *filter* do not require data shuffling, the first four operations are grouped in a single stage (*Stage 0*). Conversely, *reduceByKey* requires shuffling because tuples with the same key are not guaranteed to be all in the same partition. For this reason, *Stage 1* is created and depends on *Stage 0*. So, it can be scheduled only when *Stage 0* has completed its execution.

Figure 2 shows a more complex Spark application composed of 4 stages. *Stage 0* reads a dataset from a text file and transforms it with two *map* operations. *Stage 1* and *Stage 2*, which follow the execution of *Stage 0*, can be executed in parallel (the actual scheduling depends on different factors such as data locality). Finally, *Stage 3* merges the two datasets produced by *Stage 1* and *Stage 2* using a cartesian product.

2.2. Constraint LTL over clocks

CLTLoc [BRS16] is an extension to LTL that allows clocks to occur in atomic formulae. Being based on LTL, the semantics of the logic is defined in terms of ordered *positions* of time, which are given by the elements of set $\mathbb{N}$. CLTLoc clocks behave in the same manner as clocks of TA, hence their value is taken from a dense domain (such as $\mathbb{R}$). For instance, consider two clocks x_1 and x_2. A possible (portion of) sequence of values for the clocks, assuming that both x_1 and x_2 start with an initial value 0, can be $\{0,0\},\{0.7,0.7\},\{1.1,0\},\{0,2.5\},\ldots$. All clocks progress with the same amount of (real) time between any two consecutive positions unless they are reset, in which case their value is set to 0. Clocks can be compared with constants in formulae of the form $x \sim c$ (where c is a natural number and $\sim \in \{<,=\}$). Since the logic does not have resets, clocks are "reset" when their value is 0—i.e., when $x = 0$ holds. Atomic formulae can also predicate over arithmetical variables (counters) that have no semantic restrictions. Therefore, the value of the counters in a given position is determined by the arithmetical formulae that hold at that time position and there are no restrictions on their value nor on their behavior over time. The logic exploits a special modality X (arithmetical next), introduced in [DD07], with the following meaning: if y is an arithmetical variable, Xy is the value of y in the next position of time. By means of X, the increment of y by 1 is expressed by the formula $\mathrm{X}y = y + 1$.

Let W be a set of arithmetical variables and $\Theta(W)$ the set of all the quantifier-free Presburger formulae over terms α of the form y or $\mathrm{X}y$, where y is an element of W. For instance, the formula $y_1 + \mathrm{X}y_2 = \mathrm{X}y_3 + y_4$ is a formula in $\Theta(W)$, when W includes y_1, y_2, y_3 and y_4. Let AP be a set of atomic propositions, K be a set of clock variables, CLTLoc formulae ϕ are defined as follows:

$$\phi := p \mid x \sim c \mid \theta \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi$$

where $p \in AP$, $x \in K$, $\theta \in \Theta(W)$, $c \in \mathbb{N}$, $\sim \in \{<, =\}$, and $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since" operators of LTL. Operators $\mathbf{F}$ ("eventually"), $\mathbf{G}$ ("globally"), and $\mathbf{P}$ ("previously") are defined through the customary abbreviations: $\mathbf{F}\phi = \top\mathbf{U}\phi$, $\mathbf{G}\phi = \neg\mathbf{F}(\neg\phi)$, and $\mathbf{P}\phi = \top\mathbf{S}\phi$.

An *interpretation* of a formula is a pair (π, σ) defined over the naturals, where $\pi(i) \subseteq \wp(AP)$ (where $\wp$ is the powerset operator), and $\sigma(i, x)$, $\sigma(i, y)$ are, respectively, the value of clock x and variable y at position i. The semantics of the evolution of time adopted for CLTLoc is strict, namely the value of a clock must strictly increase in two adjacent time positions, unless it is reset (i.e., for all $i \in \mathbb{N}$, $x \in K$, it holds that $\sigma(i+1, x) > \sigma(i, x)$, unless $\sigma(i+1, x) = 0$ holds). To ensure that time strictly progresses at the same rate for every clock, σ must satisfy the following condition: for every position $i \in \mathbb{N}$, there exists a "time delay" $\delta_i > 0$ such that for every clock $x \in K$:

$$\sigma(i+1, x) = \begin{cases} \sigma(i, x) + \delta_i & \text{progress} \\ 0 & \text{reset } x \end{cases}$$

A clock assignment is such that $\sum_{i \in \mathbb{N}} \delta_i = \infty$, that is time is always progressing. The semantics of CLTLoc is defined as for LTL, except for formulae $x \sim c$ and θ. Let A_W be the ordered set of all terms α, of the form y and $\mathrm{X}y$, with $y \in W$, and let n be its cardinality; for each $\alpha_j \in A_W$, the depth $|\alpha_j|$ is such that $|\alpha_j| = 0$ if $\alpha_j = y$, and $|\alpha_j| = 1$ if $\alpha_i = \mathrm{X}y$, for some $y \in W$. For every $\theta \in \Theta(W)$, $\theta[k_0, \ldots, k_{n-1}]$ is the valuation of θ through the values $k_0, \ldots, k_{n-1} \in \mathbb{Z}$, which is obtained by replacing each term α_j occurring in θ with value k_j. Let $t(\alpha_j) = y$ if α_j is either y or $\mathrm{X}y$. Hence, for every $i \geq 0$:

$$\begin{array}{ll} (\pi, \sigma), i \models x \sim c & \textbf{iff } \sigma(i, x) \sim c \\ (\pi, \sigma), i \models \theta & \textbf{iff } \theta[\sigma(i + |\alpha_0|, t(\alpha_0)), \ldots, \sigma(i + |\alpha_{n-1}|, t(\alpha_{n-1}))] \end{array}$$

For instance, let W be the set $\{y_1, y_2\}$ and σ be such that $\sigma(i, y_1) = 0$ holds for all $i \geq 0$, and $\sigma(0, y_2) = 1$, $\sigma(1, y_2) = -1$ also hold. Then, if $\theta = y_1 > \mathrm{X}y_2$ and $i = 0$, it holds that $\theta[\sigma(i+0, y_1), \sigma(i+1, y_2)] = (0 > -1) = \mathit{true}$ (indeed, 0 is the depth of y_1, and 1 is the depth of $\mathrm{X}y_2$), and $y_1 > \mathrm{X}y_2$ holds at position 0; similarly, if $\theta = y_2 < \mathrm{X}y_1$ and $i = 1$, then $\theta[\sigma(i+1, y_1), \sigma(i+0, y_2)] = (-1 < 0) = \mathit{true}$ holds, and $y_2 < \mathrm{X}y_1$ holds at position 1. Hence, $(\pi, \sigma), 0 \models y_1 > \mathrm{X}y_2$ and $(\pi, \sigma), 1 \models y_2 < \mathrm{X}y_1$ hold. If ϕ is a formula, interpretation (π, σ) is a *model* for ϕ if $(\pi, \sigma), 0 \models \phi$ holds.

The satisfiability problem CLTLoc is decidable [BRS16] and can be computed through a Bounded Satisfiability Checking approach [BRS16, BPKR16]. In Sect. 3, we use CLTLoc with counters, hence embedding CLTL [DD07], to define the model of Spark computation. We show that, since the value of arithmetical variables is bounded by some value that depends on the problem instance, we can still use the approach introduced in [BRS16, BPKR16].

2.3. Timed Automata

Timed Automata (TA) are a well-known formalism; for this reason, only the main definitions are recalled. Let $\Gamma(K)$ be the set of *clock constraints* defined as $\eta \stackrel{\text{def}}{=} x \sim c \mid \neg\eta \mid \eta \wedge \eta$, where $\sim \in \{<, =\}$, $x \in K$ and c is a natural number; and let $\Gamma(W)$ be the set of *variable constraints* ζ defined as $\zeta \stackrel{\text{def}}{=} y \sim d \mid y \sim y' \mid \neg\zeta \mid \zeta \wedge \zeta$, where y and y' are variables in W and d is an integer. Let $\mathit{assign}(W)$ be the set of assignments of the form $y := \mathit{exp}$, where $y \in W$ and exp is a Presburger formula over W and the integers (e.g., $y_1 + 3y_2 - 1$ is a Presburger formula).

Given a set of atomic propositions AP, a *timed automaton* is a tuple $\langle Q, q_0, v^0, I, L, T \rangle$, where: Q is a finite set of locations, $q_0 \in Q$ is the initial location, v^0 is a function that assigns an integer value to each variable in W, $I : Q \to \Gamma(K)$ is an invariant assignment function, $L : Q \to \wp(AP)$ is the labeling function and $T \subseteq Q \times Q \times \Gamma(K) \times \Gamma(W) \times \wp(K) \times \wp(\mathit{assign}(W))$ is a finite set of transitions.

The standard semantics of a TA [AD94] is given in terms of *configurations*, i.e., pairs (q, v) defining the current location of the automaton and the value of all clocks and variables, where $q \in Q$ and v is a function over $K \cup W$ that assigns every clock of K with a real value and every variable of W with an integer. A configuration change from (q, v) to (q', v') can happen because either a transition in T is taken or because time elapses. The adopted semantics is standard, and the formal definition is omitted. Let A be a finite set of assignments in $assign(W)$ and S be a set of clocks in K. Informally, a *discrete transition* determined by the tuple $(q, q', \eta, \zeta, A, S)$ can be fired when: (i) the clock and the variable values, defined by the valuation v, satisfy η and ζ and v' satisfies the invariant $I(q')$; (ii) $v'(x) = 0$ holds for all clocks in S, and $v'(x) = v(x)$ otherwise (time does not advance in a discrete transition); (iii) finally $v'(y) = exp(v)$ holds, for all the variables in $y \in W$, where $y := exp$ is an assignment of A and $exp(v)$ is the value of exp obtained by replacing the occurrences of the variables in exp with the variable values defined by v. In the case of *time (elapsing) transitions*, (q, v) and (q', v') are such that $q = q'$, all the variables $y \in W$ retain the value, $v'(x) = v(x) + \delta$, for all $x \in K$ and for some $\delta \geq 0$. In addition, the invariant $I(q)$ is satisfied by all the assignments of the clocks from v to v'.

The model-checking of TA is a well-known problem in the area of formal verification and efficient decision procedures are implemented in several tools, such as Uppaal [BDL+06], Kronos [BDM+98], RED [Wan04] (many others can be found in [WDR13]). The widely adopted specification language is Computation Tree Logic (CTL). The definition and the semantics of CTL and the timed version TCTL are standard and can be found in [ACD93, Bou09] along with some decidability and complexity results on the model-checking problem of TA. In this work, TCTL is used to express the time-bounded reachability of specific configurations of the Spark computation, through the operator $\mathbf{F}_{<d}$. Given a TCTL formula ϕ, $\mathbf{F}_{<d}(\phi)$ means that ϕ holds in the future within d time units from the current instant.

3. Modeling

The section outlines the abstract model of Spark computations (a preliminary version of the model was introduced in [MQB+18]).

3.1. Problem statement

The model of Spark computations only refers to the temporal properties of applications and their functional aspects are not considered. The model represents the *temporal ordering* and the *temporal distance* among the events that occur in a computation. The relevant events are the beginning and the end of stages and of batches of tasks that occur throughout the computation of DAGs. The temporal ordering among the events is determined by the dependency relation of the DAG whereas the temporal distance between the beginning and the end of a task is defined by the duration of the task on a single core.

3.1.1. Assumptions

The following assumptions are considered in the design of the abstract model.

- Each application is deployed on a homogeneous cluster of identical machines;
- The workload managed by the cluster that executes the application is not subject to oscillations that might alter the execution of the running tasks (i.e., the performance of the cluster is stable and does not vary over time);
- Network overhead is predictable;
- Applications are CPU-bounded;
- The input datasets provided to the application are homogeneous; (i.e., the so-called data skewness is negligible).

The previous assumptions reflect realistic scenarios. Applications are often executed on clusters of identical machines that comply with a specific Service Level Agreement (in terms of number of cores, characteristics of the CPUs and total amount of available memory). Cluster providers allow users to purchase the required computational power in advance of the actual application deployment and guarantee stable performance throughout the execution of the application.

Private clusters, for instance, are a solution adopted in case of highly demanding applications when ad-hoc techniques to manage variable network performance fail. In addition, a vast literature on network guarantees is available, and several works in this area propose techniques that can be adopted by cloud providers to achieve stable performance, i.e., a predictable overhead. For instance, *Silo* [JSBM15] is "a system that allows datacenter operators to give tenants guaranteed bandwidth, delay and burst allowances for traffic between their VMs". Cloud providers, very often, allow customers to rent machines that are close to one another, in so-called "availability zones". Clusters that consist of a single availability zone provide stronger guarantees on bandwidth performance and latency than those that are geographically scattered.

Morover, as stated by Ousterhout et al. [ORR+15], Spark applications are usually bound to CPU allocation, which becomes the main bottleneck. Indeed, they quantify that network optimization reduces execution time by just 2%. Note that one of the main features and advantages of Spark is that it favors in-memory processing, and users can even disable the use of disk.

The skewness of the dataset affects the performance of the application and, in particular, it influences its duration. The more the dataset is skewed, the more the task duration oscillates around a mean value, as tasks process data partitions that are very diverse. The presence of skew datasets actually indicates the lack of implemented procedures limiting the skewness and poses a threat against efficiency. Best practices, in fact, advise against processing skew datasets to prevent the performance degradation and an inefficient use of the cluster. According to Ousterhout et al. [ORR+15], skewed data can make applications 20% slower. However, there are approaches in literature that focus on techniques to automatically split data into a more homogeneous data partitioning [YCH15], or on controlling applications affected by skewed data at runtime [BQ18]. For these reasons, the skewness of the dataset can be considered to be negligible, as an effective data processing is presupposed.

As a corollary, since the cluster is homogeneous, the network overhead is predictable and every Spark task is executed on a single core, task duration is an abstraction of a complex execution, that includes not only the time needed by the cluster to execute a task but also other phenomena, such as the network overhead (i.e., *shuffling* read/write) and possible performance loss that are caused, for instance, by data skewness when no optimization of input data can be performed.

In addition, cores in the system can be abstracted as part of a single machine, i.e., they are not distributed across the machines of the cluster, while actual task durations capture the difference between local and remote executions. The main difference between having many machines with a single core and having a single machine with many cores is that, in the former case, data are transfered either in memory or using the local network interface, whereas, in the latter, they require remote connections. Given the assumptions, these predictable overheads can be considered as a fraction of the total duration of tasks. Therefore, modeling the behavior of a single machine is not relevant and the tasks of a stage are executed in batches, whose size depends only on the total number of cores available in the system.

3.1.2. DAG-based model and feasible executions

Let D be a DAG (S, E) where S is a finite set of N stages $\{S_0, \ldots, S_{N-1}\}$ and E is a set of edges, that is, a subset of $S \times S$ that represents the precedence relation among the stages. For instance, $(S_1, S_2) \in E$ means that S_2 *subscribes* to the output of S_1—hence, S_1 is a predecessor of S_2. Let T_i be a finite set of tasks associated with S_i such that any pair of tasks $(T_i, T_{i'})$ are disjoint for any $0 \leq i, i' < N$ (with $i \neq i'$) and let $\bar{T}$ be the set $\bigcup_i T_i$. Hereafter, consider variable i to be $0 \leq i < N$.

An *execution* η for D with tasks in $\bar{T}$ is a finite sequence of H tuples $e^0, e^1, \ldots, e^{H-1}$ of the form $e^j = (T_0^j, \ldots, T_{N-1}^j)$, called *execution steps*, for some $H > 2$. Hereafter, consider variable j to be $0 \leq j < H$. T_i^j is a—possibly empty—subset of T_i, and it contains the *active* tasks of S_i at step j when T_i^j is non-empty. The active tasks in T_i^j are executed in parallel; so, every T_i^j is a *batch* of running tasks. Executions satisfy the following constraints:

- For every stage S_i, each task in T_i appears in the execution exactly once; also, if some task of T_i occurs at step j, then all tasks associated with all stages $S_{i'}$ preceding S_i, with respect to E, occur before j;
- For each execution step e^j there exists at least one set of active tasks.

For any stage S_i in S, let τ_i be a strictly positive constant in $\mathbb{R}$ that defines the time needed to compute a generic task of T_i (since all the tasks in T_i are homogeneous, their execution requires the same amount of time). Let I and I' be two convex and bounded sets in $\mathbb{R}$. I *precedes* I' when all the elements in I are

strictly smaller than all the elements in I'. Given an execution η for D, define function $active(t)$ specifying the set of active tasks of $\bar{T}$ at any time instant t, such that for every $t \in \mathbb{R}$:

- If a batch T_i^j is active at t, i.e., $T_i^j \in active(t)$, then there is an interval I of τ_i time units, including t, where T_i^j is active and no task of T_i^j is active in any time instant t' not belonging to I;
- Every batch T_i^j is eventually active, i.e., there exists a time instant t such that $T_i^j \in active(t)$;
- If batch T_i^j occurs before batch $T_i^{j'}$ in η (i.e., $j < j'$), then the interval of time where T_i^j is active precedes the interval of time where $T_i^{j'}$ is active.

Let $p > 0$ be the number of available CPU cores in the cluster. An execution $\eta = e^0, e^1, \ldots, e^{H-1}$ for D is *p-feasible* if $|active(t)| \leq p$, for all $t \geq 0$, i.e., in every time instant t the number of active tasks does not exceed the number of cores. The *time span* $ts(\eta)$ of η is defined as the maximum time instant where at least one task in active. The value $ts(\eta)$ is always finite because all the tasks of all the stages are executed by means of a finite number of execution steps and the durations of the tasks are finite.

3.1.3. Feasibility problem statement

Let D be a DAG (S, E) of N stages, let T_i, τ_i and p be defined as before and let d be a strictly positive integer. A d-bounded solution (or, simply, solution) of the feasibility problem for D with tasks in $\bar{T}$ is a p-feasible execution $\eta = e^0, e^1, \ldots, e^{H-1}$ such that $ts(\eta) < d$. Let FD be the set of values $\{d : \exists \eta \ ts(\eta) < d\}$ of the feasible deadlines, i.e., the set of all the possible deadlines d such that there exists a feasible execution whose duration is less than d. The *minimum feasible deadline* (mfd) is the minimum of FD.

3.2. CLTLoc model

This section presents the CLTLoc model that allows for the automated resolution of the previously described deadline-feasibility problem.

To represent the set of possible executions defined by a DAG, the CLTLoc model makes use of finite sets of atomic propositions, discrete counters and clocks. For instance, $\texttt{runT}_i$ is the atomic proposition that represents that a batch of tasks of stage i is in execution, i being the index identifying the stage S_i. Propositions, counters and clocks are respectively used to:

(i) model the current status of the stages and their tasks (i. e., started, running or completed, as described in Sect. 3.2.1);

(ii) keep track of the number of CPU cores that are either available, or are allocated to run the active tasks (Sect. 3.2.3);

(iii) enforce temporal constraints on the different tasks (Sect. 3.2.4).

Corresponding to these aspects, three groups of formulae can be identified in the model, mainly associated with the different kinds of variables. All the formulae described hereafter, except (7a)-(7c) in Sect. 3.2.5, are implicitly universally quantified over time through the **G** temporal operator.

For the sake of readability, and when the context is clear, we abuse the notations $i \in S$ and $(j, i) \in E$ to indicate a stage $S_i \in S$ and the pair $(S_j, S_i) \in E$, respectively.

3.2.1. State formulae for the stages

A stage S_i can be either running (i. e., the atomic proposition $\texttt{runS}_i$ holds) or idle (i. e., $\neg\texttt{runS}_i$ holds). A stage becomes running—i.e., $\texttt{startS}_i$ holds—when there is at least one task that has started its execution and none of the tasks in T_i has been executed so far. If no tasks were executed then the number of tasks still to be processed, represented by discrete integer variable $\texttt{remTC}_i$, is equal to the total number of tasks that the stage has to execute ($|T_i|$). A stage terminates—i.e., $\texttt{endS}_i$ holds—when the last batch of tasks

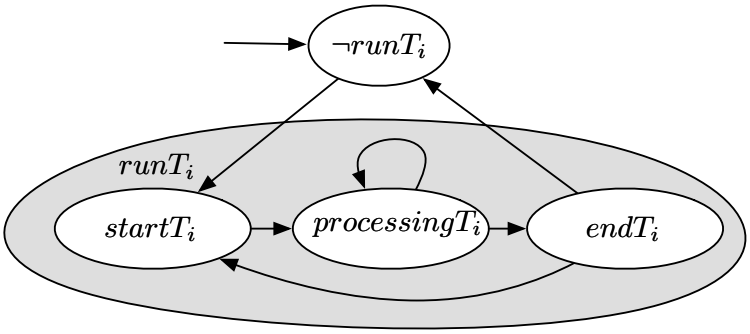


Fig. 3. Finite state machine representing the state evolution of a set of tasks.

terminates and there are no more tasks to be processed—i. e., when $\texttt{endT}_i$ holds and $\texttt{remTC}_i$ is equal to 0. This property is modeled in Formula (1).

$$\bigwedge_{i \in S} \begin{pmatrix} (\texttt{startT}_i \wedge \texttt{remTC}_i = |T_i| \iff \texttt{startS}_i) \wedge \\ (\texttt{endT}_i \wedge \texttt{remTC}_i = 0 \iff \texttt{endS}_i) \end{pmatrix} \tag{1}$$

A stage is completed (i.e., $\texttt{completedS}_i$ holds) if it has already been terminated in the past (i.e., there is a time position before the current one in which $\texttt{endS}_i$ held); it is enabled (i.e., $\texttt{enabledS}_i$ holds) when all the predecessor stages S_j, such that (S_j, S_i) belongs to E, have been completed.

$$\bigwedge_{i \in S} \begin{pmatrix} (\texttt{completedS}_i \iff \mathbf{P}(\texttt{endS}_i)) \wedge \\ (\texttt{enabledS}_i \iff \bigwedge_{j \in S,\ (j,i) \in E} \texttt{completedS}_j) \end{pmatrix} \tag{2}$$

3.2.2. State formulae for tasks

The behaviour of each batch of tasks is summarized in Fig. 3. In order for the batch to start processing ($\texttt{runT}_i$ becomes true), the stage must be enabled (i. e., $\texttt{enabledS}_i$ holds), and some conditions on the resources (explained later) must hold. Every batch is $\texttt{processingT}_i$ in all time instants strictly included between the beginning of the batch (in which $\texttt{startT}_i$ holds) and the end of the batch (in which $\texttt{endT}_i$ holds). Atom $\texttt{processingT}_i$ is used here for illustrative purposes and it corresponds to the state where $\texttt{runT}_i \wedge \neg\texttt{startT}_i \wedge \neg\texttt{endT}_i$. This execution cycle can be repeated many times depending on the available resources and the number of tasks to be executed. The CLTLoc formulae capturing the state machine of Fig. 3 are shown below in Formula (3).

Formula (3a) specifies the necessary conditions that must be true when a batch of tasks starts. When $\texttt{startT}_i$ holds then (i) the execution cannot be finished at the same time (i. e., $\neg\texttt{endT}_i$ must hold), (ii) in the previous time position, the stage was enabled to run and (iii) a new batch cannot start (i. e., $\neg\texttt{startT}_i$ holds) until the termination of the current one. Formula (3b) imposes that any execution of a batch of tasks is started with $\texttt{startT}_i$ and ends with $\texttt{endT}_i$, respectively; and that if a batch is running, then the corresponding stage is running at the same time. Similarly to Formula (3a), Formula (3c) defines the necessary conditions so that $\texttt{endT}_i$ holds. The termination of a batch of tasks imposes that $\neg\texttt{endT}_i$ holds since the time position where the current batch was started. Notice that formula $\phi\mathbf{R}\psi$ is the abbreviation of $\neg(\neg\phi\mathbf{U}\neg\psi)$. Intuitively, $\phi\mathbf{R}\psi$ holds when either ψ holds forever, or ψ holds until a time position where ϕ and ψ hold together. Moreover, $\texttt{orig}$ is the abbreviation of $\neg\mathbf{Y}(a \vee \neg a)$, that is $\texttt{orig}$ holds only at time position 0.

$$\bigwedge_{i \in S} \begin{pmatrix} (\texttt{startT}_i \Rightarrow \texttt{runT}_i \wedge \neg\texttt{endT}_i \wedge \mathbf{Y}\texttt{enabledS}_i \wedge \mathbf{X}(\texttt{endT}_i \mathbf{R} \neg\texttt{startT}_i)) \wedge & (3a) \\ (\texttt{runT}_i \Rightarrow \texttt{runS}_i \wedge (\texttt{runT}_i\, \mathbf{S}\, \texttt{startT}_i) \wedge (\texttt{runT}_i\, \mathbf{U}\, \texttt{endT}_i)) \wedge & (3b) \\ (\texttt{endT}_i \Rightarrow \texttt{runT}_i \wedge \mathbf{Y}\neg\texttt{endT}_i\, \mathbf{S}\, (\texttt{orig} \vee \texttt{startT}_i)) & (3c) \end{pmatrix}$$

3.2.3. Counter-related formulae

Counter variables are used to define the constraints on system resources and the evolution of the tasks that are executed within a stage. For example, Formula (4) limits the number of cores that are allocated to execute the active tasks. In particular, the sum of the number of available (avaCC) and allocated cores is always equal to p, where $\texttt{runTC}_i$ is the cardinality of the set T_i^j for the current execution step j.

$$\sum_{i \in S} (\texttt{runTC}_i) + \texttt{avaCC} = p \tag{4}$$

Formula(5) links $\texttt{startT}_i$, $\texttt{runT}_i$ and $\texttt{endT}_i$ with the value of counters $\texttt{runTC}_i$ and $\texttt{remTC}_i$. In particular, Formula (5a) defines that a batch is running (i.e., $\texttt{runT}_i$ holds) whenever the value of $\texttt{runTC}_i$, which counts the active tasks, is strictly positive. Formula (5b) imposes that the number of the remaining tasks of a stage decreases during its execution, i.e., the value of $\mathrm{X}\texttt{remTC}_i$ (that is, the value of $\texttt{remTC}_i$ in the next time position) is not greater than the value of $\texttt{remTC}_i$ in the current position. Formulae (5c) and (5d) constrain the values of $\texttt{runTC}_i$ and $\texttt{remTC}_i$ to change only when a batch starts or terminates. Specifically, Formula (5c) imposes that a variation of the value of $\texttt{runTC}_i$ between two adjacent positions is a sufficient condition to make $\texttt{startT}_i$ or $\texttt{endT}_i$ true. Formula (5e) defines the relation between the variables $\texttt{remTC}_i$ and $\texttt{runTC}_i$. When a batch of tasks is ending, the number $\texttt{remTC}_i$ of remaining tasks (to be executed) in the batch is the difference between the number of remaining tasks in the preceding time position (i. e., value $\mathrm{Y}\texttt{remTC}_i$) and the number $\texttt{runTC}_i$ of tasks running in the batch being completed.

$$\bigwedge_{i \in S} \begin{pmatrix} (\texttt{runT}_i \Leftrightarrow \texttt{runTC}_i > 0) \ \wedge & (5a) \\ (\texttt{remTC}_i \geq \mathrm{X}\texttt{remTC}_i) \ \wedge & (5b) \\ ((\texttt{runTC}_i \neq \mathrm{X}\texttt{runTC}_i) \Rightarrow (\mathbf{X}\texttt{startT}_i \vee \texttt{endT}_i)) \ \wedge & (5c) \\ ((\texttt{remTC}_i \neq \mathrm{X}\texttt{remTC}_i) \Rightarrow \mathbf{X}\texttt{endT}_i) \ \wedge & (5d) \\ (\texttt{endT}_i \Rightarrow (\texttt{remTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{runTC}_i)) & (5e) \end{pmatrix}$$

3.2.4. Constraints on clocks

The processing duration of the batches of tasks is constrained in Formula (6) by means of a clock variable $\texttt{clk}_{\texttt{runT}_i}$, that is defined for each stage S_i. Specifically, $\texttt{clk}_{\texttt{runT}_i}$ measures the duration of the $\texttt{runT}_i$ phases, for every possible batch of stage S_i. Formula (6a) specifies first that $\texttt{clk}_{\texttt{runT}_i}$ is reset in the origin or every time a new batch of tasks starts running for node S_i. In addition, Formula (6b) limits the duration of the execution of a batch of tasks by imposing that the termination of the batch occurs when the value of clock $\texttt{clk}_{\texttt{runT}_i}$ is in the interval $[\tau_i - \epsilon, \tau_i + \epsilon]$.

$$\bigwedge_{i \in S} \begin{pmatrix} ((\texttt{clk}_{\texttt{runT}_i} = 0) \iff (\texttt{orig} \vee \texttt{startT}_i)) & (6a) \\ \left(\begin{array}{l} \texttt{runT}_i \Rightarrow \\ \left(\begin{array}{l} \texttt{runT}_i \wedge \neg \texttt{endT}_i) \mathbf{U} ((\texttt{clk}_{\texttt{runT}_i} \geq \tau_i - \epsilon) \wedge (\texttt{clk}_{\texttt{runT}_i} \leq \tau_i + \epsilon) \wedge \texttt{endT}_i) \\ \wedge (\texttt{remTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{runTC}_i) \end{array} \right) \end{array} \right) & (6b) \end{pmatrix}$$

3.2.5. Initialization

The initial condition of any modeled Spark application obeys the following constraints (recall that the next formulae are stated only for the first time position): (i) no tasks are running in the origin (Formula (7a)); (ii) for each node S_i, the number of remaining tasks is $|T_i|$, which is the total number of tasks to be processed by node i (Formula (7b)); (iii) the number of available cores avaCC is the total number of cores p (Formula (7c)).

$$\bigwedge_{i \in S} \begin{pmatrix} (\neg \texttt{runT}_i \wedge \neg \texttt{runS}_i) \ \wedge & (7a) \\ (\texttt{remTC}_i = |T_i|) \wedge (\texttt{runTC}_i = 0) & (7b) \end{pmatrix}$$

$$(\texttt{avaCC} = p) \tag{7c}$$

3.2.6. Problem reduction

Let `totaltime` be a clock variable that is only reset when the application starts (i.e., `totaltime` $= 0$ in the origin), the reachability of the termination of all the stages earlier than the deadline d is reduced to the satisfiability problem of the conjunction of all the previous CLTLoc formulae and the following one:

$$\mathbf{F}\left(\bigwedge_{i \in S} (\texttt{completedS}_i) \wedge \texttt{totaltime} < d\right)$$

3.3. TA model

This section presents the new and alternative model of Spark computations built with TA and defines the reduction of the p-feasibility problem to a model-checking one.

The TA-based model is introduced to show the generality of the optimization approach. Given a DAG, one could decide to solve the corresponding p-feasibility problem using either the CLTLoc, or the TA-based model. However, the two models are not meant to be equivalent from a formal language perspective (i.e., given the same DAG, the CLTLoc model and the TA-based one do not necessarily capture the exact same sets of timed words). Indeed, they have been developed to leverage the specific features of each formalism for the description of Spark computations, rather than to define a language-preserving translation of one model into the other (and vice versa). The reason for this is twofold. On the one hand, a model produced through a language-preserving translation (for example, from CLTLoc to TA) would not be optimized from the point of view of its size, or the efficiency of the formal verification activity. On the other hand, our goal is to show how the optimizations introduced in this paper follow from the features of the analyzed DAGs as captured by Theorem 4.1, rather than from the specific features of the underlying formalism. For these reasons, and from the fact that, at its core, each model is an abstraction of the actual system, there can sometimes be a slight difference in the results (i.e., the minimum feasible deadline *mfd* obtained through the verification activity) obtained with different modeling approaches, but such discrepancy is not significant.

In the TA-based model, every stage of a DAG is associated with a TA that describes the execution of the stage over time, and which is an instance of the template shown in Fig. 4. Then, given a computation represented through a DAG, the corresponding TA-based model is the so-called "network of TA" obtained through the parallel composition of the TA capturing the various stages.

The rest of this section briefly describes the features of the TA of Fig. 4. The TA-based model uses identifiers that were also introduced in the logical specification presented in Sect. 3 (e.g., $completedS_i$). However, the two models are independent and the same identifier in different models can capture different elements (for example, $\texttt{completedS}_i$ in CLTLoc identifies a propositional letter, but $completedS_i$ in TA corresponds to a Boolean variable); hence, we write the identifiers used in the TA-based model in italic.

The TA of Fig. 4 uses variables and clocks to capture various features of the stage. In particular, Boolean variable $completedS_i$ is used to represent when stage i has completed the computation of all the tasks of set T_i (it holds value true when all tasks have been processes, false otherwise); this allows us to capture the precedence relation among stages encoded in the DAG. Variable $avaCC$, instead, whose domain is $[0, p]$ (where p is total number of cores in the cluster, see Sect. 3.1), keeps track of the number of cores that are available during the computation. It is initially set to p, and changes whenever a core starts/completes a computation. Finally, clock c_i measures the duration of the computation of a single batch, and variable $ntasks_i$ (whose domain is $[0, |T_i|]$) counts the total number of completed tasks along the computation.

Locations $Init_i$ and $Idle_i$ model, respectively, the initial state of stage i, when it is waiting for the termination of all its predecessors, and the state in which stage i is waiting for some core to be available to execute a batch of tasks. The automaton also includes p locations 1-$task \ldots p$-$task$, where each location j-$task$, with $1 \leq j \leq p$, captures the fact that j tasks are being processed. A stage can start the computation when all its predecessors have completed their tasks (function $pre(i)$ indicates the set of predecessors of stage i). The transition from $Init_i$ to $Idle_i$ is taken only when guard $\bigwedge_{s \in pre(i)}(completedS_s)$ holds, which corresponds to checking the termination of all predecessors of stage i, and thus captures the dependency relation among stages; in addition, the transition sets the value of $ntasks$ to 0.

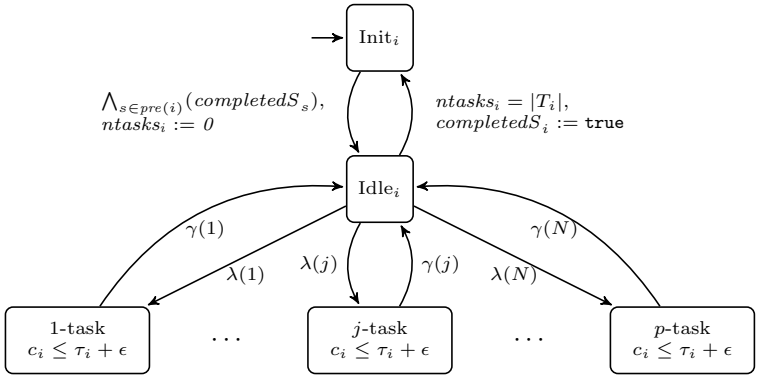


Fig. 4. Timed automaton modeling the i-th stage.

The computation of a stage terminates when the total number of elaborated tasks is equal to $|T_i|$. Therefore, the transition from $Idle_i$ to $Init_i$ modeling the termination of the stage is taken only when condition $ntasks = |T_i|$ holds. The effect of the transition is to set variable $completedS_i$ to `true`. When the stage is idle, the processing of a new batch of j tasks can start when:

- there are enough available cores to elaborate the batch, i.e., when the number of available cores is greater than or equal to j (this captures condition $|active(t)| \leq p$ of the problem statement); this is formalized by constraint $avaCC \geq j$;
- the number of tasks that still have to be completed is greater than or equal to j; this is captured by condition $ntasks_i \leq |T_i| - j$.

Every time a batch is started, clock c_i measuring the duration of the computation is reset and the number of available cores is decremented by the size of the batch (this is captured by assignment $avaCC := avaCC - j$). In Fig. 4, label $\lambda(j)$, with $1 \leq j \leq p$, stands for:

- the guard $avaCC \geq j \wedge ntasks_i \leq |T_i| - j$;
- the clock reset $c_i := 0$;
- the variable assignment $avaCC := avaCC - j$.

After τ_i time units the batch is completed, the stage becomes idle again, and the allocated cores are released. This behavior is modeled through the transition from location j-*task* to $Idle_i$, which is labeled with the assignment $avaCC := avaCC + j$ and a guard γ on c_i capturing the duration of the stage. To make the modeling of the task duration more adherent to the actual runtime behavior of Spark applications, the TA model tolerates a small deviation—of ϵ time units, with ϵ a constant—from τ_i of the task duration. More precisely, guard γ is defined as $c_i \geq \tau_i - \epsilon$, which expresses a lower bound on the task duration. In addition, to guarantee that the batch is processed in a finite amount of time, every location j-*task* is labeled with invariant $c_i \leq \tau_i + \epsilon$, which defines an upper bound on the task duration. Then, in Fig. 4, label $\gamma(j)$, with $1 \leq j \leq p$, indicates:

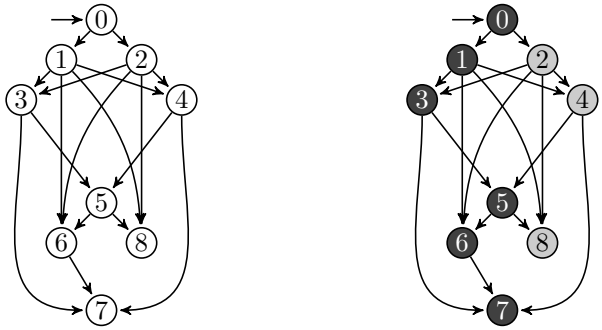
- the guard $c_i \geq \tau_i - \epsilon$;
- the assignment $avaCC := avaCC + j$.

3.3.1. Problem reduction

In the TA-based approach, the *p-feasibility* problem is reduced to the problem of checking that TCTL formula (8) holds for the network of TA.

$$\exists \mathbf{F}_{<d} \left(\bigwedge_{i \in S} completedS_i \right) \tag{8}$$

Indeed, Formula (8) holds if there exists an execution of the network of TA such that all stages are eventually completed within d time units from the origin (i.e., from the start of the computation).



(a) A DAG. (b) A labeled DAG.

Fig. 5. A DAG and a possible partitioning of its nodes into two chains shown in different colors.

4. Optimization

This section discusses the technique applied for reducing the overall time and memory consumption of the verification procedure. The technique is based on a general result—Dilworth's theorem—that holds for any partially ordered set (poset), hence also applies to the DAGs describing the executions of Spark applications. In fact, a DAG defines a partial order among the nodes: nodes that are linked with an edge are strictly ordered, whereas nodes that are not related through the predecessor relation (i.e., parallel nodes) are not. More precisely, this section presents the theoretical foundations required to state Dilworth's theorem and Dilworth's partitioning algorithm. Sections 4.1 and 4.2 show the modifications to the CLTLoc model in Sect. 3.2 and to the TA-based model in Sect. 3.2 taking advantage of the partitioning result.

The optimization exploits a specific property related to the *width* of posets, which is a general notion that applies also to Spark DAGs. Formally, a finite poset is a (finite) set equipped with a binary relation that is reflexive, antisymmetric and transitive. Two elements that are related through the relation are called *comparable*, indicating that one precedes the other in the ordering. However, not every pair of elements of a poset are required to be pairwise comparable; in such a case, two unrelated elements are called *incomparable*. A poset can be visualized as a graph where edges connect comparable elements as, for instance, the one in Fig. 5a (the edges deriving from the transitive closure of those depicted in the graph are omitted). Intuitively, the width of a poset is the maximum number of mutually incomparable elements of the poset and corresponds to the maximum number of possibly concurrent stages of a feasible execution obtained from a Spark DAG.

Definition 4.1. An *antichain* is a set of elements of a poset that are pairwise incomparable and the *width* of a poset is the size of the largest antichain. A *chain* is a set of elements of a poset that are all pairwise comparable.

In the case of Spark computations, a chain is a set of stages in a DAG that *must be executed sequentially* because they are related to each other through the precedence relation. The chains of a poset can be overlapping, but a poset can always be partitioned into disjoint chains. Dilworth's theorem binds the width of the poset and the number of its disjoint chains.

Theorem 4.1. [Dil50] Let P be a finite poset and k be such that every set of $k+1$ elements of P contains at least a pair of comparable elements and there is at least one set of k elements of P that is an antichain. Then, k is the *minimum* cardinality of the family of chains into which P can be partioned (Dilworth's chains).

By Theorem 4.1, a poset having width k can be partitioned into k disjoint chains. Figure 5a shows a DAG where arrows indicate the precedence relation between the nodes; then, Figure 5b shows a possible partitioning of the DAG in Fig. 5a into two chains that are indicated with light gray and dark gray nodes.

Modeling the computation of the stages belonging to the *same* chain can be done efficiently, because at most one stage per chain can be active at a time. All the nodes belonging to the same chain can be modeled together as they are never executed concurrently. In fact, only the computation of the active stage has to be represented, whereas the modeling of the inactive stages is straightforward as they are simply idle. Therefore, this property can be used to reduce the state-space of the verification as stated below.

Optimization. *Every chain in a DAG can be modeled with a unique modeling construct (e. g., a*

set of CLTLoc formulae or a TA). Every construct is a modeling template that can be instantiated with specific parameter values representing one chain. Various instances of the template, with different parameter values, capture all the chains in the DAG.

The modeling templates are described in Sect. 4.1 and Sect. 4.2.

There are various algorithms to obtain a Dilworth partition of a poset. The one that has been used to carry out the experiments[7] in Sect. 5 is shown in the Appendix (Fig. 15). Given a poset of n elements, it obtains in $O(n^3)$ the Dilworth partition (the set of chains with the smallest cardinality) and assigns a label to each chain of the poset. The procedure, which uses Bogart-Magagnosc's algorithm [IG04], starts with the smallest partitioning strategy, that is the one containing a single node per chain. Then, the algorithm iteratively tries to aggregate the partitions until no possible reduction can be made. When this happens, a Dilworth partition is found. The algorithm uses a breadth-first search to find *reducing sequence*s, that are identified when a node a has an ancestor b that is the last element of a chain.

For instance, consider the DAG shown in Fig. 5a that represents the graph generated by Spark when it executes application *PageRank*. The algorithm starts with the partition (or set of chains) $C = \{\{0\}, \{1\}, \{2\}, \{3\}..., \{8\}\}$ that is the one with the maximum cardinality where each single node is a chain. Then, for each chain, the algorithm searches for a reducing sequence. In the example, at first, chain $\{0\}$ is considered, and no reduction is performed, since node 0 has no ancestor. Instead, chain $\{1\}$ can be aggregated into chain $\{0\}$, because node 0 is a predecessor of node 1, and node 0 is also the last element of a chain. Therefore, after the first reduction, the resulting partition is $C = \{\{0, 1\}, \{2\}, \{3\}..., \{8\}\}$. Then, chain $\{2\}$ is not reduced with $\{0, 1\}$ since its last element (node 1) is not a predecessor of node 2. On the other hand, chain $\{3\}$ is aggregated into chain $\{0, 1\}$, while chain $\{4\}$ is reduced with chain $\{2\}$. Finally, chains $\{5\}, \{6\}$ and $\{7\}$ are iteratively aggregated into chain $\{0, 1, 3\}$, while chain $\{8\}$ is reduced with chain $\{2, 4\}$, since node 4 is an ancestor of node 8, and it is also the last element of a chain. Therefore, the outcome of the algorithm is the partition $C = \{\{0, 1, 3, 5, 6, 7\}, \{2, 4, 8\}\}$, whose cardinality is 2. Fig. 5b shows the resulting labeled DAG. The 9 nodes are annotated with 2 labels (represented by different colors), corresponding to each set of C. The degree of parallelism of the DAG (i.e., its width or, equivalently, the size of the maximum antichain) is also 2.

4.1. Optimized CLTLoc model

The optimized version of the CLTLoc model introduces a few—yet impactful in terms of model size—changes with respect to the model depicted in Sect. 3.2. In this context, the size of a formula is determined by the number of symbols (atoms, logical connectives and temporal modalities) in the formula. For instance, the size of formula $a\mathbf{U}(b \wedge c)$ is 5 (brackets are not counted). When a formula is parametric, such as all those defining our model, its size depends on the indexes occurring in it. For instance, the size of formula $\bigwedge_{i \in S} a_i \mathbf{U}(b_i \wedge c_i)$ is $5n$, where $n = |S|$, that is, it is linear in the size of set S. Once the DAG is labeled, most of the formulae referring to the computation of each node (stage) in the DAG (e. g., clock formulae, state formulae for tasks) capture the computation of each *group of stages* corresponding to a label. Likewise, atomic propositions and counters that were specific to each stage's batch of tasks (e. g., $\texttt{runT}_i$, $\texttt{startT}_i$, $\texttt{endT}_i$, $\texttt{runTC}_i$), are now related to each label's batch of tasks. For example, Formula (9) below corresponds to Formula (3) after the application of labeling. One main difference between (3) and (9) lies in the fact that the former is repeated for all the elements of the set of labels L, rather than for all the stages in S (indeed, the formula corresponding to Formula (3c) is not shown here, because it is the same as the latter, except for the index domain). Moreover, for this set of formulae and variables, the size is reduced by a factor of $\frac{|S|}{|L|}$.

$$\bigwedge_{j \in L} \left(\begin{array}{ll} (\texttt{startT}_j \Rightarrow \texttt{runT}_j \wedge \neg\texttt{endT}_j \wedge \mathbf{Y}(\neg\texttt{runT}_j \,\mathbf{S}\, (\texttt{orig} \vee \texttt{endT}_j)) \wedge \mathbf{X}(\texttt{endT}_j \mathbf{R} \neg\texttt{startT}_j)) \ \wedge & \text{(9a)} \\ \left(\texttt{runT}_j \Rightarrow \bigvee_{i \in S:\ \ell(i)=j} (\texttt{runS}_i) \wedge (\texttt{runT}_j \,\mathbf{S}\, \texttt{startT}_j) \wedge (\texttt{runT}_j \,\mathbf{U}\, \texttt{endT}_j) \right) & \text{(9b)} \end{array} \right)$$

[7] The source code of the algorithm is available at [dag19].

Some additional constraints need to be specified in order to determine, for each label, which stage is currently active. Let $\ell : S \to L$ be the labeling function that associates each stage with a label according to the Dilworth partitioning. Let $\ell(i)$ be the *label associated with the stage* i; the stage-specific start and end conditions, expressed in Formula (1), are enriched in the following way in Formula (10): since $\texttt{startT}_j$ and $\texttt{endT}_j$ might now refer to many stages (all the stages i such that $\ell(i) = j$), an additional check on $\texttt{completedS}_i$ and $\texttt{enabledS}_i$ is needed to disambiguate which stage is actually active. Analogously, Formula (3b) is enriched in Formula (9b) by adding the constraint that, when $\texttt{runT}_i$ holds for a label i, $\texttt{runS}_j$ must hold for one of the stages j such that $\ell(j) = i$. The definitions of $\texttt{completedS}_i$ and $\texttt{endS}_i$ presented in Formula (2) are kept unchanged in the optimized model.

$$\bigwedge_{i \in S} \left(\begin{array}{rcl} \texttt{startS}_i & \iff & \left(\begin{array}{c} \texttt{runS}_i \wedge \texttt{startT}_{\ell(i)} \wedge \texttt{remTC}_{\ell(i)} = |T_i| \\ \wedge \mathbf{Y}\texttt{enabledS}_i \wedge \neg\texttt{completedS}_i \end{array} \right) \wedge \\ \texttt{endS}_i & \iff & \left(\begin{array}{c} \texttt{runS}_i \wedge \texttt{endT}_{\ell(i)} \wedge \texttt{remTC}_{\ell(i)} = 0 \\ \wedge \mathbf{Y}\texttt{enabledS}_i \wedge \neg\mathbf{Y}\texttt{completedS}_i \end{array} \right) \end{array} \right) \begin{array}{l} \text{(10a)} \\ \text{(10b)} \end{array}$$

Let $\texttt{startEnabledS}_i$ be the shorthand for $\texttt{enabledS}_i \wedge \neg\mathbf{Y}\texttt{enabledS}_i$. As shown in Formula (11), differently from the original CLTLoc model, variable $\texttt{remTC}_{\ell(i)}$ is reset to $|T_i|$ every time a stage S_i corresponding to its label becomes enabled (Formula (11a)) and it is decremented when the related batch of tasks terminates—i. e., $\texttt{endT}_{\ell(i)}$ holds, as in Formula (11c)—and has a monotonically decreasing trend only when S_i is executing (i. e., $\texttt{runS}_i$ holds, as for Formula (11b)). Formulae (11) do not include the ones corresponding to Formulae (5a), (5b), (5d) and (5e), as they differ only for the index, that now ranges over the set of labels.

$$\bigwedge_{i \in S} \left(\begin{array}{l} \left(\texttt{startEnabledS}_i \Rightarrow \mathrm{X}\texttt{remTC}_{\ell(i)} = |T_i|\right) \wedge \\ \left(\texttt{runS}_i \Rightarrow \texttt{remTC}_{\ell(i)} \leq \mathrm{Y}\texttt{remTC}_{\ell(i)}\right) \end{array} \right) \begin{array}{l} \text{(11a)} \\ \text{(11b)} \end{array}$$

$$\bigwedge_{j \in L} \left((\texttt{remTC}_j \neq \mathrm{Y}\texttt{remTC}_j \Rightarrow \texttt{endT}_j \vee \bigvee_{i \in S:\ \ell(i)=j} (\mathbf{Y}\texttt{startEnabledS}_i)) \wedge \right) \quad \text{(11c)}$$

Formula (6), Formula (7) and Formula (4) also need to be modified. More precisely, in Formula (6) clock variables $\texttt{clk}_{\texttt{runT}_i}$, counters $\texttt{runTC}_i$, $\texttt{remTC}_i$, and atomic propositions $\texttt{runT}_i$, $\texttt{endT}_i$ now range over labels L, rather than stages (for example, $\texttt{clk}_{\texttt{runT}_i}$ is replaced with $\texttt{clk}_{\texttt{runT}_{\ell(i)}}$, and similarly for the other elements). In addition, the antecedent of Formula (6b) is modified to explicitly include the condition that stage i is executing (i.e., $\texttt{runS}_i$ holds). In Formula (7) and Formula (4), instead, the index must now simply range over set L instead of S.

The next section examines how the optimization affects the complexity of the CLTLoc model.

4.1.1. Complexity analysis of CLTLoc models

The satisfiability of CLTLoc is in PSPACE [BRS16]. The complexity of the satisfiability problem depends exponentially on the number of subformulae and on the number of clocks in the CLTLoc formula. Even if both the CLTLoc models contain counters, their domain is finite and, hence, their formulae can be converted into equivalent ones using a finite number of atomic propositions. The two parameters that affect the overall complexity are the number of subformulae (added or removed because of the optimization) and the number of clocks (the maximum constant remains the same).

Number of subformulae. Every group of formulae defining the models, i.e., formulae in Sect. 3.2.1 to 3.2.4, undergoes a change in the optimized version; their size, however, only changes linearly in $|S|$ or in the difference $|S| - |L|$. For instance, the size of task formulae and of counter formulae, in Sect. 3.2.2 and 3.2.3 respectively, decreases linearly in $|S| - |L|$ whereas the size of stage formulae increases by a linear term in $|S|$.

Arithmetical operations on finite counters can be encoded linearly in the cardinality of the domain by means of propositional formulae. The domain of the counters that are used in both models does not vary (the number of tasks to be computed is fixed) whereas the number of counters changes; in the optimized

version the set of counters is smaller than the same set in the non-optimized version. In fact, only $|L|$ pairs of counters `remTC` and `runTC` are used in the optimized version whereas $|S|$ pairs of counters `remTC` and `runTC` are adopted in the non-optimized one (`avaCC` is used in both models).

Number k of clocks. The number of clocks adopted in the optimized model is smaller than that in the non-optimized one. In particular, the number of clocks in the optimized model depends linearly on $|L|$ instead of $|S|$. Hence, the overall complexity of the p-feasibility problem encoded through a CLTLoc formula is reduced by a term which exponentially depends on the value $|S| - |L|$.

4.2. Optimized TA model

The optimized version of the TA modeling the nodes is similar to the one presented in Sect. 3.3 and it is shown in Fig. 6. The fundamental difference between the optimized version and the non-optimized one is that, in the optimized version, one TA models the computation of all the nodes with the same label. For instance, the graph in Fig. 5b shows a labeled DAG where dark gray and light gray stages define two disjoint chains. Using the encoding of Sect. 3.3, every stage is modeled by a distinct TA (9 in total), whereas two distinct TA are enough to represent the two chains with the optimized version.

Given a DAG, let $L = \{l_0, \ldots, l_{m-1}\}$ be the set of the m labels extracted with the algorithm presented in Sect. 4 and let S_l be the set of the stages associated with label l. Similarly to the non-optimized model, the computation of the DAG is represented through a network of TA. For every label $l \in L$, there is an automaton A_l which keeps track of the computation of all stages labeled with l. Every automaton A_l has two locations $Init_l$ and $Idle_l$ that have the same meaning as those in the automaton in Fig. 4. Since at most one stage per label is active at a given time (the stages with the same label are executed sequentially), a variable $active\_stage_l$, whose value ranges from 0 to $|S_l|$, is introduced to indicate the active stage among those modeled by A_l (assuming that the nodes in S_l are numbered). The value of $active\_stage_l$ is set initially to 0 and it is incremented every time the active stage has completed the computation of all of its associated tasks (see the update on transition from $Idle_l$ to $Init_l$). Moreover, $active\_stage_l = |S_l|$ holds when all the stages labeled with l have been completed. Variable $active\_stage_l$ is part of the definition of the global state of the network of TA, hence it can be read by all the automata modeling the DAG. Hereafter, variable $active\_stage_l$ is used to easily refer to the stages in S_l. For instance, instead of saying that "if $active\_stage_l$ is equal to i, then the duration of tasks of stage i is τ_i", $\tau_{active\_stage_l}$ succinctly indicates the duration of the tasks of the active stage.

In general, the stages labeled with the same label l behave differently at runtime, because their tasks have different duration, or because they include different number of tasks and dependencies (determined by the DAG). Hence, to properly represent the computation of every stage modeled by automaton A_l, the following information must be taken into account:

- the dependency relation between the stage $active\_stage_l$ and the other stages of the DAG,
- the duration of the tasks $\tau_{active\_stage_l}$ and
- the number of tasks in $T_{active\_stage_l}$.

This information is required to model the duration of the batches and to establish the beginning and the termination of the computation of the active stage. While, in the non-optimized solution, they are hard-coded in the automaton modeling a stage, automaton A_l uses suitable guards and updates to correctly set the values of $\tau_{active\_stage_l}$ and $T_{active\_stage_l}$ according to the value of $active\_stage_l$.

The condition on the transition between $Init_l$ and $Idle_l$ captures the dependency relation of the DAG. This condition in A_l is more complex than the corresponding one in the non-optimized version, as the set of predecessors of the active stage modeled by A_l now depends on $active\_stage_l$. Given a stage s, let $label(s)$ be its associated label, and let $\#(s)$ be the index of s given by the enumeration of the element in the set $S_{label(s)}$ containing s. The guard between $Init_l$ and $Idle_l$ is a conjunction of implications, one for every stage in S_l, enforcing the predecessor relation (consequent) based on the value of $active\_stage_l$ (antecedent):

$$\bigwedge_{s \in S_l} (active\_stage_l = \#(s) \Rightarrow \bigwedge_{s' \in pre(s)} active\_stage_{label(s')} > \#(s')).$$

Every condition $active\_stage_{label(s')} > \#(s')$ of the conjunction in the right-hand side guarantees that every

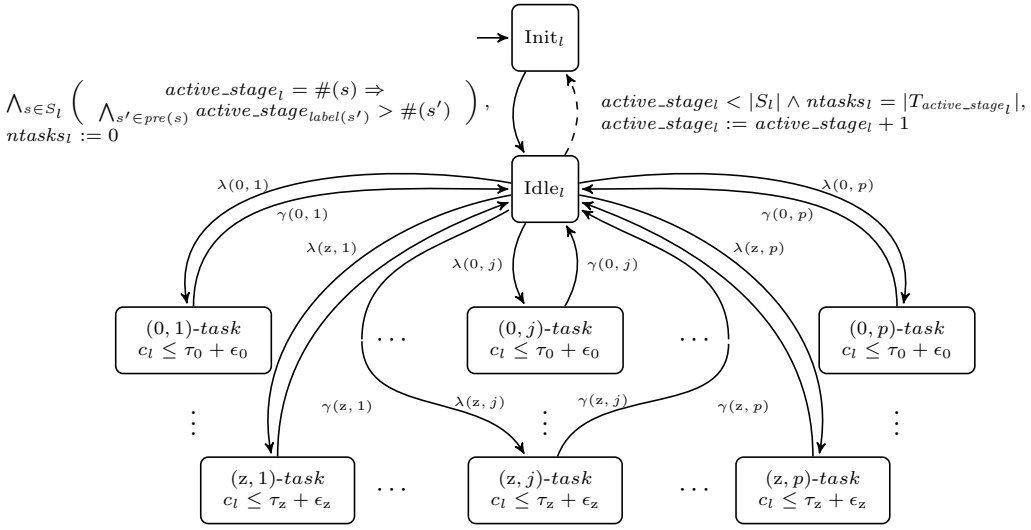


Fig. 6. Timed automaton representing all the nodes labeled with the same label. Dashed lines indicate a set of transitions: every transition is associated with a specific set $T_{active\_stage_l}$. Constant $|S_l| - 1$ is abbreviated with symbol z in Roman.

stage s'—labeled with $label(s')$—that is a predecessor of the active stage s has been completed. More precisely, the condition requires that the value of the current active stage in the automaton $A_{label(s')}$ is greater than its index $\#(s')$; hence, the computation of $\#(s')$ is completed (recall that the stages in S_l are numbered and, when a stage terminates, variable $active\_stage_l$ is incremented). For instance, assume that the stage with label l and index 2 depends on the stage with label l' and index 1 and the stage with label l'' and index 3; then, the corresponding implication (which is part of the—possibly more complex—conjunction defining the guard between $Init_l$ and $Idle_l$) is $active\_stage_l = 2 \Rightarrow active\_stage_{l'} > 1 \wedge active\_stage_{l''} > 3$.

To model different task durations, automaton A_l uses copies of the locations j-$task$ already used in the non-optimized TA, indexed with the identifier of the stages in S_l labeled with l. For every stage in S_l, one copy of every location (i,j)-$task$ is defined to indicate that stage i is currently running a batch of j tasks. Every location (i,j)-$task$ has a specific invariant that depends on the value of i that is stored in $active\_stage_l$. In addition, every transition between $Idle_l$ and any location (i,j)-$task$ has the same updates as the corresponding one in the non-optimized automaton, but it differs from the latter because the guard also includes a constraint over the index of the current active node. In particular, abbreviations λ and γ now depend on both the active node i and the size j of the batch that is going to be executed. Label $\lambda(i,j)$ contains the guard $active\_stage_l = i \wedge avaCC \ge j$, and the assignments are $avaCC := avaCC - j$ and $c_l := 0$. Similarly to the model of Sect. 3.3, lower and upper bounds to the duration of tasks of stage i are expressed by considering a deviation of ϵ_i time units from the nominal one τ_i. Hence, to guarantee that the duration of every batch of tasks is bounded, every location (i,j)-$task$ is labeled with an invariant $c_l \le \tau_i + \epsilon_i$. Label $\gamma(i,j)$ contains guard $c_l \ge \tau_i - \epsilon_i$ and the assignment $avaCC := avaCC + j$.

4.2.1. Problem reduction

The reachability of the termination of all the stages earlier than the deadline d is encoded with the TCTL Formula (12), that constraints the index of the variables $active\_stage_l$ to be equal to $|S_l|$: this is the case when the stage in S_l numbered with $|S_l| - 1$ has completed the computation of all the tasks, and variable $active\_stage_l$ has been incremented by the transition from $Idle_l$ to $Init_l$.

$$\exists \mathbf{F}_{<d} \left(\bigwedge_{l \in L} (active\_stage_l = |S_l| \right). \tag{12}$$

The impact of the optimization in terms of model complexity is discussed in the next session.

4.2.2. Complexity analysis of TA models

The PSPACE-completeness of the model-checking problem of TA with respect to TCTL formulae is shown in [ACD93], where the upper bound for the problem is determined by the size of the region automaton that is constructed for the evaluation of the TCTL formula. The size of the region automaton is exponential in the number of clocks and in the size of constants (assuming a binary encoding) occurring in the TA and in the formula, while it is linear in the size of the TA (i.e., locations and edges) and in the number of TCTL subformulae. The reachability problem defined through the formulae (8) and (12) is, however, only affected by the number of locations in the TA modeling a p-feasibility problem, the greatest constant in the problem and the number of clocks used by the TA. In fact, the size of the formulae (8) and (12) is constant, and the number of edges of the whole automaton is always linear in n, where n is the size of the structure of every TA modeling the stages (both in the non-optimized and optimized version) fixed a-priori. Hence, given a TA, the number of vertices in the region automaton is $O(n \cdot k! \cdot 2^k \cdot (2c_{max}+2)^k)$, that is $n \cdot 2^{O(k \log(kc_{max}))}$, where n is the number of locations in the TA, c_{max} is the greatest constant of the problem and k is the number of clocks used by the TA. For this reason, to establish the complexity of the procedure, only parameters n and k of the network of TA modeling a p-feasibility problem have to be estimated. Their value depends on the size of the feasibility problem to be solved, which is a function of the size of the DAG (i.e., the number of nodes $|S|$), the number of cores p and the deadline d. In the optimized version, the set L of labels defined by the partitioning algorithm is also considered. The constant c_{max} is equal to the deadline d, which is the greatest constant in the problem.

Number k of clocks. The value of k in the optimized solution is equal to the number $|L|$ of labels produced by the labeling algorithm, whereas it is determined by $|S|$ in the non-optimized model. Therefore, the cost that is introduced by the clocks is always smaller in the case of the optimized model and, in particular, the overall complexity is reduced by a term which exponentially depends on the value $|S| - |L|$. Hence, the magnitude of the reduction of k depends on the DAG and, in some cases, k turns out to be logarithmic in the size $|S|$ of the DAG in the optimized model, whereas it is linear in $|S|$ in the non-optimized version.

Number n of locations. The number of locations of the automata in Fig. 6 and Fig. 4 depends on:

- the number of cores p considered in the p-feasibility problem,
- the maximum value of *ntasks* and $completedS_i$ in the model in Fig. 4, and
- the maximum value of *ntasks*, *avaCC* and $active\_stage_l$ in the model in Fig. 6.

The integer counters have finite domains which can be embedded in the control states of a TA without counters which is equivalent to the original one with counters.

Variables $active\_stage_l$ are only used in the optimized version, and they are bounded by $|S|$; for every label $l \in L$, a variable $active\_stage_l$ is introduced in the model. Conversely, the variables $completedS_s$ are only used in the non-optimized version and they are Boolean; for every stage $s \in S$, a variable $completedS_s$ is introduced in the model.

To determine the overall size of a network of TA, built based on the template of Fig. 4 in the case of the non-optimized translation, and on the template of Fig. 6 in the optimized one, the size of a single automaton is calculated first. In the analysis, variable *avaCC* is omitted because the variable is used in both models, hence it affects their size in the same way.

Assume that *ntasks* is bounded by $\mathtt{n}_{\mathrm{MAX}}$. The TA in Fig. 4 has size $p \cdot \mathtt{n}_{\mathrm{MAX}} \cdot 2^{|S|}$, where factor $2^{|S|}$ ensues from the use of variable $completedS_s$ in the TA, defined for every $s \in S$ (every automaton can read the variable associated with any $s \in S$). Every variable $completedS_s$ is global and it is not replicated in every automaton with different values. Hence, representing $|S|$ stages, each one with a distinct TA, requires a network of size $(p \cdot \mathtt{n}_{\mathrm{MAX}})^{|S|} \cdot 2^{|S|}$.

Consider $|S_l|$ stages with label l. Then, the TA in Fig. 6 has size $(|S_l| \cdot p \cdot \mathtt{n}_{\mathrm{MAX}}) \cdot \prod_{i=0}^{|L|-1}(|S_i|+1)$, where the factor $|S_l|$ is determined by the $|S_l|$ copies of the locations (i,j)-*tasks*, for $0 \leq j < |S_l|$, and $\prod_{i=0}^{|L|-1}(|S_i|+1)$ is determined by the possible values of variable $active\_stage_l$ in the TA, for every $l \in L$ (every automaton can read all the variables $active\_stage_l$, having at most $|S_l|$ distinct values). The domain of every $active\_stage_l$ contains at least 2 elements (when $|S_l| = 1$ then $active\_stage_l$ is binary). Similarly to the previous case, every variable $active\_stage_l$, for $l \in L$, is global, and it is not replicated in every automaton with different values. Hence, representing $|L|$ labels, each one with a distinct TA, requires a network of size

$\prod_{i=0}^{|L|-1}(|S_i| \cdot p \cdot \mathtt{n}_{\mathrm{MAX}}) \cdot \prod_{i=0}^{|L|-1}(|S_i|+1)$ that is $(p \cdot \mathtt{n}_{\mathrm{MAX}})^{|L|} \cdot (\prod_{i=0}^{|L|-1} |S_i|(|S_i|+1))$. If the labeling assigns every stage of S with a label, then the previous formula reduces to the one obtained in the non-optimized case.

The size of the optimized and of the non-optimized solutions are now compared to determine their relation. Since it always holds that $(p \cdot \mathtt{n}_{\mathrm{MAX}})^{|L|} < (p \cdot \mathtt{n}_{\mathrm{MAX}})^{|S|}$, the following relation is investigated:

$$2^{|S|} \sim \prod_{i=0}^{|L|-1} |S_i|(|S_i|+1) < \prod_{i=0}^{|L|-1} (|S_i|+1)^2. \tag{13}$$

The right-hand side of (13) depends on the size S_l, for every label $l \in L$, which only depends on the DAG and cannot be written in terms of S. For this reason, $\prod_{i=0}^{|L|-1}(|S_i|+1)$ is analyzed in the worst case to identify the values for S_i maximizing the product and such that $\sum_{i=0}^{|L|-1} |S_i| = |S|$. If $\prod_{i=0}^{|L|-1}(|S_i|+1)$ is maximum, then so is $\prod_{i=0}^{|L|-1}(|S_i|+1)^2$. The method of Lagrange multipliers [Haz87] can be applied to identify the maximum (or minimum) of a function that is subject to additional constraints. By applying the method to $\prod_{i=0}^{|L|-1}(|S_i|+1)$ with constraint $\sum_{l=0}^{|L|-1} |S_i| = |S|$, the maximum of the product is obtained when $|S_i| = \frac{|S|}{|L|}$ for every $0 \leq i < |L|$. Let m be $\frac{|S|}{|L|}$; relation (13) can be rewritten as $2^{|S|} \sim (m+1)^{2|L|}$, hence also as

$$2^{|S|} \sim 2^{2\frac{|S|}{m}\log_2(m+1)}, \tag{14}$$

because it holds that $(m+1)^{2|L|} = 2^{2|L|\log_2(m+1)} = 2^{2\frac{|S|}{m}\log_2(m+1)}$. Since $\frac{\log_2(m+1)}{m}$ is always lower than $\frac{1}{2}$ for $m > 6$, it turns out that $|S| > 2\frac{|S|}{m}\log(m+1)$ holds for every $|S|$ when $m > 6$. In such a case, i.e., every label labels at least 6 stages, the number of locations in the non-optimized model is always greater than the number of locations in the optimized one, for any S such that $|S| > 6$.

5. Evaluation

This section presents two sets of experiments carried out on real-world Spark applications to evaluate the effectiveness of the approach. The first validation activity focuses on the evaluation of the accuracy of the formal model in characterizing the feasibility of a deadline for a given Spark application. The second one aims at assessing the practical impact of the optimizations presented in Sect. 4. A number of well-known benchmark applications were selected to perform the analysis and evaluate the technique against realistic use-cases: the simple *SortByKey* operation; the graph processing algorithm *PageRank* [BP98]; the clustering procedure *K-Means* [Mac67], the machine learning algorithm *SVM* [Bur98], *TPCH_22*, a benchmark query from the TPC suite[8], and Louvain, a community detection algorithm in large graphs[9]. Specifically, *TPCH_22* is the 22nd query, called "Global Sales Opportunity Query", of class H. Two versions of the query have been considered in the experiments. They are called *TPCH_22_6* and *TPCH_22_7*, where the numbers 6 and 7 are parameter values that identify two distinct sets of input data that are filtered out by the query to obtain the result. *SortByKey*, *PageRank* and *K-Means* were employed for both evaluation activities, while *SVM*, *TPCH_22* and *Louvain* were used only in the second assessment.

Figure 7 shows the DAGs associated with every benchmark considered in the analysis. Colors highlight the different labels that are assigned to the nodes by applying the algorithm described in Sect. 4 (whose details are in Fig. 15 of the Appendix) with the intended meaning that nodes with the same color have the same label, as they belong to the same partition.

All the Spark applications were executed on virtual machines purchased from Azure[10], the cloud computing platform developed by Microsoft. The adopted virtual machines, called *Standard_D14_v2*, are optimized for memory usage, with a high memory-to-core ratio, and are equipped with 16 CPUs, 112GB of memory,

[8] TPC is a non-profit corporation that produces benchmarks to measure performance of transaction processing and databases. `http://www.tpc.org/tpch/`

[9] `https://sourceforge.net/projects/louvain/`

[10] `https://azure.microsoft.com/`

and 800GB of local SSD storage. Each machine ran Canonical Ubuntu Server 14.04.5-LTS, Oracle Java 8, and Apache Spark 2.0.2.

5.1. Accuracy evaluation

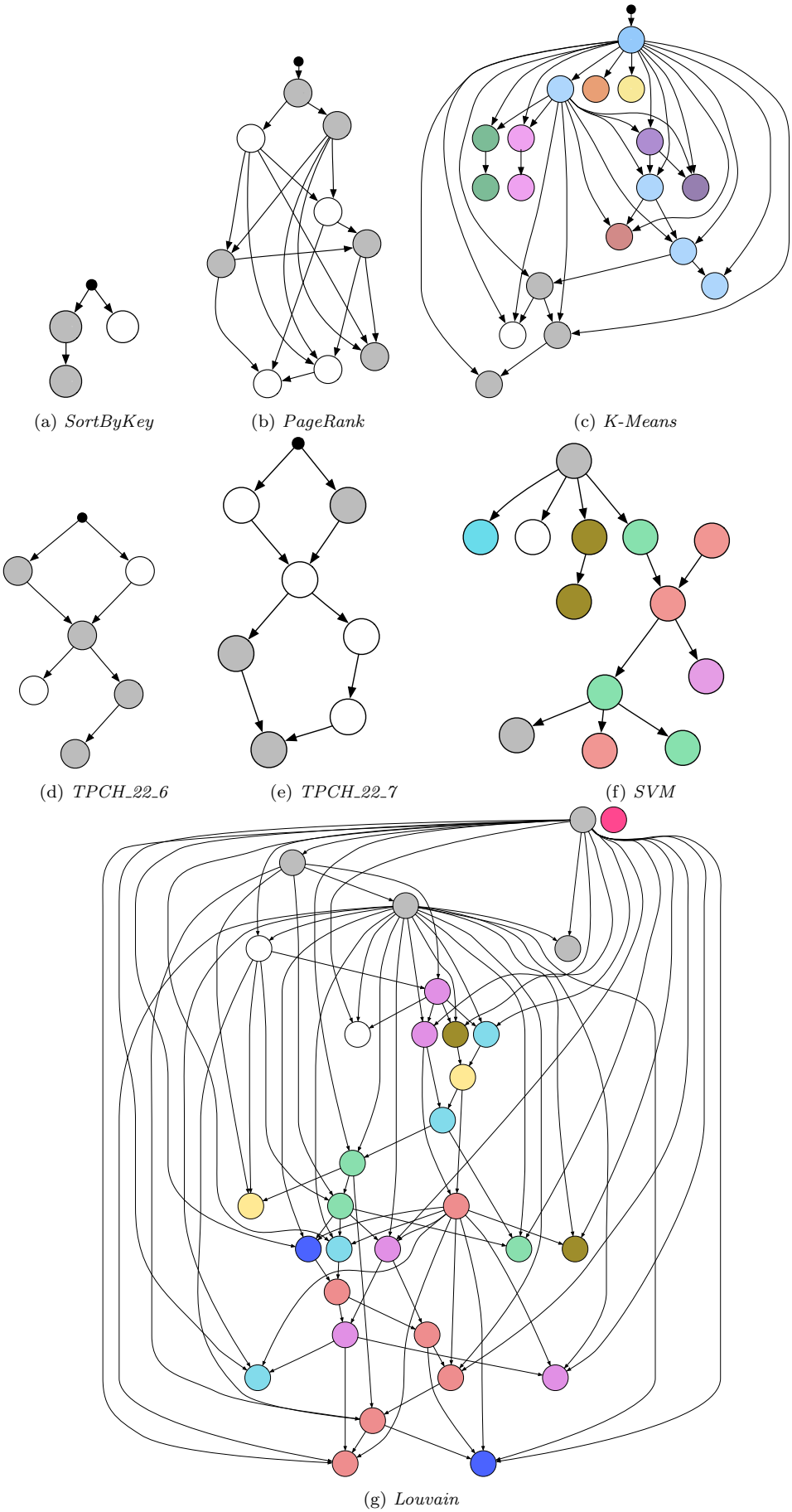
The experiments were carried out considering applications *SortByKey*, *PageRank*, and *K-Means*, and by applying the following methodology.

- Six different settings are chosen for every benchmark, in terms of the configuration of the underlying cluster: two different numbers of available cores for each cluster node, and three different dimensions of the input dataset split in a fixed number of partitions.
- For every setting, the benchmarks are executed three times on two different versions of the Spark framework: *sequential* Spark and the regular version of Spark. In *sequential* Spark, the scheduler was modified to always launch all the stages sequentially (i. e., no more than one stage can be simultaneously in execution), and to have a clean isolation of the stages and tasks for a precise identification of the task durations, without the noise introduced by the concurrent execution of multiple stages.
- The timing information for each task is collected by means of a profiling tool:[11]
  - the average task durations obtained from executions with *sequential* Spark are used to automatically generate instances of the formal model with such characteristics;
  - the *average execution time*s of all applications collected with Spark (from now on $avg(t_{exec})$) are used as reference to compare the results of the analysis carried out by means of the formal models.
- For every benchmark and for every setting, two formal models are built, one based on TA and the other based on CLTLoc. A set of feasible deadlines *FD* (defined in Sect. 3.1.3) for each model is determined by solving several *p*-feasibility problems, instantiated with various deadline values. The number of *p*-feasibility problems is determined based on a heuristic explained later.
- For every benchmark and for every setting, the *mfd* (i.e., the minimum value in *FD*) is compared against the corresponding average execution time $avg(t_{exec})$. The difference is used to evaluate the accuracy of the model (expressed as percentage error *err*).

All the use cases were verified by means of two state-of-the-art tools: Zot [Pol18]—supporting the CLTLoc model—and Uppaal [BDL+06]—supporting the TA model. Each tool was used in two different configurations. Since Zot provides many plugins, each of them supporting a different encoding of the temporal logic formulae, the two most relevant ones for the verification of CLTLoc formulae, *ae2sbvzot* [BPKR16] and *ae2zot* [MAA+14], were adopted. Verification with Uppaal was carried out by considering two different search order strategies: *depth-first* and *breadth-first*.

Since, in the neighborhood of *mfd*, the verification time can be extremely high and the memory demand can exceed available RAM, running an extensive deadline feasibility analysis was not always possible. Preliminary experimental results showed a significant difference between the verification times for feasible and unfeasible deadlines in the neighborhood of *mfd* in almost all the cases, except for the Uppaal breadth-first algorithm. In addition, both the verification time and memory usage grow significantly with the size of the DAG with Uppaal, whereas a significant correlation between verification time and the deviation between deadline and *mfd* is typical of Zot and Uppaal using a depth-first searching algorithm. For this reason, the results of Table 1 and of Sect. 5.2 were collected by considering specific timeouts and memory upper bounds that allowed us to conclude that certain deadlines were *reasonably* not feasible. For every benchmark and setting, and for each verification tool, an approximation of *mfd* was calculated according to the following heuristic approach. The search for the feasible deadlines proceeded iteratively by decreasing the deadline, initially set to a trivial value (i.e., the time of the serial execution of the application), until an unfeasible deadline is found or the timeout/memory bound is reached. The initial deadline is such that its value always allows a feasible (trivial) execution to exist, and it is calculated as the sum of the duration of all the stages of the benchmark that were observed in the execution performed by sequential Spark. The timeout and the memory bound that are considered in a verification experiment (carried out with Zot or Uppaal) are based on the times and used memory observed for the feasible deadlines previously obtained. For instance, consider

[11] github.com/deib-polimi/xSpark-bench



(a) *SortByKey* (b) *PageRank* (c) *K-Means*

(d) *TPCH_22_6* (e) *TPCH_22_7* (f) *SVM*

(g) *Louvain*

Fig. 7. Labeled DAGs of selected applications. Nodes of different colors belong to different partitions.

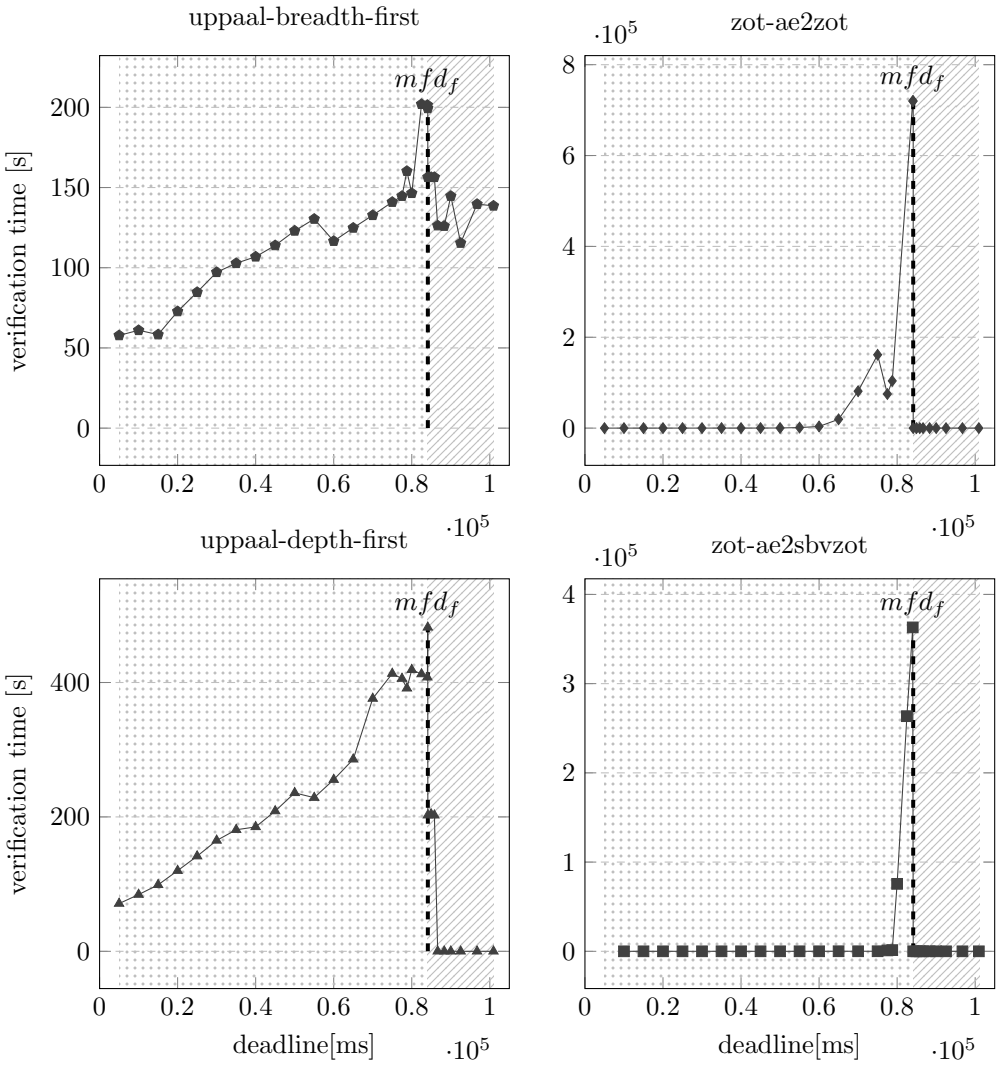


Fig. 8. Verification time statistics on *SortByKey* (22 cores, 100 tasks and 300M input records) with different deadlines.

a feasible deadline d, verified by using memory m and time t. In such a case, the search continues, because d is feasible, and the next deadline to be verified is set to $0.99 \cdot d$; the timeout is set to $20 \cdot t$, and the memory bound is set to $50 \cdot m$. In the other cases, i.e., when d is unfeasible or the timeout/memory bound are reached, the search terminates. The increments in memory bound and timeout between two generic consecutive experimental verifications are based on empirical evidence: given two close *feasible* deadlines (i.e., within a time difference of less than 1%), if the largest is successfully verified in a certain amount of time t, then the shorter is generally verified within the newly computed bounds based on t. For this reason, a deadline is reasonably considered to be not feasible if no result is returned by the tool within the timeout or without exceeding the memory bound. The only exception is represented by the verification tasks performed using *ae2zot*, whose execution time was, in some use-cases, extremely higher than the ones previously obtained for greater deadlines. In this case, our policy would have led to excessively high timeouts (up to many days). Since the feasibility of such deadlines was already assessed with *ae2sbvzot* in much lower time, for practical reasons, the verification was stopped after several hours. For a given setting, the smallest feasible deadline obtained by applying the heuristic is indicated with mfd_f.

Figures 8 and 9 together illustrate the claim above and the identification of the *mfd* for *SortByKey*.

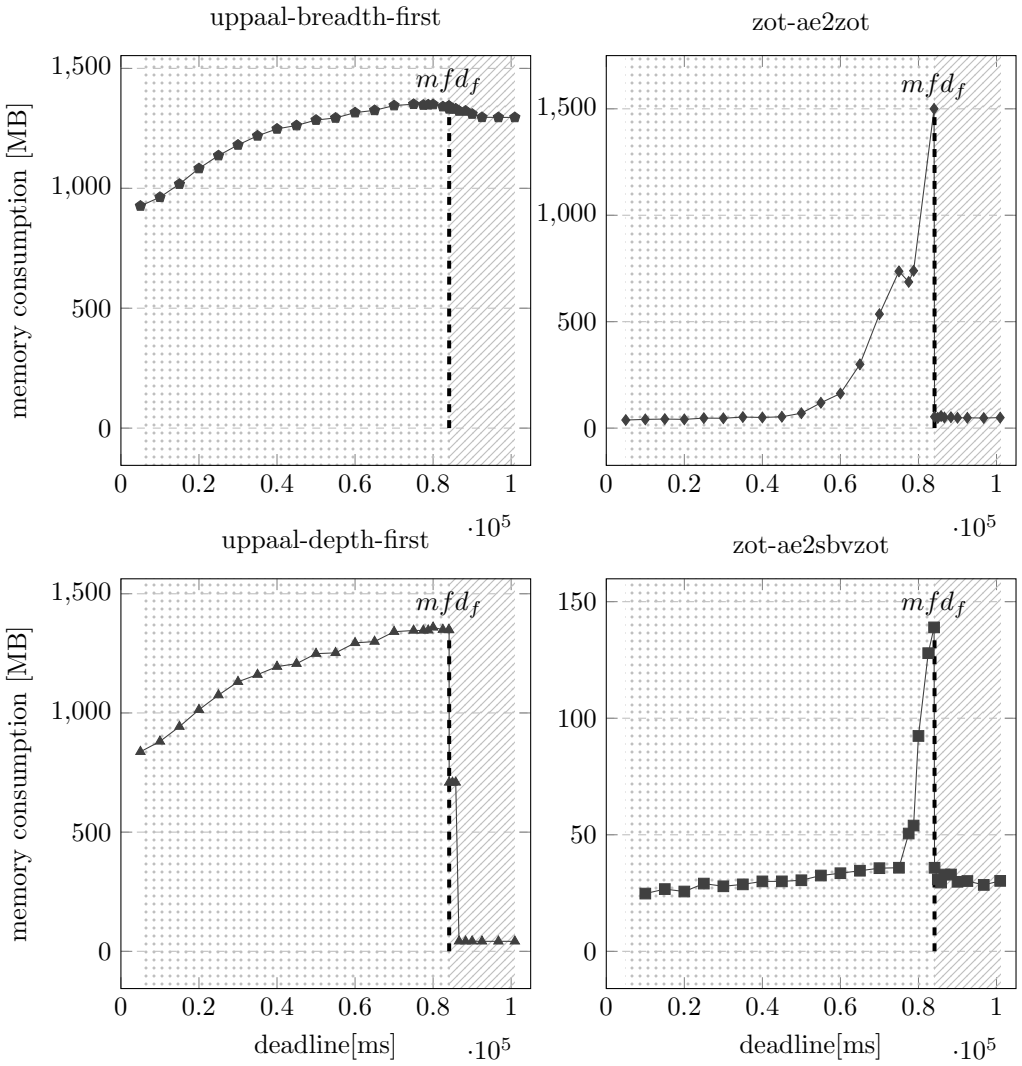


Fig. 9. Memory consumption statistics of the verification tasks on the *SortByKey* use case (22 cores, 100 tasks and 300M input records) by providing different deadlines.

They show the time and memory consumption registered for verification tasks performed with the four tool configurations on a single setting of *SortByKey*, the only one that allowed for comprehensive view of all the experimental verifications with no timeouts or memory saturation—except for *ae2zot*, which reached the timeout for a small number of deadlines. The horizontal axes represent the deadline values, while the vertical axes display respectively the times (Fig. 8) and memory statistics (Fig. 9) of the verification tasks. *mfd* is highlighted by a vertical black dashed line that separates the feasible deadlines (light gray lines background pattern) from the unfeasible deadlines (light gray dotted background pattern). The mfd_f is 84118 ms, therefore all the deadlines higher than that value are feasible. On the other hand, deadlines of 84117 and below were unfeasible. Since, for this setting, $avg(t_{exec})$ is 82133 ms, the percentage error *err* is about 2.4%. This analysis highlighted a strong dependency of verification time and memory consumption on the closeness of the analyzed deadline to the minimum feasible deadline for three tool configurations out of four: the only exception is represented by Uppaal breadth-first, whose space exploration strategy led to time and memory statistics more uniform across the different deadlines analyzed.

As reported in Fig. 8, in the case of Zot, the verification time is in the order of seconds for all deadlines lower than 60000 ms (75000 ms for *ae2sbvzot*) and greater than or equal to 84118 ms (mfd_f), whereas it

Table 1. Accuracy evaluation: experimental results.

app	*cores*	*tasks*	$records_{in}$	$avg(t_{exec})$	mfd_f	*err*
SortByKey	12	100	260M	88386	91384	3.3%
			280M	100769	98420	2%
			300M	107054	105443	1.5%
	22	100	260M	74919	72904	2.6%
			280M	77884	78500	0.7%
			300M	82133	84118	2.4%
PageRank	28	128	200M	60028	62500	4%
			300M	87787	94000	7%
			400M	116810	120000	2%
	48	128	200M	48805	47000	3.6%
			300M	66636	65100	2.3%
			400M	88320	86000	2.6%
K-Means	24	18	80M	77651	69000	11.1%
			120M	103492	100000	2.5%
			160M	131600	126725	3.7%
	32	24	80M	64565	58000	10.1%
			120M	81299	80000	1.5%
			160M	101483	100000	1.4%

grows exponentially for increasing deadline values between 60000 ms (75000 ms respectively) and 84117 ms. *ae2sbvzot* registered a peak of 100 hours for 84000 ms, while *ae2zot* reached the timeout of 200 hours for the analyzed deadlines between 84000 ms and 84117 ms. The notable growth is therefore registered for those deadlines that were unfeasible, but close to mfd_f. Uppaal depth-first experiments showed a less steep growth in verification time (from 70 to 400 seconds) for unfeasible deadlines, and a sharp drop when passing to the smallest feasible deadlines. The growth in Uppaal breadth-first is more moderate, and there is little difference between feasible and unfeasible deadlines.

Fig. 9 shows similar trends for memory consumption statistics. The main difference lies in the fact that the memory usage of Uppaal is generally higher by an order of magnitude than that of Zot. As already mentioned, memory usage represents the main obstacle to an exhaustive analysis with Uppaal to the same extent as excessive verification time prevents Zot from practically performing verification on some unfeasible deadlines. The trend depicted in Fig. 8 and 9 has been observed also for the other benchmark applications, except for the presence of timeouts or memory saturation.

Table 1 shows the experimental findings of the validation activity for the three applications. Each row represents a different application setting, characterized by a specific number of cores in the cluster, a number of tasks (i. e., partitions) for each stage, and a dimension of the input dataset in terms of number of records ($records_{in}$). The measures of interests are the previously defined $avg(t_{exec})$, mfd_f and the related percentage error *err*. Results show that the adherence of the model to the actual execution times with Spark (i. e., of mfd_f to $avg(t_{exec})$) is not particularly affected by the application and the configuration (i.e., number of cores and tasks). The average *err* across all the configurations is 3.5%, the median is 2.7% and the maximum is 11.1%. The error is higher than 10% only in two configurations used for *K-Means*, while it is lower than 3.4% across all the six settings of *SortByKey* and it is at most 7% for all the *PageRank* settings.

The experimental evidence overall shows that verification time and memory consumption are mostly affected by DAG size and number of cores, which directly impact the model size. On the other hand, the input size $records_{in}$ does not influence the time and memory characteristic of the verification. For this reason, in the experiments carried out on additional use cases, and discussed in Sect. 5.2, only one input size and two different core configurations for each application were selected. In particular, the two settings considered in the experiments on *TPCH_22* both use 60 million input records, and 16 or 32 cores. *SVM* was analyzed with 100 million input records, and 32 or 64 cores, and *Louvain* was analyzed with 16M input records, and 64 or 128 cores.

TPCH_22_7		Uppaal DF		Uppaal BF		ae2sbvzot		ae2zot	
res.	deviation	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving
unsat	-22%	1.43x	0%	1.47x	0%	n. a.	n. a.	n. a.	n. a.
unsat	-11%	1.44x	0%	1.40x	0%	n. a.	n. a.	n. a.	n. a.
sat	0%	1.00x	25%	1.43x	0%	3.36x	48%	4.83x	52%
sat	6%	1.00x	24%	1.35x	0%	2.76x	46%	4.39x	53%
sat	11%	1.00x	24%	1.36x	0%	1.71x	43%	7.74x	61%
sat	17%	1.00x	24%	1.55x	0%	1.22x	42%	3.03x	50%
sat	22%	1.00x	24%	1.39x	0%	1.25x	42%	3.57x	51%

Fig. 10. Effects of optimization on the different tools for the *TPCH_22_7* configuration with 32 cores and 60M input records.

5.2. Evaluation of the optimized models

The experiments discussed in this section aim at assessing the effects of the model optimization techniques presented in Sect. 4. They employ some additional benchmark applications with respect to those used in Sect. 5.1, such as *SVM*, *TPCH*, and *Louvain*. The verification tasks are performed on different settings, for each benchmark, and use the two models (i. e., CLTLoc and TA) both in the optimized and in the non-optimized version. Moreover, as for accuracy evaluation, two different plugins of Zot and two different search orders for the state space exploration in Uppaal were used. For every verification experiment, the time and memory consumption statistics[12] are compared to estimate the benefits of the optimization.

Fig. 10 to 14 show, for the use cases *PageRank*, *K-Means*, *SVM*, *TPCH_22_7*, and *Louvain*, respectively, a summary of the experiments carried out for a selected setting among those considered[13], and highlight the impact of the optimization in terms of both verification time and memory across the different configurations of the verification tool. Each row in the tables shows the statistics regarding the verification of the feasibility problem against a specific deadline. The column *deviation* indicates the percentage difference between the *deadline* of the row and the mfd_f for that specific case, and expresses how much the deadline is higher or lower (negative percentage values) than the mfd_f. The impact on verification time is expressed in terms of *speedup*. The value is the ratio between the smallest and the biggest of the two verification times with and without optimization, respectively. When the time needed to solve the optimized version is smaller, then there is a speedup, that is highlighted in the figures with a bar from the center of the column to the right. Conversely, when the time to solve the non-optimized version is smaller, there is a slowdown (or negative speedup), represented in the figures with a negative sign and a bar from the center to the left of the column. Analogously, the impact on memory is expressed in terms of *memory saving*—that is, the percentage difference between the smallest and the biggest of the two memory consumption statistics with and without optimization, respectively. Positive values represent the percentage of memory that can be saved by using the optimization, while negative values express how much more memory (in percentage) is required by the optimized version with respect to the non-optimized one.

Tables 2 and 3 provide a compact overview of the impact of the optimization strategy on the two tools across the different benchmarks. Since various settings are considered, the cells of the tables show ranges that indicate aggregated data obtained from all the settings considered in the tests. The wording *"$N\times$ speedup with x hours timeout"* indicates that the experimental verification using the non-optimized version of the model reached the timeout, the optimized model was solved N times faster than the non-optimized one, and the speedup N is determined by considering the verification time of the non-optimized version to be equal to the timeout.

ae2sbvzot shows significant benefits on memory consumption across all use cases, while in some (fewer) cases—probably due to the heuristics used by the solver—the impact on the verification time is negative. The best improvements (in terms of both memory and time) are registered with the largest DAGs we considered. *ae2zot* is positively affected by the optimization for all use cases, and—as with *ae2sbvzot*—the benefits

[12] All the verification experiments were conducted on machines running Ubuntu GNU/Linux 14.04, with 16 Intel Intel(R) Xeon(R) CPU E5-2673 v3, 2.40GHz, and 112GB RAM; CLTLoc models were analyzed by means of Z3 SMT solver v.4.7.1 (64 bit), while the TA models were analyzed through UPPAAL 4.1.19 (64 bit).

[13] The complete results can be found at `https://github.com/deib-polimi/DAG-ver`

SVM		Uppaal DF		Uppaal BF		ae2sbvzot		ae2zot	
res.	deviation	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving
sat	0%	1.00x	26%	n. a.	n. a.	33.88x	67%	n. a.	n. a.
sat	1%	1.00x	26%	n. a.	n. a.	-1.83x	31%	1.12x	n. a.
sat	3%	1.00x	26%	n. a.	n. a.	1.80x	44%	3.28x	52%
sat	5%	1.00x	27%	n. a.	n. a.	2.78x	51%	11.16x	n. a.
sat	10%	1.00x	27%	n. a.	n. a.	2.70x	55%	2.57x	51%
sat	15%	1.00x	34%	n. a.	n. a.	-3.05x	21%	7.21x	57%
sat	20%	1.00x	27%	n. a.	n. a.	2.42x	52%	10.31x	61%

Fig. 11. Effects of optimization on the different tools for the *SVM* configuration with 64 cores and 160M input records.

Pagerank		Uppaal DF		Uppaal BF		ae2sbvzot		ae2zot	
res.	deviation	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving
unsat	-3%	2.35x	8%	1.68x	4%	n.a.	n.a.	n.a.	n.a.
unsat	-2%	3.26x	10%	1.68x	5%	n.a.	n.a.	n.a.	n.a.
sat	0%	3.28x	26%	1.72x	4%	-1.95x	56%	n.a.	n.a.
sat	2%	13.29x	49%	1.68x	4%	3.57x	60%	12.25x	77%
sat	3%	1.00x	29%	1.73x	4%	5.45x	61%	11.20x	75%
sat	11%	1.00x	29%	1.80x	4%	4.49x	60%	6.27x	68%
sat	19%	1.00x	28%	1.61x	4%	4.98x	60%	5.85x	64%

Fig. 12. Effects of optimization on the different tools for the *PageRank* configuration with 28 cores, 128 tasks and 200M input records.

K-Means		Uppaal DF		Uppaal BF		ae2sbvzot		ae2zot	
res.	deviation	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving
sat	0%	3321.92x	99%	n. a.	n. a.	1.28x	49%	n.a.	n.a.
sat	1%	3.70x	17%	n. a.	n. a.	4.08x	54%	n.a.	n.a.
sat	5%	1.00x	22%	n. a.	n. a.	-1.18x	53%	n.a.	n.a.
sat	10%	1.00x	22%	n. a.	n. a.	3.09x	55%	1.82x	n.a.
sat	15%	1.00x	22%	n. a.	n. a.	4.49x	58%	2.67x	n.a.
sat	20%	1.00x	22%	n. a.	n. a.	2.08x	55%	2.57x	62%

Fig. 13. Effects of optimization on the different tools for the *K-Means* configuration with 24 cores, 18 tasks and 160M input records.

Louvain		Uppaal DF		Uppaal BF		ae2sbvzot		ae2zot	
res.	deviation	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving	speedup	mem. saving
sat	0%	n. a.	n. a.	n. a.	n. a.	1.16x	66%	n. a.	n. a.
sat	4%	n. a.	n. a.	n. a.	n. a.	-3.13x	66%	n. a.	n. a.
sat	13%	n. a.	n. a.	n. a.	n. a.	7.87x	76%	n. a.	n. a.
sat	17%	n. a.	n. a.	n. a.	n. a.	36.34x	84%	n. a.	n. a.
sat	21%	n. a.	n. a.	n. a.	n. a.	7.67x	79%	n. a.	n. a.
sat	25%	n. a.	n. a.	n. a.	n. a.	3.57x	72%	n. a.	n. a.

Fig. 14. Effects of optimization on the different tools for *Louvain* with 128 cores, 160 tasks and 16M input records.

Table 2. Overview of the impacts of optimization on the two Zot plugins across the different use cases.

use case	*ae2sbvzot*		*ae2zot*	
	speedup	*memory saving*	*speedup*	*memory saving*
SortByKey	• 1.5×-2.6× speedup far from *mfd* (12 cores) • 1.1×-9× speedup (22 cores) • 1.45×-8.9× slowdown on deadlines close to *mfd* (12 cores)	• 7%-25% (12 cores) • 23%-34% (22 cores)	• 1.1×-7× speedup in most of cases (37/50) • 1.1×-3× slowdown in remaining cases (13/50)	• 20%-45%(12 cores) • 8%-17% (22 cores) • peak of 60% (unfeasible deadlines)
TPCH_22_6	• 1.5×-6.5× speedup	• 33%-44%	• > 3× speedup • > 2600× speedup with 14 hours timeout	• 40%-50%
TPCH_22_7	• > 1.2× speedup • > 8000× speedup with 8 hours timeout	• 32%-40% (16 cores) • 43%-48% (32 cores)	• > 4.1× speedup • > 780× speedup with 14 hours timeout	• 52%-68%
PageRank	• 1.1×-5.4× speedup (18/30 cases) • 1.2×-5.1× slowdown (10/30) • 11× and 29× slowdown in two cases	• > 15% • > 50% memory saving in most of cases (20/25)	• > 4× speedup in most of cases (18/27) • peak of 55× • 1.2×-2× slowdown in few cases (3/27)	• > 39% • > 60% memory saving in most of cases (19/27) • peak of 90%
SVM	• > 2× speedup in most of cases (16/23) • > 40× speedup with 12 hours timeout • 1.8×-4× slowdown in few cases (3/23)	• > 20% • > 40% memory saving in most of cases (14/18) • 11% memory increase in one case	• > 2× speedup • > 3.5× speedup with 36 hours timeout	• 48%-53%
K-Means	• 1.1×-23× speedup in most of cases (55/77) • 1.6×-9× slowdown in other cases (18/77)	• 40%-60%	• > 1.2× speedup • > 2.5× speedup with 30 hours timeout	• 45%-69%
Louvain	• 4.6×-36× speedup in most of cases (14/22) • 3× slowdown in a case (1/22)	• 66%-85%	• > 6× speedup with 60 hours timeout	• optimization avoids memory saturation

increase together with the number of nodes in the DAG. Moreover, in many configurations, the optimization makes the verification possible even with *ae2zot* (which is typically slower than *ae2sbvzot* in solving this kind of problems) in a reasonable time, whereas the unoptimized version reached the timeout.

Given the extreme efficiency of Uppaal in completing the verification in trivial cases (feasible deadlines far from *mfd*) when the depth-first search order is used, the speedup due to the optimization is negligible in many cases; the benefits are more significant when the deadlines are feasible and close to *mfd*, or when they are unfeasible. The impact in terms of memory saving is consistent across the different use cases. It is lower in percentage than the one registered for Zot, but in absolute terms it allows for the saving of a greater amount of memory, as Uppaal requires on average much more memory to perform the verification.

Due to its memory-demanding space exploration strategy, Uppaal breadth-first is not able to manage DAGs with more than 10 nodes. For smaller DAGs, the effects of the optimization are generally limited in

Table 3. Overview of the impacts of optimization on the two selected Uppaal configurations across the different use cases.

use case	***Uppaal Depth-First***		***Uppaal Breadth-First***	
	speedup	*memory saving*	*speedup*	*memory saving*
SortByKey	• 4×-15× speedup on feasible deadlines close to *mfd* • 2× speedup on unfeasible deadlines • Negligible impact for feasible deadlines far from *mfd*	• 40%-70% on feasible deadlines close to *mfd* • 5% on unfeasible deadlines • Negligible impact for feasible deadlines far from *mfd*	• 1.1× speedup	• 2%-3%
TPCH_22_6	• 1.3× speedup on unfeasible deadlines • Negligible impact for feasible deadlines	• 10%-20% on feasible deadlines • Negligible impact on unfeasible deadlines	• 1.3× speedup	• Negligible impact on memory
TPCH_22_7	• 1.3×-1.5× on unfeasible deadlines • Negligible impact for feasible deadlines	• 15%-25% on feasible deadlines • Negligible for unfeasible deadlines	• 1.43×	• Negligible impact on memory
PageRank	• 1.1×-7× speedup in most of cases (24/36)	• 20%-50% on feasible deadlines • 5%-15% on unfeasible deadlines	• 1.7×-1.8×	• 3%-8%
SVM	• Negligible impact on time	• 20%-30%	• Memory saturation	• Memory saturation
K-Means	• Oscillating from 3000× speedup to 100× slowdown for feasible deadlines close to *mfd* • Negligible for feasible deadlines far from *mfd*	• 22%-25% on most of cases (20/30) • 99% saving and 70% loss in a few outlier cases (5/30)	• Memory saturation	• Memory saturation
Louvain	• Memory saturation	• Memory saturation	• Memory saturation	• Memory saturation

terms of memory; they are more significant in terms of speedup, as the verification time in some cases is almost halved (1.8× speedup).

Overall, the experimental results in Fig. 10 to 14 demonstrate that the use of Dilworth's partitioning applied to Spark DAGs is essential and makes the approach scalable, as it enables the analysis of Spark applications with tens of nodes. The partitioning significantly reduces the size of the models, hence the memory footprint required to solve the feasibility problem. This fact is evident from the reduction of memory that can be observed in all the experiments and for all the verification approaches. In the case of *ae2zot* and *ae2sbvzot*, the reduction ranges from 7% to 90% of the memory footprint observed when partitioning is not applied, while in the case of Uppaal it ranges from 5% to 99%. A similar argument also holds for the execution time, even if the trend is not always uniform across the experiments, and a slowdown can be observed in some cases. However, this phenomenon is more often evidenced with *ae2zot* and *ae2sbvzot* than with Uppaal, and it is likely dependent on the heuristics implemented in the underlying solvers that Zot adopts to solve the CLTLoc formulae.

The number of cores used to execute the application in the target cluster is the second key parameter that influences the complexity of the analysis. The evaluation of the execution time for all the benchmark applications and for every configuration including 64 or 128 cores can always be done by using the logic-based model. Conversely, the TA-based models (both versions) have shown a more marked tendency to depend on

the number of cores, whose growth can trigger state-space explosion even when considering an application such as *K-Means*—which is not the biggest one in our benchmark—and a cluster with 64 cores.

Finally, we remark that the verification approach turned out to be accurate also in the analysis of *Louvain*, whose DAG, which includes 32 nodes, is the biggest considered in this work. The application was profiled with an input dataset that contains a randomly-generated graph of 16 millions vertexes, on two different configurations: the first one including 80 tasks on 64 cores and the second one including 160 tasks on 128 cores. As expected, *ae2sbvzot* turned out to be the best solver. The collected feasible deadlines, which led to an estimate of the *mfd* that differs from the exact one by less than 10% of the latter, were obtained with execution times that are comparable with those already observed for the other applications. In particular, the solving time needed to obtain an accuracy of 10% was approximately 2.3 hours, with 64 cores, and 24 hours, with 128 cores.

6. Related Work

This section describes a number of related works, and organize them into three distinct classes. Only few works propose specific techniques for the analysis of Spark applications, and no alternative approaches, that apply formal verification for the analysis of Spark applications, has been found in literature. Some works fall into the area of Operations Research, as the feasibility problem, posited for Spark applications, shares some similarities with the Job-shop scheduling problem. Finally, other works investigate optimization techniques for improving the efficiency of known verification algorithms.

6.1. Scheduling problems

The analysis of temporal properties of scheduling algorithms and of distributed systems has been addressed with positive outcomes using Timed Automata (TA, [AD94]) and Hybrid Automata (HA, [Hen00]). In [Cor96], TA are used for the analysis of the task scheduling of Ada programs in systems equipped with one CPU that executes both the scheduler and the Ada code. Unlike in standard schedulability analysis (e.g., [PGH98]), the use of TA—and, similarly, the use of CLTLoc in the present work—allows relevant properties of real implementations (e.g., resource constraints) to be captured. Moreover it allows for relaxing some restrictions on the software structure, that are needed for the analysis. A timed analysis for distributed systems has been addressed in [BHK99] by means of HA. HA model the execution of concurrent tasks on the available CPUs and the precedence relation among the tasks, which is specified by a graph of dependencies. The tasks are indivisible units of work with a fixed duration, they have a scheduling priority and can be preempted. [KWPH04] also uses TA to model distributed real-time applications. A distributed application consists of several concurrent tasks, each running on a single processor and communicating with the others via a network. TA are used to model the interaction among the tasks, the network (sender and receiver components) and the arbiter of the communication channel. Both the schedulability of the tasks and the application response-time are analyzed by using a state-of-the art model-checker for TA and for HA. Our model considers DAGs of stages similar to the graph of dependencies in [BHK99]. However, whereas tasks in [BHK99] and in [KWPH04] are atomic and are each executed on a single CPU, the execution of a Spark stage can be spread over different CPUs, complicating the model.

Operations Research (OR) offers a wide range of techniques for scheduling and planning problems. TA and their extensions are very effective tools to tackle non-standard problems that cannot be solved by using standard OR techniques. [BLR05] presents Priced TA (PTA), which extend TA with costs and are suitable for modeling scheduling problems with optimal goals. PTA allow for computing the minimum optimal cost of reaching a target configuration. Three standard problems of OR are dealt with using PTA and the experimental results, comparing the standard MILP-based approaches with the PTA algorithm, indicate that PTA are competitive and, in some cases, faster. The Job-shop problem, that [BLR05] addresses by means of PTA, and the extension with bounded delay uncertainty are addressed in [YAAM06] by using standard TA. The experimental results again demonstrate that the TA-based procedures applied to the problem can provide better outcomes, that is, more efficient schedules, than those produced with OR algorithms.

6.2. Simulation-based Approaches

In the domain of the analysis of Big Data frameworks, simulation, rather than formal verification is usually the approach of choice. For example, [PBM+17] considers the problem of computing the response-time of a Spark application through the simulation of a Stochastic Petri Net (SPN) model. The experimental results demonstrate that the error affecting the simulation is low (less than 10%) when the simulated application has a high number of tasks and cores (e.g., more than 12 cores and 200 tasks). For some configurations, however, the error is bigger than 30%. In [BAB+17] an ad-hoc fast event driven simulator, called *dagSIM*, has been used to simulate applications modeled as DAGs of nodes that represent the execution of batches of tasks whose average duration is described by means of a stochastic distribution. DagSIM predicts the application response time by means of a resolution procedure which is faster than the one based on SPN. However, simulation-based approaches—unlike verification-based ones—cannot offer *guarantees* about the feasibility of a desired deadline, and in particular they cannot be used to determine the *unfeasbility* of a deadline.

6.3. Model Optimizations

One of the main issues for automated verification through model checking is the *state space explosion* problem whereby the number of states of the model under analysis grows exponentially with respect to the number of components of the modeled system, sometimes making the verification of moderately large systems impractical.

Different strategies have been proposed to mitigate this problem. Two of the most relevant are *symmetry reduction* methods and *partial order reduction* methods. Symmetry reduction [CESPS98] is based on the observation that the state space underlying many systems contain sets of symmetrically equivalent states, corresponding to sets of practically identical system components. The complexity of the model can be dramatically reduced by replacing sets of equivalent states with a single representative from the equivalence class. The analysis is then performed on the reduced state space. Symmetry reduction has been implemented in many explicit state and symbolic state model checkers, and it has been applied to temporal logic [MDC06] and probabilistic model checking [DMP09]. The two techniques are different from that presented in this paper because they modify the way states are matched and stored during search, and so avoid constructing a complete model. Dilworth's partitioning theorem is applied to the model of Spark computations and transforms a model into a smaller one.

Partial orders are fundamental in the area of formal verification. The theory of well-structured transition systems [FS01] is based on the existence of a partial order of the system configurations that, under specific conditions, allows for a finite representation of an infinite state space. This fact, combined with an order-preserving transition relation, makes the definition of an algorithm solving the backward reachability problem possible. Optimization techniques that take advantage of partitioning mechanisms have been widely studied to decrease the number of representatives of system executions that satisfy a given property. Partial order reduction [CGP99, God96] exploits the fact that, often, there are sets of transitions whose order of execution does not affect the overall behavior of the system. Analogously to symmetry reduction, it only considers, in every state, one representative transition for every class of system transitions that can be executed from the state and that satisfy suitable properties. Partial order reduction has been implemented in state-of-the-art tools, such as SPIN [Hol97], as it significantly reduces the cost of the state space exploration algorithms, yielding therefore a potentially drastic reduction of the time and memory needed to carry out the verification.

The optimization presented in Sect. 4 also aims to reduce the state space to be analyzed, but it is orthogonal to the aforementioned strategies, as it acts at a different level of abstraction. Symmetry reduction and partial order reduction are general approaches that operate at the very low level of Kripke structures. Ours is a domain-specific optimization for Spark computations, and it operates at the (higher) model level. Domain-specific optimizations have been already adopted in the past to complement low-level optimization techniques. For instance, [BGM11] successfully applied optimization of models to improve the performance of the verification of distributed applications that implement a Publish/Subscribe communication paradigm.

7. Conclusions

This work shows that the estimation of the execution time of Spark applications can be done by using automata-based and logical-based formalisms with high accuracy for which (the error is less than 11%). In particular, the two adopted classes of models are TA and the formulae of CLTLoc, both capable of expressing timing constraints through dense clocks. The execution time of an application is determined by solving a limited number of the so-called p-feasibility problems on the DAG that describes the computation of the application, given the number of available cores allocated to the application. Being an instance of the p-feasibility problem reducible to the model-checking of a TA (reachability) or to the satisfiability of a CLTLoc formula, the estimation of the execution time of an application can be obtained by using standard tools that solve these problems on TA and CLTLoc. The proposed methodology has the following advantage. The adopted formalisms are solved by means of procedures that implement different approaches: TA can be solved by using standard algorithms that explore the state-space of the automata with an exhaustive search and CLTLoc formulae can be solved by means of a test for bounded satisfiability using SMT-solvers. While the former guarantees the completeness of the verification, but requires a large amount of memory, the latter is unlikely to reach the completeness bound but always assures a limited usage of memory. Even if the complexity of the problems to be solved to answer a p-feasibility problem is PSPACE, in some practical cases, the time or the memory required to analyze realistic applications still turn out to be too high. For this reason, this work also describes a technique that exploits a well-known result valid for partially ordered sets, and so also for DAGs, that lowers the practical complexity of the analysis. Dilworth's theorem states that a finite poset can be partitioned into n independent posets, where n is the cardinality of the largest set of incomparable elements. Dilworth's partioning is applied to the DAG of the application and allows for the construction of smaller models than the ones obtained without taking the partitioning into account. The technique is general as it applies to the DAG of the applications, with no distinction between TA-based models and CLTLoc-based models. The reduction of the size of the models turned out to be effective in all the experiments based on realistic applications and led to a significant reduction of both the verification time and memory usage. The investigation of the effects of the use of this technique in the verification of Spark applications has been improved with respect to our preliminary work and enriched with new case studies based on well-known applications.

Acknowledgments

This work has been partially supported by the DICE project (Horizon 2020 project no. 644869) and by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX).

References

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[BAB+17] A. Brito, D. Ardagna, I. Blanquer, A. Evangelinou, E. Barbierato, M. Gribaudo, J. Almeida, Couto. A.P., and T. Braga. D3.4 eubra-bigsea qos infrastructure services intermediate version. Technical report, EUBra-BIGSEA Consortium , 2017.

[BCC+03] A. Biere, A. Cimatti, E.C. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BDL+06] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, pages 125–126, Washington, DC, USA, 2006. IEEE Computer Society.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[BGM11] L. Baresi, C. Ghezzi, and L. Mottola. Loupe: Verifying publish-subscribe architectures with a magnifying lens. *IEEE Transactions on Software Engineering*, 37(2):228–246, March 2011.

[BHK99] S. Bradley, W. Henderson, and D. Kendall. Using timed automata for response time analysis of distributed real-time systems. In *24th IFAC/IFIP Workshop on Real-Time Programming*, pages 143–148, 1999.

[BLR05] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, March 2005.

[Bou09] P. Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323 – 341, 2009. Proceedings of the 5th Workshop on Methods for Modalities (M4M5 2007).

[BP98] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Int. World-Wide Web Conference (WWW)*, pages 107–117, 1998.

[BPKR16] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi. How Bit-vector Logic Can Help Improve the Verification of LTL Specifications over Infinite Domains. In *Proc. of the 31st Annual ACM Symposium on Applied Computing*, pages 1666–1673, 2016.

[BQ18] L. Baresi and G. Quattrocchi. Towards Vertically Scalable Spark Applications. In *Euro-Par 2018: Parallel Processing Workshops*. Springer, 2018.

[BRS16] M. M Bersani, M Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, 53(2):171–206, 2016.

[BRS17] M. M. Bersani, M. Rossi, and P. San Pietro. A logical characterization of timed regular languages. *Theoretical Computer Science*, 658:46 – 59, 2017.

[Bur98] C.J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, 1998.

[CCG+02] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 359–364, 2002.

[CESPS98] E.C. Clarke, E. A. Emerson, Jha. S., and A. Prasad Sistla. Symmetry reductions in model checking. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998.

[CGP99] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[Cor96] J. C. Corbett. Timing analysis of ada tasking programs. *IEEE Transactions on Software Engineering*, 22(7):461–483, Jul 1996.

[dag19] DAG-ver Project repository. github.com/deib-polimi/DAG-ver, 2019.

[DD07] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

[Dil50] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[DMP09] A. F. Donaldson, A. Miller, and D. Parker. Language-level symmetry reduction for probabilistic model checking. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 289–298. IEEE Computer Society, 2009.

[FS01] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63 – 92, 2001.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[GRB+17] E. Gianniti, A.M. Rizzi, E. Barbierato, M. Gribaudo, and D. Ardagna. Fluid Petri Nets for the Performance Evaluation of MapReduce and Spark Applications. *SIGMETRICS Performance Evaluation Review*, 44, 2017.

[Haz87] M. Hazewinkel. *Encyclopaedia of Mathematics (1)*. Encyclopaedia of Mathematics: An Updated and Annotated Translation of the Soviet "Mathematical Encyclopaedia". Springer, 1987.

[Hen00] T. A. Henzinger. *The Theory of Hybrid Automata*, pages 265–292. Springer Berlin Heidelberg, 2000.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279 –295, may 1997.

[IG04] S. Ikiz and V.K. Garg. Online algorithms for Dilworth's chain partition. *Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, University of Texas at Austin, Tech. Rep.*, 2004.

[JSBM15] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 435–448, 2015.

[KA10] K. Kc and K. Anyanwu. Scheduling Hadoop Jobs to Meet Deadlines. In *Proc. of the IEEE 2nd International Conference on Cloud Computing Technology and Science*. IEEE, 2010.

[KWPH04] J. Krakora, L. Waszniowski, P. Pisa, and Z. Hanzalek. Timed automata approach to real time distributed system verification. In *Proc. of the IEEE Int. Work. on Factory Communication Systems, 2004*, pages 407–410, Sept 2004.

[LHW+14] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace. WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters. In *Proc. of the IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014.

[MAA+14] Bersani M.M., Frigeri A., Morzenti A., Pradella M., Rossi M., and San Pietro P. Constraint LTL satisfiability checking without automata. *J. Applied Logic*, 12(4):522–557, 2014.

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[MDC06] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Computing Survey*, 38(3):8, 2006.

[MQB+18] F. Marconi, G. Quattrocchi, L. Baresi, M.M. Bersani, and M. Rossi. On the timed analysis of big-data applications. In Dutle. A., C. A. Muñoz, and A. Narkawicz, editors, *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, volume 10811 of *Lecture Notes in Computer Science*, pages 315–332. Springer, 2018.

[ORR+15] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making Sense of Performance in Data Ana-

lytics Frameworks. In *Proc. of the 12th USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2015.

[PBM+17] D. Perez, S. Bernardi, J. Merseguer, JoJ. Requeno, G. Casale, and L. Zhu. DICE simulation tools - Final version. Deliverable, DICE Consortium, 2017.

[PGH98] J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the IEEE Real-Time Sys. Symp.*, pages 26–37, Dec 1998.

[Pol18] Politecnico di Milano. The Zot Bounded Model/Satisfiability Checker. github.com/fm-polimi/zot, 2018.

[Wan04] F. Wang. Efficient verification of timed automata with bdd-like data structures. *International Journal on Software Tools for Technology Transfer*, 6(1):77–97, Jul 2004.

[WDR13] M.T.B. Waez, J. Dingel, and K. Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1 – 26, 2013.

[YAAM06] Y. Yasmina Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *Theor. Comput. Sci.*, 354(2):272–300, 2006.

[YCH15] J. Yu, H. Chen, and F. Hu. SASM: Improving Spark Performance with Adaptive Skew Mitigation. In *2015 IEEE International Conference on Progress in Informatics and Computing (PIC)*, Dec 2015.

A. Full CLTLoc model - referring to Sect. 3

A.1. Symbols description

A.1.1. Atomic Propositions

Node-specific Atomic Propositions

- $\texttt{runS}_i$
- $\texttt{startS}_i$
- $\texttt{endS}_i$
- $\texttt{completedS}_i$
- $\texttt{enabledS}_i$
- $\texttt{startEnabledS}_i$

Task-specific Atomic Propositions

- $\texttt{runT}_i$
- $\texttt{startT}_i$
- $\texttt{endT}_i$

A.1.2. Global Atomic Propositions

- `idleCores`

A.1.3. Counters

- $\texttt{runTC}_i$ - (*runningTakskCounter*) Number of tasks currently running for node i
- $\texttt{remTC}_i$ - (*remainingTasksCounter*) Number of tasks that still have to be executed for node i
- `avaCC` - (*availableCoresCounter*) Number of cores currently available (global).

A.1.4. Constants

- $|T_i|$ - Total number of tasks needed to complete stage i
- p - Total number of cores available in the cluster

A.2. Temporal logic model

A.2.1. Node formulae

The state evolution of each node is defined by the following constraints.

Normal Version

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{startT}_i \wedge \mathtt{remTC}_i = |T_i| \iff \mathtt{startS}_i) \tag{15}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{endT}_i \wedge \mathtt{remTC}_i = 0 \iff \mathtt{endS}_i) \tag{16}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{completedS}_i \iff \mathbf{P}(\mathtt{endS}_i)) \tag{17}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{enabledS}_i \iff \bigwedge_{\substack{j \in \mathbf{S}: \\ Dep(i,j)}} \mathtt{completedS}_j) \tag{18}$$

Labeling Version

$$\bigwedge_{i \in \mathbf{S}} \left(\mathtt{startS}_i \iff \left(\begin{array}{c} \mathtt{runS}_i \wedge \mathtt{startT}_{\ell(i)} \wedge \mathtt{remTC}_{\ell(i)} = |T_i| \\ \wedge \mathbf{Y}\mathtt{enabledS}_i \wedge \neg \mathtt{completedS}_i \end{array} \right) \right) \tag{19}$$

$$\bigwedge_{i \in \mathbf{S}} \left(\mathtt{endS}_i \iff \left(\begin{array}{c} \mathtt{runS}_i \wedge \mathtt{endT}_{\ell(i)} \wedge \mathtt{remTC}_{\ell(i)} = 0 \\ \wedge \mathbf{Y}\mathtt{enabledS}_i \wedge \neg \mathbf{Y}\mathtt{completedS}_i \end{array} \right) \right) \tag{20}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{completedS}_i \iff \mathbf{P}(\mathtt{endS}_i)) \tag{21}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{enabledS}_i \iff \bigwedge_{\substack{j \in \mathbf{S}: \\ Dep(i,j)}} \mathtt{completedS}_j) \tag{22}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{startEnabledS}_i \iff \mathtt{enabledS}_i \wedge \neg \mathbf{Y}\mathtt{enabledS}_i) \tag{23}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{runS}_i \Rightarrow \mathtt{runS}_i \mathbf{S} \mathtt{startS}_i \wedge \mathtt{runS}_i \mathbf{U} \mathtt{endS}_i) \tag{24}$$

A.2.2. Tasks formulae

Let `orig` be a shorthand for $\neg \mathbf{Y} \top$. The formula is true only in the origin. The behaviour of each batch of tasks is defined by the following formulae.

Normal Version

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{startT}_i \Rightarrow \mathtt{runT}_i \wedge \neg \mathtt{endT}_i \wedge \mathbf{Y}\mathtt{enabledS}_i \wedge \neg \mathtt{runT}_i \, \mathbf{S} \, (\mathtt{orig} \vee \mathtt{endT}_i) \wedge \mathbf{X}\mathtt{endT}_i \mathbf{R} \neg \mathtt{startT}_i) \tag{25}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{runT}_i \Rightarrow \mathtt{runS}_i \wedge (\mathtt{runT}_i \, \mathbf{S} \, \mathtt{startT}_i) \wedge (\mathtt{runT}_i \, \mathbf{U} \, \mathtt{endT}_i)) \tag{26}$$

$$\bigwedge_{i \in \mathbf{S}} (\mathtt{endT}_i \Rightarrow \mathtt{runT}_i \wedge \mathbf{Y} \neg \mathtt{endT}_i \, \mathbf{S} \, (\mathtt{orig} \vee \mathtt{startT}_i)) \tag{27}$$

Labeling Version

$$\bigwedge_{i \in \mathbf{L}} (\texttt{startT}_i \Rightarrow \texttt{runT}_i \wedge \neg \texttt{endT}_i \wedge \mathbf{Y} \neg \texttt{runT}_i \, \mathbf{S} \, (\texttt{orig} \vee \texttt{endT}_i) \wedge \mathbf{X} \texttt{endT}_i \mathbf{R} \neg \texttt{startT}_i) \tag{28}$$

$$\bigwedge_{i \in \mathbf{L}} (\texttt{runT}_i \Rightarrow \bigvee_{\substack{j \in \mathbf{S}: \\ \ell(j) = i}} (\texttt{runS}_j) \wedge (\texttt{runT}_i \, \mathbf{S} \, \texttt{startT}_i) \wedge (\texttt{runT}_i \, \mathbf{U} \, \texttt{endT}_i)) \tag{29}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{endT}_i \Rightarrow \texttt{runT}_i \wedge \mathbf{Y} \neg \texttt{endT}_i \, \mathbf{S} \, (\texttt{orig} \vee \texttt{startT}_i)) \tag{30}$$

A.2.3. Resource Constraints

The number of available cores in the system is constrained as follows:

Normal Version

$$\sum_{i \in \mathbf{S}} (\texttt{runTC}_i) + \texttt{avaCC} = p \tag{31}$$

Labeling Version

$$\sum_{i \in \mathbf{L}} (\texttt{runTC}_i) + \texttt{avaCC} = p \tag{32}$$

A.2.4. Counters Formulae

Normal Version

$$\bigwedge_{i \in \mathbf{S}} (\texttt{runTC}_i \geq 0 \wedge \texttt{runTC}_i \geq 0) \wedge \texttt{avaCC} \geq 0 \tag{33}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{runT}_i \Leftrightarrow \texttt{runTC}_i > 0) \tag{34}$$

$$\bigwedge_{i \in \mathbf{S}} ((\texttt{runTC}_i \neq \mathrm{X}\texttt{runTC}_i) \Rightarrow (\mathbf{X}\texttt{startT}_i \vee \texttt{endT}_i)) \tag{35}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{remTC}_i \geq \mathrm{X}\texttt{remTC}_i) \tag{36}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{remTC}_i \neq \mathrm{X}\texttt{remTC}_i \Rightarrow \mathbf{X}\texttt{endT}_i) \tag{37}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{endT}_i \Rightarrow (\texttt{remTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{runTC}_i)) \tag{38}$$

Labeling Version

$$\bigwedge_{i \in \mathbf{S}} (\texttt{startEnabledS}_i \Rightarrow \mathbf{X}\texttt{remTC}_{\ell(i)} = |T_i|) \tag{39}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{runS}_i \Rightarrow \texttt{remTC}_{\ell(i)} \leq \mathrm{Y}\texttt{remTC}_{\ell(i)}) \tag{40}$$

$$\bigwedge_{i \in \mathbf{L}} (\texttt{runTC}_i \geq 0 \wedge \texttt{runTC}_i \geq 0) \wedge \texttt{avaCC} \geq 0 \tag{41}$$

$$\bigwedge_{i \in \mathbf{L}} (\texttt{runT}_i \Leftrightarrow \texttt{runTC}_i > 0) \tag{42}$$

$$\bigwedge_{i \in \mathbf{L}} ((\texttt{runTC}_i \neq \mathrm{X}\texttt{runTC}_i) \Rightarrow (\mathbf{X}\texttt{startT}_i \vee \texttt{endT}_i)) \tag{43}$$

$$\bigwedge_{i \in \mathbf{L}} (\texttt{remTC}_i \neq \mathrm{Y}\texttt{remTC}_i \Rightarrow \texttt{endT}_i \vee \bigvee_{\substack{j \in \mathbf{S}: \\ \ell(j) = i}} (\mathbf{Y}\texttt{startEnabledS}_j)) \tag{44}$$

$$\bigwedge_{i \in \mathbf{L}} (\texttt{endT}_i \Rightarrow (\texttt{remTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{runTC}_i)) \tag{45}$$

A.2.5. Initialization

Normal Version

$$\bigwedge_{i \in \mathbf{S}} (\neg \texttt{runT}_i \wedge \neg \texttt{runS}_i) \tag{46}$$

$$\bigwedge_{i \in \mathbf{S}} (\texttt{remTC}_i = |T_i|) \wedge (\texttt{runTC}_i = 0)) \tag{47}$$

$$(\texttt{avaCC} = p) \tag{48}$$

Labeling Version

$$\bigwedge_{i \in \mathbf{L}} (\neg \texttt{runT}_i \wedge \texttt{runTC}_i = 0) \tag{49}$$

$$(\texttt{avaCC} = p) \tag{50}$$

A.2.6. Clocks Formulae

In order to represent the duration of the various processing phases of each node we introduce different clocks:

- $\texttt{clk}_{S_i}$ measures the duration of the $\texttt{runT}_j$ phase for each node/label j.

  Reset conditions

- $\texttt{clk}_{S_j}$ **for each node/label** - clock resets every time a new batch of tasks of node/label j starts running.

$$\bigwedge_{j \in \mathbf{S/L}} \left((\texttt{clk}_{S_j} = 0) \iff (\texttt{orig} \vee \texttt{startT}_j) \right) \tag{51}$$

Task running duration Single interval, **Normal** variant:

$$\bigwedge_{i \in S} \left(\begin{array}{l} \texttt{runT}_i \Rightarrow \\ \left(\begin{array}{l} \texttt{runT}_i \wedge \neg \texttt{endT}_i) \mathbf{U} ((\texttt{clk}_{S_i} \geq \alpha_i - \epsilon) \wedge (\texttt{clk}_{S_i} \leq \alpha_i + \epsilon) \wedge \texttt{endT}_i) \\ \wedge (\texttt{remTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{runTC}_i) \end{array} \right) \end{array} \right) \tag{52}$$

Single interval, **Labeling** variant:

$$\bigwedge_{i \in \mathbf{S}} \left(\begin{array}{l} (\texttt{runS}_i \wedge \texttt{runT}_{\ell(i)} \Rightarrow \\ \left(\begin{array}{l} (\texttt{runT}_{\ell(i)} \wedge \neg \texttt{endT}_{\ell(i)}) \mathbf{U} ((\texttt{clk}_{S_{\ell(i)}} \geq \alpha_i - \epsilon) \wedge (\texttt{clk}_{S_{\ell(i)}} \leq \alpha_i + \epsilon) \wedge \texttt{endT}_{\ell(i)}) \\ \wedge (\texttt{remTC}_{\ell(i)} = \mathrm{Y}\texttt{remTC}_{\ell(i)} - \texttt{runTC}_{\ell(i)}) \end{array} \right) \end{array} \right) \tag{53}$$

```
function LABEL(G)
  I, C = ∅
  for all node ∈ G do
    I = I ∪ ⟨node, getAncestors(node)⟩
    C = C ∪ {node}
  do
    oldC = C
    REDUCE(G, C, I)
  while C ≠ oldC
  id = 0
  for all chain in C do
    for all node in chain do
      node.label = i
    i = i + 1

function REDOCHAINS(C, a, b, PAIR)
  modtail(b, tail(a, C), C)
  p = PAIR[a]
  if p ≠ null then
    REDOCHAINS(C, p[0], p[1], PAIR)
  else
    C = C \ tail(a, C)

function REDUCE(G, C, I)
  N, M, PAIR, USED = ∅
  for all chain in C do
    N = N ∪ {chain[0]}
  while |N| > 0 do
    for all head in N do
      J = I[head]
      for all anc in J do
        if anc ∉ USED then
          T = tail(anc, C)
          if |T| > 1 then
            PAIR = PAIR ∪ ⟨T[1], (head, anc)⟩
            USED = USED ∪ {anc}
            M = M ∪ T[1]
          else return
            REDOCHAINS(C, head, anc, PAIR)
    N = M, M = ∅
```

Fig. 15. DAG labeling algorithm. (see Section 4)