Multi-path Coverage of all Final States for Model-Based Testing Theory using Spark In-memory Design

Wilfried Yves Hamilton Adoni¹, Moez Krichen²,
Tarik Nahhal¹, and Abdeltif Elbyed¹

¹ Hassan II University of Casablanca, Morocco
adoniwilfried@gmail.com,
t.nahhal@fsac.ac.ma,
abdeltif.elbyed@univh2c.ma

² FCSIT, Albaha University, Albaha, Saudi Arabia
ReDCAD Laboratory, University of Sfax, Tunisia
moez.krichen@redcad.org

Abstract. This paper deals with an efficient and robust distributed framework for finite state machine coverage in the field model based testing theory. All final states coverage in large-scale automaton is inherently computing-intensive and memory exhausting with impractical time complexity because of an explosion of the number of states. Thus, it is important to propose a faster solution that reduces the time complexity by exploiting big data concept based on Spark RDD computation. To cope with this situation, we propose a parallel and distributed approach based on Spark in-memory design which exploits A* algorithm for optimal coverage. The experiments performed on multi-node cluster prove that the proposed framework achieves significant gain of the computation time.

Keywords: Model-Based Testing · Coverage · Big Data · Big Graphs · Apache Spark · Apache Hadoop · Parallel and Distributed Computing

1 Introduction

Since we moved into the era of new technologies such as IoT, Cloud service, artificial intelligence, big data and 5G, the systems test attracts more interest for industrial and scientific communities. The reliability test of systems is based on a large number of compliance actions. The best known testing technique where the behavior of the system under test is checked against a prediction model is the Model-based testing [14, 12].

The basic idea of this technique is to create a set of functional tests based on some requirements, input sequences and actions. Once created, this model can be applied for testing both software and hardware. In this context, several research projects have been conducted in the field of system testing [17, 1, 15, 7].

Moez Krichen et al. [16] proposed a conformance testing for real-time systems. The proposed framework is designed for non-deterministic timed automata. This allows the user to define essential features and assumptions based on variants of the tested model. Moreover, they proposed a dynamic testing for real-time applications [7], it is dynamic in the sense that it provides an automatic generation of observers of time automata.

The checking of advanced models emanated from concurrent, dynamic or real-time systems is challenging and has been the focus of several decades of research. Indeed, the main problem that affects checking of sophisticated software or hardware is the state explosion issue [18] because of large amount of stored and shared information between many components of the system under test. To cope with this situation, the most efficient techniques to cope with this problem are based on symbolic bounded model checking, partial order reduction and symbolic model [11]. These breakthrough approach have enabled the testing of complex systems with large amount of states. Nevertheless, these techniques present some drawbacks. In fact, symbolic techniques disable the computation for specific states like state probabilities of Petri nets [6].

Another idea consists of using a computer cluster to achieve verification of a system represented by very complex model. These computers are connected under a master-slave architecture. Each computer checks a part of the model in parallel. Thus the analysis of very complex systems is done in a fast way with a quasi-optimal verification of the system.

Although big data computation techniques have so far been poorly explored for verifying software systems. We believe that the main issues to be tackled in conformance testing can benefit a lot from big data concepts. In this paper, we are interested to the problem of multipath coverage of all final states in large-scale automaton with state explosion. This task is very expensive with impracticable computation time.

The major contributions of the paper are summarized below.

- Introduction of big data (big-graph) concepts in the area of model based testing.
- An optimized coverage technique for generating test cases based on the extended version of MapReduce-A* [4, 3].
- A framework designed for large-scale finite automaton coverage.

The remainder of this paper is organized as follows. Section 2 gives background about Model-based testing and big data framework. In Section 3, we introduce the problem formulation. Section 4 presents the design of the proposed framework. Next, we expose our experiment results in section 5 and conclude this work in section 6.

2 Background

2.1 Model-based testing

Model-based testing (MBT) is a variant of test techniques that is based on explicit behavior models, describing expected behaviors of the system under

test (SUT), or the behavior of its environment, constructed from functional requirements. The MBT is an evolving approach that aims to automatically generate from one of these models, test cases to play on the SUT. The interest of adopting model-based testing in projects is to improve the detection of SUT bugs and improve software quality. Moreover, Model-based testing can reduce the time and effort spent on testing since the test scripts are derived automatically from the model of the SUT.

2.2 Apache Spark

Hadoop is an Apache open source framework allowing parallel and distributed processing of large amount of data sets. It is based on Google File System and works with multi-nodes connected under a master-slaves architecture. Hadoop consists of two main components: HDFS [9] and MapReduce [8, 19]. HDFS component is the Hadoop Distributed File System, it is designed to store very big dataset. The dataset is split into many chunk files and distributed across the cluster nodes in order to prevent fault-tolerance. The MapReduce component is designed for intensive computation on large data file in parallel and distributed way. The MapReduce programming model consists of two main functions : *map()* and *reduce()* functions.

Spark [20] is an Apache open source framework allowing parallel and distributed processing of large amounts of data sets. Spark is designed to face some limitations of Hadoop [19] such as iterative processing, in-memory computing and near real-time processing. Spark works with multi-nodes cluster connected under a master-slaves architecture, it provides a Resilient Distributed Datasets (RDDs) that computing operations in memory. RDD is a immutable object which allows transformation and action operations to do parallel processing on multi-node cluster.

3 Problem of coverage

Let $A = (Q, \Sigma, s_0, F, \delta)$ be a determinist finite automaton (DFA) such as:

- Q is a finite set of states.
- δ is a finite set of inputs.
- $s_0 \in Q$ is the initial state.
- $F \subseteq Q$ is a set of final states.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.

An automaton $A_i = (Q_i, \Sigma_i, s_0^i, F_i, \delta)$ with the initial state changed to s_0^i is a sub-automaton of A if $Q_i \subseteq Q$, $\Sigma_i \subseteq \Sigma$ and $F_i \subseteq F$. In MBT, the reliability test generally passes through the coverage of all states reachable by the system. This task consists of starting from the initial state of the system to find all the optimal or quasi-optimal paths that reach the set of final states of the system. However, in an automaton with state explosion, the search of all paths covering the final states of the system is very expensive.

Current coverage techniques [17, 1, 16, 7] are not suitable for large-scale automaton. One technique that has attracted attention is the divide-to-conquer which consists of partitioning the automata into a sub-automata and then finding the intermediate coverage. However, the quality of the result of this technique depends on the partitioning technique adopted. There are parsing algorithms that allow for quick coverage while others are well suited for optimal coverage [2, 5].

4 Proposed Spark coverage technique

The framework we propose is an adapted version of MRA* framework [4, 3]. The cluster is setup so that the RDD is used for partitioned and distributed the automaton across the cluster. Then, we process the big automaton data using Spark in-memory computation. As shown in Figure 1, the conceptual model of our framework works under a master-slaves architecture and it consists of three mains stages:

- Input stage which consists in the automaton partition.
- Map stage which consists in computing of intermediate states coverage.
- Reduce stage which consists in merging of all obtained coverages.

First of all, the coverage job submitted from the master is split into two sets of mapper and reducer tasks and distributed across the Spark cluster. The coverage job consists of finding all paths reaching the set of final states $s \in F$. The master manages the coverage job, assigns each map and reduces task to all workers. The coverage job is synchronized so that the output of the previous stage is taken as the input of the next stage. It is important to note that the input stage is the pre-processing stage and intermediate coverages are performed into map and reduce stages. Thus the total time t_σ required to find all paths that cover the set of final states $s_i \in F$ is calculated as follows:

$$t_\sigma = t_{map} + t_{red} \quad (1)$$

where t_{map} and t_{red} are respectively the time passed in the map and reduce stage.

4.1 Input stage: automaton partition

First, we need to load the automaton graph A as a parallelized RDD object. Then, A is partitioned into k sub-automata. A good partitioning strategy allows to schedule all tasks so that the load balancing among all nodes in the cluster is balanced. Spark provides different types of partitioning scheme. But in our work, we have used the “Edge-partition” technique [10] which equally distributes the automaton among all the workers in the cluster.

As shown in Figure 2 the automaton partition is defined as subsets of transitions such that each transition edge belongs to exactly one partition. The states which appear in two sub-automata A_1 and A_2 are used for the communications channels [10].

From the initial state $s_0 \in Q$, we create the first sub-automaton, the intermediate states s_i located on each sub-automaton $A_i \subseteq A$ are used as initial and final states. The initial state s_i of the i^{th} sub-automaton A_i is the final state of the $(i-1)^{th}$ sub-automaton A_{i-1} so on until the last k^{th} sub-automaton A_k . To avoid overload of the cluster and maintain load balancing, we assume that the number k of partitions is:

$$k = N_{node} \times N_{core} \quad (2)$$

where N_{node} is the number of workers within the cluster and N_{core} the number of core processors used by each worker node.

Finally, when the k-partitioning of the automaton is done, we create an object $RDD(A_i)$ of each sub-automaton A_i . This means that the input dataset remains in memory, but it can also be stored on the local disk if there is not enough RAM. The idea of creating RDD object of each sub-automaton is beneficial since at any time, we can correctly re-balance the number of states or transitions in each sub-automaton.

4.2 Map stage: intermediate states coverage

The map stage consists of computing for a given sub-automaton object $RDD(A_i)$, all cover paths that reach the set of final states $s_i \in F_i$. For the cover computation, we used an adapted version of A* mapper algorithm proposed by [4, 3].

This procedure takes as input an RDD object of a sub-automaton A_i , and maps it into key-value pairs of states RDDs and transitions RDDs. Then, it initializes the initial state s_0^i and the two main queues: the *open-list* O that contains all candidates states and the *close-list* S that contains promising states. The next step, consists of exploring in depth and selecting the most promising states until the final state found. Finally, the last step consists of extracting the coverage paths from the close-list S . Afterward, the result is submitted to the reduce stage.

The time complexity for all intermediate states coverage depends on the total number of worker nodes N_{node} within the cluster and the number of core processors N_{core} per node. To cover an intermediate final state s_i into the sub-automaton A_i , the time $m_{i,j,k}$ taken by the k^{th} mapper assigned to the j^{th} core processor of the i^{th} worker is about $O(|Q_i| + |T_i| \log(|Q_i|))$ where T_i is the set of transitions within A_i . The time $\Delta m_{i,j,k}$ taken to reach all final states during the map stage is calculated as follows:

$$t_{map} = \max\left(\sum_{i=1}^{N_{node}} \sum_{j=1}^{N_{map}} \sum_{k=1}^{|F_i|} \Delta m_{i,j,k}\right) \quad (3)$$

4.3 Reduce stage: merging all states coverage

In the final stage, the intermediate coverages from the map stage are aggregated and merged based on all final states that share the same key. This is performed by an adapted version of A* reducer procedure presented in [4, 3].

It takes as input the coverage path of each state, then group and merge all paths to build the final coverage path. The coverage $\sigma_{i,j}$ is extended to the coverage $\sigma_{j,k}$ if the intermediate state j is the final state reached by $\sigma_{i,j}$ and the start state of $\sigma_{j,k}$, so on until the last coverage.

When one worker node completes its tasks, it waits until the other reducers achieve their tasks before sending the result to the master. Then the time $\Delta r_{i,j,k}$ taken to merge all intermediate coverages from state s_0 to all final state $s_i \in F$ of the original automaton A is:

$$t_{red} = \max(\sum_{i=1}^{N_{node}} \sum_{j=1}^{N_{map}} \sum_{k=1}^{|F|} \Delta r_{i,j,k}) \quad (4)$$

5 Experiment results

The experiments have been achieved on a Spark cluster composed of one master node and three worker nodes. The cluster configuration is given below in Table 5

Table 1. Cluster setup

Software	Hardware
Linux Ubuntu 18.01.1.0 amd64	Processor Intel Core i5-2410M
Apache Spark 2.4.0	4 CPU @ 2.3GHz
JDK 1.8	16Gb RAM

We have used a dataset generated from the mutants-equiv-eval dataset ³. This dataset corresponds to the finite behavioural models of equivalent mutant problem. Table 5 shows the number of states and transitions in different finite behavioural models, each automaton contains at least 5% of final states to cover.

Table 2. Mutant equivalence dataset

Automaton	States	Degree	Transitions	Type
ageRR	30,000	6	180,000	Directed
ageRRN	60,000	5	300,000	Directed
claroline	500,000	4	2,000,000	Directed
cpterminal	1,000,000	6	6,000,000	Directed
elsaRR	1,500,000	6	9,000,000	Directed

Table 3 shows the execution time between sequential and parallel approaches using single Spark node, the computation is parallelized on 4 cores processor.

³ <https://projects.info.unamur.be/vibes/mutants-equiv.html>.

First, we remark that the sequential approach takes impracticable time to achieve the coverage of all final states.

On the other hand, by using our approach, the computational time is much faster and reduces significantly the execution time. For example, the task of finding all coverage paths within the claroline automaton takes on average 6 hours with sequential approach, it is computationally expensive. In contrast, by using our approach we find all coverage paths in 25 min on disk computing versus 3 min in-memory computing.

Table 3. Runtime of sequential coverage versus Spark coverage using one node

	Sequential	Spark on disk	Spark in-memory
ageRR	26 min	2 min	9.58 sec
ageRRN	1 h	4 min	26.85 sec
claroline	6.17 h	25 min	2.63 min
cpterminal	11.24 h	45.3 min	4.7 min
elsaRR	21 h	1h24 min	8.17 min

Figure 3 shows the performance comparison of increasing workers on the computational time. In this case, the computation is distributed across the cluster and also parallelized on the 4 cores of each worker. We remark that the runtime varies strongly and is improved while extending the cluster from 1 to 4 workers. Adding new Spark worker reduces the computation time, however when we move from a very large dataset to a small automaton, the computational time is not affected by the addition of new Spark node into the cluster. This leads us to consider that there exists an optimal number of nodes that satisfies the full coverage of a given finite automaton.

6 Conclusion and perspectives

In this paper, we have proposed a parallel and distributed framework for large-scale automaton coverage. The time complexity decreases from exponential to linear time. The experimental results prove that our approach is faster and works well with very large automaton. But our approach presents some limitations: 1) the path optimality depends on the partitioning strategy and the number of sub-automatons, see [4] for more details; 2) the computation is often memory expensive. For future work, we are interested in studying the impact of automaton partitioning on the time complexity and propose an extended version of the framework for the coverage of timed-automaton and distributed systems [13].

References

1. Adamatzky, A.: Computation of shortest path in cellular automata. *Mathematical and Computer Modelling* **23**(4), 105–113 (Feb 1996)
2. Adoni, H.W.Y., Nahhal, T., Krichen, M., Aghezzaf, B., Elbyed, A.: A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems. *Distributed and Parallel Databases* pp. 1–36 (2019)
3. Adoni, W.Y.H., Nahhal, T., Aghezzaf, B., Elbyed, A.: MRA*: Parallel and Distributed Path in Large-Scale Graph Using MapReduce-A* Based Approach. In: *Ubiquitous Networking*, pp. 390–401. *Lecture Notes in Computer Science*, Springer, Cham (May 2017)
4. Adoni, W.Y.H., Nahhal, T., Aghezzaf, B., Elbyed, A.: The MapReduce-based approach to improve the shortest path computation in large-scale road networks: the case of A* algorithm. *Journal of Big Data* **5** (2018)
5. Adoni, W.Y.H., Nahhal, T., Krichen, M., El byed, A., Assayad, I.: Dhpv: a distributed algorithm for large-scale graph partitioning. *Journal of Big Data* **7**(1)
6. Bell, A., Haverkort, B.R.: Sequential and distributed model checking of Petri nets. *International Journal on Software Tools for Technology Transfer* **7**(1), 43–60
7. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing Conformance of Real-Time Applications by Automatic Generation of Observers. *Electronic Notes in Theoretical Computer Science* **113**, 23–43 (Jan 2005)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. *Communications of the ACM* **51**(1), 107–113 (2008)
9. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: *ACM SIGOPS operating systems review*, vol. 37, pp. 29–43. ACM, New York, NY, USA (2003)
10. Guerrieri, A.: Distributed computing for large-scale graphs. Ph.D. thesis, University of Trento (2015)
11. Jhala, R., Majumdar, R.: Software model checking. *ACM Comput. Surv.* **41**(4), 21:1–21:54 (Oct 2009)
12. Krichen, M.: Model-based testing for real-time systems. Ph.D. thesis, PhD thesis, Universit Joseph Fourier (December 2007) (2007)
13. Krichen, M.: A formal framework for black-box conformance testing of distributed real-time systems. *IJCCBS* **3**(1/2), 26–43 (2012)
14. Krichen, M.: Contributions to Model-Based Testing of Dynamic and Distributed Real-Time Systems. Ph.D. thesis, ENIS, Sfax (Tunisia) (2018)
15. Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: *SPIN*. *Lecture Notes in Computer Science*, vol. 2989, pp. 109–126. Springer (2004)
16. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Formal Methods in System Design* **34**(3), 238–304 (Jun 2009)
17. Maâlej, A.J., Lahami, M., Krichen, M., Jmaïel, M.: Distributed and Resource-Aware Load Testing of WS-BPEL Compositions. pp. 29–38 (Feb 2019)
18. Rafe, V., Rahmani, M., Rashidi, K.: A Survey on Coping with the State Space Explosion Problem in Model Checking p. 6 (2013)
19. Vavilapalli, V.K., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H.: Apache Hadoop YARN: yet another resource negotiator. pp. 1–16. ACM Press (2013)
20. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* **59**(11), 56–65 (Oct 2016)

Fig. 1. Overview of our framework

Fig. 2. Automaton partitioning under $k = 2$ sub-automata. State s_2 is a frontier (intermediate) state, it serves as start state of A_2 and final state of A_1

Fig. 3. Impact of number of workers on the computation time