

A sound and correct formalism to specify, verify and synthesize behavior in BIG DATA systems ^{*}

Fernando Asteasuain^{1,2} and Luciana Rodriguez Caldeira²

¹ Universidad Nacional de Avellaneda, Avellaneda, Argentina

² Universidad Abierta Interamericana - Centro de Altos Estudios CAETI, CABA, Argentina

fasteasuain@undav.edu.ar, lrcaldeira.uai.edu.ar

Abstract. In this work we consolidate our behavioral specification framework based on the Feather Weight Visual Scenarios (FVS) language as a powerful tool to specify, verify and synthesize behavior for BIG DATA systems. We formally demonstrate that our approach is sound and correct end to end, including the latest extensions such as fluents and partial specifications. In addition, our empirical validation is strengthened by adding new and complex case studies and incorporating, besides execution time, space exploration as a factor in the comparison with other approaches. We believe that the contributions introduced in this work aim to point up FVS as a solid tool to formally verify behavior in BIG DATA systems.

Keywords: Formal Verification, BIG DATA, Soundness, Correctness

1 Introduction

Modern world is surrounded by interconnected software devices and artifacts. Almost every device is equipped with some kind of sensor capable of transmitting or receiving information. The topologies and ways to communicate each other are as variable as the kind of domains they can be applied. New trends and technologies arise such as Industry 4.0 [38] or the so called Internet of Things [31]. However, perhaps the most relevant cutting edge emergent topic is built around the concept of BIG DATA systems [24]. These systems include the process of gathering and analyzing the possible huge amount of information available. It is said that the main features concerning BIG DATA systems can be summed up with the “Five Vs”: Variety, Velocity, Volume, Value and Veracity [19, 33].

The software engineering community had been trying to address the challenges involved in BIG DATA systems [7, 12, 18, 26–29, 35, 36]. That is, providing rigorous and sound methodologies, tools, processes and techniques to carry on the complex process of manipulating information in BIG DATA Systems. One of the fields that need particular attention is formal verification [21, 28]. According

^{*} Corresponding Author: Fernando Asteasuain fasteasuain@undav.edu.ar ORCID: 0000-0002-5498-6878

to [28] only 2 of nearly 200 analyzed approaches handling software engineering and big data are related to this particular area. Most of these approaches are focused on just two of the five V's: Velocity and Volume, trying to improve the performance and size exploration of the system under analyses making available, for example, parallel or distributed versions of their tools.

However, the other three V's (Variety, Value and Veracity) have been somehow neglected. These three aspects need powerful, rich and expressive formalisms and notations to reason about the expected behavior of this kind of systems. This is because they need to handle heterogeneous and unstructured data. In addition, proper software engineering approaches need to be proved sound and correct [22, 23, 34].

In [3] we presented the Feather Weight Visual (FVS) framework as a candidate tool to formally apply verification in BIG DATA systems contemplating all the five V's. For Velocity and Volume, we parallelised our scheme and combined FVS with the parallel version of the MTSA model checker (Modern Transition System analyzer), which can be downloaded from <https://mtsa.dc.uba.ar/>. Regarding Variety, Value and Veracity we extended FVS with powerful formalisms: fluents [25], partial specifications with Modal Transition Systems [30] and controller synthesis [13]. Fluents are a compelling logic which introduce a new layer of abstraction in the specification, creating and reasoning with high level events occurring between intervals of time. Partial specifications introduces the novelty of optional behavior which can be latter decided to be included or not in future versions, constituting an appealing tool to address the description of behavior in early stages. Finally, controller synthesis is a potent technique to build a system such that it satisfies its specification by construction. This is achieved employing game theory algorithms.

We now present a robust continuation of that work from two crucial perspectives: theoretical and applied. From the theoretical point of view we provide formal proofs of our approach. In particular, we formally demonstrate that FVS are capable of dealing fluents and partial specifications, the two missing links needed to state that FVS is sound and correct all the way, from the beginning to the end. From the applied perspective, we exhibit a solid empirical validation considering two case of studies available in the literature, and a comparison with other attractive approach [12]. The empirical evaluation considers not only execution time, but also space size exploration. Size and time are by far the most relevant issues in formal verification algorithms. In addition, all the source files from the case studies are available online so that results can be reproduced and replicated.

1.1 Previous Work and New Contributions

FVS was firstly introduced to specify behavior in early stages [1]. Among the distinguishable features of this language we can mention a powerful and expressive notation, being more expressive than classical temporal logics or state-based formalisms [1]. FVS was latter extended to denote branching behavior besides its original linear flavour, coping with the most classic way to reason about

the expected behavior of systems [4]. FVS continue to growth incorporating the possibility to synthesize behavior [5]. That is, FVS can not only describe behavior but also a concrete implementation can be automatically build obtaining a controller for the system under analysis.

FVS was then adapted to address with formal verification of BIG DATA systems. The first step was given in [2], where we present a parallel algorithm that translates FVS specifications into Büchi automata. The formal demonstration of the correctness and soundness of the parallel approach is given in [6]. Finally, in [3] we deepened this road by composing FVS parallel translation algorithm with a parallel version of the MTSA model checker tool [15]. In order to achieve this combination FVS was extended using fluents [25] and partial specifications [30]. We now complete this last step presenting novel and valuable contributions:

1. We formally demonstrate that FVS can denote all the behavior expressible by fluents.
2. We formally demonstrate that FVS can handle partial specifications. To do so, we proved that FVS semantics can be seen as a refinement operation between rules.
3. The formal proofs of our approach constitute a key aspect regarding the correctness and soundness of our approach.
4. We provide a more profound and enriched empiric validation of our scheme. Complete, relevant case of studies are introduced, comparing our approach with a widely known tool in the literature [12]. Besides considering execution time, we incorporated size as a comparison factor. In addition, all the examples' source files and our tool are provided online, making reproducible the results (see <https://gitlab.com/fernando.astesuasain/fvsweb>).

The rest of this paper is structured as follows. Section 2 introduces the FVS framework. Sections 3 and 4 exhibit how FVS can handle fluents and partial specifications respectively, including the formal demonstration to prove our approach to be sound and correct. Sections 5 and 6 cover the case studies whereas Section 7 condenses all the empiric validation results and analysis. Section 8 addresses related and future work and Section 9 presents the conclusions of this work.

2 Feather weight Visual Scenarios

In this section we will informally describe the main characteristics of FVS whereas the complete definition of the language can be found in [1].

FVS is a graphical language based on scenarios. Scenarios describe partial order of events through the use of labeled points and labeled arrows. Points are labeled with the events that occur on that particular moment. In particular, points can be labeled with a logic formulae. Arrows between points indicates a precedence relationship. For instance, in Figure 1-(a) *A*-event precedes *B*-event. In FVS it can be specified the very next occurrence of an event and not any other occurrence in the future. This is graphically shown with a small arrow near the

point of interest. For example, Figure 1-b shows an scenario that properly signal the first occurrence of a B event after the occurrence of a A event. Arrows can be labeled to restrict behavior. Figure 1-c exhibits a situation where a C event must not occur between events A and B . In FVS, aliasing between points can also be denoted. Scenario in 1-d indicates that a point labeled with A is also labeled with $A \wedge B$. Note that A -event is repeated on the labeling of the second point just because of FVS formal syntax. This constitute an interesting syntactic sugar mechanism of our notation.

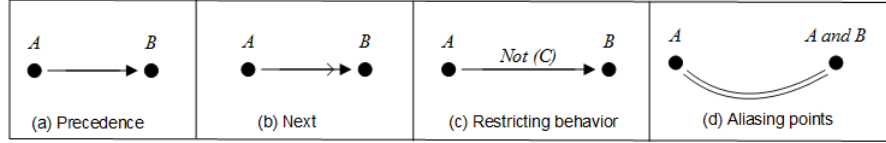


Fig. 1. Basic Elements in FVS

We now introduce the concept of FVS rules, a core concept in the language. FVS rules consists of one scenario playing the role of an antecedent or a triggering condition, and one or more consequents, playing the role of the behavior to be triggered after the condition is met. The intuition is that whenever a trace “matches” a given antecedent scenario, then it must also match at least one of the consequents. 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. Some examples are shown in Figure 2 describing the behavior of a semaphore controlling access to a critical section in a distributed environment. The rule at the top says that once a process is in the critical section no other process will enter until it exits. The second rule (the one in the middle) has two consequents. After an *Critical* event a process will either enter the critical section (Consequent 1) or it will be moved to the waiting queue (Consequent 2). Finally, rule at the bottom of Figure 2 introduces a fairness concern. It says that between two consecutive admission of a process $P1$ to the critical section, if another process $P2$ requires to enter, its access will be satisfied. This avoids that process $P2$ dies from starvation, a typical concern in distributed systems.

2.1 FVS and Ghosts Events

Besides the actual events occurring in the system, FVS is able to reason about high-level events [1]. Adding new layers of abstraction is a common and useful practice when specifying behavior. We denote this high-level events as “ghost” events while system’s events are called “actual” events. We employ FVS rules to model how “ghost” events are activated, and latter, other FVS rules expressing behavior with these events. In this way, FVS can introduce a new layer of abstraction. For example, in a software architecture system one way to address the

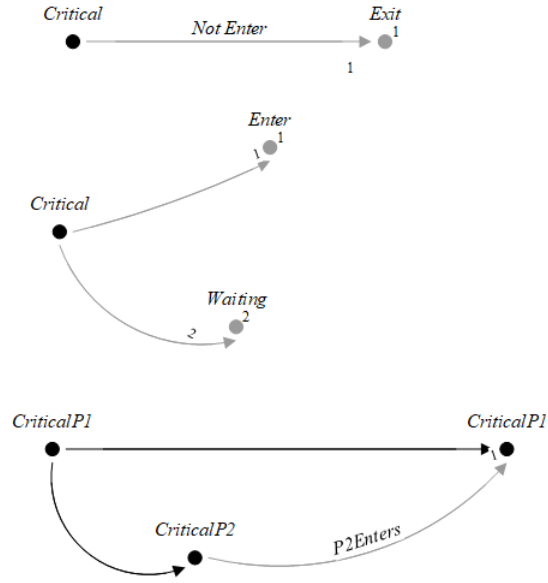


Fig. 2. FVS rules examples

availability concern is detecting when a server is near to its maximum capacity and reduces the level of services to offer to avoid a server's crash down. This downgrade mode in the server can be seen as a ghost event and occurs when the systems detects that the number of connection is too high. The first rule in Figure 3 establishes the conditions that triggers the occurrence of the ghost event *DowngradeMode*, whereas the second rule directly shapes behavior using ghosts events. It says that while working in the down grade mode the server reduces the services it can respond until the downgrade is over.

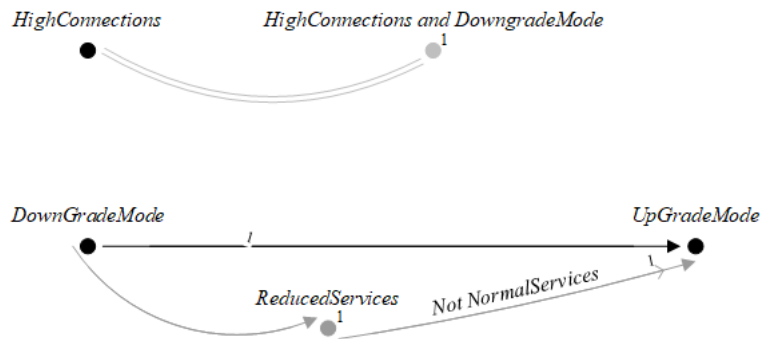


Fig. 3. An FVS ghost event behavior

2.2 Behavioral Synthesis in FVS

FVS specifications can be used to automatically obtain a controller employing a classical behavioral synthesis procedure. The complete process is fully described in [5]. In few words, FVS rules are first translated into Büchi automata. Then we relied on several tools such as [32] or [37] in order to synthesize the denoted behavior and obtaining a controller. Finally, in [3] we showed how FVS can also be combined with the parallel version of the MTSA model checker [15]. Obtaining a controller complete all the verification path point to point, from the specification to the implementation. In this way, we can state that FVS is a proper framework to handle system's behavior specification, verification and synthesis.

3 Fluents and FVS

In this section we explain how FVS can denote fluents behavior, and we formally demonstrate this claim. Fluents [25] can be seen as an extension of the LTL logic. A fluent represent a particular behavior of the system that take place between intervals or moments, introducing a new layer of abstraction in the specification. Starting and ending actions of these intervals are defined, and then properties can be stated using the mentioned intervals. For example, recall the FVS ghost event example in Figure 3. This downgrade mode can be specified using a fluent defining the *HighConnection* event as an starting event, a *LowConnection* event as a ending event, and a LTL formula such as $\Box (\text{DowngradeMode} \Rightarrow \text{ReducedServices})$.

FVS can specify fluents in a very simple and direct way employing ghosts events. Fluents starting and ending delimiters are modeled with FVS rules, fluents predicates are modeled with ghost events, and intervals behavior are simply FVS rules using those ghosts events. The next subsection addresses the formal proof showing that FVS can denote all behavior expressible in fluents.

3.1 Formal proof: FVS and Fluents

In this section we formally demonstrate that FVS can denote all the behavior expressible employing Fluents [25]. In other words, given a fluent FL we build a set $RFVS$ of FVS rules such that the language denoted by FL is included in the language denoted by $RFVS$: $L(FL) \subseteq L(RFVS)$. A few concepts must be cleared before the formal demonstration is given. First of all, we adopted the notion of traces as a linear sequence of events, and those traces accepted by the fluents or the FVS rules defines the language of the system. More formally, a trace t is seen as the sequence $\{t_0, t_1, \dots, t_k\}$. FVS semantics [1] is based on the concept of morphisms, a mathematical relationship that establishes when a given trace satisfy an FVS rule. In few words, if for every item in a trace t a morphism m exists such that t can be related to a rule R , then t satisfies R . More formally, $\forall i, 0 \leq i \leq k$ in $\{t_0, t_1, \dots, t_k\}$, there exists a morphism m_i such that $t_i \xrightarrow{m_i} t_{i+1}$. It can be said that each morphism makes an advance in the trace.

A fluent is formally defined establish a set of initiating actions, a set of terminating actions, and the behavior itself happening between the start and the end [25]. That is, a fluent $FL = \langle I_{FL}, T_{FL} \rangle$, where I_{FL} denotes the starting events, T_{FL} the ending events, and $I_{FL} \cap T_{FL} = \emptyset$. In addition, a boolean variable called *Initially* is defined to express if the fluent is true or false at the beginning of the execution. Finally, a trace $t = \{t_0, t_1, \dots, t_k\}$ is said to be valid for a fluent if one of the following conditions holds:

1. Condition 1: *Initially* $\wedge (\forall i, 0 \leq i \leq k, t_i \notin T_{FL})$. This conditions covers the case where the fluent is valid in the beginning and no terminating events occurred.
2. Condition 2: $\exists j, j \leq k, t_j \in I_{FL} \wedge (\forall i, j \leq i \leq k, t_i \notin T_{FL})$. This condition covers the case where the fluent initiates at a given point, and no terminating events occurs.

We are now ready to address the formal demonstration showing that FVS can denote any behavior expressible by a fluent. Given a fluent FL we define a set of FVS rules called *RFVS* in the following way:

1. For every action a in I_{FL} there exists a rule r_a in *RFVS*. That is, for every initiating event there is an FVS rule that starts the behavior of the fluent.
2. For every action a in T_{FL} there exists a rule r_a in *RFVS*. That is, for every terminating event there is an FVS rule that ends the behavior of the fluent.
3. For every action established for the fluent, we add a rule in *RFVS* mimicking its behavior.
4. If the variable *Initially* is set to true, we add an FVS rule activating the fluent.

We now demonstrate that the language of a fluent FL is included in the language denoted by *RFVS*: $L(FL) \subseteq L(RFVS)$. That is, given a trace $t = \{t_0, t_1, \dots, t_k\}$ active in the fluent FL we will show that $t \in L(RFVS)$.

If t is active in FL then either Condition 1 or Condition 2 must hold. Let's assume that Condition 1 holds. Since FL 's *Initially* variable is true, there exists a rule r in *RFVS* initiating the fluent behavior. So, there exists a morphism m_0 such that $t_0 \xrightarrow{m_0} t_1$. What is more, since $\forall i, 0 \leq i \leq k, t_i \notin T_{FL}$, the fluent will always advance (no ending events occurs). The set of rules *RFVS* includes, by construction, all the rules that advances a fluent. This implies that $\forall i, 1 \leq i \leq k$, there exists a morphism m_i such that $t_i \xrightarrow{m_i} t_{i+1}$. Therefore, the trace t belongs to the set of traces denoted by *RFVS*: $t \in L(RFVS)$, which was the goal of our demonstration. $\square$.

Now let's analyze the case where Condition 2 holds. If Condition 2 holds, then there exists an event j which activates the fluent. Given the way the set of rules *RFVS* was build (including one rule for each activation), we can affirm that there exists a morphism m_j such that $t_i \xrightarrow{m_j} t_j$. Since no terminating events occurs, there will always be a next morphism advancing the trace: $\exists$ morphism m_n such that $t_n \xrightarrow{m_n} t_{n+1}$, $n \leq j \leq k$. Therefore, $t \in L(RFVS)$. This concludes the second part of the demonstration. $\square$.

Since we have proved that the hypothesis holds for both conditions, we can conclude that $L(FL) \subseteq L(RFVS)$. $\square$ $\square$.

4 Partial Specifications and FVS

Partial Specifications are a crucial tool to model and shape early behavior of computer systems, introducing the possibility to distinguish between optional and required behavior. Probably the most widely known formalism addressing Partial Specifications is Modal Transitions Systems (MTS) [30]. A process called *refinement* guides the iterations between specifications versions, guaranteeing that the new version satisfy all the original assumption in the system.

FVS includes the possibility to express partial and required behavior since FVS rules can hold multiple consequents. So, a future rule modifying the number of consequents is a refinement of the previous version. So, FVS provides the refinement operation by its traces semantics definition. A rule with multiple consequents can be replaced in next versions with a rule with less consequents (the optional behavior is discarded), or combining two consequents into one (making mandatory an optional behavior). In the following subsection we formally demonstrate this claim.

4.1 Formal Proof: FVS and Partial Specifications

In this section we formally demonstrate that FVS rule semantics can be seen as a refinement relationship, making FVS a suitable notation to denote partial specifications. For the formal demonstration we employed MTS [30], since this formalism is one of the most widely known addressing partial specifications. The most distinguishable feature in partial specifications is the possibility to divide events into two categories: required and optional behavior.

The refinement operation relates two different versions of the system, where the newer version makes former optional behavior either turned into required or discarded. More formally, given two MTS M and N we define a refinement relationship H , $H \subseteq States(M) \times States(N)$ if and only if, $\forall l \in events(M) \wedge \forall (s, t) \in H$, the following conditions holds:

1. if $(s, l, s') \in Required(M)$ then $\exists t$ such that $(t, l, t') \in Required(N) \wedge (s', t') \in H$.
2. if $(t, l, t') \in Optional(N)$ then $\exists s$ such that $(s, l, s') \in Optional(M) \wedge (s', t') \in H$.

In other words, we can establish that an MTS N refines another MTS M if every required transition in M belongs to N and every optional behavior of N is present in M .

The hypothesis to be demonstrated is that an FVS rule with multiple consequents represent a valid refinement operation. That is, given an FVS rule M with k consequents, and an FVS rule N with $k-1$ consequents, where $Consequents(M) = Consequents(N) \cup C_k$, $\wedge C_k$ is the missing consequent in N . Given this setting,

we must prove that the traces accepted by N are a refinement operation for the traces accepted by M . According to the refinement operation definition, we must prove the following claims:

1. $\forall$ trace $t \in \text{required}(M)$, $t \in \text{traces}(N)$
2. $\forall$ trace $t \in \text{optional}(N)$, $t \in \text{traces}(M)$

To prove Claim 1 we consider two cases: where t satisfies the rule and where t does not satisfy the rule. If the trace t does not satisfy rule M , t does not satisfy rule N either, since N is the same rule without one consequent. Since traces that do not satisfy the rules are not included in the system by definition of FVS semantics, this case can be discarded. Therefore, we must conclude that t do satisfy M . This indicates that given $t = \{t_0, t_1, \dots, t_k\}$, there exists a morphism $m_i \forall i$, $0 \leq i \leq k$ such that $t_i \xrightarrow{m_i} t_{i+1}$, leading t to an accepting state. Let C_k be the missing consequent in N (recall that $\text{Consequents}(M) = \text{Consequents}(N) \cup C_k$). Lets suppose by the absurdum that $t \notin \text{traces}(N)$. This implies that one of the morphisms m_i belongs exclusively to consequent C_k . However, this indicates that t does not satisfy the rule, which contradicts the initial assumptions. The contradiction arose from assuming that $t \notin \text{traces}(N)$, therefore $t \in \text{traces}(N)$, which was what we aimed to prove. This concludes the first part of the demonstration. $\square$.

We now address Claim 2: $\forall$ trace $t \in \text{optional}(N)$, $t \in \text{traces}(M)$. As we did with Claim 1, the case where t does not satisfy the rule can be ignored. Therefore, we can assume that t satisfy the rule. Let assume by the absurdum that $t \notin \text{traces}(M)$. If that is the case, then there exists a morphism m_i such that $t_i \xrightarrow{m_i} t_{i+1}$ for some $i \wedge m_i$ belongs exclusively to C_k . However, if this is true, then t does not satisfy the rule, which is a contradiction of the initial assumptions. And this contradictions backtracks to assuming that $t \notin \text{traces}(M)$. This imply that $t \in \text{traces}(M)$, which was our goal to be proved. This concludes the second part of the demonstration. $\square$.

Given that Claim 1 and Claim 2 had been proved, we can affirm that FVS rules denote a refinement relationship. $\square \square$.

5 Simple Load Balancing Case Study

Inspired on [12] we completely formalized the Simple Load Balancing example, including the specification of its behavior and the automatic construction of a controller for it. The setting environment for the load balancer system consists of c clients, two servers, and a load balancer process playing the role of a proxy between them. As in [12], we studied a model with 2, 5 and 10 clients. Some rules describing the behavior can be found in Figure 4. The rule in Figure 4-a says that every client should eventually make a request whereas the rule in Figure 4-b states that the server enters the idle state when it finishes processing a client's request. Finally, rule in Figure 4-c dictates that the server can not enter the idle state while processing a request.

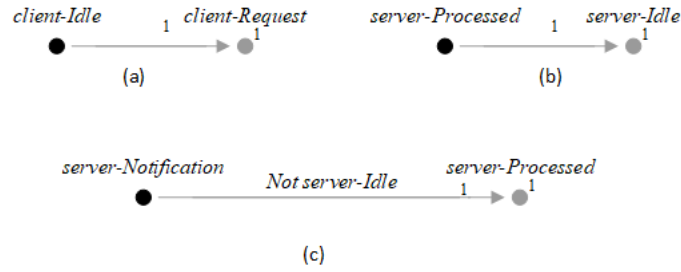


Fig. 4. Some FVS graphical rules for the Load Balancer Example

Part of the controller for the Load Balancer Server Controller is shown in Figure 5. Recall the complete files can be inspected from the FVS web site (<https://gitlab.com/fernando.asteasuain/fvsweb>), including both the server and client controllers.

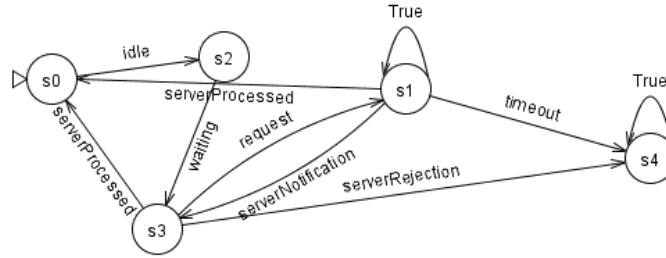


Fig. 5. Part of the Load Balancing Controller system for the server side

6 Shared Memory Case Study

Completing the empiric evaluation presented in [12] we now present the FVS specification for a shared memory example. This system represents clients trying to gain access to a shared memory through a unique bus. As in [12], we specified two variations of the system: with 5 clients and with 10 clients.

Three rules describing the behavior can be found in Figure 6. The first one (see Figure 6-a) establishes a progress property: every process must advance to the active state while waiting in the queue. The property in Figure 6-b demands that processes must be in memory in order to be in the active state. Finally, rule

in Figure 6-c says that if a process should wait in the queue if the intermediate memory is full.

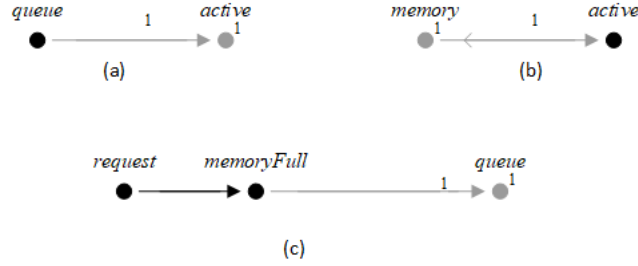


Fig. 6. Three FVS rules for the shared Memory Example

Part of the controller for the Shared Memory example is shown in Figure 7.

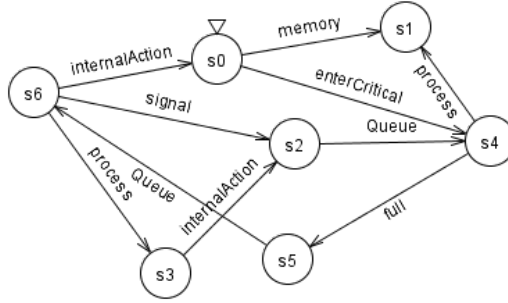


Fig. 7. A controller for the Shared Memory System

7 Empirical Evaluation

In this section we present a comparison between our approach and the technique presented in [12], which also addresses formal verification in big data systems and we conducted the same case of studies. We compared not only execution times but also the space dimension. So, both views, space and time, which are critical in any formal approach are thoroughly analyzed. Although they verified

the behavior of the algorithms and we obtained controllers instead, the involved tasks and objectives are similar enough to produce valuable results from their performance execution time and size comparison. We ran our experiments in a Bangho Inspiron5458, with a Dual Core i5-5200U and 8GB RAM memory.

Table 1 shows the empirical comparison considering the execution time view. The column MAP-Reduce-CTL stands for the technique in [12] while the column ParallelFVS stands for our approach. Execution times are expressed in seconds.

Table 1. Empirical Comparison: Execution Time

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

Table 2 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 2) and transitions (*tr* in Table 2) of the automata involved in the verification process.

Table 2. Empirical Comparison: Size

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

Valuable observations can be stated by analyzing both tables. Considering time execution, it can be observed that our approach holds a worst execution time compared with [12] in small examples. We believe the reason behind this is that the parallel overhead in our approach overcomes the parallel benefits. However, the difference diminishes as the examples become more complex, being almost negligible for the most complex cases (for example, the shared memory with 10 clients or the load balancing example with 15 clients). This is a good result for our approach, since we can offer our rich and expressive specification language without resigning performance.

Regarding the size comparison (see Table 2) it can be seen that the automata obtained in our approach are bigger, specially considering the number of states.

FVS approach almost duplicate the number of states while the number of transitions are pretty similar. Our main hypothesis to explain these numbers is that the algorithm translating FVS rules into Büchi automata is introducing more states than the ones that are actually needed. This particular point should be addressed in future work. However, as seen in Table 1 this size difference does not have an actual impact in the execution times.

In conclusion, we can state that FVS came out unhurt from the empirical validation point of view. FVS can offer a powerful and rich notation to express and synthesize behavior in a competitive manner considering execution time and size of the involved artifacts.

8 Related and Future Work

Perhaps the pioneer approach addressing formal verification in BIG DATA systems is [12, 18]. This work is focused on distributed CTL (computation tree logic) and the architecture of their scheme is based on a HADOOP MAPREDUCE fashion, a classic architecture in this domain. The distributed state space exploration rely on a framework called Mardigras [11]. This latter tool is able to work with very different formalisms such as temporal logics or Petri Nets. Given this tool flexibility we would definitively like to explore if FVS can be integrated with Mardigras since the potential benefits of this combination sounds promising.

Other valuable approaches such as [9, 10, 14, 16, 17] focused on parallel or distributed model checking. It would be interesting to check if the integration of one of this tools with FVS outcomes the results obtained with FVS and MTSA presented in this paper. Similarly, other approaches aim to speed-up the model checking by performing parallel verification of very small units pieces of behavior [8, 20]. For example, these units are called *swarms* in [20].

Regarding future work, we would like to strengthen our empirical validation by exploring the interaction of FVS with other model checkers. Another future research line is trying to optimize the algorithm translating FVS scenarios into Büchi automata. As noted in Section 7 the automata built by our algorithm do present more states than other known tools. Despite this does not have an impact on performance is definitively an issue to be addressed. Finally, we would like to add expressiveness as a factor to compare FVS with other approaches.

9 Conclusions

In this work we accentuated FVS as an appealing framework to address formal verification aspects in BIG DATA domains. From the theoretical point of view, we incorporated formal proofs for our approach, guaranteeing the soundness and correctness of all the involved steps from end to end. From the applied point of view, we analyzed a complete set of benchmarks and compare FVS with the most successful tool in the field. We believe that the theoretical and applied

contributions introduced in this work are relevant for the formal verification software engineering phase in a forefront domain such as BIG DATA systems.

References

1. F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2):239–274, 2017.
2. F. Asteasuain and L. R. Caldeira. A parallel tableau algorithm for big data verification. In *CACIC. ISBN 978-987-4417-90-9*, pp 360-369, 2020.
3. F. Asteasuain and L. R. Caldeira. An expressive and enriched specification language to synthesize behavior in big data systems. In *CACIC. ISBN 978-987-633-574-4*, pp 357-366, 2021.
4. F. Asteasuain, F. Calonge, and P. Gamboa. Behavioral synthesis with branching graphical scenarios. In *CONAIIISI*, 2019.
5. F. Asteasuain, C. Federico, D. Manuel, and G. Pablo. Open and branching behavioral synthesis with scenario clauses. *CLEI JOURNAL*, 24(3):1–20, 2021.
6. F. Asteasuain and R. C. Luciana. Exploring parallel formal verification of big-data systems. *Revista Ciencia y Tecnologia*, 21(2):7–18, 2021.
7. 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 *CLOSER*, pages 3–20. Springer, 2012.
8. J. Barnat, P. Bauch, L. Brim, and M. Češka. Employing multiple cuda devices to accelerate ltl model checking. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 259–266. IEEE, 2010.
9. J. Barnat, L. Brim, M. Češka, and P. Ročkal. Divine: Parallel distributed model checker. In *2010 ninth PDMC*, pages 4–7. IEEE, 2010.
10. A. Bell and B. R. Haverkort. Sequential and distributed model checking of petri nets. *STTT journal*, 7(1):43–60, 2005.
11. 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.
12. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Distributed ctl model checking using mapreduce: theory and practice. *CCPE*, 28(11):3025–3041, 2016.
13. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa’Ar. Synthesis of reactive (1) designs. 2011.
14. M. C. Boukala and L. Petrucci. Distributed model-checking and counterexample search for ctl logic. *IJSR* 3, 3(1-2):44–59, 2012.
15. 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.
16. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In *FMCAD*, pages 352–366, 2004.
17. L. Brim, K. Yorav, and J. Žídková. Assumption-based distribution of ctl model checking. *STTT*, 7(1):61–73, 2005.
18. M. Camilli. Formal verification problems in a big data world: towards a mighty synergy. In *ICSE*, pages 638–641, 2014.
19. F. Cappa, R. Oriani, E. Peruffo, and I. McCarthy. Big data for creating and capturing value in the digitalized environment: unpacking the effects of volume, variety, and veracity on firm performance. *Journal of Product Innovation Management*, 38(1):49–67, 2021.

20. R. DeFrancisco, S. Cho, M. Ferdman, and S. A. Smolka. Swarm model checking on the gpu. *STTT*, 22(5):583–599, 2020.
21. 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.
22. R. Drechsler et al. *Advanced formal verification*, volume 122. Springer, 2004.
23. V. D’silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
24. C. K. Emani, N. Cullot, and C. Nicolle. Understandable big data: a survey. *Computer science review*, 17:70–81, 2015.
25. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *European software engineering conference*, pages 257–266, 2003.
26. 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.
27. O. Inverso and C. Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In *PPoPP*, pages 202–216, 2020.
28. 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.
29. 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.
30. K. G. Larsen and B. Thomsen. A modal process logic. *LICS, pages 203210, IEEE*.
31. S. Li, L. Da Xu, and S. Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, 2015.
32. S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): A case study. *arXiv preprint arXiv:1602.01172*, 2016.
33. M. Naeem, T. Jamal, J. Diaz-Martinez, S. A. Butt, N. Montesano, M. I. Tariq, E. De-la Hoz-Franco, and E. De-La-Hoz-Valdiris. Trends and future perspective challenges in big data. In *Advances in Intelligent Data Analysis and Applications*, pages 309–325. Springer, 2022.
34. S. Nejati. Next-generation software verification: An ai perspective. *IEEE Software*, 38(3):126–130, 2021.
35. C. E. Otero and A. Peter. Research directions for engineering big data analytics software. *IEEE Intelligent Systems*, 30(1):13–19, 2014.
36. P. A. Sri and M. Anusha. Big data-survey. *Indonesian Journal of Electrical Engineering and Informatics (IJEI)*, 4(1):74–80, 2016.
37. Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan. Goal: A graphical tool for manipulating büchi automata and temporal formulae. In *TACAS*, pages 466–471. Springer, 2007.
38. L. D. Xu, E. L. Xu, and L. Li. Industry 4.0: state of the art and future trends. *International Journal of Production Research*, 56(8):2941–2962, 2018.