



Formally Verifying Data Science Systems with a Sound and Correct Formalism

Fernando Asteasuain^{1,2} 

¹ Universidad Nacional de Avellaneda, Avellaneda, Argentina

² Universidad Abierta Interamericana - Centro de Altos Estudios CAETI,
Buenos Aires, Argentina
fasteasuain@undav.edu.ar

Abstract. The state explosion problem arises as one of the most problematic issues to be faced against when trying to formally validate Data Science Software Systems. This challenge imposes the synergy and combination of tools. Taking this into consideration in this work we present a robust theoretical feature of our behavioral framework *VG-FVS*: we formally prove that our approach relies on a sound, complete and correct formalism. *VG-FVS* is a brand new version of our framework *FVS* (Feather weight Visual Scenarios), which is specially developed to address the state explosion problem by integrating *FVS* with *MaRDiGraS*, a generic library which eases the state space exploration using a *MAP-REDUCE* software architecture.

Keywords: Formal Verification · Data Science · State Explosion

1 Introduction

The emergence of new software paradigms pushes the Software Engineering community to adapt the current state of its tools mechanisms, procedures and methodologies to build modern software with the desired quality. For example, user interface design, validation and testing were overhauled to address the construction of systems to run in modern smart phones technologies [14, 19, 21]. The continuous growth of the so called Data Science Software Development [13, 17, 19] constitutes another important milestone for the evolution of software engineering. The vast and amorphous data and information to be explored and analyzed imposes the appearance of new ways to guarantee the quality of developed software. Some of approaches addressing this evolution by for example, optimizing resources and performance, are work like [5, 12, 15, 16, 22]. In particular, formal verification been flagged as one of the areas that more urgently need to be tackled. According to [11, 15], only two of nearly one hundred analyzed approaches addressing new software engineering methods for Data Science Software Systems were related to formal validation.

In [1] we took an important first step in the road to offer new contributions for the formal verification of Data Science Systems. We presented *VG-FVS*, a

specially designed version of our behavioral framework FVS [2,3] to address formal verification in this domain. FVS, which stands for *Feather Weight Visual Scenarios*, is a very expressive declarative graphical language to express the desired behavior of systems. Due to its flexibility it can be fully integrated with other model checking tools to expand its applicability [3]. For example, FVS specifications can be synthesized to automatically obtain a controller of the system. In [1] we combined FVS with the MaRDiGraS tool [4,5] given birth to a new flavour of our tool that we named *VG-FVS*. This version is specially suited to tackle a famous issue when formally verifying software systems: the state explosion problem [24]. Usually, the space to be explored by an automatic tool such as model checker is given as a set of states, and a mathematical function that given the occurrence of a new event indicates the next state. However, as the problem complexity increases the number of states grow exponentially and became a serious menace to formally verification tools. This problem exacerbates in the context of Data Science Software Systems. The mentioned MaRDiGraS tool employs an astute way of dealing with the explosion of states. It relies on a widely known technique used in distributed systems to optimize resources: Hadoop MapReduce architecture [10,25].

Since MaRDiGraS can be easily extended in [1] we integrated this tool with FVS. We now deepen that line of research by providing formal proofs demonstrating that this integration is *sound and complete*. To provide formal proofs of the soundness and correctness of any formalism or mechanism is a crucial and highly desirable feature, especially when dealing with formal verification techniques. For example, work like [6,18,20,26] provides formal assurance of their correct behavior in several domains such as robotic missions, probabilistic measures or temporal logics. *Besides providing the formal proof of VG-FVS soundness and correctness, in this work we also enrich the formal aspects of VG-FVS early introduced in [1] and also formal aspects of the integration of FVS and MaRDiGraS.*

The rest of this work is structured as follows. Section 2 briefly introduces FVS's main features whereas Sect. 3 describes FVS's formal characterization. This is needed in order to understand how FVS is integrated with MaRDiGraS, a procedure which is detailed in Sect. 4. Section 5 exhibits the formal proofs proving that VG-FVS is sound, correct and complete. Finally, Sects. 6 and 7 present future and related work and enumerates some final conclusions.

2 Feather Weight Visual Scenarios

In this section we will informally highlight the standing features of FVS. The reader is referred to [2] 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 Fig. 1-(a) is shown an scenarios where three events are occurring in a *Formula 1* simulator system:

two consecutive *Box* events and an *EntersPit* event. In this scenario, the *EntersPit* is preceded by the two *Box* events. We use an abbreviation for a frequent sub-pattern: a certain point represents the next occurrence of an event after another. The abbreviation is a second (open) arrow near the destination point. For example, in Fig. 1-b the scenario captures the very next *FlatEnabled* following an *DRSEnabled* event, and not any other *FlatEnabled* event. Conversely, to represent the previous occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme. This is also shown in Fig. 1-b, where the rightmost scenario handles the immediate previous occurrence of the *DRSEnabled* event given the occurrence of a *FlatEnabled* event. Finally, we provide a useful way to restrict behavior and denote prohibited situations. This is achieved by labelling arrows denoting the precedence events. For example, in Fig. 1-c the *Push* event is followed by an *Accelerate* event given that no *RedFlag* event occurs between them. This is because when a red flag is raised in Formula 1 races, all the cars must significantly reduce their speed.

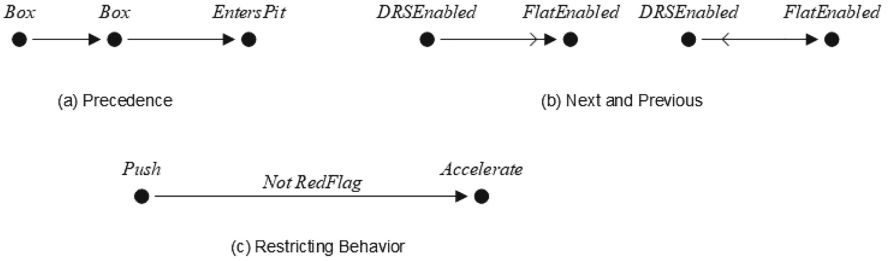


Fig. 1. Basic Elements in FVS

2.1 VG-FVS Rules

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.

Two simple examples taken from the Formula 1 simulation system are shown in Fig. 2. The first describes an overtake from a legendary *Ferrari* racing car given that the driver receives a push instruction to accelerate its car given that no red flag situation occurred. The second rule shows two possible alternative given the occurrence of a tyre degradation event. Consequent 1 reflects the behavior where a boxes call is made whereas Consequent2 shows the behavior when it is decided to maintain the car in the track.

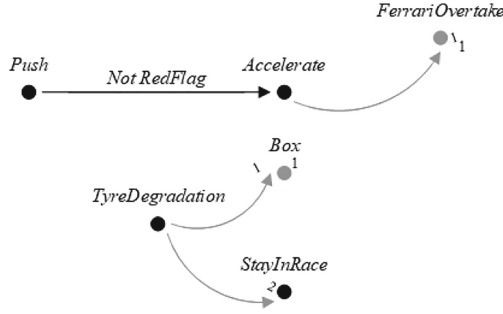


Fig. 2. Two VG-FVS rule examples

3 FVS Formal Definitions

Big journeys begin with small steps. Our final goal and main objective is to provide a formal proof that *VG-FVS*, the declarative and behavioral specification language oriented to diminish the state explosion problem, is sound and correct. In order to do so we must first provide formal definitions of VG-FVS building blocks. That is, we need to understand the theory behinds our graphical scenarios, the motion given by algebraic morphisms, the notion of what we call *situations*, which formally define a photo of our scheme in execution, how rules of our language are logically satisfied or not, and how our scenarios are translated into Büchi automata. This latter aspect involves the definition of FVS states, how the successors of states are built and how transitions between states are constructed and labeled. These formal definitions will be the first step of the ravishing journey to demonstrate that our approach is sound and complete.

Our graphical scenarios are defined first.

Definition 1 (VG-FVS Scenario). An *VG-FVS* scenario is a tuple $\langle \Sigma, P, \ell, \equiv, \neq, <, \gamma \rangle$, 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: $\ell : P \rightarrow \mathcal{PL}(\Sigma)$ is a function that labels each point with a given formula;

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

S5: $\neq \subseteq P \times P$ is an asymmetric relation among points;

S6: $\leq \subseteq (P \uplus \{0\} \times P \uplus \{\infty\}) \setminus \{\langle 0, \infty \rangle\}$ is a precedence relation between points, where 0 and ∞ represent the beginning and the end of execution, respectively;

S7: $\gamma : (\neq \cup <) \rightarrow \mathcal{PL}(\Sigma)$ 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.

Now we provide the notion of *morphisms*. A morphism can be seen as a mathematical function that relates two scenarios, indicating how an scenario can

be embedded into another one, uniting points and labeling functions. This operation can be seen as a traditional “specialization” operation [8]. points exhibiting how a scenario “specializes” another one [8].

Definition 2 (Morphism). *Given two scenarios $\mathcal{S}_1, \mathcal{S}_2$ (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 $\mathcal{S}_1$ to $\mathcal{S}_2$ (denoted $f : \mathcal{S}_1 \rightarrow \mathcal{S}_2$) 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 $p, q \in P_1$.

In particular, morphisms are important since they allow to properly define the concept of rules. Intuitively, a rule consist of an scenario playing the role of the antecedent, and one or more scenarios playing the role of the consequents. However, in order to build a rule morphisms must be provided to denote how the antecedent is connected to each consequents. This semantics “bureaucracy” is naturally hidden in our graphical flavour, but it is crucial to formally define our language. Rules formal definition is presented next.

Definition 3 (VG-FVS Rule). *Given a scenario $\mathcal{S}_0$ (antecedent) and an indexed set of scenarios and morphisms from the antecedent $f_1 : \mathcal{S}_0 \rightarrow \mathcal{S}_1$, $f_2 : \mathcal{S}_0 \rightarrow \mathcal{S}_2$, ..., $f_k : \mathcal{S}_0 \rightarrow \mathcal{S}_k$ (consequents), we call $R = \langle \mathcal{S}_0, \{(S_i, f_i)\}_{i=1\dots k} \rangle$ an FVS Rule.*

We now need to relate morphisms and events. We rely on the classical definition of traces as a linear sequence of events, and those traces accepted by the set of FVS rules defines the language of the system. The formal definitions are given below.

Definition 4 (Traces). *A trace t is seen as the sequence $\{t_0, t_1, \dots, t_k\}$.*

Definition 5 (VG-FVS Semantics). *Given a rule R and a trace $t \{t_0, t_1, \dots, t_k\}$, if $\forall t_i, 0 \leq i \leq k, t_i \in t \exists$ a morphism m $t_i \xrightarrow{m_i} t_{i+1}$ then $t \models R$.*

3.1 Tableau Algorithm: Translating FVS Scenarios into Büchi Automata

The next step in our picturesque formal journey is to establish how VG-FVS graphical scenarios are translated into Büchi Automata. This is a necessary step since it enables the possibility to combine our tool with model checkers and other verification formal tools such as MaRDiGraS [4,5] as it is currently done in this work.

The translation is given by a tableau procedure, which is fully detailed [2]. Nonetheless, we introduce in this section its main components and behavior so that this paper can be read in a stand-alone manner.

The tableau heavily relies on a semantic construction that we denominate as **situation**. In few words, a situation symbolizes all the possible combinations showing how a given sequence of events or trace can be embedded into a given rule. That is, a situation captures all partial possible matches from the antecedent to the consequent of a given rule. Consider the following example in Fig. 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).

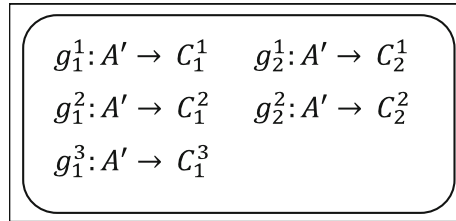


Fig. 3. A situation example

In what follows we formally define the notion of **situation**.

Definition 6 (Situation). *Given*

- $f_j : A \rightarrow C_j$, $j \in [1..n]$, standing for a rule with antecedent A and consequents $C_1, C_2, \dots, C_n$.
- $e : A' \rightarrow A$, a morphism standing for the partial matching A' of the antecedent A
- $e_j^m : C_j^m \rightarrow C_j$, $m \in [1..k_j]$, morphisms representing partial matches for each consequent C_j

a **situation** η is an indexed set of morphisms such that index η_i returns all the partial matches for consequent C_i . More formally, a **situation** η is an indexed set of morphisms $g_j^i : A' \rightarrow C_j^i$, $i \in [1..k_j]$ such that the following conditions holds:

- Condition 1: A' is a configuration for A and $e_j^i(C_j^i)$ is a configuration for $C_i \setminus \{f_j(e(A'))\}$. That is, A' and each $e_j^i(C_j^i)$ represent partial instantiations of the rule's antecedent and consequents.
- Condition 2: $f_i \circ e = e_j^i \circ g_j^i$. The morphisms g_j^i are consistent with the partial matches of the antecedent and consequents of the rule.
- Condition 3: $e_j^i(C_j^i) \cap f_j(A) = f_j(e(A'))$. The partial matches of the antecedent and consequents are complete. That is, no point has been “missed” by any of the partial matches.

Given the occurrence of an event, all the possible advances of the antecedent and consequents is reflected in the current situations. What is more, the states of the built automaton will consist of a set of situations, representing the partial advances up to that point for a given trace or sequence of events. We now define the notion of FVS state, which constitute the state of the output automaton of the tableau algorithm.

Definition 7 (FVS State). *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 $(\Upsilon_R \times \text{bool} \times \mathcal{PL}(\Sigma_R))$, 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 $\text{situations}(S)$, 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. Initially, this element is initialized with $\{ \text{true} \}$.*

The initial state S^0 is given by the empty configuration: $\Upsilon(i) = \{\emptyset\} \forall i \in [0..k]$. Starting from the initial state $(\langle \emptyset, \text{false}, \text{true} \rangle)$, the tableau procedure will aim to incrementally “construct” the pattern as events, represented by minterms, occurs. The algorithm takes into account the decision to match or not a given event by producing two different *situations*, one where the new element is added (matched), and the other one remains unchanged (not matched). Recall that we simply define states as a set of situations, denoting all the possible matches from the trace to the rule.

The heart of the algorithm is those lines where the successor of a state is obtained. In what follows we simply delineate how this is achieved. This part of the tableau is called *Succ*. It receives as input a state S and a minterm representing the actual trace, and the output is the successors of S . This *Succ* routine is invoked until all the paths are covered, obtaining a fully well formed automaton, with a initial state, accepting states, accepting conditions, and trap states (error states). This algorithm rely on two auxiliary functions: *advanceAntecedent* and *advanceConsequents*. A high-level description of both of them is given in Algorithm 1 and Algorithm 2 respectively.

```

1 Algorithm advanceAntecedent( $\eta : \text{situation}, m : \text{minterm}$ ):  $\eta$  situation ;
2 return  $\{z_j^i : A' \cup \mathcal{F}_j^i \rightarrow C_j^i \cup \{[f(\mathcal{F}_j^i)]_{\equiv}\}\}$  such that:  $g_j^i : A' \rightarrow C_j^i \in \eta \wedge$ 
    $A' \xrightarrow{m} A' \cup \mathcal{F}_j^i \wedge C_j^i \xrightarrow{m} C_j^i \cup \{[f(\mathcal{F}_j^i)]_{\equiv}\} \wedge g_j^i(A') = z_j^i(A') \wedge z_j^i(\mathcal{F}) = f(\mathcal{F}) \wedge$ 
    $m \in \mathcal{R}_{[f(\mathcal{F}_j^i)]_{\equiv}}(C_j^i)\} \cup \{g_j^i : A' \rightarrow C_j^i \text{ such that } m \in \mathcal{R}(A')\}$ 

```

Algorithm 1: Procedure *advanceAntecedent*

```

1 Algorithm advanceConsequent( $\eta : \text{situation}, m : \text{minterm}$ ):  $\eta$  situation;
2 return  $\{z_j^i : A' \rightarrow C_j^i \cup \mathcal{F}_j^i \text{ such that: } g_j^i : A' \rightarrow C_j^i \in \eta \wedge C_j^i \xrightarrow{m} C_j^i \cup \mathcal{F}_j^i \wedge$ 
    $g_j^i(A') = z_j^i(A') \wedge m \in \mathcal{R}_{\mathcal{F}_j^i}(C_j^i)\} \cup \{g_i : A' \rightarrow C_j^i \text{ such that } m \in \mathcal{R}(C_j^i)\}$ 

```

Algorithm 2: Procedure advanceConsequent

These two auxiliary functions are invoked from the main algorithm *Succ*, which can be depicted in Algorithm 3. Lines 5 and 6 of 3 invoke the *advanceAntecedent* and *advanceConsequents* routines. Line 7 deals with trap situations. A very common construction in the automata world is the error states: those states representing an unusual or error condition. In those cases, the system must remain in these states. In our case, a trap situation is given when an antecedent is found, but none of the consequents can be matched. Therefore, the rule can not be satisfied.

The next two lines (Lines 8 and 9 from Algorithm 3) raise a flag in case a consequent has been matched. This is, *goalmatched*[*i*] = *true* if and only if *consequent* *C_i* is matched. Accepting states are tagged in line 10. An accepting state contains a matched consequent and it is not a trap state. This concludes all the possible successors for a given state. The resulting set of states is returned in Line 11. If more than one consequent is matched, then all possible combinations of situations are considered. Specifically in the algorithm, non-determinism is introduced by the condition $GM \rightarrow \text{goalMatched}$ (line 11). When the latter holds the *true* value two states are added, one with $GM = \text{true}$ and the other with $GM = \text{false}$. In the first case a consequent is matched and obligations of the consequent are added as future restrictions of the trace. The second case represents the situation where the consequent achievement is omitted.

```

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

```

Algorithm 3: Successor states

4 VG-FVS: FVS Plus 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 (Sect. 4.1) and latter on, how them are extended to cope with FVS definitions (Sect. 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 [10,25].

Map Reduce constitute a classical way of computation inherited from the functional paradigm, since a couple of operations are performed as mathematical function's composition. It captures a repetitive behavior, where the same group of activities is performed over a possible very large collection of items to generate a bunch of partial results. These results are, in turn, aggregated to compute the final outcome. The map function addresses the repetitive activities whereas the reduce functions deals with the aggregation aspect. In MaRDiGraS the map function is in charge of computing the new states and the reduce function computes equivalence and the inclusion relationship between states. As in all major frameworks, most of the technical aspects of this distributed oriented architecture are transparent to the final user. This is a notable feature of the MaRDiGraS tool. The final users must only concentrate in defining notions as states, edges, the transition relationship and functions to calculate equivalence and inclusion of states. Once these aspects are defined, the framework apply the mad reduce philosophy in a transparent way.

The main formalism employed to build all the definitions are the usual labeled state transition system. Labeled transitions systems ease the application of formal techniques to address the state explosion problem. This is achieved by constructing abstraction of the actual (concrete) states so that a group of concrete states (which are somehow related) can be mapped into the same abstract state. 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]. The user must define the concept of state, how to build abstract states, how to link concrete to abstract states, how to obtain a successor of a state, and the notion of accepting and reachable states. The design implementation follows the Open Close principle. This application model can be easily extended to any other formalism. For example, in [4] it is shown how several kinds of Petri Nets can be adapted and used in the MaRDiGraS tool. By extending the application model definitions in order to match the desired formalism the tool will run under the expected modified constructions.

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. Figure 4 represents the basic functioning of the MaRDiGraS tool.

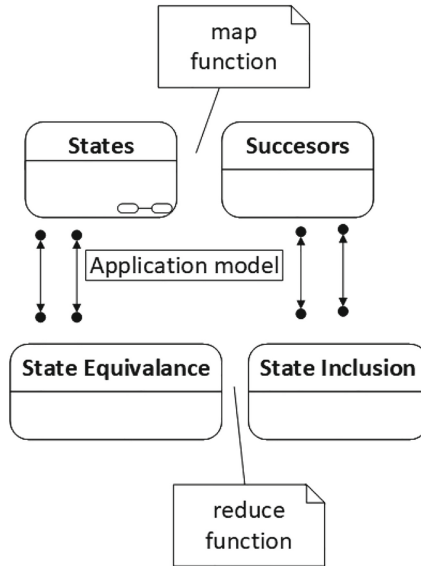


Fig. 4. MaRDiGraS architecture

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.
- addLabel: a simply method to initialize an edge between two states.

4.2 VG-FVS: Combining FVS and MaRDiGraS

Given the formal definitions characterizing our FVS language given in Sect. 3 the combination of FVS with MaRDiGraS is described as follows.

We represent MaRDiGraS **state class** employing the concept of Büchi state defined when translating our graphical scenarios into Büchi automata. Definition 7 in Sect. 3 defines the concept of state and Algorithm 3 describes how the automaton is built. Since both notions of states (MaRDiGraS and ours) rely on automata concepts, this part of the integration into MaRDiGraS was pretty straightforward.

The next challenge was to give a proper implementation of the *Successors* method. Again, this was very smoothly done since we directly applied the *Succ* method detailed in Algorithm 3.

Regarding the **Edge** class, we simply take advantage of the morphism operation. A morphism can be seen as an edge between *situations*, and our states are no more than a set of *situations*. The labelling function which assigns occurring events into points in a scenario can be straightforwardly implemented as the *addLabel* MaRDiGraS method.

To complete the integration with the MaRDiGraS tool two additional functions must be defined: the **IdentifyRelationship** function and the **GetFeatures** function.

The **IdentifyRelationship** function deals with the necessary abstraction function in states, which is a crucial aspect when dealing with the state explosion problem. This function correlates abstract and concrete states. To achieve this goal, any pair of states of interest must be analyzed to determine whether one state includes the other one, and if the two of them can be logically connected. We implemented this function relying on the notion of FVS *situations* (see Definition 6 in Sect. 3). Given two situations S_1 and S_2 this function must return one of these possible outputs:

- NONE: if S_1 and S_2 do not match.
- EQUALS: if S_1 and S_2 are equivalent.
- INCLUDED: if S_1 includes S_2 but they are not equivalent.
- INCLUDES: if S_2 includes S_1 but they are not equivalent.

The main behavior of this procedure can be seen in Algorithm 4. Line 2 verifies if every morphism in S_1 is included in S_2 where Line 3 does the dual work: to verify if every morphism in S_2 is included in S_1 . These results are stored in two boolean variables: *Condition1* and *Condition2*. Finally, lines 4 to 7 in Algorithm 4 calculate the demanded output.

The last item to be implemented to fully integrate FVS with MaRDiGraS is the **GetFeatures** function. This function analyzes equivalence between states. In our case this is reflected with the following behavior: given two states, S_1 and S_2 , the function must calculate all the possible combinations of partial matches obtained up to that states from the antecedent to the consequents simultaneously. This output is returned as a set of FVS *situations*. Algorithm 5 exhibits our implementation for the **GetFeatures** function. In Line 2 it is verified that

```

1 Algorithm IdentifyRelationship( $S_1 : \text{situation}, S_2 : \text{situation}$ );
2 condition1 = if  $\forall g_j^i : A' \rightarrow C_j^i \in S_1 \exists f_j^i : A' \rightarrow C_j^i \in S_2$  ;
3 condition2 = if  $\forall f_j^i : A' \rightarrow C_j^i \in S_2 \exists g_j^i : A' \rightarrow C_j^i \in S_1$  ;
4 if condition1 and condition2 then return EQUALS ;
5 if !condition1 and !condition2 then return NONE ;
6 if condition1 and !condition2 then return INCLUDED ;
7 if !condition1 and condition2 then return INCLUDES ;

```

Algorithm 4: Procedure IdentifyRelationship

both consequents and antecedents are matched simultaneously since morphisms must exist in the set of situations of both states. Finally, all the morphisms satisfying these conditions are returned.

```

1 Algorithm GetFeatures( $S_1 : \text{State}, S_2 : \text{State}$ );
2 return { all  $\eta$  such that exists  $g_j^i : A' \rightarrow C_j^i \in S_1 \wedge g_j^i : A' \rightarrow C_j^i \in S_2$  };

```

Algorithm 5: Procedure GetFeatures

This concludes the formal definitions needed to embed FVS mechanisms into the MaRDiGraS tool. Table 1 resumes the integration between FVS and MaRDiGraS. With the same objective, Fig. 5 exhibits this tools' combination in a graphical manner.

Table 1. FVS-MaRDiGraS Integration

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

5 VG-FVS: Sound and Complete

In this section we provide the formal proofs needed to assure that the integration between FVS and MaRDiGraS is sound and complete. FVS itself has been proved sound and correct [2]. Similarly, this has been done for the MaRDiGraS tool [4]. This indicates that we only need to prove that the FVS integration into MaRDiGraS is sound and complete. The rest of this section addresses this issue.

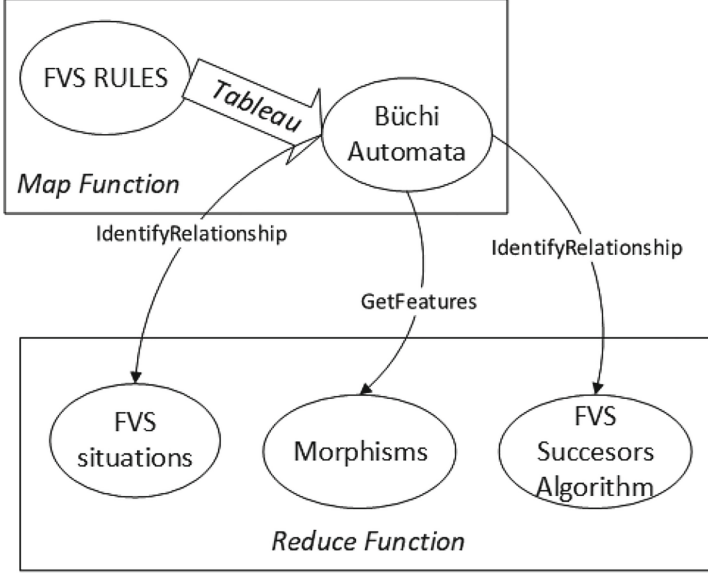


Fig. 5. VG-FVS and MaRDiGraS

Since FVS is sound and complete we know that the following condition holds: given a rule R , and a trace scenario $\mathcal{S}_\sigma$ $\mathcal{S}_\sigma \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B (i.e., $\mathcal{S}_\sigma \in \mathcal{L}(B)$), where B is the Büchi automaton built by the tableau for rule R . In order to prove that VG-FVS is sound and complete we need to verify that this condition holds for all the items shown in Table 1: *state*, *successors*, *addLabel*, *Edge* and functions *IdentifyRelationship* and *GetFeatures*. For the *state*, *successors*, *addLabel* and *Edge* constructions there is nothing to prove since they are implemented directly as FVS formal definitions, so the condition trivially holds. Therefore, to complete the sound and complete proof we need to prove the condition for the functions *IdentifyRelationship* and *GetFeatures*. This is addressed in the next two subsections.

5.1 IdentifyRelationship Function is Sound and Complete

We know that given a rule R , and a trace scenario $\mathcal{S}_\sigma$ $\mathcal{S}_\sigma \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B (i.e., $\mathcal{S}_\sigma \in \mathcal{L}(B)$), where B is the Büchi automaton built by the tableau for rule R . In particular, for each situation $\eta \in \text{States}(R)$ given a trace $t = \{t_0, t_1, \dots, t_k\} \exists$ morphism m $t_i \xrightarrow{m_i} t_{i+1}$, $0 \leq i \leq k$.

The *IdentifyRelationship* functions returns one of four possible results: EQUALS, NONE, INCLUDES or INCLUDED. We need to prove that the mentioned conditions holds in every case.

If the function returns NONE, we know that $\forall g_j^i : A' \rightarrow C_j^i \in S_1 \nexists f_j^i : A' \rightarrow C_j^i \in S_2$ and also that $\forall f_j^i : A' \rightarrow C_j^i \in S_2 \nexists g_j^i : A' \rightarrow C_j^i \in S_1$. This implies that

trace t will not arrive to an accepting state and will not satisfy rule R . Since we are only interested in accepting states to prove that $\mathcal{S}_\sigma \models R$ we can safely discard this case for the analysis.

If the function returns EQUALS, we know that $\forall g_j^i : A' \rightarrow C_j^i \in S_1 \exists f_j^i : A' \rightarrow C_j^i \in S_2$ and also that $\forall f_j^i : A' \rightarrow C_j^i \in S_2 \exists g_j^i : A' \rightarrow C_j^i \in S_1$. This implies that accepting states will be visited by any trace $t \in \mathcal{L}(B)$. Then we can conclude that $\mathcal{S}_\sigma \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B .

If the function returns INCLUDED we know that state $S_1 \subset S_2$. This indicates that for every trace $t \in S_1$, $t \in S_2$. Given this, all the accepting states in S_1 are included in the accepting states of S_2 . Therefore, if $S_1 \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B . then it must be the case that $S_2 \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B . Since the conditions holds for both states it is fully satisfied.

If the function returns INCLUDES we know that state $S_2 \subset S_1$. This indicates that for every trace $t \in S_2$, $t \in S_1$. Given this, all the accepting states in S_2 are included in the accepting states of S_1 . Therefore, if $S_2 \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B . then it must be the case that $S_1 \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B . Since the conditions holds for both states it is fully satisfied.

Since we can demonstrate that given a rule R , and a trace scenario $\mathcal{S}_\sigma$ $\mathcal{S}_\sigma \models R$ if and only if $\mathcal{S}_\sigma$ seen as a trace is accepted by B holds in every case of the function, it can be established that the *IdentifyRelationship* function is sound and complete. $\square$.

5.2 GetFeatures Function Is Sound and Complete

The *GetFeatures* function, given two states S_1 and S_2 , returns a set of situations S_3 containing all the possible matches from the antecedent to the consequents in both states. Since both S_1 and S_2 are states of the rule R , we know that for every trace $t \in S_1 \cup S_2$, $t \in \mathcal{L}(B)$, where B is the Büchi automaton built by the tableau for rule R . Since S_3 contains all situations η such that morphisms $g_j^i : A' \rightarrow C_j^i \in S_1$ and $g_j^i : A' \rightarrow C_j^i \in S_2$ we can establish that for every trace $t = \{t_0, t_1, \dots, t_k\}$, $t \in S_1 \cup S_2$, $\exists$ morphism m such that $t_i \xrightarrow{m_i} t_{i+1}$, $0 \leq i \leq k$. This implies that trace t will always lead to an accepting state concluding that $t \in \mathcal{L}(B)$, which was the result we needed to prove. $\square$.

5.3 Soundness and Completeness: Final Remarks

Since all the elements composing the integration between VG-FVS and MaRDiGraS has been proved to be sound and complete we can affirm that VG-FVS is a sound and complete formalism. $\square$.

6 Related and Future Work

Work in [9] brilliantly summarizes modern tools and techniques addressing the state explosion problem in formal methods. The common element behind all the

described approaches is abstraction. Abstraction is intensively applied in all the described and proposed methods. Given this, we would like to further extend VG-FVS in this direction.

Interesting approaches like [23] or [7] somehow relates to our work. [23] proposes a complete and sound framework to specify behavioral models of spatially-distributed systems. Requirements are expressed in Spatio-Temporal Reach and Escape Logic (STREL), a domain specific logic formalism. We would definitely like to investigate the possibility to amalgamate VG-FVS with this technique, extending even further our horizons. This work also provides formal proofs of their correctness. On the other hand, [7] focuses on formal validation of distributed systems in the cloud introducing a framework denominated *Maude*. Our tool could benefit introducing some of the architectural patterns employed in *Maude*.

7 Conclusions

In this work we provide formal proofs to guarantee the soundness and correctness of VG-FVS, the framework that combines the super powers of FVS and MaRDiGraS. Demonstrating that a formalism is sound, complete and correct is a key feature when dealing with formal verification techniques. In this way, we can state that VG-FVS can be seen as a solid alternative to verify behavior in Data Science Software systems. Besides featuring powerful results to deal with the state explosion problem a robust theoretical aspect is added to our approach to provide further assurance.

References

1. Asteasuain, F.: A parallel tableau algorithm for big data verification. In: CACIC, pp. 275–287 (2023). ISBN 978-987-9285-51-0
2. Asteasuain, F., Braberman, V.: Declaratively building behavior by means of scenario clauses. *Requirements Eng.* **22**(2), 239–274 (2017). <https://doi.org/10.1007/s00766-015-0242-2>
3. Asteasuain, F., Calonge, F., Dubinsky, M., Gamboa, P.: Open and branching behavioral synthesis with scenario clauses. *CLEI E-JOURNAL* **24**(3), 1–20 (2021)
4. Bellettini, C., Camilli, M., Capra, L., Monga, M.: Mardigras: Simplified building of reachability graphs on large clusters. In: RP workshop. pp. 83–95 (2013)
5. Bellettini, C., Camilli, M., Capra, L., Monga, M.: Distributed CTL model checking using mapreduce: theory and practice. *CCPE* **28**(11), 3025–3041 (2016)
6. Blume, M., McAllester, D.: Sound and complete models of contracts. *J. Funct. Program.* **16**(4–5), 375–414 (2006)
7. Bobba, R., et al.: Survivability: design, formal modeling, and validation of cloud storage systems using Maude. In: *Assured Cloud Computing*, pp. 10–48 (2018)
8. Braberman, Víctor., Garbervestky, Diego, Kicillof, Nicolás, Monteverde, Daniel, Olivero, Alfredo: Speeding up model checking of timed-models by combining scenario specialization and live component analysis. In: Ouaknine, Joël., Vaandrager, Frits W.. (eds.) *FORMATS 2009. LNCS*, vol. 5813, pp. 58–72. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_7

9. Camilli, M., et al.: Coping with the state explosion problem in formal methods: advanced abstraction techniques and big data approaches (2015)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
11. Ding, J., Zhang, D., Hu, X.H.: A framework for ensuring the quality of a big data service. In: 2016 SCC, pp. 82–89. IEEE (2016)
12. Hummel, O., Eichelberger, H., Giloj, A., Werle, D., Schmid, K.: A collection of software engineering challenges for big data system development. In: SEAA, pp. 362–369. IEEE (2018)
13. Kim, M., Zimmermann, T., DeLine, R., Begel, A.: Data scientists in software teams: State of the art and challenges. *IEEE Trans. Softw. Eng.* **44**(11), 1024–1038 (2017)
14. Klein, A.M., Kölln, K., Deutschländer, J., Rauschenberger, M.: Design and evaluation of voice user interfaces: what should one consider? In: Salvendy, G., Wei, J. (eds.) *International Conference on Human-Computer Interaction*, pp. 167–190. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35921-7_12
15. Kumar, V.D., Alencar, P.: Software engineering for big data projects: domains, methodologies and gaps. In: *IEEEBIGDATA*, pp. 2886–2895. IEEE (2016)
16. Laigner, R., Kalinowski, M., Lifschitz, S., Monteiro, R.S., de Oliveira, D.: A systematic mapping of software engineering approaches to develop big data systems. In: SEAA, pp. 446–453. IEEE (2018)
17. Martínez-Fernández, S., et al.: Software engineering for AI-based systems: a survey. *TOSEM* **31**(2), 1–59 (2022)
18. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. *IEEE Trans. Software Eng.* **47**(10), 2208–2224 (2019)
19. Salman, H.M., Wan Ahmad, W.F., Sulaiman, S.: A design framework of a smart-phone user interface for elderly users. *Univ. Access Inf. Soc.* **22**(2), 489–509 (2023)
20. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic (TOCL)* **15**(4), 1–37 (2014)
21. Soui, M., Haddad, Z.: Deep learning-based model using densnet201 for mobile user interface evaluation. *Int. J. Hum.-Comput. Interact.* **39**(9), 1981–1994 (2023)
22. Sri, P.A., Anusha, M.: Big data-survey. *Indonesian J. Electr. Eng. Inform. (IJEI)* **4**(1), 74–80 (2016)
23. Tsigkanos, C., Nenzi, L., Loret, M., Garriga, M., Dustdar, S., Ghezzi, C.: Inferring analyzable models from trajectories of spatially-distributed internet of things. In: SEAMS, pp. 100–106. IEEE (2019)
24. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) *LASER 2011. LNCS*, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_1
25. Zhang, J., Lin, M.: A comprehensive bibliometric analysis of Apache Hadoop from 2008 to 2020. *IJICC* **16**(1), 99–120 (2023)
26. Zhao, L., Wang, X., Shu, X., Zhang, N.: A sound and complete proof system for a unified temporal logic. *Theoret. Comput. Sci.* **838**, 25–44 (2020)