2018 IEEE Conference on Control Technology and Applications (CCTA)
Copenhagen, Denmark, August 21-24, 2018

Dynamic Models for the Formal Verification of Big Data Applications via Stochastic Model Checking

Claudio Mandrioli[1], Alberto Leva[2], Martina Maggio[1]

Abstract— **Big Data Applications (BDAs) manage so much data to require a cluster of machines for computation and storage. Their execution often has temporal constraints, such as deadlines to process the data. BDAs are executed within Big Data *Frameworks* (BDFs), that provide mechanisms to automatically manage the complexity of the computation distribution. For a BDA to fulfill its deadline when executed in a BDF, online dynamic resource allocation policies should be in place. The introduction of control for such resource allocation calls for formal verification of the closed-loop system. *Model checkers* verify the correct behaviour of programs, and in principle they could be used to prove properties on the BDF execution. However, the complexity of BDFs makes it infeasible to directly model the BDAs and BDFs. We propose a formalism to associate the execution of a BDA with a *first-principle* dynamic simulation model that can be used for model checking in the place of the real application, making the verification viable in practice. We introduce our formalism, apply it to a well assessed framework, and test its capabilities. We show that our solution is able to capture the dynamics and prove properties of the BDA execution using a stochastic model checker.**

I. Introduction

Nowadays, computing systems and networks process an unprecedented amount of data. Every day, 300 million photos are uploaded on Facebook [1]. Every minute, 300 hours of videos are uploaded on YouTube[2]. Every second, modern cars produce location data. Every millisecond, trading systems have to cope with updates on market prices. Such huge data volume gave rise to the term *big data*. In the big data era, data processing must become not only extremely fast, but also reliable, with computation results produced within a given time frame.

In fact, large amounts of data have been *already encountered in various* domains, like large-scale finite-element and computational fluid dynamic models. However, *big data* is novel because: (i) data is composed of many simple unities that can be processed independently of one another; (ii) data is generated and needs to be stored at different locations in a network; (iii) data is typically *unstructured* and not homogeneous, respectively meaning that it does not have an *a priori* known structure and its nature (e.g., type of data) might be different for the different unities.

Applications that process data with the above listed characteristics are called *Big Data Applications* (BDAs), and executed in Big Data Frameworks (BDFs). BDFs, today, are very similar to one another [20] and include the software used by BDAs to query and manipulate unstructured data that do not fit into the memory of a single computing unit. BDFs hide from the programmer the complexity of allocating resources for the BDA execution on distributed nodes.

BDFs are complex software systems and their implementation intricacies — together with the volume of data, and the dependency of the computation efficiency on the data itself — impede the direct use of verification techniques aimed at ensuring that the computational results of the BDA execution are produced timely[1]. In this paper we propose a modeling framework for BDAs. A key feature of our framework is that it allows to apply verification techniques to prove properties of the given models. The proven properties are then supposed to hold for the BDAs execution. This approach generates models of the application execution by abstracting only the characteristics of the framework and application that are relevant from the verification perspective.

The abstraction aims at modeling the intrinsic complexity and dynamics of the problem, allowing us to separate them from the complexity induced only by the framework chosen for the execution. This description of the inner complexity of the problem results in a *white box* model, describing only the dynamics of interest. In physics, this is usually called a *first principle* approach. One of the advantages of first principle models is the possibility to use them to describe reality at different levels. For instance, when modeling the behavior of a gas, we could either represent the single molecules dynamics or to use the ideal gas theory. With first principle models for BDAs, we are able to choose the abstraction level for the application description. This choice will be related to the type of property to verify.

Verification is here conducted using a *model checker* [17], [8]. Model checkers receive as input a system description and a property to be checked against that description, either in an absolute or in a probabilistic sense. The system is specified using an operational formalism—in our case timed automata—and the property is specified as a logic formula that should be proven valid for the given model. The property can be proven in an exhaustive way on every possible execution of the model, or in a stochastic way via simulations of the (probabilistic) model. In this work, we exploit the latter, often called *statistical model checking* [14], [26], [21].

The paper is organized as follows. Section II offers an historical perspective on BDFs. Section III describes the modeling paradigm, and Section IV provides a validation

C. Mandrioli and M. Maggio are members of the LCCC Linnaeus Center and the ELLIIT Excellence Center at Lund University. This work was partially supported by the Wallenberg Autonomous Systems Program.

[1]Department of Automatic Control, Lund University, Sweden, corresponding author: `claudio.mandrioli@control.lth.se`, `martina.maggio@control.lth.se`

[2]Department of Electronics, Information and Bioengineering, Politecnico di Milano, Italy, `alberto.leva@polimi.it`

[1]Formal verification techniques have been used to ensure that code-related properties are enforced, e.g., data protection, security and privacy-related properties [5].

example. Finally, Section V concludes the paper, pointing out in retrospect the usefulness of the proposed formalism to ensure reliability and predictability in controlled BDAs.

II. Historical Perspective

Several Big Data Framework have been developed, most of them based on similar concepts and building principles [20]. This section briefly provides an overview of the main technological advances for big data computation: distributed file systems, the MapReduce paradigm, the directed graph abstraction, and in-memory computation.

Google designed the Google File System, *"a scalable distributed file system for large distributed data-intensive applications"* [11]. The paper describes the structure and operations of a distributed file system, later been open-sourced in the Hadoop File System[2] (HDFS). HDFS, now adopted by many big data frameworks, replicates data over the cluster of machines to guarantee fault tolerance. Data access is centrally managed by a single node, keeping track of the data location in the form of metadata.

Arguably, the main turning point in big data analytic has been the *MapReduce* programming model [10]. The MapReduce framework introduces a functional programming model, based on the concept of two different execution steps: a *map*, and a *reduce* phase. The map phase consists in processing the elements of a dataset in a fully parallel way, i.e., each *datum* is processed independently from the others, hence preserving the cardinality of the dataset. The reduce phase consists in processing independently subsets of the original dataset, merging data that belong to the same subset. Between the two operations, the dataset must be re-organized, identifying data belonging to the same subset. This operation is usually referred to as *shuffle*. A shuffle does not necessarily produce any data movement, but rather determines how to partition the dataset resulting from the map operation and how to locate each *datum* of each subset in the cluster.

Processing parallelism is the most important point for optimizing big data manipulations. For a map operation, thanks to the processing being independent at each *datum* level, the cluster manager can arbitrarily generate data partitions and distribute data to the processing nodes in the cluster. The parallelism of reduce operation, on the contrary, depends on the number of different data subsets (also called data keys) in the dataset. Manipulation of data belonging to different keys can be done in parallel. Within a single key, the parallelism depends on the specific operations to be executed.

Another turning point in big data processing has been the introduction of a high-level description of the operations to be executed on the dataset and of their control flow in the form of a Directed Acyclic Graph (DAG) [13], [27]. Vertices in the DAG are used to represent instructions to be executed on the data, while edges highlight the input and output dependence between each different vertices. An edge $v_i \rightarrow v_j$ between two generic vertices v_i and v_j imposes that the execution of the code for v_i must terminate before the code for v_j can be run. Initially introduced in the Dryad Framework [13] (now discontinued), the DAG model has been widely adopted by most big data frameworks, like Spark [28], Tez [20], Nephele [25], Hyracks [6].

While the programming model is important for application writing, the entire big data idea would fail if computation speed was not adequate. Resilient Distributed Datasets (RDDs) [27] were introduced to boost the performance of big data frameworks and allow them to efficiently reuse intermediate computation results via in-memory computation [19], [29]. The general idea behind in-memory execution is to store the intermediate results in the computer Random Access Memory (RAM), instead of writing them on the hard drive. Minimizing the amount of stored intermediate results is crucial because writing to disk takes from 10 to 100 times longer than reading and writing from memory, allowing for significant advantages. In-memory execution does not directly process data from memory, but keeps track of how the input *datum* of the following operation should be retrieved, or of the so-called *data lineage*, and process it only whenever necessary, i.e., when the actual value is used.

Many works analyzed the performance offered by big data frameworks, highlighting the difficulties in offering formal guarantees on said performance, due to the frameworks implementation intricacies, e.g., [18], [7], [12]. A solution could be the use of simulators: Mumak (2009)[3] is the Apache open source simulator for Hadoop; MRperf (2009)[4], YARNsim (2015) [16], and SimMR (2011) [23] are MapReduce simulators. Simulators have been built to replicate the behavior of BDAs for performance analysis purposes, and they respectively focus on some elements of the execution (e.g, YARNsim[5] simulates the behavior of YARN and SimMR is used to compare different scheduling solutions).

To highlight that existing big data frameworks operate similarly to one another, we mention Apache Tez [20], a *"library to build data-flow based engines"*. Tez can be used to produce code to be executed using different BDFs — MapReduce, Hive, Pig, Spark, Flink, Cascading, and Scalding — confirming the *"intuition that most of the popular vertical engines can leverage a core set of building blocks"*. This idea is the basis of our systemic approach, and reinforces the importance of looking for commonality in all the frameworks, the first principles that rule the processing of big data. Specifically, according to the creators of Tez [20], the frameworks commonalities are: (i) a description of BDAs in the form of DAGs, (ii) the environment in which they run, Hadoop and HDFS, (iii) a policy for resource allocation, (iv) mechanisms for recovering from hardware failures, (v) mechanisms for publishing metrics and statistics.

In our work, we assume that these commonalities are known. An application is given in the form of a DAG, and a running environment. We want to obtain guarantees on the use of resource allocation policies for the timely execution of the BDA. None of the previous work has taken a system-theoretical approach to this problem and none of the models

[2]https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html

[3]https://issues.apache.org/jira/browse/MAPREDUCE-728

[4]http://research.cs.vt.edu/dssl/mrperf/

[5]YARNsim [16] evidences a lognormal distribution of the execution time for map tasks with the average value linearly related to: (i) blocksize of input chunks, (ii) size of intermediate data and (iii) level of concurrency.

```
text = read_input(in1_location)                          // Call #1
counts = text.flatmap(fun line -> line.split(" "))       // Call #2a
  .map(fun word -> (word, 1))                            // Call #2b
  .reducebykey(fun a, b: a+b)                            // Call #2c
counts.write_output(out1_location)                       // Call #3
articles = {a, an, the}
no_articles =
  text.flatmap(fun line -> line.split(" "))              // Call #4a
      .filter(fun word not in articles)                  // Call #4b
no_articles.write_output(out2_location)                  // Call #5
```

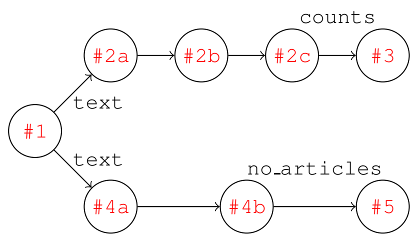


Fig. 1. Pseudo-code and example of DAG representation of a BDA.

used to determine BDAs properties is dynamical. The existing proposals for predicting whether an application will meet its deadline are purely statistical approaches or simulation-based analysis tools. On the contrary, this work investigates abstractions — or first-principle relations between the involved quantities. Despite that, similar considerations (e.g., on the relevance of the degree of concurrency or on the data locality) are recurrent in the different studies. In our opinion these work lack investigation on why these relations emerge, on the nature of probabilistic distributions of certain quantities, and on system dynamics.

III. The Modeling Framework

A major issue in modeling a BDA is correctly representing the parallelism inherently induced by distributed execution and possibly asynchronous data communication. BDAs can exploit parallelism in two different ways. The first one is *application-level parallelism*—parallelism that arises from the possibility of executing multiple parts of the application in parallel, e.g., retrieving the marks given to students in two different exams and computing their averages can be done in parallel for the two exams. The second way is *data-level parallelism*—parallelism that is intrinsic in the operation that is being executed on the data, e.g., increasing every element of a vector by 1 can be done in parallel on each element.

The DAG description of a BDA is useful, as it allows a framework to clearly identify which operations are executed on which data. The DAG for one application represents the application code in terms of vertices and edges, and is generally not unique, as it reflects different execution choices[6]. Each vertex v_i in the DAG represents a series of operations that can be executed sequentially on the data, without the need for synchronization. Each edge $v_i \to v_j$ represents data dependencies and enforces synchronization between the different operations. An example of the translation from the application code to its DAG can be seen in Figure 1, where the code (on the left) is paired with the graph representation of the sequence of operations (on the right). In the shown DAG, each framework call in the code is associated with one vertex, according to its label. Each BDF generates its own DAG according to its optimization policies. We use the generated DAG as an input for our models, capturing therefore the optimization featured in the BDF implementation.

Our modeling approach is inspired by first principle models, that aim at describing only the *phenomena* that is necessary to achieve the specific goal the model is needed for, in our case a description of the core components of applications and frameworks, and the verification of relevant properties. BDAs run on clusters (networks of computing machines) and usually do not get exclusive access to the hardware resources, being these computational or networking resources. These resources can be allocated by a different manager: Google, for instance, uses Borg [24]; the Hadoop environment uses YARN [22] or Mesos[7]. Managers receive resource requests (e.g. <4GB RAM, 8 CPUs>) and correspondingly allocate sets of resource units (e.g. <1GB RAM, 1 CPU>).

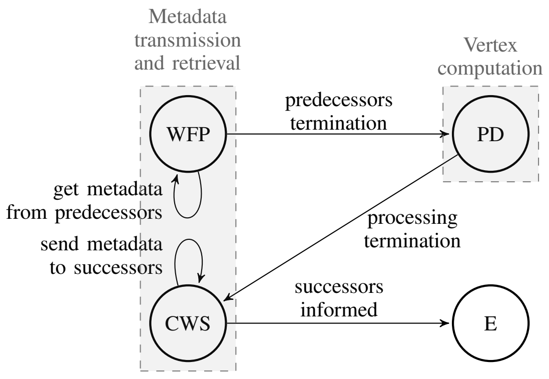
The first principle in the execution of BDAs and BDFs is the presence of queues of data and of operations executed by each vertex, with a given amount of resources. Queues describe two very important aspects of the BDA execution: (i) data processing, and (ii) data transmission through the network. A third concept, directly related to data transmission, is (iii) data locality.

- **Data processing:** Data are queued to be processed. The processing of data is defined as a set of couples of *metadata* and *operations sequences* (vertices). The computational units receive these couples and perform the given operations (the content of a vertex) on the data identified by the metadata. Each BDF can use a different way to map the received computational units to the actual entities that are performing the computation (e.g., threads, executors, etc). For our model, we abstract from the specific BDF implementation and deployment, aiming at generically describing the response of the computation progress of the given vertex to the given amount of resource.
- **Data transmission:** Operations and metadata are small with respect to the actual data, making their transmission negligible with respect to the network load generated to transmit the data. A good distribution of the computational resources will try to assign to a processing node operations on the data stored in the same physical node. This reflects the idea that, in the context of big data, "moving the computation is cheaper than moving the data".[8] The transmission of the actual data, on the contrary, directly affects the execution

[6]Equivalent DAGs represent (different) sequences of operations (vertices) that, given the same set of input data, produce the same set of output data. The differences are related to code execution optimizations (for example, re-executing the same operation to avoid data transmission). A natural consequence of this property is that the DAG can be manipulated to improve the computational efficiency of the application through better parallelism exploitation.

[7]http://mesos.apache.org

[8]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html


Metadata transmission and retrieval
Vertex computation
WFP
PD
predecessors termination
get metadata from predecessors
send metadata to successors
processing termination
CWS
E
successors informed
Fig. 2. State machine of the execution of a vertex.

progress of each vertex.

- **Data locality:** The allocation and synchronization of these executors is the level at which we wish to exploit data locality, i.e., a good BDF will try to allocate computational units in the cluster close to where the data they are supposed to process are located. On the one side, data locality has a strong impact on the execution of the application. On the other side, it strongly depends on the specific execution instance and on the cluster on which the application is run. We assume to capture data locality implicitly using the response of the BDA to resource changes. We generically define the given resources as a single variable and relate the progress of data processing to this single variable, being it executors, cores, CPU time, or other alternative representations.

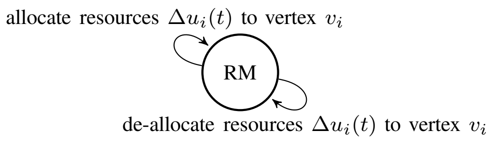
The execution of BDAs in a BDF can be modeled as a set of parallel processes. For each BDA, we need a parallel process for each vertex of the application DAG. These processes synchronize their execution based on the DAG information on data dependency (the output data of a DAG vertex is the input data of the successors of that vertex in the DAG). The remaining parallel process is the execution of the resource manager in the BDF. The resource manager process allocates resources to the different running vertices of the different BDAs, affecting their computation speed. Section III-A discusses in details how we model the processing of a vertex, while Section III-B explains how to model a BDF resource allocation policy. Section III-C finally describes how we implement the model in the *UPPAAL*[9] model checker [9].

A. Vertex Execution Model

The execution of the code for each vertex can be represented as a state machine, like the one shown in Figure 2. This state machine states are: (i) Waiting for Predecessors (WFP), (ii) Processing Data (PD), (iii) Communicating with Successors (CWS), and (iv) End (E).

In WFP, the vertex synchronizes with its predecessors, waiting for their completion, obtaining the metadata determining the data location, and then retrieving the needed data for the operations execution. The computation can start as soon as the data input set is completely defined. The processing and the retrieval of the data are run asynchronously

[9]http://www.uppaal.org/


allocate resources $\Delta u_i(t)$ to vertex v_i
RM
de-allocate resources $\Delta u_i(t)$ to vertex v_i
Fig. 3. State machine of the Resource Manager.

and in parallel, and modeled in the processing phase.

PD captures the asynchronous execution of the computation. For vertex v_i, the resource manager (implemented in the BDF) allocates resources $u_i(t)$ for the computation, affecting the progress speed. We model the vertex computation progress η_i as an integrator. Apparently, $\dot{\eta}_i$ represents the progress rate. To be as general as possible and capture the behavior of different BDFs and clusters, $\dot{\eta}_i$ is a generic function of the allocated resources and of other parameters

$$\dot{\eta}_i(t) = g_i\left(u_i(t), \eta_i(t), d_i(t), \theta\right), \tag{1}$$

where $d_i(t)$ is a positive quantity that represents a time varying efficiency of the computation, and θ is a vector of parameters describing the cluster. The time varying efficiency is a disturbance on the system representing all the possible inputs influencing the computation progress, including both external and internal disturbances. External disturbances include phenomena that happens at the cluster's machines level and influence the computation speed, like the co-location of additional load. Among the internal disturbances, $d_i(t)$ models the varying computational load generated by each *datum* and data locality. A negative value of $g_i()$ would imply the loss of processed data during the application execution, which is unlikely, as HDFS takes care of replicating data and guaranteeing fault tolerance. Saturation on $g_i()$ models the total available parallel units in the cluster (function of θ), and the maximum parallelism that the operations included in the vertex expose.

Once the processing is completed, the vertex state can switch to CWS state (dual of the WFP state), modeling the transmission of metadata to the successor nodes in the DAG. Finally, when this operation is completed, the vertex transitions to the end state E.

B. Resource Allocation Policy

BDFs include a Resource Manager, that can realize many different resource allocation policies[10]. In general, a resource manager individually allocates resources to the vertices that are in the PD state — i.e., to all the vertices whose operations are currently being executed. Irregardless of the framework implementation, a generic resource allocation policy can be realized as the *automaton* shown in Figure 3. With arbitrary (transition) activation time and resource variation $\Delta u_i(t)$, it is possible to implement any allocation policy. The allocation is subject to the constraint that $\forall t, \sum_i u_i(t) \leq U$, where U represents the amount of resource available in the cluster. Therefore vertices that are being executed concurrently compete for the same pool of resources.

We believe that allocating the correct amount of resources for a BDA to terminate within its deadline is a control

[10]For example, Hadoop include Apache YARN [22]. Standard allocation policies are: FIFO scheduling, the "Fair scheduler" (developed by Facebook), and the "Capacity scheduler" (developed by Yahoo!)

problem, where the application should follow a target computation progress. An approach based on this intuition is implemented in xSpark, an extension of the Spark framework [4]. We therefore model the xSpark allocation policy for our validation purposes.

C. Implementation in UPPAAL

The BDF and BDAs model is then the concurrent execution of switching systems like the ones described in Sections III-A and III-B. Suppose that n BDAs are run in parallel in one BDF. The j-th BDA is described by its DAG, that includes a number V_j of vertices. Our model includes one automaton for each vertex of each BDA, each of them with the states represented in Figure 2, for a total of $\sum_{j=1}^{n} V_j$ state machines. It also features one automaton for the resource allocation, as the one shown in Figure 3.

This representation precisely fits the model used in the UPPAAL model checker [9], which we have used for our implementation. Model checking can be extended to include stochastic behavior of systems, like delays following a specific probability distribution. This extension comes at a high cost, as the model checking problem for such systems is undecidable. To overcome this limitation UPPAAL includes *statistical model checking* (SMC) [15]. The idea of SMC is to run a finite number of simulations of the system, and then use results from statistics to bound the confidence interval for the property verification—this positions SMC between exhaustive model checking and testing [9]. In fact, while on one side SMC it is not exhaustive as pure model checking, on the other side it is more expressive and requires by far less memory and time for computing.

UPPAAL allows us to study stochastic switching systems. The properties that we would like to verify belongs precisely to this class — i.e., they are probability distributions and expected values of convenient indexes characterizing the evolution of the system. These properties are statements about the system expressed with an extension of the temporal logic, namely the Mixed Interval Temporal Logic (MITL) [3]. MITL includes operators like "eventually in the future" and "always in the future", together with the classical operators.

IV. Validation and Verification Experiment

The aim of this Section is twofold: (i) showing that the model described in Section III captures the relevant phenomena needed to describe the execution of a BDA in a BDF; (ii) showing the capabilities of our model checking implementation with respect to property verification. To this end, we use data generated by an execution of the "sort-by-key" application in xSpark.

Sort-by-key orders the input data set, hence preserving the cardinality of the data. Figure 4 shows the application DAG, composed of three stages. Even though stage #1 and the sequence of stages #2 and #3 could be executed in parallel, xSpark executes them sequentially (in this case in ascending order). Figure 5 shows the percentage of resources that are allocated to the vertex of the three stages by the xSpark resource allocator during the execution on the cluster, when the target deadline for the BDA is set to 160 seconds.

To simulate the *sort-by-key* execution in UPPAAL, we need to specify some parameters: (i) the volume of input data

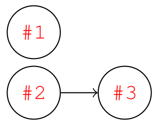


Fig. 4. Example: execution DAG of the sort-by-they BDA.

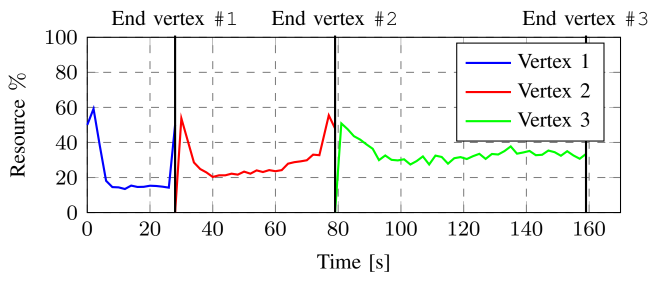


Fig. 5. Application execution on cluster: plot of allocated resources.

for each vertex, (ii) the DAG and the vertices connections, (iii) the resource allocation policy and its parameters, (iv) the cluster composition and parameters, (v) the function $g_i : U \rightarrow \mathbb{R}$ that – for each vertex v_i – maps the set of possible allocated resources U to the vertex execution progress $\dot{\eta}_i$.

In the given example, the input dataset is composed of 500'000'000 records and each vertex preserves its cardinality. The application DAG is shown in Figure 4. The resource allocation policy and its parameters are specified in xSpark [4], realizing a Proportional and Integral controller with $k_p = 70$ and $T_i = 50000$. The cluster is composed of 4 machines, each with 16 cores. The HDFS runs on different machines, to avoid resource competition. For each vertex, we implemented g_i as a first-order linear system with time-varying gain. We define the gain as a stochastic process, with a uniform distribution between a maximum (1'050'000) and a minimum (250'000) value.

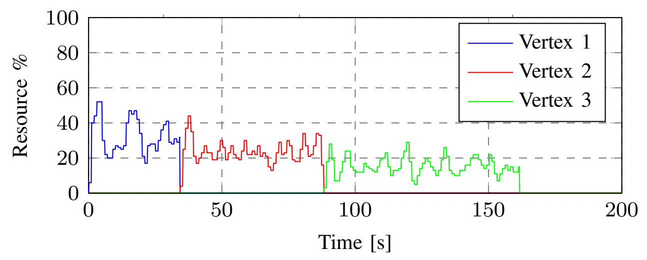


Fig. 6. Stochastic simulation of the implemented model, 1 trace: plot of allocated resources.

Figure 6 shows the simulation of one execution performed by the model checker, plotting the percentage of resources allocated to the vertices $u\%$. The model captures the linear behavior of the system, having a similar response than the one collected from the cluster execution and shown in Figure 5. The model checker takes care of the stochastic element of the simulation. Figures 7 and 8 show the results obtained repeating the simulation 50 times plotting respectively $u\%$ and the progress percentage η over time. The stochastic behavior gives us a way to quantify the system's variability. We can appreciate how the xSpark resource allocation controller rejects the disturbance introduced by the stochastic variation of the progress rate, making the application terminate always

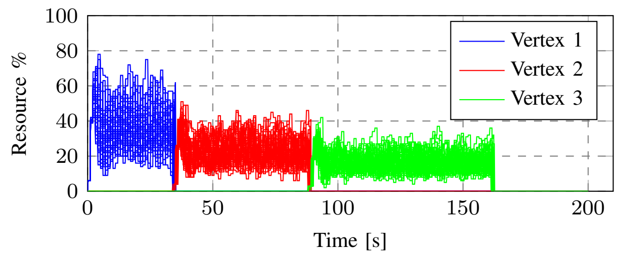


Fig. 7. Stochastic simulation of the implemented model, 50 traces: plot of allocated resources.

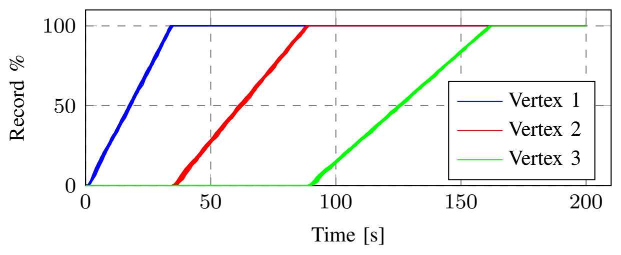


Fig. 8. Stochastic simulation of the implemented model, 50 traces: plot of processing progress.

in a neighborhood of the prescribed deadline. This experiments validates our model, showing that the model is able to capture the relevant phenomena and describes the execution of the BDA.

We now focus on using the model checker to verify relevant properties. One of such properties is the probability of not respecting the deadline. An observer is defined – in UPPAAL – to check whether the execution reaches the state in which the deadline has been missed. Running the model checker with the query of determining the probability of this event returns a value that is extremely close to zero, since the cluster has the needed computational capacity to satisfy the application. A more interesting query could be the maximum expected value for the total amount of allocated resources. This can be performed – in the stochastic sense – by determining a maximum amount of simulations and a maximum duration for each of the simulations. The model checker then provides an estimation of the average upper bound for the resource allocated to each of the vertices. In the example above, the model checker returns a little more than 40 cores over the 64 available ones in the cluster, corresponding to 62.5% of the total resources.

V. Conclusion

In this paper we presented a first-principle modeling framework for the execution of BDAs. First-principle modeling provides guarantees on the generailty of the model, and most important, allows us to synthesize control strategies (e.g., the resource allocation policy in xSpark) and validate them. Without a proper model abstraction, any such verification would be infeasibly complex in practice. We implemented the proposed model family in the stochastical model checker UPPAAL and we used the implementation to show: (i) that the model captures the behavior of an application running on a cluster of machines; (ii) that probabilistic queries can be done on the behavior of the application and of the resource allocation policy.

References

[1] https://zephoria.com/top-15-valuable-facebook-statistics/.
[2] https://fortunelords.com/youtube-statistics/.
[3] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1), January 1996.
[4] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. A discrete-time feedback controller for containerized cloud applications. In *ACM SIGSOFT Foundations of Software Engineering*, 2016.
[5] Marcello M. Bersani, Francesco Marconi, Matteo Rossi, and Madalina Erascu. A tool for verification of big-data applications. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, 2016.
[6] Vi. Borkar, M. Carey, R. Grover, N. Onose, and R Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *27th International Conference on Data Engineering*, 2011.
[7] T.C. Bressoud and Q. Tang. Analysis, modeling, and simulation of hadoop YARN mapreduce. In *22nd IEEE International Conference on Parallel and Distributed Systems*, 2016.
[8] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.
[9] A. David, K.G. Larsen, A. Legay, M. Mikučionis, and D.B. Poulsen. Uppaal smc tutorial. *Int. J. Softw. Tools Technol. Transf.*, 17(4), 2015.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.
[11] S. Ghemawat, H. Gobioff, and S.T. Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles*, 2003.
[12] G.P. Gibilisco, M. Li, L. Zhang, and D. Ardagna. Stage aware performance modeling of dag based in memory analytic platforms. In *9th IEEE International Conference on Cloud Computing*, 2016.
[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *2nd EuroSys Conference on Computer Systems*, 2007.
[14] J.P. Katoen. The probabilistic model checking landscape. In *31st ACM/IEEE Symposium on Logic in Computer Science*, 2016.
[15] A. Legay, B. Delahaye, and S. Bensalem. *Statistical Model Checking: An Overview*. Springer Berlin Heidelberg, 2010.
[16] N. Liu, X. Yang, X.H. Sun, J. Jenkins, and R. Ross. YARNsim: Simulating hadoop YARN. In *ACM Digital Symposium on Cluster, Cloud, and Grid Computing*, 2015.
[17] J. Maluszyski. *Model Checking*. MIT Press, 1997.
[18] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris. I/o performance modeling for big data applications over cloud infrastructures. In *IEEE Conference on Cloud Engineering*, 2015.
[19] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer Publishing Company, Incorporated, 2011.
[20] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *2015 ACM SIGMOD International Conference on Management of Data*, 2015.
[21] K. Sen, M. Viswanathan, and G. Agha. *On Statistical Model Checking of Stochastic Systems*. Springer Berlin Heidelberg, 2005.
[22] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *4th Symposium on Cloud Computing*, 2013.
[23] A. Verma, L. Cherkasova, and R.H. Campbell. Play it again, simmr! In *IEEE International Conference on Cluster Computing*, 2011.
[24] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *10th European Conference on Computer Systems*, 2015.
[25] D. Warneke and O. Kao. Nephele: Efficient parallel data processing in the cloud. In *2nd Workshop on Many-Task Computing on Grids and Supercomputers*, 2009.
[26] H.L.S. Younes. *Planning and Verification for Stochastic Processes with Asynchronous Events*. PhD thesis, CMU, 2004.
[27] M. Zaharia, M. Chowdhury, T Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Conference on Networked Systems Design Implementation*, 2012.
[28] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.
[29] H. Zhang, G. Chen, B.C. Ooi, K.L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2015.