Exploring parallel formal verification of BIG-DATA systems (*Explorando verificación formal paralela para sistemas de BIG-DATA)*

Fernando Asteasuain[1] & Luciana Rodriguez Caldeira[2]

Campo temático: Ciencias de la Computación.

Resumen

La Ingeniería de Software viene adaptando sus herramientas, métodos y técnicas para enfrentar los desafíos de los denominados sistemas de BIG-DATA. En particular, el área de verificación formal ha sido señalada como unas de las áreas de las que se requiere inmediatas contribuciones. En este trabajo se presentan aspectos claves buscando adaptar al lenguaje FVS como un lenguaje de verificación formal para BIG DATA. Por un lado se presenta una demostración formal de la correctitud del esquema paralelo de FVS. Por otro, se presenta una desafiante validación empírica del enfoque propuesto utilizando un protocolo relevante a nivel industrial con un balanceador de carga y comparando varias implementaciones.

Palabras Clave: verificación formal; big data; algoritmos paralelos; model checking.

[1] Universidad Nacional de Avellaneda -Centro de Altos Estudio CAETI -UAI. fasteasuain@undav.edu.ar

[2] Universidad Abierta Interamericana- Centro de Altos Estudio CAETI. luciana.rodriguezcaldeira@alumnos.uai.edu.ar

Abstract

Software Engineering is trying to adapt its tools, mechanisms and techniques to cope with the challenges involved when developing BIG DATA software systems. In particular, formal verification in one of the areas that more urgently is required to step in. In this work we introduce two crucial aspects aiming to adapt FVS to cope with BIG Data requirements. For one side, FVS's parallel algorithm is proved to be sound and correct. For the other side, we developed a compelling empirical validation of our approach, employing a communication protocol relevant in the industrial world within a context of parallel systems, introducing a load-balancer process and comparing several implementations.

Keywords: formal verification; big data; parallel algorithms; model checking.

1. Introduction

Big DATA systems and applications are incredible present in everyday life. Huge amounts of data and information become available every second from diverse sources like sensors or Internet of Things (IoT)-based systems. The software Engineering Community has been adapting its traditional tools, methods and techniques in order to cope with the challenges that BIG DATA systems involve (Hummel, O, et al. 2018, Kumar, V. D., & Alencar, P. 2016, Laigner, R. et all, 2018, Otero, C. E., & Peter, A. 2014, Camilli,M. 2014, Ding, J., Zhang, D., & Hu, X. H. 2016). **Formal verification of big DATA systems has been pinpointed as one of the main software engineering's areas that more urgently need to be explored and adapted**. For example, according to the results presented in (Kumar, V. D., & Alencar, P. 2016) only two of nearly two hundreds analyzed approaches addressing software engineering activities regarding BIG DATA systems deal with the formal verification phase.

Some approaches tried to expand traditional tools like model checking involving techniques as parallel model checking or Cloud-Model checking (Camilli, M.2014). However, a prior step in the formal verification road has been somehow neglected, which is the way the behavioral properties to be verified in the model checker are built and specified (Clarke et.al 2011, Asteasuain, F., Caldeira Rodriguez L., 2020). In (Asteasuain, F., Caldeira Rodriguez L, 2020) a parallel tool for formal verification of big DATA systems is presented tackling this issue. The tool is based on a graphical language called FVS Feather Weight Visual Scenarios (Asteasuain, F., & Braberman, V. 2017), a simple and yet powerful and expressive language to denote the expected behavior of the system. **We now expand and spread the potential of that approach by presenting two relevant aspects from the theoretical and empiric point of view.** From the theoretical perspective we introduce a formal proof of correctness of the main parallel algorithm involved in the approach, a process which translates FVS graphical scenarios into Büchi automata. From the empirical perspective we developed a more complex and thorough evaluation of the case of study, introducing a load-balancer process to manage parallel activities. These two aspects allow considering FVS as a potential parallel tool for big DATA SYSTEMS. **It is worth noticing that these aspects represent a first step aiming to adapt FVS to BIG DATA requirements. Once this stage is finished, as a second step we will apply FVS to formally verify BIG DATA systems per se, a research line which is included in our short-term future work**.

The rest of this paper is structured as follows. Section 1.1 mentions some observations about the selected case of study. Section 2 briefly presents the FVS language, and the parallel algorithm to translate FVS into Büchi automata, a required step to employ FVS in formal verification tools like model checkers. Section 3 presents the proof of correctness of our approach. Section 4 presents the empirical evaluation of our approach. Finally, Sections 5 and 6 end this work by mentioning related and future work and the final conclusions.

1.1 On the selected case of study

Our case study is a relevant protocol with a widespread use in the industrial world: the MS-NNS protocol, specified in (Asteasuain, F., & Braberman, V. 2017). Although it is not directly related to big data systems, the space explored to formally verify the system is similar enough to big data systems (Bellettini, C., Camilli, M., Capra, L., & Monga, M. 2016). Since space is an important dimension in BIG DATA systems, we considered that the analysis and conclusions of the case study are significant for an initial step in the FVS's road to formally validate BIG DATA systems. We aim to continue this trip formally verifying a BIG DATA systems in future work.

2. FVS: Feather Weight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is referred to (Asteasuain, F., & Braberman, V. 2017) for a formal characterization of the language. FVS is a graphical language based on scenarios. Scenarios are partial order of events, consisting of points, which are labeled with a logic formula expressing the possible events occurring at that point, and arrows connecting them. An arrow between two points indicates precedence. For instance, in Figure 1-(a) A-event precedes B-event. In Figure 1-b the scenario captures the very next B-event following an A-event, and not any other B-event. Events labeling an arrow are interpreted as forbidden events between both points. In Figure 1-c A-event precedes B-event such that C-event does not occur between them. Finally, FVS features aliasing between points. Scenario in 1-d indicates that a point labeled with A is also labeled with A ^ B. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxes. Aliasing allows the possibility of adding new behavior or renaming existing behavior by saying two events are equivalent in terms of behavior.

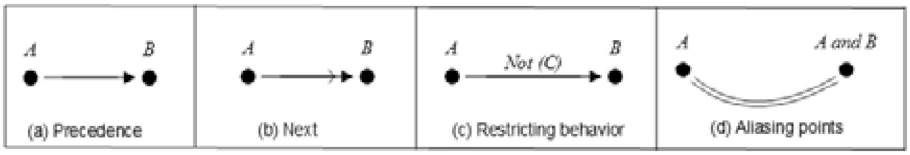

Figure 1. FVS Basic Features

We now introduce the concept of FVS rules. A rule consists of a scenario playing the role of an antecedent and at least one scenario playing the role of a consequent. Graphically, the antecedent is shown in black, and consequents in grey.

As an example, we show two FVS rules modeling a testing technique for BIG DATA called *metamorphic* testing (Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortés, A. 2016, Ding, J., Zhang, D., & Hu, X. H. 2016) . In few words, the technique says that similar inputs should behave equivalently and opposite inputs should have opposite results. In this way, simple tests can be generated. The FVS rules in Figure 2 show an example in a system analyzing user's reviews to distinguish good reviews from bad reviews. These two metamorphic rules apply to good reviews and they specify that if words are replaced with synonyms the review should be classified as a good review too. The second rule says that if words are replaced with antonyms, then the outcome should change and the review should be classified as a bad one. Similar rules could be added for bad reviews.

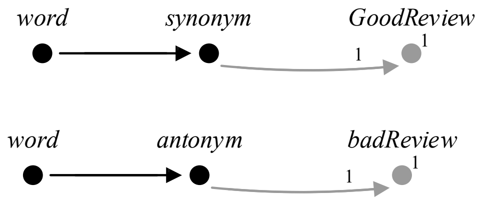

Figure 2. FVS Rules Example

2.1 A Parallel Algorithm translating FVS Scenarios into Büchi Automata

This section describes the parallel tableau algorithm which translates FVS scenarios into Büchi automata. We first introduce some basic notions. FVS scenarios can be defined as *morphisms* from the antecedent to the consequent. The algorithm relies on the notion of ***situations*** (Asteasuain, F., & Braberman, V. 2017). A situation represents for a given rule possible combinations of partial matches from the antecedent to the consequent. Consider the following example in Figure 3, where a rule with two consequents is shown. There are three partial matches for consequent one, and two for consequent two. Therefore, situation *η1* consists of the three morphisms in the first column (g_1^1, g_1^2, g_1^3) whereas situation *η2* consists of the two morphisms in the second column (g_2^1, g_1^2, g_2^2).

$$\begin{array}{ll} g_1^1 : A' \rightarrow C_1^1 & g_2^1 : A' \rightarrow C_2^1 \\ g_1^2 : A' \rightarrow C_1^2 & g_2^2 : A' \rightarrow C_2^2 \\ g_1^3 : A' \rightarrow C_1^3 & \end{array}$$

Figure 3. A situation example

The sequential algorithm is detailed in (Asteasuain, F., & Braberman, V. 2017). Starting from the initial state the automata will try to incrementally "construct" the pattern as events, represented by minterms, occur. For every minterm, the algorithm computes all possible matchings considering matchings in the antecedent and also in each consequent. The set *situation(S)* symbolically represents all the possible combination of partial matches obtained up to that state from the antecedent to each consequent.

After a rigorous analysis of it we detected two natural points suitable for parallelization: the computation of all the possible antecedents and consequents and tagging all the possible matches and checking whether any consequent has been matched by the last move. These are tasks that can be easily divided into different nodes to be realized and then the main algorithm can continue once every one is finished. The parallel pseudo-code for the parallel algorithm is depicted in Algorithm 1.

```
1.  Algorithm Parallel Succ(S : State,m : minterm) : set of states;
2.  Precondition : m ∧ obligations(S) is satisfiable;
3.  newSits := ∅;
4.  PrepareNodes ($N_1$,$N_2$,…,$N_k$)
5.  istributeAdvancesCalculation(($N_1$,$N_2$,…,$N_k$),Situations(S))
6.  JoinNodes(($N_1$,$N_2$,…,$N_k$), newSits)
7.  trapSituation : ∃η ∈ newSits such that the antecedent is matched and non of the
    consequents.
8.  PrepareNodes ($N_1$,$N_2$,…,$N_k$)
9.  DistributeGoalMatches(($N_1$,$N_2$,…,$N_k$),Situations(S))
10. JoinNodes(($N_1$,$N_2$,…,$N_k$), goalMatched)
11. goalMatched:= (∃j (goalmatched[j])) ∧ (¬trapSituation)
12. return <newSits,GM,Obligations> such that GM goalMatched ∧ GM = true
    ∃j(goalmatched[j]) ∧ Obligations = Obligations(S)
```

Algorithm1. Parallel Algorithm's sketch

Nodes preparation and setup is done in Line 4. Line 5 is in charge of distributing the task of obtaining the advances of antecedent and consequents in each situation η among the nodes. In Line 6 all the tasks done by the nodes are united and the new situations set represented by the variable *newSits* are obtained. Line 7 analyzes whether any successor reaches a trap situation, a situation where the antecedent has been matched, but matching for all consequents is known unfeasible. Parallel instructions in lines 8 to 11 deal with of goal calculation and verify if any antecedent has been satisfied. Finally, line 12 returns the expected output.

Figure 4 exhibits a simplified version of the automaton obtained using the FVS rules in Figure 2 as input.

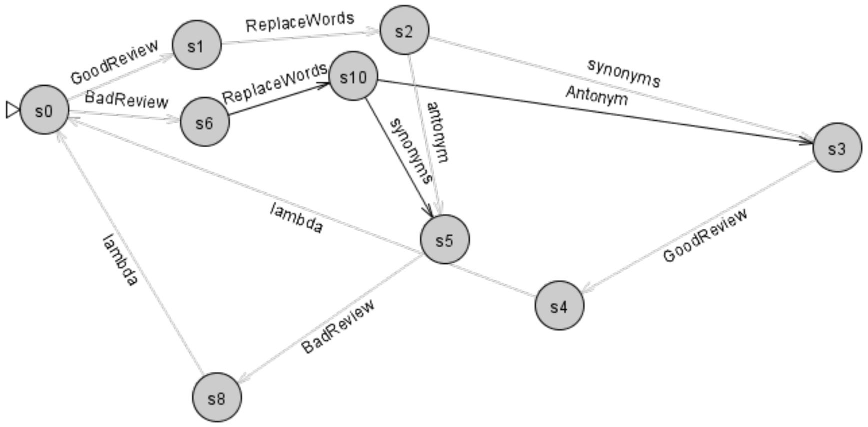

Figure 4. FVS-based Automaton

3. Proofs of Correctness of the Parallel Algorithm

In this section we prove that the parallel algorithm shown in Section 2.1 is sound and correct. We know that the sequential algorithm is sound and correct (Asteasuain, F., & Braberman, V. 2017). That is, that the set of traces satisfying a given rule R is equivalent to the language accepted by the automaton B, built by the tableau. More formally, we know that $traces(R) \equiv L(B)$. Work in (Asteasuain, F., & Braberman, V. 2017) also contains a proof of an important lemma called "Traces-States" relating the traces of an FVS rule with the states of the automaton. The lemma says that for every trace t satisfying R such that t leads to a given state S of B, then t can be "matched" (i.e. a morphism exists) with a situation $\eta \in S$. The "Traces-States" lemma plays an important role since it allows going back and forth between traces of rules and states of the automaton.

We now need to prove that $traces(R) \equiv L(B)$ also holds for the parallel version of the algorithm. We will achieve this by demonstrating that $traces(R) \subseteq L(B)$ for one side and that $L(B) \subseteq traces(R)$ for the other side, following a classic equivalence proof for set's languages.

Part 1: $traces(R) \subseteq L(B)$: *if* $t \in traces(R) \rightarrow t \in L(B)$. We know that $\forall t, t \in$ traces (R) , $t \subseteq L(B)$ for the sequential algorithm. Given this, and by the "Traces-States" lemma, we know that for every prefix of t, (where prefix is the usual function that returns a ordered subset of t: $\forall t_i$, $i<=k$, *where* $t_1,t_2,\ldots,t_k = t$) and a morphism m such that $t_i \rightarrow^m t_{i+1}$ (t_i advances to t_{i+1} by m) there exists a situation $\eta_1 \in$ states (B) such that $\eta_1 \rightarrow^m \eta_2$, where $\eta_2 \in$ Succesors(B) for η_1. In few words, for every prefix of t if there exists a morphism *m* that makes *t* to advance recognizing the rule *R,* then the resulting situation η_2 belongs to the next states of *B*, advancing both the trace and the automaton.

Given the codification of the parallel algorithm we know there exist a node N_i such that N_i calculates the successor of η_1 (observe instructions in lines 4 and 5 which distribute in *N* nodes the calculation of the next states, assigning one node for every situation). Let N_k *(1<=k<=i)* be that node. Once node N_k and all the other nodes finish their work the *join* instructions in line 6 of Algorithm 1 simply obtain all the successors by a employing the union operation of all the successors obtained by every node. That is, $\cup_{i(1<=i<=k)} Succ(N_i)$. Then, given that $\eta_2 \in$ Successors (*B*) for η_1 then it can be stated that $\eta_2 \in \cup_{i(1<=i<=k)} Succ(N_i)$. More simply, for every prefix of *t* the successor of *t* will be present in the next state of the automaton. Finally, since $t \in$ *traces(R)* when *t* finishes the automaton will be in a accepting state, concluding that $t \subseteq L(B)$.

If it is the case that *t* does not advance for any morphism *m*, then the algorithm guarantees that η_1 will belong to the next state of the automaton since they are always included by default. **This concludes part 1 of the proof**.

Part 2: *L(B) ⊆ traces(R): if t ∈ L(B) → t ∈ traces(R).* We know that ∀*t, t ∈ L(B)* → *t ⊆ traces(R)* for the sequential algorithm. For every accepting state *S* of *B* there exist a morphism *m*, such that *m* can be matched with a situation η , η ⊆ *S ,* and a trace *t, t ∈ traces(R).* By the "Traces-States" lemma we can affirm that for every trace *t* leading to an accepting state *S* of *B* we can find a morphism *m* relating a situation η in S with *t*. More formally, ∀ *t, t ∈ L(B)* → ∃ morphism *m*, such that m can be matched with *t* and also with a situation η, η ∈ *Situations(S) where S =AcceptingStates(B).*

The codification of the parallel algorithm in lines 8,9 and 10 specifies that deciding whether an state is accepting or not is done in parallel where every node N_i performs this calculation for every situation η_i. Let N_k be the node that performs an advance for *t*. The join instruction in Line 10 in Algorithm 1 simply merges all the results obtained by every node by a employing the union operation of them. That is, $\cup_{i(1<=i<=k)} Accepting(N_i)$. Given this, we can conclude that the next state for *t,* calculated by N_k will be included in the next state of the automaton.

We can then affirm that for every advance of *t* the next state will be included in the automaton. In particular, this holds for all the traces leading *t* to an accepting state *S* of *B (those t ∈ L(B)).* And this is also true for every prefix of *t*: if $t_i \rightarrow^m t_{i+1}$ (t_i advances to t_{i+1} by morphism *m*) then t_{i+1} is included in the next state of the automaton. Since this holds for every t_i in $t=t_1,2,\ldots t_k$. *1<=i<=k*, and given that *t* ∈ *L(B)* (t leads to an accepting state)*, t* will always satisfy *R*. In other words, *if t ∈ L(B) then t ∈ traces(R).* Note that this is also true for those cases where no advance is produced, since in those cases all of these situations are included in the next step by default. **This concludes part 2 of the proof**.

Given that *traces(R)* ⊆ *L*(*B*) and that *L(B)* ⊆ *traces(R)* then *traces(R)* ≡*L*(*B*), which was what we aim to prove.

4 .Empirical Validation

We implemented three different implementations for the parallel algorithm delineated in Algorithm 1. In the first one we simply use Java threads. The others two version handle two parallel libraries for Java: Open MPI (Vega-Gisbert, O., Roman, J. E., & Squyres, J. M. 2016) and MPJ Express (Shafi, A., Carpenter, B., & Baker, M. 2009).

We employed as case of study the verification of the MS-NNS protocol specified in (Asteasuain, F., & Braberman, V. 2017). This protocol was introduced as a lightweight option to provide authenticated and confidential communication between a server and a client over a TCP connection protocol.

We took the initial empirical validation in (Asteasuain, F., & Rodriguez Caldeira L. 2020) one step further considering more clients and a load-balancer process, therefore making the case of study more complex. We considered from 32 to 512 clients together with an environment where the available nodes were not enough to assign one node for every situation or "for the goal matched calculation". This was resolved in two ways: by simulating parallelism through the concurrency provided by the underlying operating systems and by introducing an extra process playing the role of a "load-balancer". This process basically assigns as many tasks as nodes are available and the rest of the tasks are assigned later as soon as a node become available.

We compared both flavors: with and without the load-balancer process in all of the versions (threads, OpenMPI and MPJ Express), together with the sequential version. We considered 32 clients, 64 clients, 128 clients, 256 clients and 512 clients. The results are shown in Table 1, where the advantages of introducing parallelism are clearly seen. It can also be noted from Table 1 that not using the load balancer is better than using it in the first case, caused since the overhead that the load-balancer imposes outcomes the benefits it provides. However, as the number of clients' increases this situation is turned around. Regarding the parallel libraries, the MPJ Express implementation is slightly better than the Open MPI implementation.

# Clients	Sequential	Threads with LB	MPJ with LB	Open MPI with LB	Threads without LB	MPJ without LB	Open MPI without LB
32	1364 sec	280 sec	103 sec	112 sec	200 sec	91 sec	96 sec
64	>10 mins	330 sec	107 sec	115 sec	343 sec	112 sec	120 sec
128	>10 mins	407 sec	140 sec	210 sec	413 sec	180 sec	250 sec
256	>15 mins	503 sec	205 sec	290 sec	568 sec	300 sec	310 sec
512	>22 mins	650 sec	330 sec	414 sec	700 sec	430 sec	470 sec

Table 1: Performance Evaluation Results

Regarding the execution times, it should be noted that the experiments took into account not only the execution time of the protocol but also the time to formally verify it. These algorithms involve non trivial data and structures manipulation obtaining exponential complexity in some cases (Vardi, M. Y. 2001). The size of the problem is also important. For the case of study analyzed in this work there average size of the automata involved was 1863 states and 7522 transitions.

As a final conclusion we can observe that the load-balancer process results in a valuable asset for the system under analysis. It is worth noticing that in the software framework for FVS the load-balancer can be activated (or deactivated) by simply clicking in an option tab when setting the environment for the verification phase.

5. Related and Future Work

Several approaches aim to adapt current formal verification techniques to BIG DATA systems. In (Matilli, M. 2014, Bellettini, C., Camilli, M., Capra, L., & Monga, M. 2016) an interesting framework for distributed CTL (computation tree logic) model checker is presented. We would definitely like to explore in future work the combination of this advanced tools with our specification language FVS.

Other approaches like (Boukala, M. C., & Petrucci, L. 2012, Brim, L., Yorav, K., & Žídková, J. 2005, Brassesco,M.V. 2017) provide some tools implementing different versions of parallel model checking. We believe that a natural continuation of this work is to provide the automata build by the parallel tableau as the behavioral properties to be checked in any of the mentioned approaches. In a different direction, work like (Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortés, A. 2016, Ding, J., Zhang, D., & Hu, X. H. 2016) employ metamorphic testing as the alternative to validate BIG DATA results. We would like to extend this notion to formally model check behavior pursuing the notion of "metamorphic" properties. Our next desired step is to apply FVS in a BIG DATA system.

6. Conclusions and Observations

In this work two main aspects are presented making FVS a suitable to formally verify BIG DATA systems. For one side, the parallel algorithm is proved to be sound and correct. For the other side, we developed a compelling empirical validation, employing a communication protocol relevant in the industrial world, introducing a load-balancer process and comparing several implementations.

We are aware there are some limitations in the current status of our approach. Fundamentally, we need to explore our tool in a BIG DATA system, challenging the promising results we have obtained so far.

7. References

Asteasuain, F., & Braberman, V. (2017). Declaratively building behavior by means of scenario clauses. Requirements Engineering, 22(2), 239-274.

Asteasuain, F., & Rodriguez Caldeira L (2020). A Parallel Tableau Algorithm for BIG DATA Verification. CACIC 2020.

Bellettini, C., Camilli, M., Capra, L., & Monga, M. (2016). Distributed CTL model checking using MapReduce: theory and practice. Concurrency and Computation: Practice and Experience, 28(11), 3025-3041.

Boukala, M. C., & Petrucci, L. (2012). Distributed model-checking and counterexample search for CTL logic. International Journal of Critical Computer-Based Systems 3, 3(1-2), 44-59.

Brassesco,M.V. 2017. Síntesis concurrente de controladores para juegos definidos con objetivos de generalized reactivity(1). Tesis de Licenciatura., http://dc.sigedep.exactas.uba.ar/media/academic/grade/thesis/tesis_18.pdf UBAFCEyN Dpto Computacion

Brim, L., Yorav, K., & Žídková, J. (2005). Assumption-based distribution of CTL model checking. International Journal on Software Tools for Technology Transfer, 7(1), 61-73.

Camilli, M. (2014, May). Formal verification problems in a big data world: towards a mighty synergy. In Companion Proceedings of the 36th International Conference on Software Engineering (pp. 638-641).

Clarke, E. M., Klieber, W., Nováček, M., & Zuliani, P. (2011, September). Model checking and the state explosion problem. In LASER Summer School on Software Engineering (pp. 1-30). Springer, Berlin, Heidelberg.

Ding, J., Zhang, D., & Hu, X. H. (2016, June). A framework for ensuring the quality of a big data service. In 2016 IEEE International Conference on Services Computing (SCC) (pp. 82-89). IEEE.

Hummel, O., Eichelberger, H., Giloj, A., Werle, D., & Schmid, K. (2018, August). A collection of software engineering challenges for big data system development. In 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 362-369). IEEE.

Kumar, V. D., & Alencar, P. (2016, December). Software engineering for big data projects: Domains, methodologies and gaps. In 2016 IEEE International Conference on Big Data (Big Data) (pp. 2886-2895). IEEE.

Laigner, R., Kalinowski, M., Lifschitz, S., Monteiro, R. S., & de Oliveira, D. (2018, August). A systematic mapping of software engineering approaches to develop big data systems. In 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 446-453). IEEE.

Otero, C. E., & Peter, A. (2014). Research directions for engineering big data analytics software. IEEE Intelligent Systems, 30(1), 13-19.

Segura, S., Fraser, G., Sanchez, A. B., & Ruiz-Cortés, A. (2016). A survey on metamorphic testing. IEEE Transactions on software engineering, 42(9), 805-824.

Shafi, A., Carpenter, B., & Baker, M. (2009). Nested parallelism for multi-core HPC systems using Java. Journal of Parallel and Distributed Computing, 69(6), 532-545.

Vardi, M. Y. (2001, April). Branching vs. linear time: Final showdown. In International conference on tools and algorithms for the construction and analysis of systems (pp. 1-22). Springer, Berlin, Heidelberg.

Vega-Gisbert, O., Roman, J. E., & Squyres, J. M. (2016). Design and implementation of Java bindings in Open MPI. Parallel Computing, 59, 1-20.