Actas

XXVI Congreso Argentino de Ciencias de la Computación

cacic

2020

05 al 09 de Octubre

Congreso Virtual

RedUNCI

DIIT

Universidad Nacional de La Matanza

Registro del presente Libro de Actas:

Congreso Argentino de Ciencias de la Computación
2020 CACIC: XXVI Congreso Argentino de Ciencias de la Computación / Coordinación General de Alicia Mon. - 1a ed. - San Justo: Universidad Nacional de La Matanza, 2021.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-987-4417-90-9

1. Computación. 2. Congreso. I. Mon, Alicia, coord. II. Título.
CDD 004.071

CACIC 2020
DIIT UNLaM / Red UNCI

Workshop Ingeniería de Software

Coordinadores

Patricia Pesado (UNLP)
Elsa Estevez (UNS)
Alejandra Cechich (UNCOMA)
Horacio Kuna (UNaM)

CACIC 2020
RedUNCI
DIIT
Universidad Nacional de La Matanza

A Parallel Tableau Algorithm for BIG DATA Verification

Fernando Asteasuain[1,2] and Luciana Rodriguez Caldeira[2]

[1] Universidad Nacional de Avellaneda, Argentina
fasteasuain@undav.edu.ar
[2] Universidad Abierta Interamericana - Centro de Altos Estudios CAETI, Argentina
luciana.rodriguezcaldeira@alumnos.uai.edu.ar

Abstract. BIG DATA systems are becoming more and more present in our everyday life generating data and information that needs to be explored and analyzed. In this sense, formal verification tools and techniques must provide solutions to face with these new challenges since they been pointed out as one of the most needed software engineering activities to consolidate BIG DATA modern systems. In this work we present a parallel implementation of a tableau algorithm aiming to improve the performance of our formal verification scheme. The pursued objective behind this transformation is to adapt our framework to deal with BIG DATA systems.

Keywords: BIG DATA, Formal Verification, Parallel Programming

1 Introduction

Modern systems reside in a world where everything is connected and information is generated, consumed and exchanged at a surprisingly increasing growth rate. This enormous amount of data and information needs to be explored and analyzed, activities that originated new disciplines such as BIG DATA [34, 16] or Data Science [32] whereas other important areas such as Artificial Intelligence turned their attention into this topic applying techniques such as Automated Reasoning [27], Machine Learning [2] or Neural Networks [22].

In the last years the Software Engineering community did also make an effort to leverage on this emerging topic. Several works like [24, 15, 26, 23, 30, 25] thoroughly present a complete state of the art stating how software engineering tools, methodologies and techniques are applied and adapted to deal with BIG DATA implications, covering all the very well known software developing phases, from requirements gathering to product release, including modeling, design, testing, validation and verification of Big Data Software Systems. In particular, all the mentioned approaches pinpoint the urgent need of a new arising of more research lines focusing on the formal verification of BIG DATA systems. This is a very challenging path to take since it involves dealing with rigorous performance

requirements in a context where data and information is highly informal and unstructured [25, 24, 20].

In this sense, approaches like [10] try to expand current formal verification tools such as model checking [17] to cope with new architectures for BIG DATA systems such as Cloud Computing [19] and distributed environments. Other approaches involves a classic migration from sequential to parallel model checking [11, 13, 7, 8, 14]. 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 [31, 18].

Given this context in this work we present a parallel implementation of a tableau algorithm which translates graphical scenarios specified in the FVS formal specification language [3, 4] describing behavioral properties into Büchi Automata. The automata built by the tableau can latter be used to feed a model checker in order to perform verification tasks [3]. FVS (Feather weight Visual Scenarios) is a formal and graphical specification language which can be used to specify, verify and synthesize behavior [4, 5]. It denotes a very rich and expressive notation (being for example more expressive than Linear Temporal Logics) and linear as well as branching-time type properties can be specified [3]. In order to adapt FVS for formal verification in BIG DATA system in this work we parallelized the tableau algorithm. **This new parallel implementation of the tableau enables the possibility for FVS to make a solid contribution for the formal verification phase applied to BIG DATA systems.** We developed three different implementations of the tableau algorithm, one using *Java* threads and other two employing two well known libraries for parallel programming: *Open MPI* [33] and *MPJ Express* [29]. Since our framework is developed using the *Java* programming language we relied on *Java* oriented tools. We compared the three versions against each other and against the sequential version of the tableau taking as a case study a complex and industrial relevant protocol verification: the *MS-NNS* protocol [1], a lightweight option to provide authenticated and confidential communication between a server and a client over a TCP connection. Although this early results are preliminaries, we believe are encouraging enough to continue exploring FVS's contributions for formal verification in BIG DATA systems.

The rest of this paper is structured as follows. Section 2 briefly presents the FVS specification language and Section 3 introduces the sequential version of the tableau. Section 4 exhibits the details behind the parallelization of the tableau, considering three different versions, a performance comparison between them and some final observations. Finally, Section 5 comments some related and future work while Section 6 concludes this work highlighting its conclusions.

2 Feather weight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is referred to [3] for a formal characterization of the language. FVS is a graphical language based on scenarios. Scenarios are partial order of events, con-

sisting 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 $A \wedge B$. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxis.

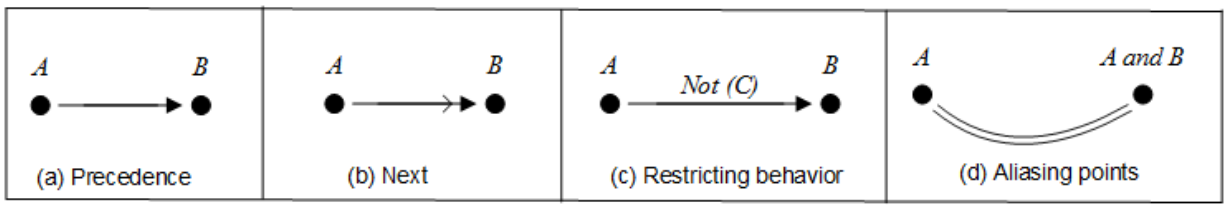

Fig. 1. Basic Elements in FVS

We now introduce the concept of FVS rules, a core concept in the language. Roughly speaking, a rule is divided into two parts: a scenario playing the role of an antecedent and at least one scenario playing the role of a consequent. The intuition is that whenever a trace "matches" a given antecedent scenario, then it must also match at least one of the consequents. In other words, rules take the form of an implication: an antecedent scenario and one or more consequent scenarios. Graphically, the antecedent is shown in black, and consequents in grey. Since a rule can feature more than one consequent, elements which do not belong to the antecedent scenario are numbered to identify the consequent they belong to. An example is shown in figure 2. The rule describes requirements for a valid writing pipe operation. For each write event, then it must be the case that either the pipe did not reach its maximum capacity since it was ready to perform (Consequent 1) or the pipe did reach its capacity, but another component performed a read over the pipe (making the pipe available again) afterwards and the pipe capacity did not reach again its maximum (Consequent 2).

3 Tableau Algorithm: From FVS Scenarios to Büchi Automata

We now present some basic concepts to understand the tableau algorithm while the reader is referred to [3] for a more detailed version of it. From a formal point of view, FVS scenarios can be defined as morphisms from the antecedent to the consequent. The algorithm relies on the notion of situations [3]. In few words, a situation represents for a given rule possible combinations of partial matches from the antecedent to the consequent. Consider the following example

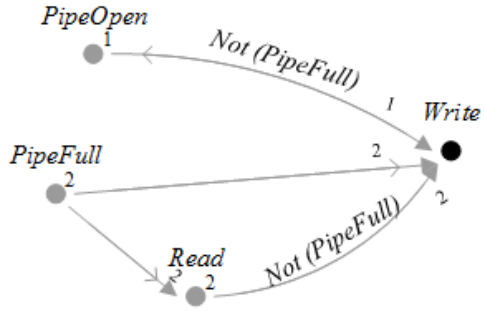

Fig. 2. An FVS rule example

in figure 3. In this case, a rule with two consequents is shown. Furthermore, there are three partial matches for consequent one, and two for consequent two. Therefore, η_1 consists of the three morphisms in the first column (g_1^1, g_1^2, g_1^3), whereas η_2 consists of the two morphisms in the second column (g_2^1, g_2^2).

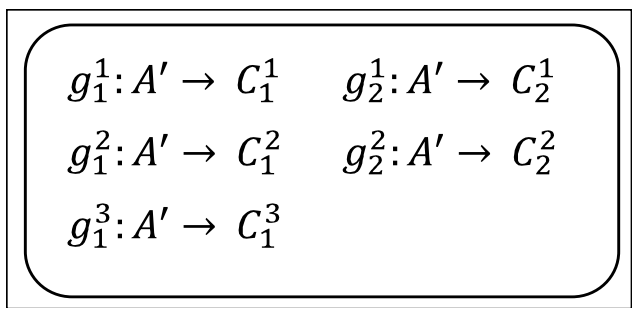

Fig. 3. A situation example

Given a rule R, the tableau builds a Büchi Automaton $\mathcal{B} = \langle \Sigma, S, S^0, \Delta, F \rangle$ such that Σ constitutes minterms over Σ_R and the set of states S are triples $(\Upsilon_R \times bool \times \mathcal{PL}(\Sigma_R))$, where $\mathcal{PL}(\Sigma)$ is a function that labels each point with a given formula. The set Υ_R associated to a state (a set of situations η), denoted *situations*(S), symbolically represents all the possible combination of partial matches obtained up to that state from the antecedent to each consequent. The second term of the triple identify accepting states. This boolean variable is set to **true** when the pattern is completely matched and will make the state transient. Finally, a third element is needed to maintain future obligations of the trace. These formulas are needed when rules predicate about conditions that must hold until the end of the trace.

The pseudo-code sketched in Algorithm 1 computes the successor states for transition relation Δ. Starting from the initial state $(\langle \emptyset, false, true \rangle)$, the automata will try to incrementally "construct" the pattern as events, represented by minterms, occurs. For every minterm, algorithm 1 computes all possible matchings considering matchings in the antecedent and also in each consequent. This is obtained trough two auxiliary algorithms, *advanceAntecedent*(line 5) and *advanceConsequent* (line 6). Line 7 analyzes if any successor reaches a *trap sit-*

uation, a situation where the antecedent has been matched (a morphism such that $A^{'} = A$), but matching for all consequents is known unfeasible. Lines 8 and 9 check if any consequent has been matched by the last move. This is, $goalmatched[i] = true$ if and only if *consequent* C_i is matched. Line 10 analyzes if the next state is an accepting state: a consequent has been matched and it is not a trap situation. Finally, line 11 returns the expected output.

1 $Algorithm\ Succ(S : State, m : minterm)$: set of states;
2 $Precondition : m \wedge obligations(S)$ is satisfiable;
3 $newSits := \emptyset$;
4 **foreach** $\eta \in Situations(S)$ **do**
5 $\quad newSits := add(newSits, advanceAntecedent(\eta, m))$;
6 $\quad newSits := add(newSits, advanceConsequent(\eta, m))$;
7 $trapSituation : \exists \eta \in newSits\ \forall i \forall j \in [1..n]\ g_j^i : A^{'} \rightarrow C_j^i \in \eta \wedge A^{'} = A \wedge C_j^i$ it is not a configuration of C_j;
8 **foreach** $j \in [1..n]$ **do**
9 $\quad goalmatched[j] := \exists \eta \in situations(S) \wedge$
$\quad g_j^i : A^{'} \rightarrow C_j^i \in \eta \wedge C_j^i \xrightarrow{m} C_j \cup \mathcal{F}_j^i \wedge m \in (R_F(C_j^i)) \wedge C_j^i \cup \mathcal{F}_j^i = C_j$;
10 $goalMatched := (\exists \text{j}\ (goalmatched[j])) \wedge (\neg trapSituation)$;
11 return $\{\ \langle newSits, GM, Obligations \rangle$ such that
$GM \rightarrow goalMatched \wedge GM = true \rightarrow \exists j(goalmatched[j]) \wedge Obligations = Obligations(S) \wedge \bigwedge_{j \in I} \mathcal{R}(C_j) \wedge GM = false \rightarrow Obligations = Obligations(S)$

Algorithm 1: Successor states

As an example of the application of the algorithm consider the FVS rule in Figure 4, which represents a classic instantiation of the Response pattern [21]. The automaton built by the tableau is depicted in Figure 5.

Fig. 4. An FVS Rule for the Response Pattern

4 Parallel Tableau Implementation

We now describe the main features of the parallel algorithm. We developed three different versions of it, whose implementations details are presented in Section 4.1. Finally, Section 4.2 presents some final remarks.

After a rigorous analysis of Algorithm 1 we detected two natural points suitable for parallelization. Those two points are: the computation of all the possible

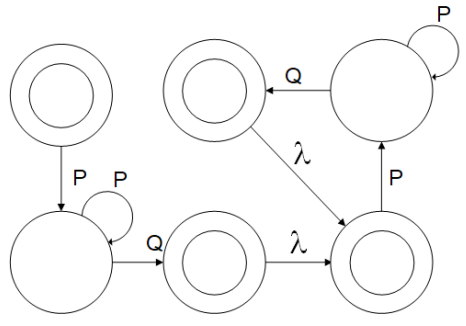

Fig. 5. The automaton built by the tableau for the Response Pattern

antecedents and consequents and tagging all the possible matches (the *For Each* constructor from lines 4 to 6) and checking wether any consequent has been matched by the last move (line 9). These are individual, orthogonal and repetitive tasks that can be easily divided into different nodes to be realized and then the main algorithm can continue once every one is finished. For example, the calculation of all the possible matches for every situation η can be done in parallel where each node calculates the possible matching in antecedent and consequents for every η. Once all the nodes are finished the results are sent to the main algorithm which simple merges the results. Similarly, for the calculation in line 9 the detection of any matching in the consequent can be done in parallel where a node contemplates one situation. When the nodes are finished the main algorithm can obtain the final result. The parallel pseudo-code for the parallel implementation of the tableau is depicted in Algorithm 2.

1 *Algorithm* Parallel $Succ(S : State, m : minterm)$: set of states;
2 $Precondition : m \wedge obligations(S)$ is satisfiable;
3 $newSits := \emptyset$;
4 $PrepareNodes(N_1, N_2, ..., N_m)$;
5 $DistributeAdvancesCalcution(N_1, N_2, ..., N_m, Situations(S))$;
6 $JoinNodes(N_1, N_2, ..., N_m, newSits)$;
7 $trapSituation : \exists \eta \in newSits\ \forall i \forall j \in [1..n]\ g_j^i : A^{'} \rightarrow C_j^i \in \eta \wedge A^{'} = A \wedge C_j^i$ it is not a configuration of C_j;
8 $PrepareNodes(N_1, N_2, ..., N_m)$;
9 $DistributeGoalMatched(N_1, N_2, ..., N_m, Situations(S))$;
10 $JoinNodes(N_1, N_2, ..., N_m, goalMatched)$;
11 $goalMatched := (\exists \text{j}\ (goalmatched[j])) \wedge (\neg trapSituation)$;
12 return { $\langle newSits, GM, Obligations \rangle$ such that
$GM \rightarrow goalMatched \wedge GM = true \rightarrow \exists j(goalmatched[j]) \wedge Obligations = Obligations(S) \wedge \bigwedge_{j \in I} \mathcal{R}(C_j) \wedge GM = false \rightarrow Obligations = Obligations(S)$

Algorithm 2: Parallel Successor states Calculation

Line 4 in Algorithm 2 deals with the nodes preparation and setup, a typical task in parallel systems. Line 5 is in charge of distributing the task of obtaining

the advances of antecedent and consequents in each situation η among the nodes. Finally, in Line 6 all the tasks done by the nodes is united and the new situations set represented by the variable *newSits* is obtained. Similarly, lines 8 to 10 deal with the parallelization of goal calculation and verify if any antecedent has been satisfied.

4.1 Algorithm Implementation and Evaluation

We developed three different implementation for the parallel algorithm delineated in Algorithm 2. We choose *Java* related tools since our algorithm is implemented in that programming language. In the first one we simply use *Java* predefined constructors to deal with parallelism: *threads*. The others two version handle different parallel libraries for *Java*: *Open MPI* [33] and *MPJ Express* [29].

Open MPI is one of the most popular implementations of *MPI*, the Message-Passing Interface, which is one of the predominant programming paradigm for parallel applications on distributed memory computers [33]. This work enables Java MPI bindings which have been included in the *Open MPI* distribution, exposing MPI functionality to Java programmers. It can be easily downloaded and installed from its website[3]. The implementation of the algorithm was very straightforward since parallel constructors application is declarative and intuitive.

The third version was implemented using *MPJ Express* [29], an open source Java message passing library that enables the possibility of introducing parallel instructions in *Java* programs. As in the previous case, it can be easily downloaded and installed from its website[4]. The setup and integration did not result in an straightforward task since several difficulties arose when trying to integrate it to our *Java* framework (for example, library versions incompatibility). However, once these not that unexpected situations when integrating software tools were solved, the codification of the algorithm was achieved without major problems.

We conducted a typical performance comparison including the three versions of the parallel tableau and also its sequential version. We employ as case of study the verification of the *MS-NNS* protocol specified in [3]. In few words, 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. In [3] both the server and the client behavior is specified and verified. For this performance evaluation we consider four cases. In all of them we employ one server whereas the amount of clients was different in each one. In other words, we consider these four cases: *Case 1:* One server and two clients ; *Case 2:* One server and four clients; *Case 3:* One server and eight clients and *Case 4:* One server and sixteen clients. The results are shown in Table 1 .

[3] https://www.open-mpi.org/
[4] http://mpj-express.org/

Case Number	Sequential	Threads	MPJ	Open MPI
1	60 sec	48 sec	53 sec	50 sec
2	109 sec	80 sec	60 sec	70 sec
3	360 sec	115 sec	70 sec	83 sec
4	600 sec	143 sec	80 sec	98 sec

Table 1. Performance Evaluation

4.2 Some Observations

It can be noted from the results in Table 1 that there is a considerable gain when the parallel version of the tableau is employed. The threads version had the best performance in the first case. We believe that the node preparation and other necessary parallel settings was too much overload for a simple case with only two clients. In the following cases the *Open MPI* and the *MPJ Express* versions dethrone the threads implementation and the difference increases when more complex case studies are presented. Between the *Open MPI* and the *MPJ Express* the latter version turns out to be slightly more competitive regarding performance.

We are aware there are several threads to the validity of this initial and exploratory results. First of all, we assume there are always lazy nodes available to receive new tasks. It would be interesting to analyze scenarios where this condition is not necessarily meet since more parallel overhead is naturally expected. Secondly, in all the analyzed cases a similar load balance was assigned to each node. It might occur a different setting in other contexts and examples making necessary to introduce some load balancer strategies who will certainly impact in the performance of the algorithm. Finally, more examples are also needed to further validate this initial experimentation.

However, taking all these facts into consideration we believe the results are promising enough to continue exploring this line of research.

5 Related and Future Work

Several approaches aim to adapt current formal verification techniques to BIG DATA systems.

In [10, 15] a interesting framework for distributed CTL (computation tree logic) model checker is presented. They present a novel architecture employing *HADOOP MAPREDUCE* as its computational engine. They provide a very solid empiric evaluation with several case of studies employing *Amazon Elastic MapReduce* [15] and the *GRID5000* cloud infrastructure [6]. For generating and building distributed state space exploration they rely on a framework called *Mardigras* [9]. We would definitely like to explore in future work the combination of these advanced tools with our specification language FVS.

Other approaches like [11, 13, 7, 8, 14] provide some tools implementing different versions of parallel model checking algorithms for both linear and branching-

time properties. 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 particular, we would like to explore the integration of FVS with the parallel implementation of MTSA (The Modal Transition System Analyser) [12].

In a different direction, work like [20, 28] employ metamorphic testing as an alternative to validate BIG DATA results. We would like to extend this notion to formally model check behavior pursuing the notion of "metamorphic" properties.

6 Conclusions

In this work we present a parallel implementation of the tableau algorithm which translates FVS scenarios into Büchi automata. Actually, three different implementations were developed and analyzed taking as a case of study an industrial relevant protocol with complex behavior. This new version of the algorithm allows FVS specifications to be constructed in a much efficient way, easing the adoption of our language to model and verify behavior in BIG DATA systems. The next step in this direction is to combine FVS specification with parallel and distributed architectures which are the most frequent ones in BIG DATA systems.

References

1. [ms-nns]: .net negotiatestream protocol specification v2.0. http://msdn.microsoft.com/en-us/library/cc236723.aspx, July 2008.
2. E. Alpaydin. *Introduction to machine learning*. MIT press, 2020.
3. F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2):239–274, 2017.
4. F. Asteasuain, F. Calonge, and M. Dubinsky. Exploring specification pattern based behavioral synthesis with scenario clauses. In *CACIC*, 2018.
5. F. Asteasuain, F. Calonge, and P. Gamboa. Behavioral synthesis with branching graphical scenarios. In *CONAIISI*, 2019.
6. D. Balouek, A. C. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, et al. Adding virtualization capabilities to the grid5000 testbed. In *International Conference on Cloud Computing and Services Science*, pages 3–20. Springer, 2012.
7. J. Barnat, L. Brim, M. Češka, and P. Ročkai. Divine: Parallel distributed model checker. In *2010 ninth PDMC*, pages 4–7. IEEE, 2010.
8. A. Bell and B. R. Haverkort. Sequential and distributed model checking of petri nets. *STTT journal*, 7(1):43–60, 2005.
9. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Mardigras: Simplified building of reachability graphs on large clusters. In *RP workshop*, pages 83–95, 2013.
10. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Distributed ctl model checking using mapreduce: theory and practice. *CCPE*, 28(11):3025–3041, 2016.
11. M. C. Boukala and L. Petrucci. Distributed model-checking and counterexample search for ctl logic. *IJSR 3*, 3(1-2):44–59, 2012.

12. M. V. Brassesco. 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 UBA FCEyN Dpto Computacion 2017.
13. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In *International Conference on Formal Methods in Computer-Aided Design*, pages 352–366. Springer, 2004.
14. L. Brim, K. Yorav, and J. Žídková. Assumption-based distribution of ctl model checking. *STTT*, 7(1):61–73, 2005.
15. M. Camilli. Formal verification problems in a big data world: towards a mighty synergy. In *ICSE*, pages 638–641, 2014.
16. M. Chen, S. Mao, and Y. Liu. Big data: A survey. *Mobile networks and applications*, 19(2):171–209, 2014.
17. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
18. E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *LASER School*, pages 1–30. Springer, 2011.
19. T. Dillon, C. Wu, and E. Chang. Cloud computing: issues and challenges. In *2010 AINA*, pages 27–33. Ieee, 2010.
20. J. Ding, D. Zhang, and X.-H. Hu. A framework for ensuring the quality of a big data service. In *2016 SCC*, pages 82–89. IEEE, 2016.
21. M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
22. M. H. Hassoun et al. *Fundamentals of artificial neural networks*. MIT press, 1995.
23. O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid. A collection of software engineering challenges for big data system development. In *SEAA*, pages 362–369. IEEE, 2018.
24. V. D. Kumar and P. Alencar. Software engineering for big data projects: Domains, methodologies and gaps. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2886–2895. IEEE, 2016.
25. R. Laigner, M. Kalinowski, S. Lifschitz, R. S. Monteiro, and D. de Oliveira. A systematic mapping of software engineering approaches to develop big data systems. In *SEAA*, pages 446–453. IEEE, 2018.
26. C. E. Otero and A. Peter. Research directions for engineering big data analytics software. *IEEE Intelligent Systems*, 30(1):13–19, 2014.
27. A. J. Robinson and A. Voronkov. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.
28. S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.
29. A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core HPC systems using java. *J. Parallel Distributed Comput.*, 69(6):532–545, 2009.
30. P. A. Sri and M. Anusha. Big data-survey. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)*, 4(1):74–80, 2016.
31. A. Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.
32. W. Van Der Aalst. Data science in action. In *Process mining*, pages 3–23. Springer, 2016.
33. O. Vega-Gisbert, J. E. Roman, and J. M. Squyres. Design and implementation of java bindings in open mpi. *Parallel Computing*, 59:1–20, 2016.
34. P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.