ISBN 978 -987-633-574-4

An expressive and enriched specification language to synthesize behavior in BIG DATA systems

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

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

Abstract. In this work we extend our behavioral specification and controller synthesis framework FVS to deal with BIG DATA requirements. For one side, we enriched FVS expressive power by exhibiting how our language can handle fluents and partial specifications. For the other side, we combined FVS with a parallel model checker in order to automatically obtain a controller given the behavior specification. In this way, FVS can be presented as an attractive tool to formally verify and synthesize behavior for BIG DATA systems. Our approach is compared to other well known parallel tool analyzing a complex big data system.

Keywords: Formal Verification, BIG DATA, Parallel Model Checkers

1 Introduction

Nowadays BIG DATA systems are surprisingly present in every day life. The Software Engineering community has been trying to adapt, to extend or to create tools, techniques and methods to deal with the new conditions and requirements that these systems expose [9, 24, 15, 30, 21, 31, 25]. For example, data warehouse approaches have emerged since traditional ways to structure data such as relational data bases are no that useful in this domain.

According to some approaches, one of the software engineering areas that more urgently need attention and contributions is formal verification [24, 17]. The challenges to be addressed are inherent to BIG DATA systems: handling and reasoning about tons of unstructured, informal and heterogeneous data and information. To deal with this kind of context flexible and expressive formalism are needed to express, validate and reason about the expected behavior of the systems [25, 24, 17]. In addition, more efficient mechanisms are needed since performance is a crucial feature to achieve when dealing with BIG DATA systems. In this sense, most of the formal verification approaches to big data try to obtain better execution times by providing parallel version of the algorithms involved. However, the flexibility and expressive power of the formalisms has been somehow neglected.

ISBN 978 -987-633-574-4

One of the most applied techniques in traditional formal verification is controller synthesis [18, 10]. In these approaches, a controller is automatically build upon the expected behavior of the system and the environment it interacts. Usually, the controller takes the form of an automaton which decides which actions to take based on the received information (mostly provided by external sensors). The controller is built using game theory concepts, obtaining a winning strategy that takes the system to an accepting state no matter which actions the environment chooses [10]. Regarding expressive power, fluents [20] and partial specifications are powerful formalisms to deal with unstructured behavior. A fluent allows to model behavior 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. Partial specifications introduce the possibility of specifying optional behavior, a feature that is highly appreciated when dealing with requirements in early stages since conditions and behavior itself are not clearly determined. MTS (Modal Transition Systems) [26] is one of the most widely known formalisms. Since controller synthesis deal with exponential algorithms, some parallel and distributed extensions have emerged. One of them is the parallel version of the MTSA (Modal Transition System Analyzer) model checker [12], which is available at: https://mtsa.dc.uba.ar/. MTSA allows to efficiently obtain controllers in different domains [28, 29]. However, it has not been explored in the BIG DATA domain.

In previous work we took an initial step to adapt our behavioral specification language FVS (FeatherWeight Visual Scenarios) to deal with BIG DATA requirements [2]. Specifically, we have parallelized the way our specification is build, introducing a parallel algorithm to translate FVS graphical scenarios into Büchi automata. In this work we continued this path by combining FVS specifications with the MTSA parallel model checker. *In this way, we end up with a flexible and extremely rich expressive power formalism to specify behavior and perform controller synthesis in BIG DATA systems in a efficient way.* In order to interact with the MTSA tool we enriched FVS in two orthogonal aspects, illustrating how FVS can express fluents and partial specifications, and providing a combination to a parallel model checker. As a case of study we analyzed a big data system provided in the literate [9] and validate our approach by comparing execution times with another parallel technique [9].

The rest of this work is structured as follows. Section 2 briefly presents FVS and explains how a controller can be obtained. Sections 3 and 4 show how FVS can denote fluents and partial specifications. Section 5 exhibits the case of study and the interaction with the MTSA model checker while Section 6 discusses the obtained results. Finally, Sections 7 and 8 analyze some related and future work and the conclusions of this research.

ISBN 978 -987-633-574-4

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.

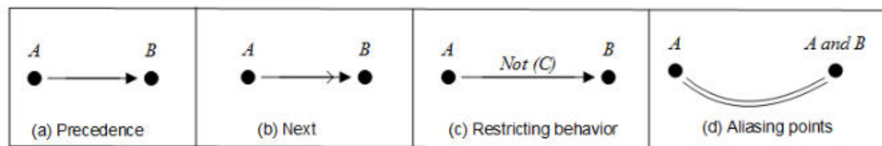

Fig. 1. Basic Elements in FVS

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

2.1 FVS and Ghosts Events

FVS can denote high level behavior. This is due to the introduction of abstraction, which is incorporated in our notation by introducing a new type of events.

ISBN 978 -987-633-574-4

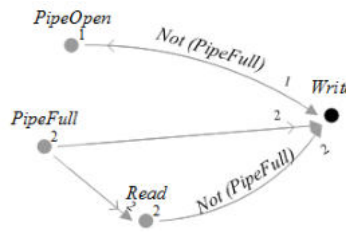

Fig. 2. An FVS rule example

By using these events the user can abstract behavior and reason about events that are not present in the system traces, but actually represent a higher level of abstraction. We call these special events as "ghost" events, in contrast with "actual" events, the set of events present in the system's specification. In order to verify that a rule containing ghosts events satisfies a certain trace of the system (which only contains actual events) there is a internal process based on morphisms that discards ghost events based on a classic process of existential elimination [1].

2.2 Behavioral Synthesis in FVS

FVS specifications can be used to automatically obtain a controller employing a classical behavioral synthesis procedure. We now briefly explains how this is achieved while the complete description is available in [3]. Using the tableau algorithm detailed in [1] FVS scenarios are translated into Büchi automata. Then, if the obtained automata is deterministic, then we obtain a controller using a technique [27] based on the specification patterns [19] and the GR(1) subset of LTL. If the automaton is non deterministic, we can obtain a controller anyway. Employing an advanced tool for manipulating diverse kinds of automata named GOAL [32] we translate these automata into Deterministic Rabin automata. Since synthesis algorithms are also incorporated into the GOAL tool using Rabin automata as input, a controller can be obtained. In this work, we add a new way to obtain a controller, combining FVS with the MTSA model checker, as shown in the remaining sections.

3 Fluents and FVS

Fluents [20] constitute a powerful variant of LTL. A fluent allows to model behavior 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. In [20] a simple example is given to illustrate how fluents works. Suppose we are validating a new decentralized system for organizing television control software. The property to verify is the following: *If the TV tuner is tuning, then the screen*

ISBN 978 -987-633-574-4

must be blanked, and the available events are: *blank* (blanks the screen), *unblank* (displays the new channel signal), *tune* (the tuner starts tuning into the new channel) and *endtune* (the tuner finishes). The tunning interval is defined as beginning with the *tune* event and ending with the *endtune* event. Similarly, the blanking interval starts with the *blank* event and finishes with the *unblank* event. Once these intervals are defined then the property to be checked can be simply formulated as $\Box$ (Tunning $\Rightarrow$ *Blanked*).

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. Rules in Figure 3 specify the TV control system property mentioned before, using two ghosts events, namely *Tunning* and *Blanked*. The rule in the top of the picture defines the *Tunning* event, where the rule in the middle does the same for the *Blanked* event. Finally, the rule in the bottom models the desired property.

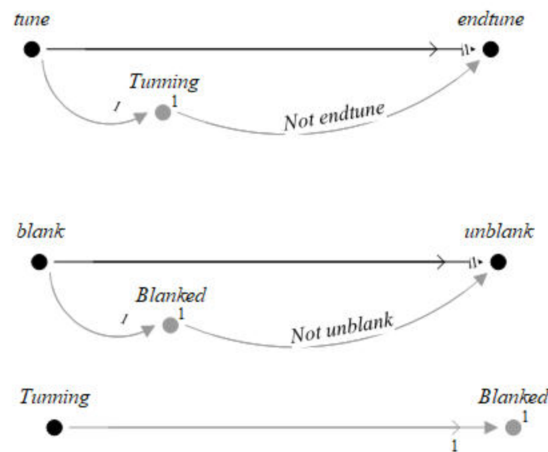

Fig. 3. Fluents in FVS

4 Partial Specifications and FVS

Partial Specifications are a crucial tool to model and shape early behavior of computer systems. They aim to capture early interactions between the elements involved in a stage where the requirements are not yet thoroughly defined. Transitions are divided into *required* and *maybe* categories, where the latter introduce partial or optional behavior. In successive versions of the system *maybe* transitions are either discarded or turned into required behavior. This process

ISBN 978 -987-633-574-4

in known as *refinement*. Perhaps the most known formalism addressing Partial Specifications is Modal Transitions Systems (MTS) [26].

In FVS partial specifications are inherently included since it features optional behavior by employing multiples consequents in its rules. The semantic of the system is given by those traces satisfying all the rules, and a rule with two or more consequents is satisfied if at leat one of them is found, or all of them, if that is the case. 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).

As an example, suppose a new requirement arises in the TV control system previously described. A new publicity system might be added to the main system. In few words, when the screen is blanked two events could happen: either a publicity is shown or the blanking process ends normally. At this stage, the publicity feature is handled as an optional behavior. In FVS, this is achieved by adding a new consequent, as shown in figure Figure 4.

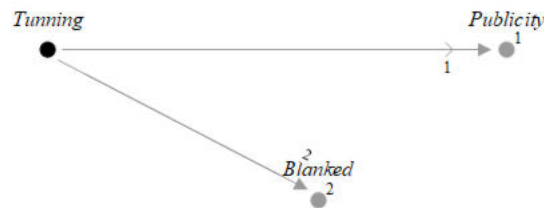

Fig. 4. Describing partial behavior in FVS

5 Case Study: Dekker Algorithm

The case of study we analyzed is introduced on [9] based on the benchmarks in [23]. This model represents a variant of Dekker's mutual exclusion algorithm. As described in [23], the main functioning of this algorithm is the following. Each process has three states, *p0*, *p1*, and *p3*. *p0* is initial. From there, the process executes try and raises its flag, reaching *p1*. In *p1*, if at least one of the other process has a high flag, it withdraws its intent and goes back to *p0*. In *p1*, it enters the critical section if all other process flag is zero. From *p3,* the process can only exit the critical section. The rules in Figure 5 show some of the FVS specification fulfilling the algorithm's requirements. The rules considered actions for one process. The complete specification is obtained by composing all the rules for every process involved. We employed several ghosts events like *Flag*, *EnterCritical* and *ExitCritical* and rules with several consequents to handle partial specifications.

ISBN 978 -987-633-574-4

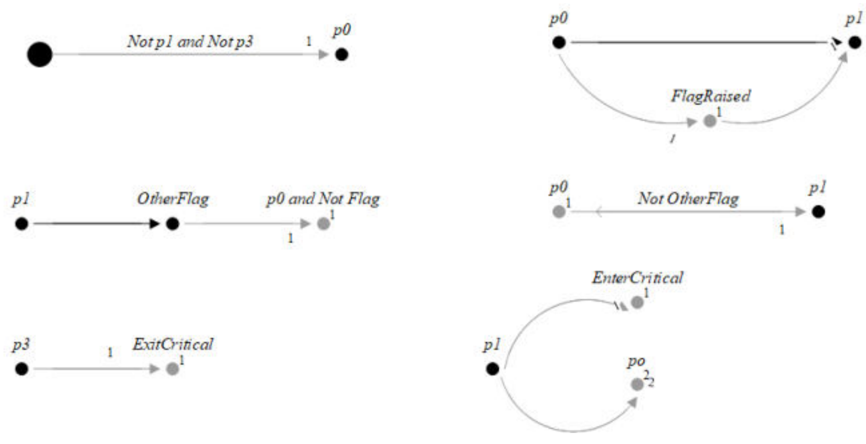

Fig. 5. FVS rules describing the behavior of the Dekker's algorithm

We modeled the complete behavior of the algorithm, and then we obtained a controller using FVS rules as input in the MTSA model checker. Part of the controller is shown in Figure 6.

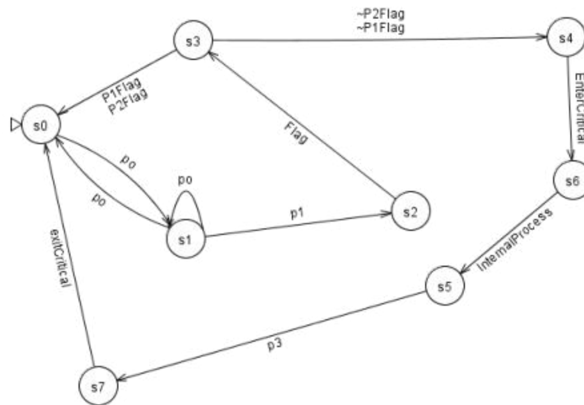

Fig. 6. Part of the behavior of the Dekker's Algorithm Controller

ISBN 978 -987-633-574-4

6 Experimental Results

In this section we describe the results we obtained aiming to measure FVS distributed model checking and controller synthesis performance regarding execution time. The system under analyses consisted of the Dekker algorithm detailed in Section 5. We compared our execution times against the technique in [9], which proposed this case of study. Although they verified the behavior of the algorithm and we obtained a controller instead, the involved tasks and objectives are similar enough to produce valuable results from their performance execution time comparison. We ran our experiments in a Bangho Inspiron5458, with a Dual Core i5-5200U and 8GB RAM memory.

As in [9], we conducted the experiment running the algorithm 10, 15 and 20 processes. Table 1 subsumes the obtained results, where the column Map-Reduce CTL stands for the technique described in [9] and times is denoted in seconds. It can be noted that although our execution times are worst the difference is not a critical value, and it reduces as the complexity of the problem increase. Thus, it can be preliminary observed that FVS provides great flexibility and expressive power to synthesize behavior in complex cases without neglecting performance.

Table 1. Dekker Algorithm Execution Time

Example	Map-Reduce CTL	Parallel FVS
10-Dekker	50 sec	87 sec
15-Dekker	825 sec	965 sec
20-Dekker	11134 sec	11529 sec

7 Related and Future Work

There are several approaches implementing different versions of parallel model checking algorithms for both linear and branching [22, 11, 13, 6, 7, 14] . It would be interesting to compare the FVS-MTSA duo explored in this work with some of the mentioned tools. Similarly, other approaches aim to speed-up the model checking by performing parallel verification of very small units pieces of behavior [16, 5] . For example, these units are called *swarms* in [16].

In [9, 15]] a interesting framework for distributed CTL (computation tree logic) model checker is presented. They introduce a novel architecture employing HADOOP MAPREDUCE as its computational engine. They provide a very solid empiric evaluation with several case of studies employing Amazon Elastic MapReduce [15] and the GRID5000 cloud infrastructure [4]. For generating and building distributed state space exploration they rely on a framework called Mardigras [8]. This framework introduces a general scheme to verify systems, allowing behavior to be specified using logics, Petri Nets and other formalisms. It would be interesting to explore if FVS can be added in this list.

Regarding future work, we would like to deepen our empirical evaluation by introducing more case of studies and also a space comparison besides the execution time. We believe a comparison taking into account, for example, the number of states and transitions of the automata involved in the verifying process can enrich the results analyzed in this work. Similarly, from the theoretical view we would like to provide formal proofs regarding the equivalence with the fluents and partial specification mechanisms.

8 Conclusions

In this work we present a powerful, flexible and highly expressive specification language to denote behavior and perform controller systems in BIG DATA systems. In particular, we show how FVS is able to express behavior in terms of fluents and partial specifications. In order to deal with BIG DATA performance requirements we combine FVS specifications with the parallel model checker MTSA. In this way, a controller can be found using FVS specifications as input. We compared our executions times with other well known parallel approach analyzing a compelling case of study. By looking at the preliminary results obtained so far we can conclude that FVS exhibits great flexibility and expressive power without a significative loss in performance.

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*, 2018.
3. F. Asteasuain, F. Calonge, and M. Dubinsky. Exploring specification pattern based behavioral synthesis with scenario clauses. In *CACIC*, 2018.
4. 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.
5. 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.
6. J. Barnat, L. Brim, M. Češka, and P. Ročkai. Divine: Parallel distributed model checker. In *2010 ninth PDMC*, pages 4–7. IEEE, 2010.
7. A. Bell and B. R. Haverkort. Sequential and distributed model checking of petri nets. *STTT journal*, 7(1):43–60, 2005.
8. 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.
9. C. Bellettini, M. Camilli, L. Capra, and M. Monga. Distributed ctl model checking using mapreduce: theory and practice. *CCPE*, 28(11):3025–3041, 2016.
10. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar. Synthesis of reactive (1) designs. 2011.
11. M. C. Boukala and L. Petrucci. Distributed model-checking and counterexample search for ctl logic. *IJSR 3*, 3(1-2):44–59, 2012.

12. M. V. Brassesco. Síntesis concurrente de controladores para juegos definidos con objetivos de generalized reactivity(1). *Tesis de Licenciatura.*, http://dc.sigedep.exactas.uba.ar/media/academic/grade/thesis/tesis_18.pdf UBA FCEyN Dpto Computacion 2017.
13. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In *FMCAD*, pages 352–366, 2004.
14. L. Brim, K. Yorav, and J. Žídková. Assumption-based distribution of ctl model checking. *STTT*, 7(1):61–73, 2005.
15. M. Camilli. Formal verification problems in a big data world: towards a mighty synergy. In *ICSE*, pages 638–641, 2014.
16. R. DeFrancisco, S. Cho, M. Ferdman, and S. A. Smolka. Swarm model checking on the gpu. *STTT*, 22(5):583–599, 2020.
17. 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.
18. N. DIppolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran*, 22(9), 2013.
19. M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
20. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *European software engineering conference*, pages 257–266, 2003.
21. 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.
22. O. Inverso and C. Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In *PPoPP*, pages 202–216, 2020.
23. F. Kordon, A. Linard, M. Becutti, D. Buchs, L. Fronc, L. M. Hillah, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, et al. Web report on the model checking contest@ petri net 2013. 2013.
24. V. D. Kumar and P. Alencar. Software engineering for big data projects: Domains, methodologies and gaps. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2886–2895. IEEE, 2016.
25. R. Laigner, M. Kalinowski, S. Lifschitz, R. S. Monteiro, and D. de Oliveira. A systematic mapping of software engineering approaches to develop big data systems. In *SEAA*, pages 446–453. IEEE, 2018.
26. K. G. Larsen and B. Thomsen. A modal process logic. *LICS, pages 203210, IEEE.*
27. S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): A case study. *arXiv preprint arXiv:1602.01172*, 2016.
28. L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel. Dynamic update of discrete event controllers. *IEEE Transactions on Software Engineering*, 46(11):1220–1240, 2018.
29. L. Nahabedian, V. Braberman, N. Dippolito, J. Kramer, and S. Uchitel. Dynamic reconfiguration of business processes. In *International Conference on Business Process Management*, pages 35–51. Springer, 2019.
30. C. E. Otero and A. Peter. Research directions for engineering big data analytics software. *IEEE Intelligent Systems*, 30(1):13–19, 2014.
31. P. A. Sri and M. Anusha. Big data-survey. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)*, 4(1):74–80, 2016.
32. 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.