CACiC 2023
CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Addressing the State Explosion Problem for Big Data Systems Formal Verification

Fernando Asteasuain[1,2]

Universidad Nacional de Avellaneda, Argentina
fasteasuain@undav.edu.ar
Universidad Abierta Interamericana - Centro de Altos Estudios CAETI, Argentina

Abstract. The formal verification of BIG DATA systems remains as a challenging task to be addressed since a very large and complex state space describing the behavior of the system must be explored and verified. In particular, the state explosion problem arises as one of the most problematic issues to be faced against. Some approaches have leveraged on some architectural patterns used in BIG DATA system development, especially those focused on the MAP-REDUCE architecture. Taking this into consideration in this work we present VG-FVS, a new version of our framework FVS (Feather weight Visual Scenarios), which is specially developed to address the state explosion problem. This is achieved by integrating FVS with MaRDiGraS, a generic library which eases the state space exploration using a MAP-REDUCE software architecture. Empirical validation analyzing BIG DATA systems was carried on, showing promising results for our approach.

Keywords: Formal verification, BIG DATA Systems, State Explosion

1. Introduction

Modern software development sometimes includes the outstanding presence of a vast amount of data to be analyzed, explored and inspected. Valuable information and analysis can be automatically gathered employing several artificial intelligence techniques. This phenomena may be the origin of the so called *Data Science Software Development* [12,19,17]. It is clear that systems in this domain also need to be properly formally verified. That is, their expected behavior must be first specified and then it must be verified that the implementation fulfills the specification. Common formal verification techniques and tools were adapted to fit the new challenges that these kind of systems impose [5,13,8,18,11,20,15].

In this sense, the most usual extensions in the software verification phase were focused on improving performance issues by parallelising tools and algorithms or enriching the expressive power of the specification language [5,8,2,16,9]. When advancing in this direction, and old “enemy” of formal software verification must be faced: *the state explosion problem*[21]. This problem can be stated in simple words in the following manner: the number of possible states delineating the behavior of the system under analysis is way too extensive to apply formal verification tools. Perhaps the MaRDiGraS framework [4] represents one of the most meaningful milestones to address this problem. This framework employs a very general formalism to simplify the construction of very large state transition systems on large clusters and cloud

computing platforms. It is based on the Hadoop MapReduce architecture [10,23], which makes it specially suited for Data Science Software Verification.

One of the most distinguished characteristics of this tool is that its mains software components can be easily extended to cope with different formalisms. For example, in [4] it is shown how several formalisms based on Petri Nets can be adapted and used in the MaRDiGraS tool. Taking this fact into consideration in this work we present a novel extension of our behavioral specification tool FVS (Feather Weight Visual Scenarios) [1–3]. **This extension, named VGFVS, serves the purpose of integrating FVS with MaRDiGraS**. FVS is a very simple yet powerful behavioral specification language based on visual scenarios depicting the expected behavior of the system under analysis. It has been extended to cope with temporal and branching behavior, as well as being able to synthesize behavior [3]. A parallel version of FVS integrated with a parallel model checker was also previously introduced in [2]. We now build on the top of these extensions introducing the *VG-FVS version*, integrating FVS with the MaRDiGraS tool so that now FVS is also being able to tackle the state explosion problem. Our approach is empirically validated by analyzing several examples found in the literature considering Data Science and Big Data Systems [5]. These results show a promising future for FVS to formally validate Data Science and BIG Data Systems.

The rest of this work is structured as follows. Section 2 briefly introduces FVS's main features whereas Section 3 describes FVS's formal characterization. This is needed in order to undertand how FVS is integrated with MaRDiGraS, a procedure which is detailed in Section 4. Section 5 exhibits the empirical validation of our approach. Finally, Section 6 presents future and related work and enumerates some final conclusions.

2. Feather weight Visual Scenarios

In this section we will informally describe the standing features of FVS. The reader is referred to [1] 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 $A \wedge B$. It is worth noticing that *A*-event is repeated on the labeling of the second point just because of FVS formal syntaxis.

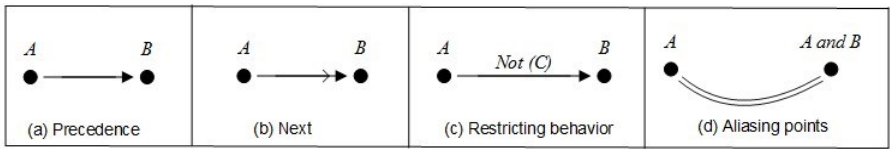

Fig.1. Basic Elements in FVS

We now introduce the concept of FVS rules, a core concept in the language. 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 a very simple requirement from a communication protocol to be embedded on an onboard satellite computer: if some data is ready to be transmitted and the satellite is connected, then transmission can begin.

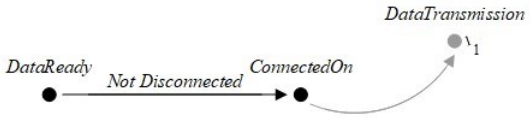

Fig.2. An FVS rule example

3. FVS Formal Definitions

In this section we sketched the most representative formal definitions of the FVS language. For a more comprehensive understanding of FVS's formal syntax and semantic please see [1]. These definitions are necessary to understand how FVS con be embedded into the MarDIGRaS formalism and tool. We need to define the following FVS concepts: *scenarios, morphisms, FVS translation into Büchi automata, how states are represented in FVS, and how state's successors are calculated*. With these definitions in mind the inclusion of FVS into MaRDiGraS can be straightforwardly delineated.

We first define the concept of scenarios.

Definition 1 (FVS Scenario). *An FVS* scenario *is a tuple* $(\Sigma, P, l, \equiv, \not\equiv, <, \gamma)$ *where:*

S1: Σ *is a finite set of propositional variables standing for types of events such that* $\Sigma = \Sigma_c \cup \Sigma_{uc}$ *where* Σ_c *represents controllable events and* Σ_{uc} *non controllable events*

S2: P *is a finite set of points;*

S3: $\ell: P \rightarrow \mathcal{P}\mathcal{L}(\Sigma)$ *is a function that labels each point with a given formula;*

S4: $\equiv \subseteq P \times P$ *is an equivalence relation;*

S5: $\not\equiv \subseteq P \times P$ *is an asymmetric relation among points;*

S6: $< \subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \setminus \{(0, \infty)\}$ *is a precedence relation between points, where* $\mathbf{0}$ *and* ∞ *represent the beginning and the end of execution, respectively; S7:* $\gamma : (\not\equiv \cup <) \rightarrow \mathcal{P}\mathcal{L}(\Sigma)$ *assigns to each pair of points, related by precedence or separation, a formula which constrains the set of events occurrences that may occur between the pair.*

We now formally define morphisms between scenarios. Intuitively, we would like to obtain a matching between scenarios ,i.e., a **mapping** between their points exhibiting how a scenario "specializes" another one [7].

Definition 2 (Morphism). *Given two scenarios* S_1, S_2 *(assuming a common universe of event propositions), and f a total function between* P_1 *and* P_2 *we say that f is a morphism from* S_1 *to* S_2 *(denoted* $f : S_1 \rightarrow S_2$*) iff*

M1: $\ell_2(a) \Rightarrow \ell_1(p)$ *is a tautology for all* $p \in P_1$ *and all* $a \in P_2$ *such that* $a \equiv_2 f(p)$;

M2: $\gamma_2(f(p), f(q)) \Rightarrow \gamma_1(p,q)$ *is a tautology for all* $p, q \in P_1$;

M3: *if* $p \equiv_1 q$ *then* $f(p) \equiv_2 f(q)$ *for all* $p, q \in P_1$;

M4: *if* $p \not\equiv_1 q$ *then* $f(p) \not\equiv_2 f(q)$ *for all* $p, q \in P_1$;

M5: *if* $p <_1 q$ *then* $f(p) <_2 f(q)$ *for all* $p, q \in P_1$.

3.1 Tableau Algorithm: Translating FVS Scenarios into Büchi Automata

We now present some basic concepts to understand the tableau algorithm while the reader is referred to [1] 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 [1]. 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 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).

Given a rule *R*, the tableau builds a Büchi Automaton 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 ($Y_R \times bool \times P\mathcal{L}(\Sigma_R)$), where $P\mathcal{L}(\Sigma)$ is a function that labels each point with a given formula. The set Y_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.

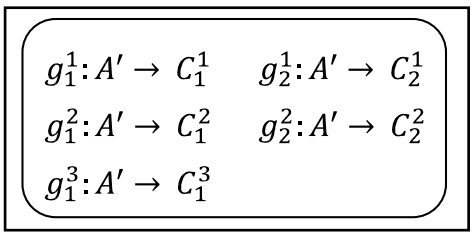

Fig.3. Asituationexample

The pseudo-code sketched in Algorithm 1 computes the successor states for transition relation Δ. Starting from the initial state (⟨∅,false,true⟩), 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 situation*, a situation where the antecedent has been matched (a morphism such that $A' = A$), but matching for all consequents is known unfeasible. A configuration of an scenario stands for a valid set of events occurring in the scenario.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.

4. VG-FVS: FVS meets MaRDiGraS

In this section we show how FVS is connected to the MaRDiGraS tool. In order to achieve this, we first explain MaRDiGraS main concepts (Section 4.1) and latter on, how them are extended to cope with FVS definitions (Section 4.2).

4.1 MaRDiGraS

As explained in [4] MaRDiGraS is a distributed software tool aimed to built big states spaces for different kinds of formalisms. It is a very helpful tool since it is designed for simplifying the task of dealing with a large amount of reachable states by exploiting large clusters of machines. It was conceived as a generic library built on top of Hadoop MapReduce [10,23]. One of the most remarkable features of this tool is that its mains software components can be easily extended to cope with different formalisms. For example, in [4] it is shown how several kinds of Petri Nets can be adapted and used in the MaRDiGraS tool.

```
1   Algorithm Succ(S : State,m : minterm) : set of states;
2   Precondition : m ∧ obligations(S) is satisfiable;
3   newSits := ∅;
4   foreach η ∈ Situations(S) do
5       newSits := add(newSits,advanceAntecedent(η,m));
6       newSits := add(newSits,advanceConsequent(η,m));
7   trapSituation : ∃η ∈ newSits ∀i ∀j ∈ [1..n] g_j^i : A' → C_j^i ∈ η ∧ A' = A ∧ C_j^i it
    is not a configuration of C_j;
8   foreach j ∈ [1..n] do
        goalmatched[j] := ∃η ∈ situations(S) ∧
        g_j^i : A' → C_j^i ∈ η ∧ A' ⟶m C_j ∪ F_j^i ∧ m ∈ (R_F(C_j^i) ∧ C_j^i ∪ F_j^i = ∪ C_i;
9   goalMatched := (∃j (goalmatched[j])) ∧ (¬trapSituation) ;
10  return { ⟨newSits, GM, Obligations⟩ such that
11  GM → goalMatched ∧ GM = true → ∃j(goalmatched[j]) ∧ Obligations =
    Obligations(S) ∧ ⋀_{j∈I} R(C_j) ∧ GM = false → Obligations = Obligations(S)
```

Algorithm 1: Successor states

In the tool the behavior is given in terms of a labeled state transition system. The state explosion problem is addressed by building an abstraction of the original (concrete) state transition system. It is key in this process to properly define the notion of state, reachable states and successors states. Computation starts by considering the initial state of the system under analysis and goes on with a sequential state-space building phase until the set of states not yet explored becomes large enough, where "large enough" is a threshold to be defined for each system [4].

Architecturally, the MaRDiGraS contains two main artifacts: The Data component and the Core Component. The Data component consists of the "business" entities, such as the *State* or the *Edge* classes. This is the component to be addressed when trying to incorporate a given formalism into the tool. On the other hand, the Core components contains the implementation of all the algorithms that implements MaRDiGraS's main functioning.

In concrete, in order to adapt any formalism to this tool the following elements from the MaRDiGraS software must be extended:

- State: is an abstract class which should be extended to instantiate the state concept in a particular formalism.
- Successors: this method must return a list of new State objects representing the states directly reachable from the subject of the call. It must be defined for each formalism to be employed.
- IdentifyRelationship: this method must evaluate the actual relationship between (abstract) states sharing some specific features. The possible output values are: NONE, EQUALS, INCLUDED and INCLUDES.
- Edge: an abstract class which should be extended to represent the edge concept.
- GetFeatures: this functions analyses the equivalence between states.

4.2 VG-FVS: Adding FVS into the MaRDiGraS oily machinery

Given the formal definitions characterizing our FVS language given in Section 3 the combination of FVS with MaRDiGras is described as follows. We defined the MaRDiGrass **state class** as the states of the Büchi Automaton representing an FVS Scenario (see section 3.1). Similarly, the **Successors method** implementation follows the successors algorithm depicted in Algorithm 1. For the **Edge** class we implemented the notion of FVS morphisms. For the **IdentifyRelationship** method we implemented the notion of FVS configurations viewed as a set of FVS *situations*. This method returns NONE if the two configurations given as input do not match, returns EQUALS if they contain the same elements, INCLUDED in the case where the first configurations includes the second one but they are not the same one, and INCLUDES where the second configuration includes the first one but they are not the same one. Perhaps the most challenging implementation arose when dealing with the **GetFeatures** function. In this case, we implemented this function such as it returns the set of morphisms in each situation that fulfill the given consequent of the FVS rule. It was the only definition that requires some extra work since this concept was not included in the FVS formal characterization. Table 1 resumes the integration between FVS and MaRDiGraS.

Table 1. FVS-MaRDiGraS Integration

MaRDiGraS Element	FVS Element
State	FVS Büchi State
Successors	FVS Succ Method
Edge	Morphisms
IdentifyRelationship	Configurations Equivalence
GetFeatures	Morphisms Fulfilment

5. Experimentation Results

The empirical validation of the novel integration introduced in this work was based on previous experiments shown in [2], where we compared the parallel version of FVS combined with a distributed model checker against the technique presented in [5], which also addresses formal verification in big data systems. Also, we conducted the same case of studies and we compared not only execution times but also the space dimension. **In this occasion, we build on the top of that experiment comparing also the performance of *VG-FVS*.** We ran our experiments in a Bangho Inspiron5458, with a Dual Core i5-5200U and 8GB RAM memory. The analyzed case of studies, taken from [5], were a simple *Load Balancing System* and a *Shared Memory Example*, where clients try to gain access to a shared memory through a unique bus. Different configurations considering several number of clients were considered.

Table 2 shows the empirical comparison considering the execution time view. The column MAP-Reduce-CTL stands for the technique in [5], the column Parallel FVS stands for the previous version of our approach whereas the third column corresponds to VG-FVS. Execution times are expressed in seconds.

Table 2. Empirical Comparison: Execution Time

Example	Map-Reduce CTL	Parallel FVS	VG-FVS
5-Shared	110 sec	188 sec	165 sec
10-Shared	1032 sec	1341 sec	1005 sec
2-Load Balancing	43 sec	76 sec	71 sec
5-Load Balancing	114 sec	136 sec	120 sec
10-Load Balancing	18958 sec	20022 sec	18902 sec

Table 3 shows the empirical comparison considering the size dimension. In this case, we present the results for two concrete examples: the shared Memory example with 5 clients and the Load Balancing example with 5 clients. Similar results were obtained for the other versions of the examples considering more clients. For measuring size, we considered the number of states (*st* in Table 3) and transitions (*tr* in Table 3) of the automata involved in the verification process.

Table 3. Empirical Comparison: Size

Example	Map-Reduce CTL	Parallel FVS	VG-FVS
5-Shared	1863 st 7528 tr	3600 st 6500 tr	2160 st 3771 tr
5-Load Balancing	116175 st 425604 tr	301001 st 432653 tr	216720 st 302857 tr

Observations A few important notes can be taken from the empirical evaluation we conducted. Perhaps the most scintillating one is that using the new version of FVS the size of the automata involved was significatively reduced. In average, the size of the automata was reduced by a thirty percent, which represent a notable impact in the verification process. In addition, execution time was also reduced empowering the results of the validation of our approach. It must be noted that our results are still a bit worse than those ones produced by the technique [5]. However, this is somehow expected since our language is more expressive and flexible [1].

6. Conclusions and Future and Related Work

In this work we present VG-FVS, a novel extension of FVS to cope with the state explosion problem, a classic issue to be tacked when dealing with large systems. This is achieved by integrating FVS with the MaRDiGraS tool [4], a very well known software library specially built to address this problem. The results show the clear benefits of this integration, making FVS a very solid alternative to formally validate BIG DATA systems.

With respect to related work the most obvious comparison is against [5]. In this work the MaRDiGraS tool is also employed to formally validate distributed systems in the cloud. The empirical validation of our work shown in Section 5 is based on this approach ([5]). As a first difference we can say that our approach reflects nearly the same performance results while denoting a more expressive and flexible specification language. In the same line, we believe that the declarative nature of FVS specifications makes a more suitable approach to be adopted in the software industry. Work in [6] presents *Maude*, a technique to formally validate systems in the cloud. This tool is specially designated to validate cloud systems, while we pursue a more general objective. However, the architectural solution they provide it is a very skilful one. FVS could benefit introducing some of the architectural patterns employed in *Maude*. We would also like to interact with approaches like [22], which presents techniques in the BIG DATA Testing domain. In [14] an interesting approach based on micro services systems is presented. The technique explains how micro services architecture came in handy for big data systems' formal verification, a future line of work we would also like to address. In addition, we would like to provide a formal proof of the integration presented in this paper, assuring the correctness and soundness of this new FVS version.

References

1. F. Asteasuain and V. Braberman. Declaratively building behavior by means ofscenario clauses. *Requirements Engineering*, 22(2):239–274, 2017.

2. F. Asteasuain and L. R. Caldeira. A sound and correct formalism to specify, verify and synthesize behavior in big data systems. In *Communications in Computer and Information Science, vol 1584*, pages 109–123. http://doi.acm.org/10.1145/587051.587064, Springer, 2021.
3. F. Asteasuain, F. Calonge, M. Dubinsky, and P. Gamboa. Open and branchingbehavioral synthesis with scenario clauses. *CLEI E-JOURNAL*, 24(3), 2021.
4. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Mardigras: Simplified buildingof reachability graphs on large clusters. In *RP workshop*, pages 83–95, 2013.
5. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Distributed ctl model checkingusing mapreduce: theory and practice. *CCPE*, 28(11):3025–3041, 2016.
6. R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. C. Olveczky, and S. Skeirik.¨ Survivability: design, formal modeling, and validation of cloud storage systems using maude. *Assured cloud computing*, pages 10–48, 2018.
7. V. Braberman, D. Garbervestky, N. Kicillof, D. Monteverde, and A. Olivero. Speeding up model checking of timed-models by combining scenario specialization and live component analysis. In *FORMATS*, pages 58–72. Springer, 2009.
8. M. Camilli. Formal verification problems in a big data world: towards a mightysynergy. In *ICSE*, pages 638–641, 2014.
9. A. Cerone and M. Roggenbach. *Formal Methods-Fun for Everybody*. Springer,
10. 2021.
11. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters.*Communications of the ACM*, 51(1):107–113, 2008.
12. O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid. A collection ofsoftware engineering challenges for big data system development. In *SEAA*, pages 362–369. IEEE, 2018.
13. M. Kim, T. Zimmermann, R. DeLine, and A. Begel. Data scientists in softwareteams: State of the art and challenges. *IEEE Transactions on Software Engineering*, 44(11):1024–1038, 2017.
14. V. D. Kumar and P. Alencar. Software engineering for big data projects: Domains,methodologies and gaps. In *IEEEBIGDATA*, pages 2886–2895. IEEE, 2016.
15. R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda,S. Lifschitz, and Y. Zhou. From a monolithic big data system to a microservices event-driven architecture. In *2020 SEAA*, pages 213–220. IEEE, 2020.
16. 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.
17. C. Mandrioli, A. Leva, and M. Maggio. Dynamic models for the formal verificationof big data applications via stochastic model checking. In *2018 CCTA*, pages 1466– 1471. IEEE, 2018.
18. S. Mart´ınez-Ferna´ndez, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz,A. M. Vollmer, and S. Wagner. Software engineering for ai-based systems: a survey. *TOSEM*, 31(2):1–59, 2022.
19. C. E. Otero and A. Peter. Research directions for engineering big data analyticssoftware. *IEEE Intelligent Systems*, 30(1):13–19, 2014.
20. A. Paleyes, R.-G. Urma, and N. D. Lawrence. Challenges in deploying machinelearning: a survey of case studies. *ACM Computing Surveys*, 55(6):1–29, 2022.
21. P. A. Sri and M. Anusha. Big data-survey. Indonesian Journal of Electrical Engineering and Informatics (IJEEI), 4(1):74–80, 2016.
22. A. Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

23. A. Vianna, F. K. Kamei, K. Gama, C. Zimmerle, and J. A. Neto. A grey literaturereview on data stream processing applications testing. *Journal of Systems and Software*, page 111744, 2023.
24. J. Zhang and M. Lin. A comprehensive bibliometric analysis of apache hadoopfrom 2008 to 2020. *IJICC*, 16(1):99–120, 2023.