CONAIISI 2018
6to Congreso Nacional de Ingeniería Informática - Sistema de Información

Expressing Early Behavior Specifications with Branching Visual Scenarios

Fernando Asteasuain[1,2], Federico Calonge[1], Federico D'Angiolo[1], Federico Diaz[1], Pablo Gamboa[2]
{fasteasuain,fcalonge,fediaz,fdangiolo}@undav.edu.ar, pablodaniel.gamboa@alumnos.uai.edu.ar
[1]*Universidad Nacional de Avellaneda, España 350, Avellaneda, BsAs.*
[2]*UAI-CAETI*

Abstract

Branching logics enable the software engineer to express interesting type of properties and feature more efficient algorithms for model checking than linear logics. In this work we present an extension of the FVS language (based on a linear representation of systems' execution) in order to contemplate branching properties. The formal semantics of this extension, named Branching FVS, is also introduced in this work. As a case of study we model the behavior of a FLASH memory test chip, a classical hardware verification example. This is a particular domain where branching logics are heavily used to specify the expected behavior of systems.

1. Introduction

Early specification of behavior has been pinpointed by the community as one of the main problems to be addressed to consolidate the transference of software formal validation and verification techniques such as model checking [5] from the academic to the industrial world [11, 10]. In this context, the specification language chosen to describe the expected behavior of the system is a key factor. Most of the approaches rely on temporal logics as the formalism used to specify behavior. However, usability and expressivity of temporal logics have been challenged by several approaches. A plethora of extensions have been proposed [1-4] in order to provide more expressive or more user friendly formalisms. Among them, FVS [4-5] results in an attractive option. FVS is a declarative language based on graphical scenarios and features a flexible and expressive notation with clear and solid language semantics. FVS expressivity is a distinguished characteristic among declarative approaches since it is able to denote Ω-regular properties, being for example, more expressive than LTL (Linear Temporal Logic) [4]. In [5] all the specification patterns [6] were modeled in FVS, and their specification was compared against other notations. The results showed that FVS specification turn out to be more succinct and easier to manipulate and validate. Furthermore, a tool named GTxFVS was developed giving support to all FVS's features [7]. However, FVS only contemplates linear specifications leaving out the possibility to denote branching properties. Branching flavored specifications represents an important way of reasoning when dealing with early behavior of systems. As explained in [8] one of the major aspects of all temporal languages is their underlying model of time. In linear temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing a behavior of a single computation of a program. In branching temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program. In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connective G ("always"), F ("eventually"), X ("next"), and U ("until"). These temporal connectives can be seen as state quantifiers. The branching temporal CTL augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). So, in CTL every temporal connective is preceded by a path quantifier [8].

Although in terms of expressiveness linear and branching temporal logics are not comparable (there are properties specified in LTL that cannot be expressed in CTL and vice versa) [8] specific aspects in where one of the two outlines the other one can be stated. Most of the specifications and formal validations in domains such as hardware design are done through the use of branching logics [3, 8-10]. In addition, branching logics result in more efficient model checking given the complexity of the algorithms involved. Suppose we are given a transition system of size **n** and a temporal logic formula of size **m**. For the branching temporal logic CTL, model-checking algorithms run in time $O(nm)$ [12], while, for the linear temporal logic LTL, model-checking algorithms run in time $n2^{O(m)}$ [11]. Since LTL model

checking is PSPACE-complete [13], the latter bound probably cannot be improved [8].

Given this context in this work we present an extension of FVS named Branching FVS based on branching temporal logics. A new branching semantic of the language is introduced and Branching FVS specifications are given in the hardware design world modeling the behavior of a FLASH memory test chip, an example taken from the literature [3].

The rest of the paper is structured as follows. Section 2 describes Branching FVS main features whereas Section 3 introduces the formal semantics of the language. Section 4 analyzes the case of study and Section 5 presents the conclusions of the work. Finally, Section 6 mentions Future Work and Section 7 discusses Related Work.

2. Branching FVS Main Features

In this section we will informally describe the standing features of Branching FVS, a simple branching extension of the FVS language [4-5]. The reader is referred to the next section 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 of the source with respect to the destination: for instance, in Figure 1-a A-event precedes B-event. 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 Figure 1-b the scenario captures the very next B-event following an A-event, and not any other B-event. Conversely, to represent the previous occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme. For example, in Figure 1-c the scenario captures the immediate previous occurrence of a B-event from the occurrence of the A-event, and not any other B-event. Events labeling an arrow are interpreted as forbidden events between both points. In Figure 1-d A-event precedes B-event such that C-event does not occur between them. FVS features aliasing between points. Scenario in 1-e indicates that a point labeled with A is also labeled with A ^ B. It is worth noticing that A-event is repeated on the labeling of the second point just because of Ω-FVS formal syntaxes [15]. Finally, two special points are introduced as delimiters to denote the beginning and the end of an execution. These are shown in Figure 1-f.

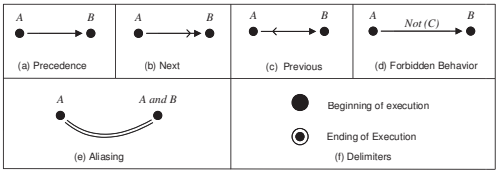

Figure 1. Basic Elements in Branching FVS

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. In Branching FVS two types of rules can be defined: rules defining behavior with an existential path quantifier (FVS-E rules) and rules defining behavior with a for all path quantifier (FVS-A) rules. For the FVS-E rules the intuition is that if **at least one time** the trace "matches" a given antecedent scenario, then it must also match at least one of the consequents. For the FVS-A rules the intuition is that **every time** the trace "matches" a given antecedent scenario, then it must also match at least one of the consequents. In both cases, 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. In addition, FVS-E rules are denoted with a letter **E** whereas FVS-A rules are denoted with a letter **A**, in order to distinguish both types of rules.

Two examples are shown in Figure 2 modeling the behavior of a client-server system with FVS-A rules. That is, these properties must be satisfied for all possible computations. The rule in the top of Figure 2 establishes that every request received by a server must be answered, either accepting the request (consequent 1) or denying it (consequent 2). The rule at the bottom of Figure 2 dictates that every granted request must be logged due to auditing requirements.

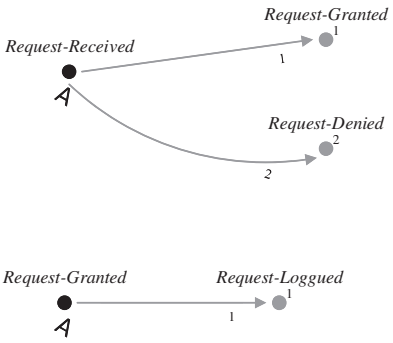

Figure 2. FVS-A rules' examples with a For All Path Quantifier

An additional requirement for the server-client system is added next featuring an existential path quantifier. The requirement is the following: It is possible to get to a state where *started* holds, but *ready* does not hold, where *started* and ready denote two possible states of the system. The CTL formula modeling this behavior is the following one*: EF (started ∧ ¬ready)*, where *E* stands for the existential path quantifier and *F* stands for the *future* operator. Figure 3 shows a Branching FVS specification for this requirement through an FVS-E rule. The rule demands that at least one time since the beginning of the trace both events *started* and ¬ *ready* must hold simultaneously.

Figure 3. FVS-E rule example with an Existential Path Quantifier

3. Branching FVS Semantics

The semantics of Branching FVS is based on the formal definition of the FVS language detailed in [4,5]. In few words, morphisms between scenarios are defined in order to establish when a certain trace of the system satisfies a given rule. Then, the semantics of a system in FVS is given by the set of traces that satisfies all the specified rules modeling the behavior of the system. Based on these concepts and following the two types of path quantifiers introduced in CTL semantics of Branching FVS is tackled providing definitions of scenarios, morphims, rules and satisfiability for the **path quantifier E** and also for the **path quantifier A**. Finally, the semantics of a system in Branching FVS is denoted by those traces satisfying both types of rules. Section **3.1** introduces the semantics for path quantifier E, Section **3.2** introduces the semantics for path quantifier A whereas Section **3.3** exhibits the semantics of Branching FVS combining both type of rules.

3.1. Branching FVS: Existential Path Quantifier

An FVS scenario is described by the following definition.

Definition 1. (FVS-E) An FVS scenario for the path quantifier E or simply an FVS-E scenario is a tuple $\langle \sum, P, l, \equiv, \neq, <, \partial \rangle$ where:

- $\sum$ is a finite set of propositional variables standing for types of events;
- P is a finite set of points;
- $l : P \rightarrow PL(\sum)$ is a function that labels each point with a given formula precisely and PL is the set of propositional formulas that can be obtained from the alphabet $\sum$;
- $\equiv \subseteq P \times P$ is an equivalence relation;
- $\neq \subseteq P \times P$ is an asymmetric relation among points;
- $< \subseteq P \oplus \{0\} \times P \oplus \{\infty\} \setminus \{\langle 0, \infty \rangle\}$ is a precedence relation between points, where 0 and ∞ represent the beginning and the end of execution, respectively;
- ∂: $(\neq \cup <) \rightarrow PL(\sum)$ assigns to each pair of points, related by precedence or separation, a formula which constrains the set of events occurrences that may occur between the pair.

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

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

- $l_2(a) \rangle \Rightarrow l_1(p)$ is a tautology for all $p \in P_1$ and all $a \in 2\ P_2$ such that $a \equiv_2 f(p)$;
- $\partial_2(f(p),f(q)) \Rightarrow \partial_1 (p; q)$ is a tautology for all $p; q \in P_1$;
- if $p \equiv_1 q$ then $f(p) \equiv_2 f(q)$ for all $p; q \in P_1$;
- if $p \neq_1 q$ then $f(p) \neq_2 f(q)$ for all $p; q \in P_1$;
- if $p <_1 q$ then $f(p) <_2 f(q)$ for all $p; q \in P_1$.

We now formally define FVS-E rules as a rule scenario playing the role of the antecedent, one or more consequents scenarios and finally, morphisms from the antecedent to the consequents. More formally:

Definition 3 (FVS-E Rule) Given a scenario S_0 (antecedent) and an indexed set of scenarios and morphisms from the antecedent $f_1 : S_0 \rightarrow S_1$; $f_2 : S_0 \rightarrow S2$; …; $f_k : S_0 \rightarrow S_k$ (consequents), we call $R \langle S_0, \{f_i: S_i\}_{i=1...k} \rangle$ an FVS-E Rule.

Finally, we can state when a trace satisfies a given rule. Intuitively, the rule will be satisfied if at least one time the antecedent is "matched" at least one of the consequents is matched. Since an existential path quantifier is being defined, it is enough to check if the rule is satisfied at least one time, and not every time. This is why the definition requires consequent(s) to be matched if some morphism (and not all of them) from the antecedent to the scenario is found.

Definition 4 (FVS-E rules' semantics) A scenario *S* satisfies an FVS-E rule *R* ($S \vDash R$) iff **for some morphism *m***: $S_0 \rightarrow S$ there exists $m_i: S_i \rightarrow S$, for some $i \in \{1..k\}$ such that $m = m_i \circ f_i$.

3.2. Branching FVS: For All Path Quantifier

For the path quantifier A ("for all computations") FVS semantics is given as follows. The exact same definitions for scenarios, morphisms and rules that were introduced for the path quantifier E ("there exists a computation") apply for this quantifier. So, FVS-A scenarios, FVS-A morphisms and FVS-A rules are trivially defined. The only difference is given in the **rules' semantics definition**. For the path quantifier E the rule is satisfied if at least one time the antecedent is "matched" at least one of the consequents is matched. However, for the path quantifier A **every time** the antecedent is matched at least one of the consequents must be matched. The formal definition establishing the rule satisfiability for the path quantifier A in Branching FVS is given next.

Definition 5 (FVS-A rules' semantics) A scenario *S* satisfies an FVS-A rule *R* ($S \vDash R$) iff **for every morphism *m***: $S_0 \rightarrow S$ there exists $m_i: S_i \rightarrow S$, for some $i \in \{1..k\}$ such that $m = m_i \circ f_i$.

Note that the only difference between this definition (**Definition 5**) and the previous one (**Definition 4**) is the amount of morphisms needed to be satisfied (at least one for the path quantifier E and all of them for the path quantifier A).

3.3. Branching FVS Semantics

Now that the semantics for both kinds of path quantifiers are defined we can establish the formal semantics of the system based on the definitions **4** and **5**: **The semantic of the system is given by those traces satisfying the set of FVS-A and FVS-E rules**. More formally:

Definition 6 (Branching FVS Semantics) The semantics of a system *S* specified in Branching FVS is given by the set of traces *T* such that $T \vDash ER \wedge T \vDash AR$ where *ER* denotes the set of all the FVS-E rules and *AR* the set of all the FVS-A rules.

4. Case Study

Our case of study is based on one example introduced in [3]. In few words, the subject of the case study is the "Tricky" technology FLASH memory test chip in 0.13µs process developed in *ST Microelectronics*. In particular, imitating the methodology introduced in [3] we focused on modeling some requirements for its external interface. The memory cell can be in one of the programming, reading or erasing modes. The correct functioning of the design at its analog level of abstraction in a given mode is determined by the behavior of the following interface signals: **bl**: the matrix bit line terminal; **pw**, the matrix p-well terminal; **wl**, the matrix word line; **s**: the matrix source terminal; **vt** the threshold voltage of cell and **id**, the drain current of cell.

The specification in Branching FVS for the case of study takes some considerations. Our language does not feature timing constructors such as clocks variables or other similar strategies. A typical way to introduce them is to consider the instants when a signal changes its value, since it can be interpreted as instantaneous events [3]. To this end we propose events representing changes on the signals such as: *vt_threshold* (the *vt* signal crosses the threshold) or *wl_below* (the *bl* signal is below its acceptable minimum). Given this context we modeled four properties describing the correct behavior of the cell in the programming mode and one property to detect the start of the erasing mode.

The first property modeling the programming mode requires that whenever the *vt* signal crosses its threshold, both *vt* and *id* have to remain continuously steady until the *id* signal falls below its threshold. Events *vt_threshold, vt_steady, id_steady and id_falls* are introduced to model this behavior in Branching FVS. The *vt_threshold* event represent the moment the signal crosses its threshold, steady and unstable events models different states of the given signals and finally *id_falls* represents the moment *id* signal falls below its threshold. This requirement is modeled in the figure 4. Since this property must be satisfied in all moments the for all path

quantifier is used. Every time the *vt_threshold* event occurs followed by the *id_falls* event then *id* and *vt* steady events must occur in the middle, and they should remain in that state until *id_falls* event occurs (this is modeled by requiring the absence of the events *id_unstable* and *vt_unstable*).

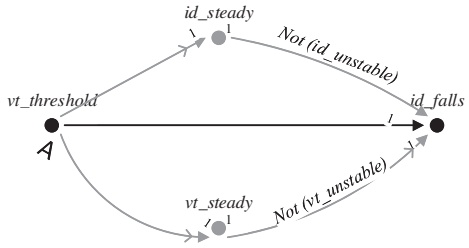

Figure 4. An FVS-A rule for the first property of interest.

The second property requires that whenever the wordline *wl* is below its threshold (represented by the event *wl_below*) but there exists a moment in the future where *wl* will jump to above its threshold (*wl_up* event) and the cell is not in the programming mode (represented by the ProgMode *event*) then the bitline signal *bl* should cross its threshold (represented by the *bl_up* event) before the end of the simulation. Figure 5 shows an FVS-E rule modeling this behavior using the mentioned events. The *bl_up* event (the consequent scenario) must occur when the conditions required by the antecedent scenario are matched.

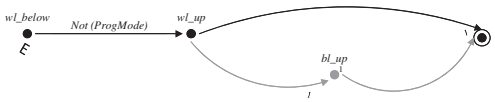

Figure 5. An FVS-E rule for the Programming Mode.

The third property specifies that whenever the programming procedure starts, the bitline signal *bt* should not fall below its threshold until the signal vt becomes up a certain value and the absolute value of the source current id goes below its threshold. The following events are introduced to shape this requirement: *ProgMode, bt_down, vt_up* and *id_below* as shown in the FVS-A rule in figure 6.

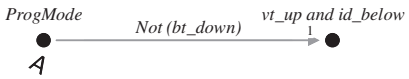

Figure 6. An FVS-A rule for the Programming Mode.

Finally, we modeled a fourth property describing the expected behavior of the programming mode. This property requires that whenever the bitline *bl* and wordline *wl* signals are above their thresholds, the p-well signal *pw* has to be below its threshold. The required events for this property are: *bl_up, wl_up* and *pw_below*. Figure 7 shows the FVS-A rule for this requirement. In this rule the consequent demands that every time events *bl_up* and *wl_up* occur the *pw_below* should also simultaneously occur.

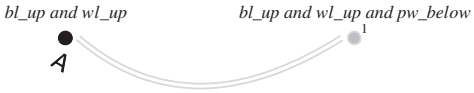

Figure 7. The fourth rule for the Programming Mode.

Regarding the erasing mode, we introduce one property which aims to detect when the erasing mode should begin (represented by the *ErasingMode* event). According to the specification detailed in [3] this occurs if there exists a computation where wordline signal *wl* is lower than a certain value (represented by the *wl_down* event) and p-well *pw* is above its threshold (represented by the *pw_up event*). Figure 8 shows the existential FVS rule modeling this property.

Figure 8. An FVS-E rule detecting the Erasing Mode

Summing up, we showed a Branching FVS specification modeling the behavior of a FLASH memory test chip. Four rules were introduced shaping the behavior in the *programming mode* plus one additional rule to detect the beginning of the *erasing mode*. Three rules were expressed with the path quantifier *A* and two with the path quantifier *E*.

5. Conclusions

In this work we present an extension of the FVS language named Branching FVS which aims to cope with properties expressed in branching temporal logic. Branching logics enable the software engineer to express interesting properties and it is heavily used for hardware design and formal validation, besides being more efficient in terms of execution time when used as input into model checkers than linear temporal logics. Branching FVS features two different types of rules: rules preceded with a universal path quantifier and rules preceded with an existential path quantifier. Branching FVS was shown in action modeling a case of study in the

hardware design world. The formal semantics of the language is also presented in this work. Although further work is needed to consolidate our approach, we consider this extension a solid first step for FVS in the branching logic world.

6. Future Work

There are several lines of research which may continue this work. First of all, we would like to use Branching-FVS specifications as input to formal validation tools like model checkers. In order to accomplish this objective, we need to revisit the tableau algorithm that translates FVS scenarios into Büchi automata [4] so that Branching-FVS scenarios can be translated too. We consider that this will be shortly achieved since Branching-FVS is a simple extension of FVS. This line of research also includes incorporating Branching FVS features into our tool GTxFVS [7] which currently holds FVS features only.

Secondly, we would like to compare Branching FVS with other known notations such as Petri Nets or other CTL extensions like [3,15] taking into account issues like usability, flexibility and expressive power. We also would like to continue exploring Branching-FVS in the hardware verification domain considering case of studies such as [16].

Finally, inspired in the work of [15] we would like to explore Branching-FVS and formal verification in the Artificial Intelligence domain.

7. Related Work

The language PSL [17-18] is widely used by chip design and verification engineers across the hardware verification community. Hardware properties can be specified in PSL in order to verify the expected behavior. The language originated as the Sugar language and later evolved into an IEEE standard. It is heavily based on temporal logics (LTL), augmented with regular expressions. We share some objectives with PSL. In the same spirit of our work, they acknowledge the need and value of reviewing and validating properties specification. However, this feature is achieved by introducing tools build on the top of PSL [19-20]. So, contrary to Branching-FVS, validation capabilities require tool support and cannot be obtained directly from PSL specifications.

On the top of SPL work in [3] introduces Signal Temporal Logic (STL) specification language, and is implemented in a stand-alone monitoring tool (AMT), which constitutes a solid approach for hardware verification purposes. Timed requirements can be denoted in STL Logics since timed constructors are available in the language. However, specifications in STL resemble programming languages constructions which may lead to premature operational decisions. In this sense, graphical and declarative notations such as Branching FVS might be closer to the way requirements are expressed [21], which make easier the behavioral exploration and specification of systems.

Real-time monitoring of the timed LTL (TLTL) logic is studied in [9]. TLTL specifications are interpreted over finite traces with the 3-valued semantics. The extensions of temporal logics that deal with richer properties were also considered in monitoring tools such as LOLA [10]. Other known realtime extensions of LTL are Temporal logics MTL [23] and MITL [22]. Branching FVS is yet only a specification language and needs further work in order to incorporate Branching-FVS specifications into validations tools such as model checkers [24-25]. Clearly, the road taken by these approaches from specification languages to formal validation tools will inspire our future work.

Work in [14] presents a technique based on Petri Nets which focuses on the design and verification methods of distributed logic controllers supervising real life processes. We believe comparing usability, flexibility and expressivity of Petri Nets, temporal logic and other approaches like Branching FVS constitutes an appealing line of research to address in the mid-term future.

Among branching logics extensions with formal verification purposes in other domains it is worth mentioning [15] and MCK [26]. They define some CTL extensions focused in the Artificial Intelligence and multi agent system domain. We share with these and other similar approaches the need for more expressive specification languages. In this sense, we believe that there is an opportunity for Branching FVS in this domain given its flexibility and expressive power.

8. Acknowledgements

This work was supported in part by a grant from UNDAVCYT 2014 and UAI-CAETI founding.

9. References

[1] Bouajjani A, Lakhnech Y, Yovine S (1996) Model checking for extended timed temporal logics. In: Formal techniques in real-time and fault-tolerant systems. Springer, Berlin,Heidelberg, pp 306–326

[2] Vardiy M, Wolperz P (1994) Reasoning about infinite computations. Information & Computation, 115(1), 1-37.

[3] Maler, O., & Ničković, D. (2013). Monitoring properties of analog and mixed-signal circuits. International Journal on Software Tools for Technology Transfer, 15(3), 247-268.

[4] Asteasuain, F., & Braberman, V. (2017). Declaratively building behavior by means of scenario clauses. Requirements Engineering, 22(2), 239-274.

[5] Asteasuain, F., & Braberman, V. (2015). Specification patterns: formal and easy. International Journal of Software Engineering and Knowledge Engineering, 25(04), 669-700.

[6] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999, May). Patterns in property specifications for finite-state verification. In Proceedings of the 21st international conference on Software engineering (pp. 411-420). ACM.

[7] F. Asteasuain, f. Tarulla & P. Gamboa. Using the power of abstraction to express high-level behavior in aspect oriented approaches. In CONAIISI, 2017. Congreso Nacional de Ingeniería Informática – Sistemas de Información.

[8] Vardi, M. Y. (2001, April). Branching vs. linear time: Final showdown. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 1-22). Springer, Berlin, Heidelberg.

[9] Andreas Bauer,Martin Leucker, and Christian Schallhart. Monitoring of Real-Time Properties. In FSTTCS, pages 260–272, 2006.

[10] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra & Zohar Manna. LOLA: Runtime Monitoring of Synchronous Systems. In TIME, pages 166–174, 2005.

[11] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In Proc. 12th ACM Symp. on Principles of Programming Languages, pages 97–107, New Orleans, January 1985.

[12] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems, 8(2):244–263, January 1986.

[13] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. Journal ACM, 32:733–749, 1985.

[14] Grobelna, I., Wiśniewski, R., Grobelny, M., & Wiśniewska, M. (2017). Design and verification of real-life processes with application of Petri nets. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 47(11), 2856-2869.

[15] Lomuscio, A., Pecheur, C., & Raimondi, F. (2007). Automatic verification of knowledge and time with NuSMV. In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (pp. 1384-1389). IJCAI/AAAI Press.

[16] D´Angiolo, Federico Gabriel; Suarez Martene, Juan Cruz , Lipovetzky José. Memoria SRAM de 1 kbit integrada en un proceso CMOS estándar. Congreso de Microelectrónica Aplicada. 2010. Buenos Aires.

[17] Eisner C, Fisman D (2006) A practical introduction to PSL (series on integrated circuits and systems). Springer, Secaucus.

[18] IEEE-Commission et al (2005) Ieee standard for property specification language (psl). Tech. rep., Technical report, IEEE, 2005. IEEE Std 1850-2005

[19] David S, Orni A (2005) Property-by-example guide: a handbook of psl/sugar examples-prosyd deliverable d1. 1/3

[20] Bloem R, Cavada R, Eisner C, Pill I, Roveri M, Semprini S (2004) Manual for property simulation and assurance tool (deliverable 1.2/4–5). In: Technical report, PROSYD Project, Technical Report.

[21] Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on (pp. 249-262). IEEE.

[22] Rajeev Alur, Tom´as Feder, and Thomas A. Henzinger. The Benefits of Relaxing Punctuality. J. ACM, 43(1):116–146, 1996.

[23] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. Real-Time Systems, 2(4):255–299, 1990.

[24] Holzmann, G. J. (2004). The SPIN model checker: Primer and reference manual (Vol. 1003). Reading: Addison-Wesley.

[25] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999, July). NuSMV: A new symbolic model verifier. In International conference on computer aided verification (pp. 495-499). Springer, Berlin, Heidelberg.

[26] Gammie, P., & Van Der Meyden, R. (2004, July). MCK: Model checking the logic of knowledge. In International Conference on Computer Aided Verification (pp. 479-483). Springer, Berlin, Heidelberg.