# On the Timed Analysis of Big-Data Applications

Francesco Marconi, Giovanni Quattrocchi,
Luciano Baresi, Marcello M. Bersani, and Matteo Rossi

DEIB, Politecnico di Milano, Milan, Italy
{firstname.lastname}@polimi.it

**Abstract** Apache Spark is one of the best-known frameworks for executing big-data batch applications over a cluster of (virtual) machines. Defining the cluster (i.e., the number of machines and CPUs) to attain guarantees on the execution times (deadlines) of the application is indeed a trade-off between the cost of the infrastructure and the time needed to execute the application. Sizing the computational resources, in order to prevent cost overruns, can benefit from the use of formal models as a means to capture the execution time of applications.
Our model of Spark applications, based on the CLTLoc logic, is defined by considering the directed acyclic graph around which Spark programs are organized, the number of available CPUs, the number of tasks elaborated by the application, and the average execution times of tasks. If the outcome of the analysis is positive, then the execution is feasible—that is, it can be completed within a given time span. The analysis tool has been implemented on top of the Zot formal verification tool. A preliminary evaluation shows that our model is sufficiently accurate: the formal analysis identifies execution times that are close (the error is less than 10%) to those obtained by actually running the applications.

**Keywords:** Big-Data Applications, Metric Temporal Logic, Formal Verification, Apache Spark

## 1 Introduction

Many software systems produce huge quantities of data and their processing has been studied widely over the last years. Frameworks like Hadoop (hadoop.apache.org), Spark (spark.apache.org), and Flink (flink.apache.org), have been proposed to automate and ease the computation. These frameworks allow users to carry out batch processing over a cluster of (virtual) servers. The actual size of supplied data and the number of machines used impact the execution time by which the framework provides results. Unfortunately, the actual execution time is only known at the end of the computation, and estimations are mainly based on experience and domain-knowledge. In this context, guarantees over the quality of service are often stated as *deadline*s—i.e., the maximum acceptable response times for single executions of the applications. The availability of tools that can foresee execution times, and thus help sizing the cluster, would greatly

ease the adoption of these frameworks in contexts where time and costs are key drivers: the higher the cost (hence the more machines are available), the lower the overall response time.

The work presented in this paper is part of a larger research on a model-driven approach to the formal verification of Big Data frameworks carried out within the DICE project (www.dice-h2020.eu). In [15] we tackled the formal verification of data streaming applications based on the Storm framework. This paper focuses instead on Apache Spark, one of the best known frameworks for batch processing. Spark programs are internally represented as directed acyclic graphs (DAG) of operations. We propose the definition of formal models of Spark programs based on the CLTLoc [6] logic to allow for the validation of the required resources (virtual machines and CPU cores) given a deadline. A suitable formalization of the problem requires that the execution times of the different *tasks*—that is, of the different computation units—are properly modeled. Hence, we based the formal model on CLTLoc, a metric temporal logic over dense time that extends LTL with atomic constraints on clock variables. CLTLoc is supported by formal verification tools which allow users to analyze formulae in an automated manner [6,3]. CLTLoc was also used—and extended—to model Storm topologies in [15]; this unified modeling and verification approach opens the possibility to analyze applications that are built upon heterogeneous building blocks, some tailored to stream processing, and others to batch processing.

The proposed solution builds the DAG-based representation of the program and automatically translates it into the corresponding CLTLoc model. The user then must provide the deadline, the number of available CPUs, the number of tasks elaborated by the application, and the average execution time of the different task types (e.g., obtained by profiling the program of interest). If the outcome of the analysis is positive, then the execution is feasible—that is, it can be completed by the given deadline. The prototype tool is implemented on top of Zot[1], our verification tool for solving the bounded satisfiability problem for CLTLoc, and a first evaluation witnesses good prediction capabilities with an error that is usually less than 10%.

The rest of the paper is organized as follows: Sect. 2 introduces Spark and the CLTLoc logic; Sect. 3 presents the formal model; Sect. 4 discusses an experimental evaluation of the approach; Sect. 5 surveys related solutions, and Sect. 6 concludes.

## 2 Background

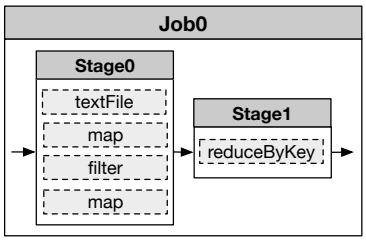
### 2.1 Apache Spark Framework

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 part of the computation to each *worker*. The main programming abstraction in Spark is the *RDD* (resilient distributed dataset), i.e., immutable and fault-tolerant collections of homogeneous objects. An RDD is distributely

[1] github.com/fm-polimi/zot

```
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'] | | | |
| 'e:3:true' | ['e','3','true'] | ['a','2','true'] | ('a', 2) | |
| 'a:3:false' | ['a','3','false'] | ['e','3','true'] | ('e', 3) | ('a', 7) |
| 'u:1:true' | ['u','1','true'] | ['u','1','true'] | ('u', 1) | ('e', 3) |
| 'o:4:false' | ['o','4','false'] | ['u','5','true'] | ('u', 5) | ('u', 6) |
| 'u:5:true' | ['u','5','true'] | ['a','5','true'] | ('a', 5) | |
| 'a:5:true' | ['a','5','true'] | | | |

(c) Data

Figure 1: Example of Spark application.

stored into workers by means of multiple redundant partitions to facilitate parallel computation. The act of a worker to read from another worker's memory or storage is called data shuffling. RDDs can be persisted in memory to improve performance through reuse. This makes Spark particularly efficient when executing iterative algorithms (e.g., machine learning and graph computations).

RDDs support two kinds of operations: *transformation*s (e.g., map, filter) create new RDDs, while *action*s (e.g., count, collect) perform computations to generate values. The former are *lazy*: they are chained together for optimization purposes, and are performed only when an action is encountered. Spark distinguishes between *narrow* and *wide* transformations, where the former do not reshuffle data (e.g., map, filter), whereas the latter do (e.g., reduceByKey).

To fully comprehend how Spark works one must first understand how the logic of a particular application is broken down into parallelized tasks. Figure 1a shows the code (in Scala) of an example Spark application that performs a simple aggregation over a dataset read from a text file containing in each line a vowel, a number and a Boolean separated by colons. The goal of the program is to sum the numbers that are labeled with the same vowel which are also not marked as *false*. To do that the program chains different operations: i) a *map* transforms each line in an array of strings by splitting it when a colon is encountered; ii) a *filter* discards the unnecessary lines (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) a *reduceByKey* is used to sum the numbers that share the same key; finally v) the dataset is returned using function *collect*. Figure 1c shows how an example dataset is transformed at each step. As soon as an application is submitted to Spark it is divided into multiple

*job*s. A job is a group of operations delimited by the presence of Spark actions within the code. When a job is scheduled for execution, a directed acyclic graph (DAG) of *stages* is created. Stages are delimited by operations that would require data shuffling, thus breaking data locality. Spark DAGs define the order among the stages of a job: two stages are connected if the second stage must read the data produced by the first, thus *a stage can 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 for the stage. A task executes all the trasformations that compose a stage over a single partition of its input RDD. Tasks are executed in parallel and are considered units of computation. Therefore, each task is executed by a single core and it is scheduled only when a core of a worker becomes free.

Figure 1b shows how logically the example program of Figure 1a is executed by Spark. Each rectangle inside a stage is an RDD that is produced by performing the associated operation; the arrows define the ordering relation between stages Stage0 and Stage1 (i.e, a DAG made of two nodes executed in sequence). Due to the lazy evaluation of transformations nothing happens until `collect` is executed; at that moment Spark allocates a job by creating a DAG of stages. Because *map* and *filter* do not require data shuffling, the first four operations are grouped in a single stage (*Stage0*). Conversely, *reduceByKey* requires an exchange of data among workers since tuples with the same key are not guaranteed to be all in the same data partition. For this reason *Stage1* is created. *Stage1* depends on *Stage0* and so it can be scheduled only when the first has completed its execution.

### 2.2 Constraint LTL over-clocks

The temporal logic model of Sect. 3 is expressed in terms of the CLTLoc logic [6] augmented with discrete counters, an extension of LTL allowing clock variables and arithmetical variables to occur in atomic formulae.

Atomic formulae over $(\mathbb{R}, \{<, =\})$ contain arithmetical variables, called *clock variables* (or simply clocks), which behave as clocks of Timed Automata [2]. A clock $x$ measures the time elapsed since the last "reset" of $x$, which occurs when $x = 0$ holds. Since the values of clocks can be compared with constants in formulae of the form $x \sim c$ (where $c \in \mathbb{N}$ and $\sim \in \{<, =\}$), clocks are used to constrain the time elapsing between the events that characterize Spark computations.

Atomic formulae over $(\mathbb{N}, \{<, =\}, +, 0, 1)$ predicate over arithmetical variables, called *counters*, that have no semantic restrictions. For instance, an atomic formula is $y + z < 4$, where both $y$ and $z$ are in $\mathbb{N}$. A counter stores a value that can be incremented, decremented and tested against a constant value. The logic exploits a special modality X applied to counters, that has been already introduced in [11], with the following meaning: if $y$ is a counter, X$y$ is the value of $y$ in the next position of time. Using modality X the increment of $y$ by 1 is expressed by the formula $\mathrm{X}y = y + 1$ whereas $y = \mathrm{X}y + 1$ indicates a decrement of $y$ by 1. Counters are used in the model of Sect. 3 to represent the amount of tasks that are elaborated by Spark applications.

Let $V$ be a finite set of variables over $\mathbb{N}$, $C$ a finite set of clock variables over $\mathbb{R}$ and $AP$ a finite set of atomic propositions. Atomic formulae $\theta$ over $V$ are quantifier-free Presburger formulae over terms $\alpha$ of the form $y$ or X$y$, with $y \in V$. CLTLoc formulae $\phi$ with counters are defined as:

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

where $p \in AP$, $x \in C$, $c \in \mathbb{N}$, $\sim \in \{<, =\}$, 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 $(\pi, \sigma)$, where $\pi : \mathbb{N} \to \wp(AP)$, and $\sigma : \mathbb{N} \times \{C \cup V\} \to \mathbb{R}$ is a mapping associating every variable in $C \cup V$ with a value in $\mathbb{R}$, but restricting values of the elements in $V$ to $\mathbb{N}$. The semantics of CLTLoc is defined as for LTL, except for formulae $x \sim c$ and $\theta$. Let $A_V$ be the ordered set of all terms of the form $y$ and X$y$, with $y \in V$, and let $n$ be its cardinality; for each $\alpha_j \in A_V$, its 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 $y \in V$. Given a mapping $v : A_V \to \mathbb{N}$, $\theta[v(\alpha_0), \ldots, v(\alpha_{n-1})]$ is the valuation of $\theta$ through $v$, which is obtained by replacing each term $\alpha_j$ occurring in $\theta$ with value $v(\alpha_j)$. If $\theta[v(\alpha_0), \ldots, v(\alpha_{n-1})]$ holds, we write $v \models \theta$. Let $t(\alpha_j) = y$ if $\alpha_j$ is either $y$ or X$y$. The following properties hold for each $i \in \mathbb{N}$:

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

If $\phi$ is a formula, interpretation $(\pi, \sigma)$ is a *model* for $\phi$ if $(\pi, \sigma), 0 \models \phi$ holds.

The satisfiability problem CLTLoc is decidable [6] and can be practically computed through a Bounded Satisfiability Checking approach [6,3]. Conversely, CLTLoc with Presburger arithmetics is undecidable, since so is its subset without clocks, CLTL [11], as the unboundedness of the domain of the counters and modality X allow the logic to encode the computations of 2-counter machines. Even if our formal model of Spark computations is based on CLTLoc with counters, the value of arithmetical variables occurring therein is bounded by some value that depends on the problem instance (see Sect. 3). Therefore, the technique introduced in [6,3] can still be exploited to solve the satisfiability problem for any instance of the model.

## 3 Modeling Spark Applications

This section presents the formal definition of the problem that we consider for the analysis of Spark applications and the temporal model that has been devised to solve it. Some assumptions are needed to abstract the Spark computation from details that are related to the physical infrastructure running the Spark framework and that depend on implementation aspects of the applications.

**Assumptions and level of abstraction** We make the following assumptions.

*The cluster running the Spark application is composed of homogeneous machines.* The workload of the cluster executing the application is not subject to oscillations that might alter the execution of the running jobs; hence, the performance of the cluster is stable and does not vary over time. The number of nodes in the cluster and the network latency are not explicitly represented in the model. However, they are strictly correlated as the more nodes are in the cluster, the higher the latency will be. For this reason, we decided to synthesize their effect as a single term to be included as an overhead to the task durations.

*Some features of the runtime environment of Spark are simplified;* for instance, the interaction among master and workers is not taken into account. The latency generated by the execution of services managing tasks is considered negligible with respect to the total execution time of the application.

*The input dataset provided to the application is homogeneous*; that is, the possible skewness of data is not taken into account. All tasks constituting a stage have durations that can vary non-deterministically by at most a fraction of a nominal value.

*The number of CPU cores that are available to the application is known* before starting the execution of the job and it does not vary over the computation.

*The functional aspects of executed operations are not directly considered in the model;* only their effect in terms of temporal behavior is represented.

The model is focused on the execution DAG underlying the application and it is based on an abstraction of the temporal behavior of stages and the tasks they are composed of. As explained in Sect. 2.1, the sequence of operations included in each stage is applied (possibly in parallel) on all partitions of the input dataset of the stage by means of a set of homogeneous tasks.

**Problem statement** 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 subset of $S \times S$ representing the precedence relation among stages. Let $\bar{T}_i$ be a finite set of homogeneous tasks associated with $S_i$ such that any pair of tasks $(\bar{T}_i, \bar{T}_{i'})$ are disjoint for any $0 \leq i, i' < N$ (with $i \neq i'$) and let $\bar{T}$ be the set $\bigcup_i \bar{T}_i$. Hereafter, variables $i, j$ are such that $0 \leq i < N$ and $0 \leq j < K$ hold.

An *execution* $\eta$ of $D$ with tasks in $\bar{T}$ is a finite sequence of $K$ tuples $t^0, t^1, \ldots, t^{K-1}$ of the form $t^j = (T_0^j, \ldots, T_{N-1}^j)$, called *execution steps*, where each set of *active tasks* $T_i^j$ is a—possibly empty—subset of $\bar{T}_i$ satisfying the following constraints: (i) for every stage $S_i$, each task in $\bar{T}_i$ appears in the execution sequence exactly once; also, if some task of $\bar{T}_i$ occurs at step $j$, then all tasks associated with all stages $S_i'$ preceding $S_i$ with respect to $E$ occur before $j$; (ii) for each step there is at least one set of active tasks. A non-empty set $T_i^j$ of tasks is called a *batch* of active tasks.

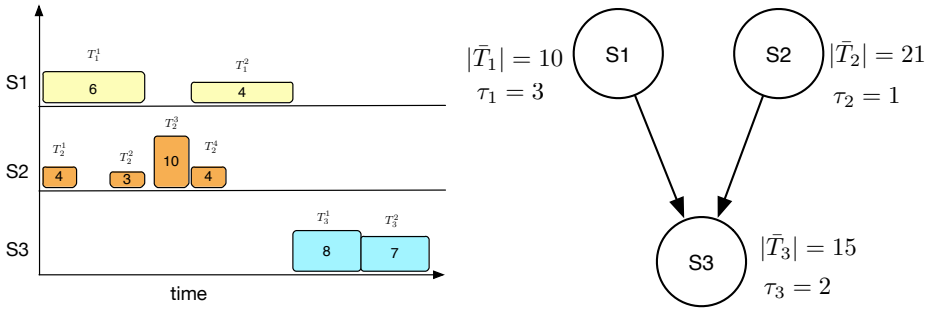
For any stage $S_i$ in $S$, let $\tau_i$ be a strictly positive constant in $\mathbb{R}$ defining the time needed to compute a generic task of $T_i$. Let $I$ and $I'$ be two convex and

bounded sets in $\mathbb{R}$. We say that $I$ precedes $I'$ when all the elements in $I$ are strictly smaller than all the elements in $I'$. Given an execution $\eta$ for $D$, define function $active(t)$ specifying the set of active tasks of $\bar{T}$ at any time instant $t$, such that for every $t \in \mathbb{R}$: (i) if a batch $T_i^j$ is active at $t$, then there is an interval $I$ of $\tau_i$ 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 $I$; (ii) every batch $T_i^j$ is eventually active; (iii) if batch $T_i^j$ occurs before batch $T_i^{j'}$ in $\eta$ (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.

Given an integer $p > 0$, an execution $\eta = t^0, t^1, \ldots, t^{K-1}$ for $D$ is *feasible* if $|active(t)| \leq p$, for all $t \geq 0$. The *time span* $ts(\eta)$ of $\eta$ is defined as the maximum time instant where at least one task in active.

The *feasibility problem* for a Spark application is defined as follows. Let $D$ be a DAG $(S, E)$ of $N$ stages, let $\bar{T}_i$, $\tau_i$ and $p$ be defined as before and let $d$ be a strictly positive integer. A solution of the feasibility problem for $D$ with tasks in $\bar{T}$ is a feasible execution $\eta = t^0, t^1, \ldots, t^{K-1}$ 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 $FD$.

Figure 2 shows a possible execution $\eta$ for the DAG depicted in Fig. 2b whose stages $S_1$, $S_2$ and $S_3$ execute, respectively, 10, 21 and 15 tasks, grouped into the sets $\bar{T}_1$, $\bar{T}_2$ and $\bar{T}_3$. Every rectangle represent a batch of running tasks and the number written therein is the size of the batch, i.e., the cardinality $|T_i^j|$. Stage 1 and 3 consists of two batches while Stage 2 is executed by means of 4 batches. The number of cores $p$ is equal to 10, hence, in every time instant, the number of running tasks is limited by 10. Assuming that the time delay between $T_1^1$ and $T_1^2$ is 1.3 time units ($\tau_2$ is 1 time unit), then the duration of the computation $ts(\eta)$ is 26.3 time units.




(a) Execution (b) DAG

Figure 2: Possible execution (2a) of the DAG in (2b).

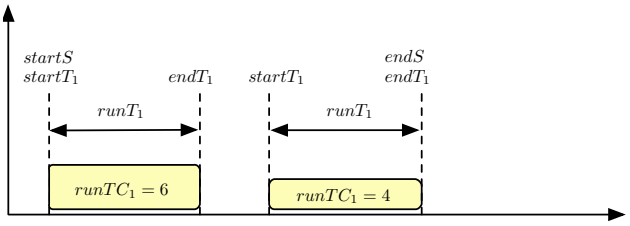



Figure 3: Atomic propositions and discrete variables used to model the running batches and the overall computation of stage $S_1$

**Temporal Logic Model of Spark Applications** Consider a Spark execution DAG $(S, E)$. Suppose that the application is running on a cluster with $p$ available cores, and each stage $S_i$ is executed by running $|\bar{T}_i|$ tasks. To represent the set of possible executions of the system, the CLTLoc model makes use of finite sets of atomic propositions, of discrete counters and of clocks. Atomic propositions are used to model the current status of stages and their tasks (i. e., whether they are started, running or completed), whereas the counters are used to keep track of the number of CPU cores that are either available, or are allocated to run the active tasks. Finally, the temporal constraints on the different tasks are expressed thanks to clocks.

Figure 3 shows the atomic propositions that are used to model the computation of the stage $S_1$, that is part of the DAG in Fig. 2b, according to the execution shown in Fig. 2a. Atoms $\texttt{startS}_1$ and $\texttt{endS}_1$ indicate the beginning and the end of the computation entailed by stage $S_1$, that is, the time instant where the first batch starts and the time instant where the last batch terminates. Batches of tasks are represented by means of $\texttt{startT}_1$, $\texttt{endT}_1$ and $\texttt{runTC}_1$ that indicate, respectively, the beginning and the end of a batch and that the batch is currently active. The value of variable $\texttt{runTC}_1$ is the number of tasks that are currently in execution, hence it corresponds to the value $|T_1^j|$, for $j \in \{1, 2\}$, representing the cardinality of the active batch.

Corresponding to the three kinds of variables mentioned above, three groups of formulae can be identified in the model: those capturing the evolution of the state of stages and tasks; those constraining the number of tasks in execution with respect to the available cores; and the set of constraints on clocks. The three groups of formulae are presented in the rest of this section. Notice that all formulae presented in this section are implicitly universally quantified over time through the **G** temporal operator.

*State formulae for stages* A stage $S_i$ can be either running (i. e., the atomic proposition $\texttt{runS}_i$ holds) or not running. A stage becomes running—i.e., $\texttt{startS}_i$ holds—when there is at least one task that starts the execution and no task 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 elaborate ($||\bar{T}_i||$). This situation is

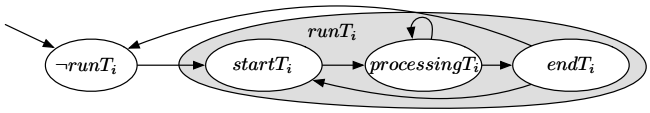



Figure 4: Finite state machine representing the state evolution of a set of tasks.

modeled through the following Formula (1).

$$\bigwedge_{S_i \in S} (\texttt{startT}_i \wedge \texttt{remTC}_i = ||\bar{T}_i|| \iff \texttt{startS}_i) \tag{1}$$

A stage terminates—i.e., $\texttt{endS}_i$ holds—when there are no more tasks to be processed—i. e., when $\texttt{remTC}_i$ is equal to 0. This is defined by Formula (2) below.

$$\bigwedge_{S_i \in S} (\texttt{endT}_i \wedge \texttt{remTC}_i = 0 \iff \texttt{endS}_i) \tag{2}$$

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

$$\bigwedge_{S_i \in S} (\texttt{completedS}_i \iff \mathbf{P}(\texttt{endS}_i)) \tag{3}$$

$$\bigwedge_{S_i \in S} (\texttt{enabledS}_i \iff \bigwedge_{S_j \in S,\ (S_i,S_j) \in E} \texttt{completedS}_j) \tag{4}$$

*State formulae for tasks* The behaviour of each batch of tasks is summarized in Fig. 4. Initially, for each stage $S_i$, the corresponding batch of tasks is not running ($\neg\texttt{runT}_i$ holds). 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 (which are explained later, when describing counter-related formulae) must hold. Every execution of a batch is characterized by an initial state (in which $\texttt{startT}_i$ holds) and a final state (in which $\texttt{endT}_i$ holds). $\texttt{processingT}_\texttt{i}$ is true in all time instants strictly included between the start and the end of a batch processing, and corresponds to $\texttt{runT}_i \wedge \neg\texttt{startT}_i \wedge \neg\texttt{endT}_i$. This execution cycle can be repeated many times depending on the available resources and the number of tasks to be executed. Being the batches of a stage sequential, they never overlap. Hence, atoms $\texttt{runT}_i$, $\texttt{startT}_i$ and $\texttt{endT}_i$ are used to model any active batch $T_i^j$, as they can be safely reused to model all the batches required to complete a stage. For brevity, the CLTLoc formulae capturing the behavior of the state machine of Fig. 4 are not shown here.

*Counter-related Formulae* Counter variables are used to define the constraints on system resources and the evolution of the tasks that are executed within the

stage. For example, Formula (5) translates the constraint $|active(t)| < p$, for any $t$, given in the problem statement. It limits the number of cores that are allocated to execute the active tasks. In particular, the sum of the number of available ($\texttt{avaCC}$) and allocated cores is always equal to $p$, the number of cores that is assigned to the job. The number of the remaining tasks of a stage decreases during its execution: Formula (6) imposes that the next value of $\texttt{remTC}_i$ (i.e., $\mathrm{X}\texttt{remTC}_i$) is not greater than the value of $\texttt{remTC}_i$ in the current position.

$$\sum_{S_i \in S} (\texttt{runTC}_i) + \texttt{avaCC} = p \tag{5}$$

$$\bigwedge_{S_i \in S} (\texttt{remTC}_i \geq \mathrm{X}\texttt{remTC}_i) \tag{6}$$

The following formulae link the truth value of the events $\texttt{startT}_i$ and $\texttt{endT}_i$ with the value of counters $\texttt{runTC}_i$ and $\texttt{remTC}_i$. Formula (7) correlates variable $\texttt{runTC}_i$ with proposition $\texttt{runT}$ by imposing that a batch is running (i.e., $\texttt{runT}$ holds) when the value of $\texttt{runTC}_i$ of active tasks is strictly positive. The two formulae (8) and (9) determine the value of $\texttt{runTC}_i$ and $\texttt{remTC}_i$ during the execution of the batch. Since the model is not designed to represent core re-balancing operations, the formulae enforce a variation of $\texttt{runTC}_i$ or $\texttt{remTC}_i$ to occur when a batch starts or terminates. In particular, Formula (8) 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. Therefore, between $\texttt{startT}_i$ and $\texttt{endT}_i$ $\texttt{runTC}_i$ cannot vary. Similarly, Formula (9) imposes that a variation of the value of $\texttt{remTC}_i$ is the sufficient condition to activate the execution of a batch (i.e., $\texttt{startT}_i$ holds). Finally, Formula (10) defines the relation between the variables $\texttt{runTC}_i$ and $\texttt{remTC}_i$. It states that, if the execution of a batch of tasks is starting, the number $\texttt{runTC}_i$ of running tasks in the batch is the difference of the (number of) remaining tasks at the beginning of the batch (i.e., value $\texttt{remTC}_i$) and the remaining tasks in the preceding position (i.e., value $\mathrm{Y}\texttt{remTC}_i$).

$$\bigwedge_{S_i \in S} (\texttt{runT}_i \Leftrightarrow \texttt{runTC}_i > 0) \tag{7}$$

$$\bigwedge_{S_i \in S} ((\texttt{runTC}_i \neq \mathrm{X}\texttt{runTC}_i) \Rightarrow (\mathbf{X}\texttt{startT}_i \vee \texttt{endT}_i)) \tag{8}$$

$$\bigwedge_{S_i \in S} (\texttt{remTC}_i \neq \mathrm{X}\texttt{remTC}_i \Rightarrow \mathbf{X}\texttt{startT}_i) \tag{9}$$

$$\bigwedge_{S_i \in S} (\texttt{startT}_i \Rightarrow (\texttt{runTC}_i = \mathrm{Y}\texttt{remTC}_i - \texttt{remTC}_i)) \tag{10}$$

*Constraints on clocks* To represent the durations of events in the model, a clock variable $\texttt{clock}_{\texttt{runT}_i}$ has been defined for each stage $S_i$. Specifically, $\texttt{clock}_{\texttt{runT}_i}$ measures the duration of the $\texttt{runT}_i$ phases for each batch of tasks of stage $S_i$. The following formula defines the reset conditions for the clocks: $\texttt{clock}_{\texttt{runT}_i}$ is

reset every time a new batch of tasks starts running for stage $S_i$.

$$\bigwedge_{S_i \in S} ((\mathtt{clock}_{\mathtt{runT}_i} = 0) \iff (\mathtt{orig} \vee \mathtt{startT}_i)) \tag{11}$$

Formula (12) 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 $\mathtt{clock}_{\mathtt{runT}_i}$ is in interval $[\tau_i - \epsilon, \tau_i + \epsilon]$, where $\tau_i$ is the average task duration of stage $S_i$ which is given as a parameter to the model, and $\epsilon$ is a constant defining the variability in the processing duration with respect to $\tau_i$. If there is a batch currently running (i.e., $\mathtt{runT}_i$ holds) then $\mathtt{runT}_i$ holds until an instant when the value of clock $\mathtt{clock}_{\mathtt{runT}_i}$ is in $[\tau_i - \epsilon, \tau_i + \epsilon]$ and $\mathtt{endT}_i$ is true.

$$\bigwedge_{S_i \in S} \begin{pmatrix} \mathtt{runT}_i \Rightarrow \\ (\mathtt{runT}_i \wedge \neg \mathtt{endT}_i) \mathbf{U} ((\mathtt{clock}_{\mathtt{runT}_i} \geq \tau_i - \epsilon) \wedge (\mathtt{clock}_{\mathtt{runT}_i} \leq \tau_i + \epsilon) \wedge \mathtt{endT}_i) \end{pmatrix} \tag{12}$$

*Initialization* The initial condition of any modeled Spark application obeys the following constraints: (i) no tasks are running in the origin; (ii) for each stage $S_i$, the number of remaining tasks is $|\bar{T}_i|$; (iii) the number of available cores $\mathtt{avaCC}$ is the total number of cores $p$.

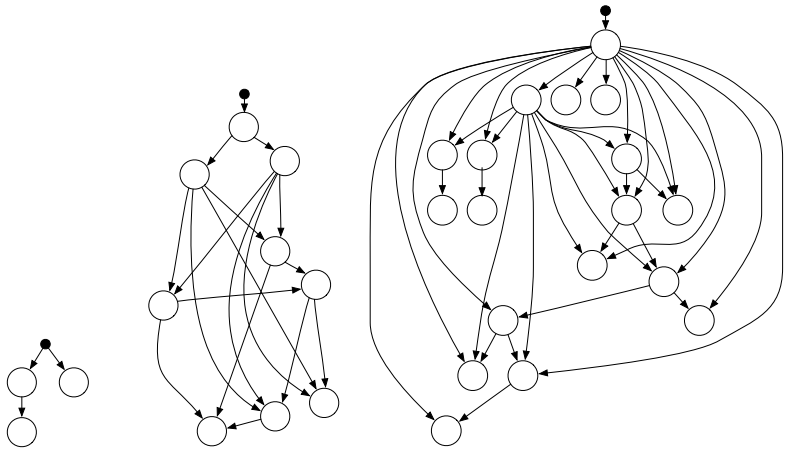
## 4 Implementation and Validation of the Model

The goals of this section are twofold. First, it briefly introduces the prototype tool that automatically generates CLTLoc formal models from high-level descriptions of Spark DAGs. Second, it presents a set of experiments carried out with real-life Spark applications to evaluate the effectiveness of the approach. The validation focuses on understanding the accuracy with which the model is able to identify the actual deadline that can be met by an implemented application.

The implemented prototype tool, D-VerT[2], takes as input a configuration file describing the Spark application to be analyzed, and uses a templating mechanism to automatically generate the corresponding formulae. The configuration file contains all the relevant information for running the analysis: the structure of the DAG, the number of tasks and the duration $\tau_i$ for each stage $i$, the deadline against which the feasibility analysis has to be performed, the number of cores in the cluster and the number of time positions to be considered for running the verification. DAG structure and timing information can be either manually provided or automatically generated by means of a benchmarking tool[3] which, as explained later in this section, allows for the profiling of running applications and provides an estimation of the timing characteristics for different settings.

D-VerT produces the corresponding instance of the formal model of Sect. 3 in the input format of the Zot verification tool, which is able to analyze CLTLoc

[2] github.com/dice-project/DICE-Verification
[3] github.com/franco-maroni/xSpark-bench



(a) Sort-by-Key (b) PageRank (c) KMeans

Figure 5: DAGs of selected applications.

formulae, and feeds the model to Zot. It then collects the outcome of the formal analysis and provides, when possible, a graphical representation of the results for better readability. The use of a declarative, logic-based modeling approach facilitates this automatic process, since the formulae are easily generalizable to any kind of DAG structure. Further details on the D-VerT toolchain can be found in [5].

We selected three well-known applications to perform the analysis and evaluate it against realistic use cases: the simple *SortByKey* operation; the graph processing algorithm *PageRank* [8]; and the clustering procedure *K-Means* [14]. As depicted in Fig. 5, the execution DAG of the three use cases have different size and level of complexity. To evaluate the model with respect to a variety of scenarios, for each one of these applications we selected six different settings in terms of both the configuration of the underlying cluster (i. e., two different numbers of available cores for each cluster node), and the configuration of the single application (i. e., three different dimensions of the input dataset and same number of partitions used for each stage). Next, we performed a profiling activity that consisted in launching several times the different applications using two different versions of Spark: one, called from now on *sequential Spark*, was slightly modified by us and the other was the regular version of Spark (i.e., *vanilla* Spark). For both cases we collected the timing information of all the stages and tasks. Our modifications in *sequential* Spark force the scheduler to launch all the stages sequentially (i. e., no more than one stage can be simultaneously in execution), allowing us to cleanly isolate the durations of each stage and its tasks, without the noise introduced by the concurrent execution of multiple stages. These durations were used to automatically generate the configuration files, therefore to instantiate the formal model in its different settings. On the other hand, the *average execution time*s

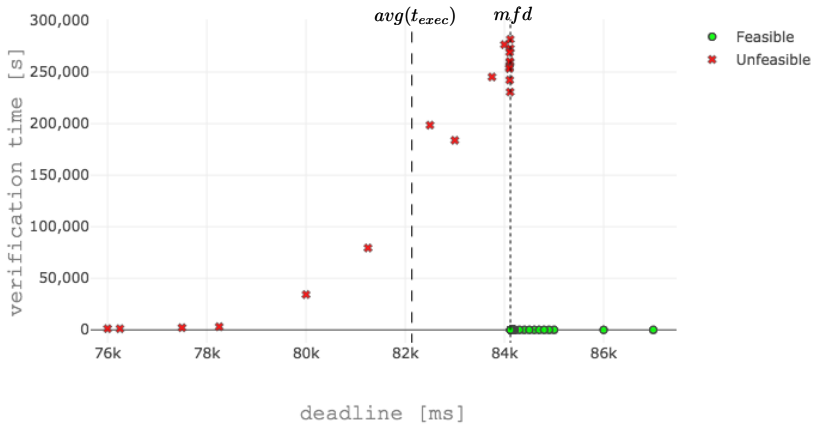



Figure 6: Times and outcomes of the verification tasks on the *SortByKey* use case (22 cores, 100 tasks and 300M input records) by providing different deadlines.

of the entire applications collected on *vanilla* Spark (from now on $avg(t_{exec})$), were used as the reference against which to compare the results of the analysis. We performed various verification tasks on each instance of the formal model to identify, for each configuration, the estimated set of feasible deadlines $FD$. Once the set was identified, we compared the *minimum feasible deadline* (*mfd*) found with the corresponding $avg(t_{exec})$, and we used the difference between them to evaluate the accuracy of the model (expressed as the percentage error *err*).

The first use case we considered is *SortByKey*. After an extensive analysis by means of multiple verification runs (each of them with a different deadline), we were able to identify the feasibility sets and the minimum feasible deadline, considering the granularity of the milliseconds. Figure 6 shows a comprehensive view of the verification tasks, performed on a single setting of the use case, with their outcomes (feasible/unfeasible) and the corresponding verification times. The *mfd* found was 84120 ms, therefore all the deadlines higher than that are feasible. On the other hand, deadlines of 84119 and below resulted unfeasible. Since, for this setting, $avg(t_{exec})$ was 82133 ms, the percentage error *err* is about 2.4%. This analysis highlighted a strong dependency of the verification time on the closeness of the analyzed deadline to the minimum feasible deadline. As reported in Fig. 6, verification time is in the order of the seconds for all deadlines lower than 75000 ms or greater than or equal to 84120 ms (*mfd*), whereas it grows exponentially for increasing deadline values between 75000 ms and 84119 ms, peaking at around 78 hours for 84117 ms. The notable growth is therefore registered for those deadlines that resulted unfeasible, but close to *mfd*. This pattern has been observed also for the other, more complex, applications we analyzed. However, since the verification times grow significantly with the size of the DAG (the analysis for feasible deadlines is generally completed in the order of minutes for *PageRank* and in the order of hours for *K-Means*) the time needed to

Table 1: Experimental results[4].

| *app* | *cores* | *tasks* | $records_{in}$ | $avg(t_{exec})$ (ms) | *mfd* (ms) | *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 | 84120 | 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 | 79000 | 1.7% |
| | | | 120M | 103492 | 107000 | 3% |
| | | | 160M | 131600 | 140000 | 6% |
| | 32 | 24 | 80M | 64565 | 63000 | 2% |
| | | | 120M | 81299 | 82000 | 1% |
| | | | 160M | 101483 | 103000 | 1% |

perform the verification of some unfeasible deadlines becomes unmanageable in practice. Therefore, since there is such a pronounced difference between the times for feasible results and the times for unfeasible deadlines in the neighborhood of *mfd*, we pursued the following heuristic approach: for each configuration we started by running the analysis for trivially feasible deadlines and then proceeded "backwards" (i. e., by lowering the deadline) until a strong discontinuity was found in the verification time. Based on the times registered for each feasible deadline, we defined some timeouts and concluded that a given deadline was reasonably not feasible if no result was returned by the tool within those timeouts. 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* and the related percentage error *err*.

Results show that adherence of the model to the actual execution times with *vanilla* Spark (i. e., of *mfd* to $avg(t_{exec})$) is not particularly affected by changes in the use case type and configuration. In fact, *err* is at most 4% across all 6 settings of *SortByKey*, at most 7% for *PageRank* and at most 6% for *K-Means*.

## 5 Related works

To the best of our knowledge, no approaches exist in literature for the formal verification of Spark applications. For this reason, we cannot directly compare

[4] Full experimental data available at 10.5281/zenodo.1162853

against other works having the same focus. In the following, we present other techniques, in some cases applied to distributed systems, that tackle problems somewhat similar to ours, starting with general scheduling problems.

The analysis of temporal properties of scheduling algorithms and of distributed systems has been addressed with positive outcomes by using Timed Automata (TA, [2]) and Hybrid Automata (HA, [12]). In [10], 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., [16]), the use of TA—and, similarly, the use of CLTLoc in the present work—allows for capturing relevant properties of real implementations (e.g., resource constraints), and for the relaxing of some restrictions on the software structure, that are needed for the analysis. A timed analysis for distributed systems has been addressed in [7] 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. [13] also uses TA to model distributed real-time applications. A distributed application in [13] consists of several concurrent tasks, each one 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 component) 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 DAG of stages similar to the graph of dependencies in [7]. However, whereas tasks in [7] and in [13] are atomic and are executed on a single CPU each, 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. [4] 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 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 [4] addresses by means of PTA, and the extension with bounded delay uncertainty are addressed in [1] 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 standard OR algorithms.

As shown in [6], CLTLoc has the same expressive power as TA. Hence, in principle any problem solved through TA can also be solved through CLTLoc, and vice-versa. The CLTLoc-based approach that we pursue in this work allows for a high degree of modularity in the generation of the formal model from its high-level description, as it is easy to focus on the various aspects of the

model (e.g., precedences among stages, timing and resource constraints) one at a time—each aspect corresponding to a different set of logic constraints. In addition, as mentioned in Sect. 1, CLTLoc is the basis for a unifying approach to the modeling of Big Data frameworks which tackles applications of different natures (stream vs. batch processing).

In the domain of the analysis of Big Data frameworks, simulation, rather than formal verification is usually the approach of choice. For example, [17] considers the problem of computing the response-time of a Spark application through 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, an error bigger that 30% is possible. In [9] an ad-hoc fast event driven simulator, called *dagSIM*, has been used to simulate applications modeled as DAGs of nodes representing the execution of batches of tasks whose average duration is described with 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 or not of a desired deadline, and in particular they cannot be used to determine the *unfeasbility* of a deadline.

As already mentioned in the introduction, an analogous—temporal logic-based—approach was followed in [15] for the analysis of Storm applications. This work and [15] are based on the same automated mechanism implemented in the D-VerT tool. However, the formal model presented in [15] represents a different computation paradigm, namely, the stream processing, by means of CLTLoc extended with discrete *unbounded* counters. The analyses performed in the two works are different as well: [15] aims at finding ultimately periodic traces witnessing the presence of bottlenecks in the application, while this work focuses on finding finite traces proving the feasibility of given deadlines.

## 6 Conclusion

This work proposed an approach and a prototype tool to formally verify the feasibility of satisfying constraints over the response time of Spark applications given a fixed amount of computational resources. An experimental evaluation shows promising results in terms of accuracy of the model with respect to real Spark executions on different use cases and settings.

Possible future works include: (i) undertaking a thorough analysis of the complexity of the model and its effects on the verification times; (ii) improvements of the verification performance by optimizing the formal model; (iii) a refinement of the profiling phase aimed at providing good estimates of the execution times against changes in the number of cores and partitions of the input dataset.

**Acknowledgment** 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


1. Abdeddaim, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoretical Computer Science 354(2), 272 – 300 (2006)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science 126(2), 183–235 (1994)
3. Baresi, L., Pourhashem Kallehbasti, M.M., Rossi, M.: 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. pp. 1666–1673 (2016)
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Optimal scheduling using priced timed automata. SIGMETRICS Perform. Eval. Rev. 32(4), 34–40 (Mar 2005)
5. Bersani, M., Erascu, M., Marconi, F., Rossi, M.: DICE verification tool - final version. Tech. rep., DICE Consortium (2017), www.dice-h2020.eu
6. Bersani, M.M., Rossi, M., San Pietro, P.: A tool for deciding the satisfiability of continuous-time metric temporal logic. Acta Informatica 53(2), 171–206 (2016)
7. Bradley, S., Henderson, W., Kendall, D.: Using timed automata for response time analysis of distributed real-time systems. In: 24th IFAC/IFIP Workshop on Real-Time Programming. pp. 143–148 (1999)
8. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proc. of the Int. World-Wide Web Conference (WWW). pp. 107–117 (1998)
9. Brito, A., Ardagna, D., Blanquer, I., Evangelinou, A., Barbierato, E., Gribaudo, M., Almeida, J., Couto, A.P., Braga, T.: D3.4 eubra-bigsea qos infrastructure services intermediate version. Tech. rep.
10. Corbett, J.C.: Timing analysis of ada tasking programs. IEEE Transactions on Software Engineering 22(7), 461–483 (Jul 1996)
11. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint LTL. Information and Computation 205(3), 380–415 (2007)
12. Henzinger, T.A.: The Theory of Hybrid Automata, pp. 265–292. Springer Berlin Heidelberg (2000), https://doi.org/10.1007/978-3-642-59615-5_13
13. Krakora, J., Waszniowski, L., Pisa, P., Hanzalek, Z.: Timed automata approach to real time distributed system verification. In: Proc. of the IEEE Int. Work. on Factory Communication Systems, 2004. pp. 407–410 (Sept 2004)
14. MacQueen, J., et al.: Some methods for classification and analysis of multivariate observations. In: Proc. of the Berkeley symposium on mathematical statistics and probability. vol. 1, pp. 281–297 (1967)
15. Marconi, F., Bersani, M.M., Erascu, M., Rossi, M.: Towards the formal verification of data-intensive applications through metric temporal logic. In: Proc. of ICFEM. pp. 193–209 (2016)
16. Palencia, J.C., Harbour, M.G.: Schedulability analysis for tasks with static and dynamic offsets. In: Proc. of the IEEE Real-Time Sys. Symp. pp. 26–37 (Dec 1998)
17. Perez, D., Bernardi, S., Merseguer, J.Z., Requeno, J.I., Casale, G., Zhu, L.: DICE simulation tools - final version. Deliverable, http://www.dice-h2020.eu/resources/