Open and Branching Behavioral Synthesis with Scenario Clauses

Fernando Asteasuain
Universidad Nacional de Avellaneda
1870, Avellaneda, BsAs, Argentina
Universidad Abierta Interamericana
Centro de Altos Estudio CAETI
fasteasuain@undav.edu.ar

and

Federico Calonge
Universidad Nacional de Avellaneda
Avellaneda, Argentina
fcalonge@undav.edu.ar

and

Manuel Dubinsky
Universidad Nacional de Avellaneda
Avellaneda, Argentina
mdubinsky@undav.edu.ar

and

Pablo Daniel Gamboa
Universidad Abierta Interamericana
CABA, Argentina
pdgamboa@uai.edu.ar

Abstract

The Software Engineering community has identified behavioral specification as one of the main challenges to be addressed for the transference of formal verification techniques such as model checking. In particular, expressivity of the specification language is a key factor, especially when dealing with Open Systems and controllability of events and branching time behavior reasoning. In this work, we propose the Feather Weight Visual Scenarios (FVS) language as an appealing declarative and formal verification tool to specify and synthesize the expected behavior of systems. FVS can express linear and branching properties in closed and Open systems. The validity of our approach is proved by employing FVS in complex, complete, and industrial relevant case studies, showing the flexibility and expressive power of FVS, which constitute the crucial features that distinguish our approach.

Keywords: Open Systems, Branching Reasoning, Behavioral Specifications, Synthesis.

1 Introduction

Early specification of behavior has been pinpointed as one of the main problems to be addressed to consolidate the transference of software formal validation and verification techniques as model checking [12] from the

academic to the industrial world [26, 27]. The software engineering community believes a solid solution should combine Open Systems [15, 26, 30] with branching reasoning [42, 43] and partial description of behavior [22, 36] utilizing specification languages that are easy to use and expressive enough to denote all types of behavior [3, 11, 14, 29, 33, 40].

The increasing demand for Open Systems [15, 26, 30] calls for the creation of tools that assist the software engineer in the complex task of specifying and describing the expected behavior of the system. There is a natural challenge involved when dealing with Open Systems since actions beyond their control must be considered, in contrast to those known as *closed* where all the events are entirely handled by the system. Open Systems interact with an environment which generates events (non-controllable by the system) which may affect in its behavior, constituting an effect known as *Controllability*.

Controllability has been addressed methodologically and algorithmically from the synthesis behavior of controllers. Synthesis behavior can be seen as an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [30]. In the case of reactive synthesis, an implementation is usually given as an automaton that accepts input from the environment, typically from sensors and produces the system's output, which are essentially instructions to actuators. By construction, every trace accepted by the automaton satisfies the specification it was synthesized from [30]. In order to reduce the time complexity of the algorithms involved, the work in [10], suggests the General Reactivity of Rank 1 (GR(1)) fragment of Linear Temporal Logics (LTL), which has an efficient polynomial time symbolic synthesis algorithm. GR(1) is a strict subset of LTL (Linear Temporal Logic) which follows an assume-guarantee scheme. A GR(1) specification consists of constraints for initial states, safety propositions over current and next states and "justice" propositions which indicate what should hold infinitely often. A GR(1) synthesis problem is defined as a game between a system player and an environment player [10]. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis have been presented in [10].

However, these approaches have some limitations regarding the specification language used. Most of them are based on temporal logics such as LTL and some extensions as fluents [20]. The expressivity of these notations has been challenged by the community [3, 14, 20, 33, 37] so there is a need to develop more expressive specification languages.

Regarding branching specifications, as it is explained in [39], one of the major aspects of all temporal languages is their underlying model of time. In LTL, time is conceived as events occurring in a unique temporal line and linear temporal logic formulas can be seen as behavioral descriptions of a single computation of a program. Linear temporal logic formulas are stated using the usual Boolean connectives plus temporal connectors such as G ("always"), F ("eventually"), X ("next"), and U ("until"). When branching temporal logics are employed, each moment in time may lead into various possible futures. In other words, program computations can be depicted as infinite trees. Therefore, the branching temporal logic CTL (Computational Tree Logic) 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 [39].

Even though, 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) [39] specific aspects in where one logic performs better than 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 [7, 13, 29, 39]. In addition, branching logics result in more efficient model checking given the complexity of the algorithms involved [39].

Given this context, in the present work, we introduce the FVS (Feather Weight Visual Scenarios) [2, 3, 6] specification language in the context of Open Systems behavior synthesis and branching behavior reasoning. We build on the top of previous work presenting an *Open and Branching version of FVS* that provide the following new capacities:

- FVS Synthesis in Open systems now includes all type of properties (in previous work we could only express a limited set of properties).
- FVS Branching specifications can now be synthesised (in previous work we could only specify behavioral branching).
- The ability to specify and synthesize behavior denoting both linear and branching type properties.
- The possibility to automatically build a controller from its specification.
- A thorough evaluation of the proposed approach with case of studies where controllers for complex and industrial relevant case studies were obtained. That is, cases of studies are not proof of concepts implementations but real and challenging systems being employed in the industry. The case study from the branching side involves dealing with the hardware design world, a particular domain where

branching specifications are vastly used. The case study for open system for Open Systems consists of two versions of a controller for a forklift robot. Evaluation is measured employing typical and pertinent concepts such as correctness and performance.

- Our tool GTxFVS (available at: https://gitlab.com/fernando.asteasuain/fvsweb) includes all these new features.

The rest of the paper is structured as follows. Section 2 briefly introduces FVS. Section 3 describes FVS in the context of Open Systems and behavior synthesis, including all the steps involved to obtain a controller using FVS specifications as input. Section 4 presents the first case study: the complete behavioral synthesis for a ForkLift robot [30] including two different controllers for the system. Section 5 introduces an FVS version incorporating branching reasoning and Section 6 shows how a controller can be obtained upon FVS branching specifications. Section 7 analyzes the case studies employed and sketches how we validated the results.Finally, Section 8 presents related while Section 9 enumerates the conclusions of this work and briefly discuss future work.

2 Background

In this section, we informally describe the standing features of FVS [3]. The reader is referred to [3] for a formal characterization of the language. FVS is a graphical language based on scenarios. Scenarios are a 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 one. Conversely, to represent the previous occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme. 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 one. Events labeling an arrow are interpreted as forbidden events between both points. In Figure 1-d A-event precedes B-event in such a way 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 and B*. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntax [3] . 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. A few observations about the notation should be mentioned. First of all, it is an event-based notation, since we aim to capture the order and occurrence (or absence) of the events of interest in the system. However, due its flexibility, states can also be captured [3]. The precedence relationship just dictates how events must occur. In other words, which events should precede or follow other events. FVS next and previous operators behave differently from those in temporal logics. We employ them as shortcuts to indicate the very next (or immediate previous) occurrence of an event. For example, we can pinpoint the first occurrence of an event as the next since the beginning of the executing. This would exclude all the following ones. Similarly, for the previous operator observe a certain connection that was established three times during the execution of the system. When considering the timeline of events after the occurrence of the third connection we can use the previous shortcut to indicate exclusively number two. Although they can be stated combining precedence and forbidden relationships, we introduce them as individual operators since they represent two very common patterns in event-based notations.

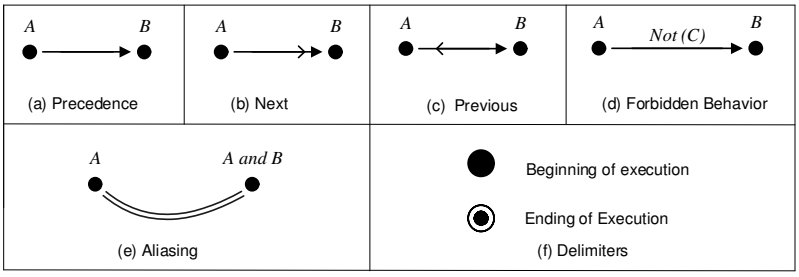

Figure 1: FVS Basic Features

We now present 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 ones. 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 one to which they belong. The semantics of the system is defined as the set of traces that satisfy all the rules [3].

Two examples are shown in Figure 2 modeling the behavior of a client-server system. The rule on the left margin 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 on the right margin of Figure 2 dictates that every granted request must be logged due to auditing requirements. In consequence, a trace where a granted request is not logged would violate the second rule in the example and would not be accepted by the behavior of the system. Accordingly, a trace where a request never got an answer would be excluded in the set of traces representing the expected behavior of the system since it would violate the first rule in the given example.

3 Open FVS and Behavior Synthesis

In this section we describe how FVS specifications can be synthesized to automatically obtain a controller. A controller is automatically obtained applying game theory algorithms. The game to solve involves two players: the system against the environment. The system wins if it can always reach an accepting state (a state that satisfies its behavior specification). Game theory concepts aim to find a winning strategy for the system that no matter what action the environment chooses, the systems always find a way to achieve its goals. If a winning strategy is not found, then the controller can not be built and rules shaping the system and the environment must be revisited. A typical way to do so is relaxing environment's conditions. When a winning strategy is found, the output generally consists of an automaton that implements the winning strategy, and moves to a next state that will eventually reach an accepting state which satisfies all the requirements.

We first present FVS's previous work and then we detailed how the new features introduced in this work were achieved.

The first step of FVS into the world of Open Systems and Behavior Synthesis (see [5]) was based on a technique introduced in [30], which is guided by the usage of specification patterns [16]. The mentioned paper presents an automated, sound, and complete translation of most of the the specification patterns [16] to the GR(1) form, which in turn were synthesized. Since FVS scenarios can be translated into Büchi automata [3], we used them as input to the synthesis scheme proposed by [30]. That is, FVS could only synthesize behavior if and only if it could be expressible by the specification patterns.

Regarding branching behavior, in [4] Branching FVS was presented and exemplified with a simple case study. However, the synthesis problem to obtain a controller over the branching specification was not been addressed yet.

We now tackle these limitations by combining FVS with the GOAL tool [38] and employing formalisms like *Rabin* automata extending FVS capabilities in order to:

- Synthesize Behavior in Open Systems including all type of properties, not being limited any more to those properties expressed by the specification patterns.
- Synthesize Behavior employing branching properties.

In what follows we delineate how this is achieved. In order to obtain a controller from its specification expressed in FVS, we rely on several techniques and tools such as [30, 45] or [38] depending on the type of behavior properties to be synthesized. In all cases, the first step is to translate FVS graphical scenarios into Büchi automata using the tableau algorithm introduced in [3]. Afterwards, if the behavior can be denoted using specification patterns, then the technique introduced in [30] is applied based on specification patterns

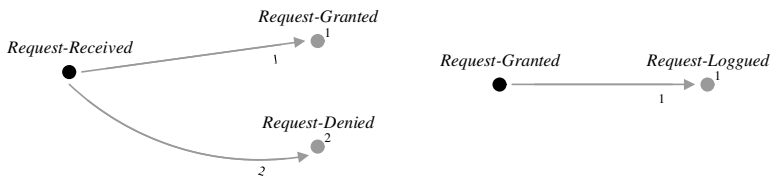

Figure 2: FVS Rules examples

and the GR(1) subset of LTL. That is, the automata generated by the FVS tableau is used as input in the [30] procedure, which translates them into GR(1) shape and apply synthesize algorithms to obtain a controller. If this is not the case, but the formula can be denoted by a deterministic Büchi automaton, the synthesized algorithms provided in the *GOAL* tool [38] are used.

Lastly, when faced with a non-deterministic Büchi automata extra steps are introduced relying on complex algorithms to manipulate infinite automata, as those available in the GOAL tool. These steps involve translating non-deterministic Büchi automata into *deterministic Rabin* automata. With these kinds of automata as input, a controller is obtained using synthesized algorithms provided in the *GOAL* tool. This tool is also used to synthesize behavior by resolving a game between the specification and the environment. GOAL provides the implementation of several game solving algorithms such as [18, 24, 35, 41].

These algorithms also can handle branching behavior, so FVS branching properties can now be synthesized. *By providing these range of possibilities we cover all the different types of properties so that any behavior specified in FVS can be used as input to obtain a controller.*

These possible ways to obtain a controller are schematized in Figure 3.

Efficiency is a key factor when automatically obtaining a controller. The best way to get a controller using FVS specifications as input is by using GR(1) subset of LTL, since more efficient algorithms are employed [10, 15, 44]. When using most of the specification patterns or deterministic Büchi automaton we fall in this category. Finally, if translation into *deterministic Rabin* automata is needed, then higher performance costs are involved. However, we believe the gain in expressivity is a solid and desirable characteristic which makes our approach distinguishable among others.

Summing up, we can obtain a controller applying different behavioral synthesis techniques employing FVS scenarios as input. When the behavior can be expressed as a specification pattern or al least is can be expressed by a deterministic Büchi automaton, the controller can be obtained efficiently. Otherwise, the expressiveness of approach is richer enough to obtain a controller by itself. However, it may take some extra time. Concerning the time complexity of our approach, it is worth mentioning that the tableau algorithm which translates FVS scenarios into Büchi automata is factorial, which is worse than the exponential tableau for linear logics or polynomial for branching logics. As it was mentioned, we prioritize expressive power over performance. Nonetheless, our performance times are similar to those approaches examined in our case studies. A main reason for this is that our automata size is similar to other approaches [3], and automata' size play a crucial role in verification algorithms performance.

It is worth mentioning that this version of FVS is fully available in the current status of GTxFVS (https://gitlab.com/fernando.asteasuain/fvsweb), the software tool which implements all the features of the language. The following section (Section 4) exemplifies our technique in action by fully modeling a complex and industrial case of study.

4 Open FVS: Case of Study

In this section, we analyze the case of study introduced in [30], the "Lego Forklift" example. The behavior of the model is given in Section 4.1. Sections 4.2 and 4.3 thoroughly describe the delayed and continuous versions of the ForkLift controller while the synthesis procedure is detailed in Section 4.4.

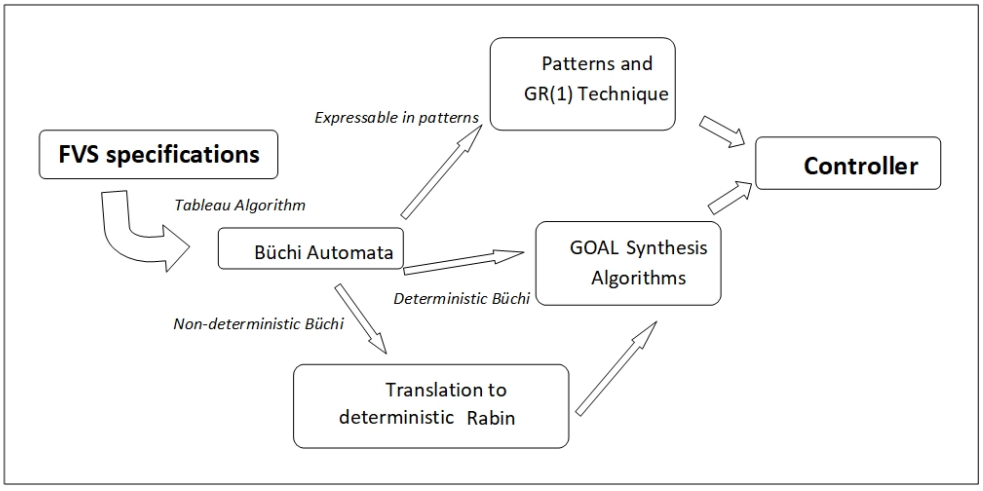

Figure 3: Different possibilities to obtain a controller with FVS scenarios as input

4.1 The "Lego Forklift" example

The behavior of the system is described in [30]. In few words, the forklift system is composed of three sensors and three motors, as well as a special button to stop it for emergency purposes. The sensors are in charge of detecting obstacles and the cargo of the forklift whereas the motors receive instructions to turn the artifact left, right, and to lift the fork. The behavior of the system does not escape from a traditional reactive system controller: a cycle where input values are obtained through the sensors, and once these values are processed instructions are sent to the actuators (instructions to the motors in this example). The system specification includes basic notions to shape the behavior of the forklift controller such as to: evade obstacles, avoid picking up cargo outside stations, keep cargo moving between stations unless the emergency button is pressed, and finally, do not attempt to lift a load again until it is delivered. The complete specification and architecture of the system is fully detailed in [30].

Following the strategy adopted in [30], we developed two different controllers for the forklift system: the Delayed controller and the Continuous controller. In the first version, a delay is introduced to give the robot time to finish the movements until the next step is processed. As in [30], we set the delay to 2000ms. For simplicity, in the rules shown we assumed that every action finishes before the next one occurs, so the delay is implicitly modeled in the specification. In contrast, in the continuous version, the controller performs its actions without delay and additional variables are introduced to track the completion of the robot's tasks.

The properties modeling the behavior of the system are described using assumptions and guarantees based on the scheme introduced in [30]. In the following two sections, we show how we modeled in our language the complete specification of both versions of the ForkLift controller. The Delayed version is tackled in Section 4.2 whereas the Continuous version is addressed in Section 4.3.

4.2 Delayed ForkLift Controller

The specification document for this version contains seven assumptions and twelve guarantees. Assumption number one says the ForkLift may escape obstacles: if the robot is backing or turning, it will reach a state where both distance sensors are clear unless it decides to go forward or to stop. This rule is captured in Figure 4. The occurrence of events *Backing* or *Turning* lead to the occurrence of the events *DistanceClear* and *CargoClear*, considering that events *Forward* or *Stop* do not occur. This behavior corresponds to a variant of the Response pattern [16]. The small numbers near the event *DistanceClear* and *CargoClear* in Figure 4 indicate the rule consequent they belong to, as explained in Section 2. These small numbers appears in all the FVS rules shown in this work.

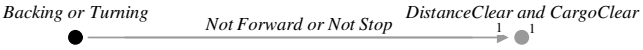

Figure 4: Assumption 1: the robot may escape obstacles

The second assumption is a safety property about the environment: the reading of the sensor station will not change when the forklift stops. This is modeled in FVS as illustrated in Figure 5. The occurrence of the Stop event at a given station implies that the robot will not leave the station when the next event occurs. We use the event *AnyEvent* as an event matching the occurrence of any event of the system. This behavior is equivalent to asking that the robot remains in the same station after stopping.

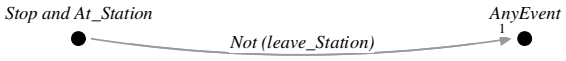

Figure 5: Assumption 2: a safety property.

The third assumption shapes an additional one for a well-behaved environment: in the given setting it is reasonable to expect that between two stations, the forklift may be blocked by obstacles at most twice. This behavior corresponds to the *Bounded Existence Pattern* [16]. The rule in Figure 6 contains four possible consequents once the forklift has left the station and the first obstacle has ocurred. Consequent 1 deals with the situation that no other obstacle is detected until the forklift arrives at the station. Consequent 2 considers the occurrence of a second obstacle, but no other obstacle must occur until the station is reached. Consequents three and four follow a similar analysis, but considering that the system may stop for some reason before reaching a station. This is why the *End* point of traces is included. Therefore, before reaching a new station or stopping the system, the forklift can be blocked by obstacles a maximum of two times.

Related to the previous requirement, assumption number four considers a liveness property of the system: going forward with both motors will lead to reaching a station unless the motors are not going forward any

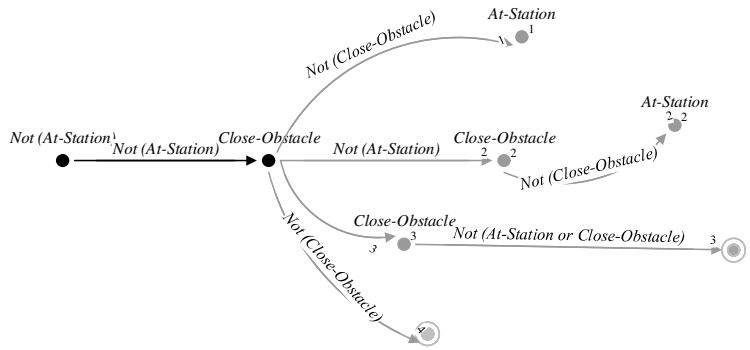

Figure 6: Assumption 3: The Bounded Existence Pattern

more. This behavior is depicted in Figure 7. Once the left and the right motors are on, then the forklift will reach a station if *MotorStopped* event does not occur.

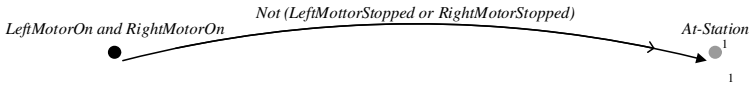

Figure 7: Assumption 4: A liveness property

Assumptions five and six deal with events related with the sensor monitoring the cargo. Assumption number five, shown in Figure -a, says that if an uncharged robot is going forward it will eventually find cargo (unless it goes backwards or it stops). Similarly, assumption number six says (-b)that when the forklift carries cargo it will eventually drop it at a given station.

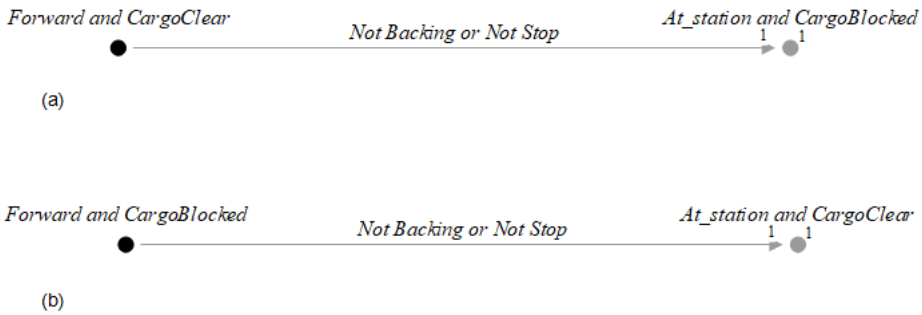

Figure 8: Assumption 5 and 6: Two cargo properties

Finally, assumption number seven says that the forklift will eventually leave the station by going forward (see Figure 9-a) or backward (Figure 9-b).

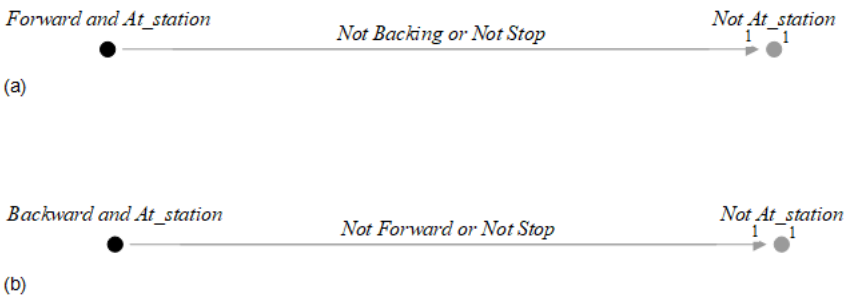

Figure 9: Assumption 7: the forklift will eventually leave the station

We now model the twelve guarantees for the ForkLift controller, which consist of one liveness property, eight safety constrains, one justice constrain and two LTL specification patterns.

The first guarantee is a liveness property which basically states that the ForkLift will always perform some actions, whether they be going back, forward, stopping or turning. The FVS rule modeling this behavior is shown in Figure 10.

Figure 10: Guarantee 1: The Forklift will continue moving

From the second to the ninth guarantees are safety properties. Guarantee number two deals with cargo and the lifting process. Basically, the system must not lift again if it contains cargo. This is captured in Figure 11. The FVS rule establishes that between the occurrence of two consecutive *CargoBlocked* event, a *Drop* event must occur. That is, the rule prohibits the system to perform two consecutive cargo operations without doing a dropping operation first.

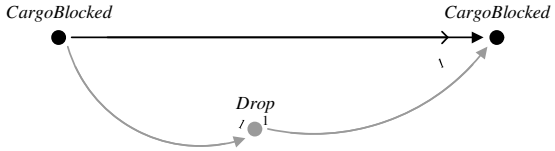

Figure 11: Guarantee 2: The system must drop the cargo before lifting a new one

The third guarantee property says that the forklift must not activate the motor in charge of lifting if no cargo is found. In other words, whenever the *CargoClear* event occurs the lifting motor must not be active. The rule modeling this requirement is shown in Figure 12.

Figure 12: Guarantee 3: No lifting movement if cargo is clear

The fourth guarantee property basically states that the forklift must do absolutely nothing if the emergency button is pressed. This behavior matches a classic Response pattern and is shown in Figure 13.

Figure 13: Guarantee 4: Emergency button stops all movements

Guarantees number five and six relate ForkLift movements when performing the lifting action or the dropping action. In few words, the robot must not move when lifting or dropping cargo. Two FVS rules model this behavior (see Figure 14, where lifting cargo is shown in Figure 14-a whereas dropping cargo is shown in Figure 14-b).

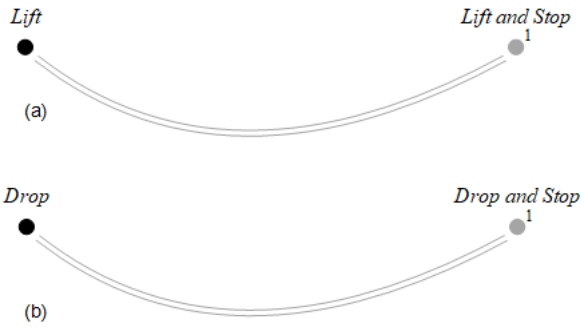

Figure 14: Guarantees 5 and 6: The robot must not move while lifting or dropping cargo

The next two guarantees (guarantees seven and eight) relate the dropping and lifting actions with the cargo of the ForkLift. In particular, if a *Lift* event occurs then eventually the sensor in charge of the cargo will achieve the *blocked* value. This is shown in Figure 15-a. Similarly, Figure 15-b shows that when the *Drop* event occurs the sensor will achieve the *clear* value.

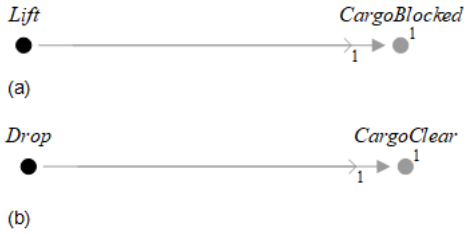

Figure 15: Guarantees 7 and 8: Dropping and lifting and the cargo sensor properties

Guarantee number nine deals with wheel motors and obstacle. The requirement demands that if the ForkLift detects an obstacle both wheel motors will stop (denoted by the occurrence of the events *LeftMotorStopped* and *RightMottorStopped*). Note that these events are modeled as occurring simultaneously, the way it is indicated in the system requirements specified in[30]. This last guarantee property is shown in Figure 16.

Figure 16: Guarantee 9: Motors should automatically stop when obstacles are detected

The tenth guarantee property is a justice constrain: it implies that the robot will be performing its task (deliver cargo) continuously. As explained in [30] this property does not only contain the expected *Drop* event (i.e., delivering cargo), but also the alternatives *Emergency* and *Obstacle* events. These alternatives represent environment actions that if occurring infinitely often liberate the forklift from its obligation. This justice type behavior is shown in Figure 17. The system will continue to lift cargo (always a new *CargoBlocked* event occurs) unless obstacles are detected or the emergency button is pressed.

The last two guarantees represent specification patterns. The eleventh guarantee of the system constitutes an instance of the *Existence Pattern* with *Between P and Q* scope [16]. The behavior to be modeled is the following: the forklift has to leave the station between lifting cargo and delivering it. This is shown in Figure 18.

Finally, the twelfth guarantee property corresponds to the *Universality* pattern with *After Q until R* scope. The requirement to be addressed is described next. After leaving a station, cargo will not be dropped until the forklift reaches another station. This is modeled in FVS as shown in Figure 19. If the *CargoBlocked* event (a cargo is lifted) occurs while the forklift is out of a station and it arrives to another one in the future, then the cargo must not be dropped during the path.

All the presented rules represent the specification for the delayed version of the ForkLift controller. The next section introduces the Continuous version, where no delay is involved in the cycle of the robot.

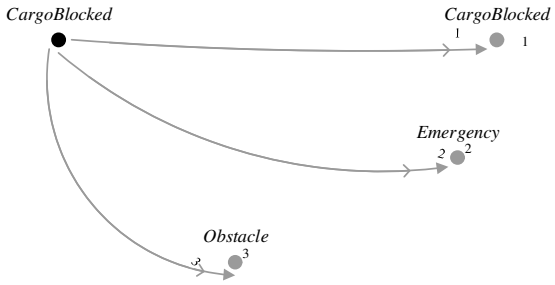

Figure 17: Guarantee 10: A justice constrain for the ForkLift system

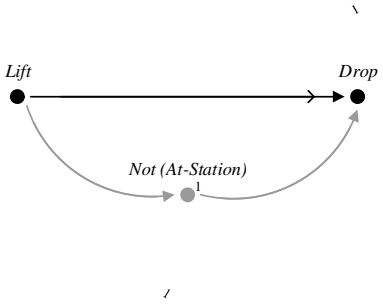

Figure 18: Guarantee 11: Existence pattern. The forklift must leave station while delivering cargo

Figure 19: Guarantee 12: A property modeled using the Universality pattern

4.3 Continuous ForkLift Controller

As fully explained in [30], the second variant applies a method inspired by Raman et al. [34] for continuous control. The main idea is to add a new variable for every action which signals completion of the action. In our case study, this method is only necessary for the completion of lifting and dropping cargo. This is achieved by introducing the event *CargoOK*.

All the properties from the delayed version are valid in the continuous version. This new controller variant adds two extra guarantees and two extra assumptions. These are described next.

The first new guarantee says that if the ForkLift starts the lifting movement then the cargo action will eventually finish. In other words, the occurrence of the *Lift* event will lead to the occurrence of the *CargoOK* event. This is reflected in Figure 20.

Figure 20: New guarantee 1 for the Continuous controller

The second extra guarantee says that *CargoOK* event should only occur if the ForkLift is lifting (i.e, it is not turning or stopping or going forward). This is shown in Figure 21. The rule in this figure establishes that if the *CargoOK* event occur it is because the event *Lift* occurred in the past and no other movement has been done since then.

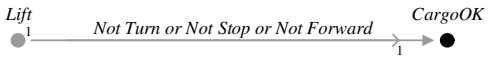

Figure 21: New Guarantee 2: Cargo action should only occur while the robot is lifting

Finally, the continuous controller adds two new assumptions. The first one checks that the ForkLift stops when it initiates a lifting action, shown in 22-a. The second one, shown in 22-b, ensures that the robot does not receive a new lifting command while waiting for the completion of a cargo.

4.4 Synthesising behavior from FVS scenarios

By employing the different alternatives described earlier (see Figure 3), we obtained Forklift controllers (both delayed and continuous versions) using as input FVS specifications. Given that all the properties considering both versions could be denoted using deterministic Büchi automata, we obtained similar performance execution times to those results obtained in [30] (see Section 4.5 for more information about this point). In addition, FVS automata turn out to be equivalent to those presented in [30]. This is an important result to validate how our approach builds controllers systems.

Figure 23 shows part of the controller obtained for the delayed version of the system. We show only a part for simplicity reasons. Similarly, Figure 24 shows a controller for the new restrictions added in the continuous version. The controller of the whole system results from the combination of the automata shown

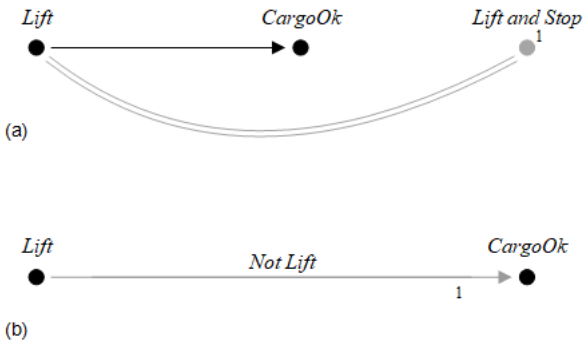

Figure 22: New assumptions 1 and 2 for the ForkLift continuous controller

in Figure 23 and Figure 24. The controllers shown in Figures 23 and 24 represent typical automata that react to events controlled by the environment by moving to other states in order to fulfill the expected behavior. For example, in Figure 24 the controller moves from the initial state to state *S1* when the *Lift* event occurs. Lambda transitions represent internal actions.

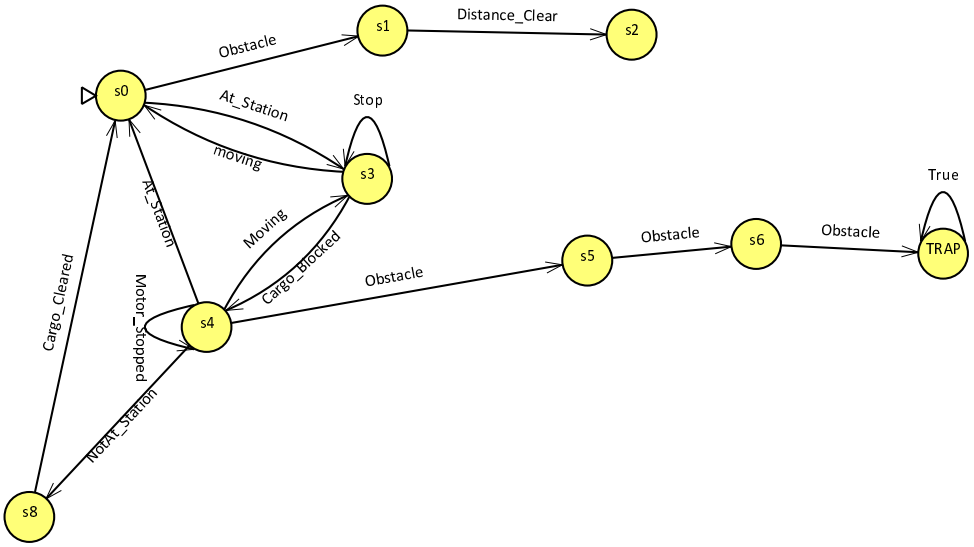

Figure 23: Part of the controller for the delayed ForkLift controller version

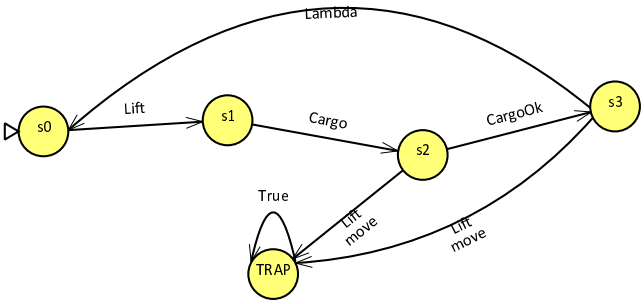

Figure 24: Part of the controller for the continuous version of the system

4.5 Time and size comparison

Comparison results are shown in Table 1. In [30], the controller for the delayed version of the system was obtained in 1.8 seconds with a size of 3412 states (the number of transitions is not reported), whereas the controller for the continuous version was obtained in 1.3 seconds with a size of 2888 states. In our approach,

Example	Time	States	Transitions
Original Delayed Forklift	1.8 sec	3412	not given
Original Continuous Forklfit	1.3 sec	2888	not given
FVS Delayed Forklift	2.6 sec	3871	13543
FVS Continuous	2.7 sec	4104	14378

Table 1: Time and size comparison

the delayed controller was obtained in 2.6 seconds with a size of 3871 states and 13543 transitions. For the continuous version, it took 2.7 seconds with a size of 4104 states and 14378 transitions. These number support the concept mentioned earlier in this paper: our approach is not as efficient, but the difference is near negligible with the plus of employing an expressive and flexible specification language. In [30], the continuous controller is obtained in less time and with a smaller automata where the opposite occurs in our approach. We believe that this arises given that we compose automata and this may cause this variation in the results. Despite the different automata obtained from both approaches they result in equivalent automata accepting the very same traces. Thus, FVS approach is proven valid.

5 Branching FVS

In this section, we briefly present an extension of FVS which is able to handle branching reasoning called Branching FVS. All the previously mentioned features of FVS hold in Branching FVS. The main distinguishable point of Branching FVS is that **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. Graphically, FVS-E rules are shown with a letter E whereas FVS-A rules are shown with a letter A, in order to distinguish both types of rules. These two points are the main items modified in FVS Branching semantics: how to satisfy existential and universal quantifiers. FVS-A rules behave exactly as the original FVS rules presented in Section 2. To illustrate this, we show again the client-server example given in Section 2 (see Figure 1) now considering branching features. Two examples are shown in Figure 25 modeling the behavior of a client-server system with FVS-A rules. That is, these properties must be satisfied for all possible computations. As it was mentioned in Section 2, the rule shown in Figure 25-a establishes that every request received by a server must be answered, either accepting the request (consequent 1) or denying it (consequent 2). The rule shown in Figure 25-b dictates that every granted request must be logged due to auditing requirements.

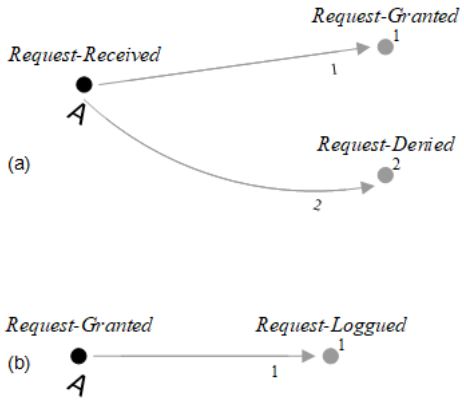

Figure 25: FVS-A rules examples with a For All Path Quantifier

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, and where *started* and *ready* denote two possible states of the system. The CTL formula modeling this behavior is the following one: $E\ F$ (started $\wedge \neg$ ready) , where E stands for the existential path quantifier and F stands for the future operator. Figure 26 shows a Branching FVS specification for this requirement

through an FVS-E rule. The rule demands that at least once, from the start of the trace, both events *started* and *Not ready* must be held simultaneously.

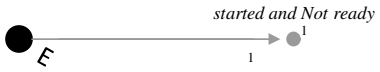

Figure 26: FVS-E rule example with an Existential Path Quantifier

6 Branching FVS synthesis

In this work, we go one step further and not only specify behavior in Branching FVS, but we also synthesize behavior and obtain a controller using the GOAL tool [38]. We based our case study on synthesizing behavior and obtaining a controller on the work introduced in [32]. In that work, a tool for embedded control software synthesis named PESSOA is presented. An attractive case study involving a DC motor is shown. In what follows we model the behavior of the DC motor using Branching FVS as the specification language. In addition, a controller of the system is obtained using the technique earlier described in Section 3.

In [32] systems are specified with logical calculus formulae describing trajectories over open sets. These trajectories involve defining the set of states X, the set of objectives Z ($Z \subseteq X$) and constraints restricting the behavior of the system (a set of constraints $W \subseteq X$). Given this context, requirements are described using only four type of properties: *Stay* properties, *Reach* properties, *Reach and Stay* properties, and *Reach and Stay while Stay* properties. *Stay* properties describe those trajectories that start and remain in the set of objectives *Z*. *Reach* properties include those trajectories that stay in the set of objectives only a finite time. *Reach and Stay* properties are those that eventually reach Z and remain there subsequently. Finally, *Reach and Stay while Stay* are those properties that reach and stay in Z if and only if the set of constraints W is satisfied.

These four templates of properties are powerful enough to express non trivial synthesis problems in the hardware domain [32]. The desired trajectories are monitored to check if they satisfy the expected thresholds defining properly the objectives and constraints set. Figure 27 shows these four properties in Branching FVS using events as *OutOfZ*, which occurs when a trajectory fall out of the Z set, *OutOfW*, which occurs when a trajectory fall out of the Z set, *InZ*, which occurs when a trajectory fall in of the Z set, and *inW*, which occurs when a trajectory fall out of the W set. More concretely, rule in 27-a describes the *Stay* properties, rule in 27-b describes the *Reach* properties, rule in 27-c describes the *Reach and Stay* properties and finally, rule in 27-d describes the *Reach and Stay while Stay* properties. The end of execution operator is introduced in this case since signals must hold the specified range until the end of the operation of the system.

By using these properties and properly defining the sets Z and W we obtained controller for the DC motor example introduced in [32]. For this case, Z is defined as $= [19.5, 20.5]x[-0.7, 0.7]$ and W = [-1,30] x [-3,3]. A sketch from the controller is depicted in Figure 28, illustrating how, from the initial state, the controller can advance to new states according to the events triggered by the environment (*stay*, *reach*,*inZ*, etc.). Recall that the automaton in Figure 28 it not complete and all events, states and transitions are shown. We only exhibit part of it for simplicity reasons. The complete automata for the all the controllers shown in this work are available at: https://gitlab.com/fernando.asteasuain/fvsweb .

Following the example in [32], we also obtained a controller for another classical problem: motion planning with obstacle avoidance by just adjusting accordingly sets Z and W.

Finally, we also modeled and obtained a controller considering the last example in [32]: **Control with shared actuators**. We consider a control system that has permanent access to a low quality actuator and sporadic access to a high quality one with a fairness constraint to access them. The fairness constraint is defined by an automaton with three states *S1*, *S2*, and *S3*. In *S1*, both actuators are available, whereas in *S2* and *S3*, only the low quality one is available. In order to obtain a controller with shared actuators, the system must be composed with this automaton. Some rules modeling the behavior of the automaton are shown in Figure 29. The rule shown in Figure 29-a demands that if the high quality actuator is used then the system must be in state *S1*. On the other hand, the low actuator can be activated in state *S1*, *S2* or *S3* (shown at Figure 29-b).

A simplification of the controller modeling the Fairness constraint for shared actuators is shown in Figure 30, stating how the controller should react to the environments' events. This controller, composed with the one shown in Figure 28 constitute the controller for the whole system.

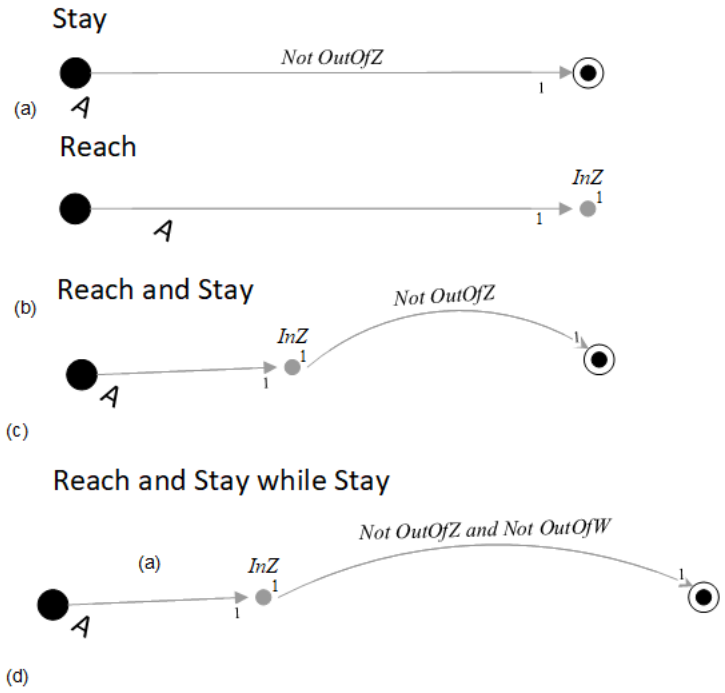

Figure 27: Four Properties for controllers' synthesis

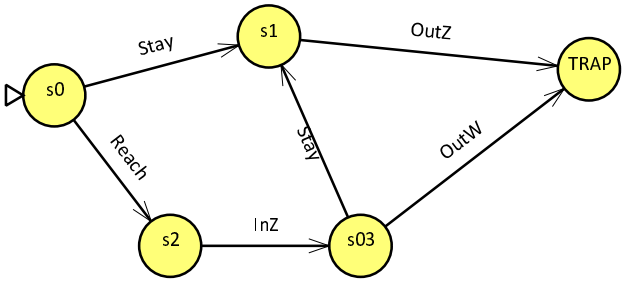

Figure 28: A controller for the DC motor system

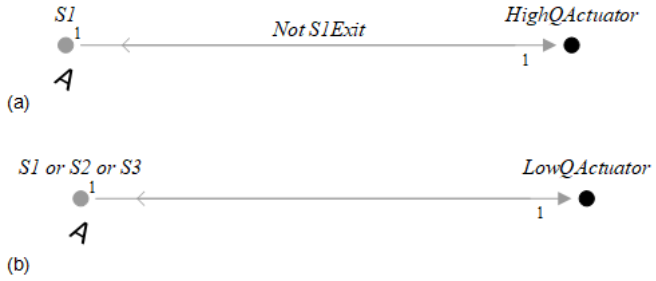

Figure 29: Fairness constraint for shared actuators

6.1 DC Motor Example's Analysis and Observations

To conclude this section, it is worth mentioning that all types of properties can be synthesized using Branching FVS specifications as input. In this case study, we only consider the four template properties introduced in [32] since we were replicating the case study presented in their work. Comparison results are shown in Table 2. It is worth noticing that the automata obtained by employing the FVS approach are equivalent (i.e, they recognize the same language) to those shown in [32], fact that proves the correctness of our specifications and synthesis procedure for this example. Regarding time and size of the controller in [32], they say that the controller is obtained in 12 seconds where the computation of the model took 205 seconds. The size of the controller is not given. In contrast, in FVS the controller is obtained in 9.9 seconds with a size of

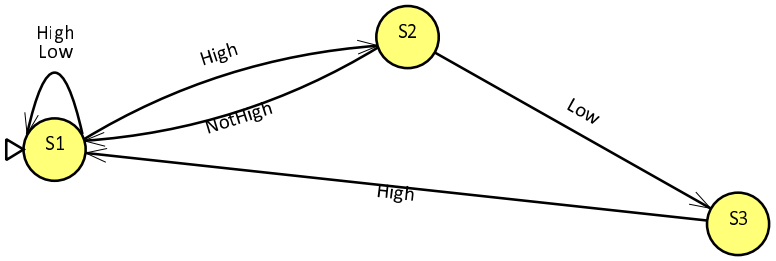

Figure 30: A controller considering shared resources

Example	Time	States	Transitions
PESSOA DC Motor Controller	12 sec	not given	not given
FVS DC Motor Controller	9.9 sec	19	35

Table 2: DC Motor Example Comparison

19 states and 35 transitions.

7 Case Studies: Analysis and Validation of the results

Two representative and challenging case studies were developed to highlight FVS main features, flexibility, and expressive power to specify and synthesize behavior, considering stimulating and appealing concepts such as *linear timed behavior*, *branching behavior*, and *Open Systems*. For linear time behavior reasoning in Open Systems, we considered the ForkLift system [30]. For branching reasoning, we moved to the hardware world synthesizing the behavior of a controller for an embedded system, a typical domain where branching behavior reasoning predominates. Both examples show the flexibility and expressive power of our notation to achieve meaningful results covering all the mentioned aspects in complex problems. In addition, FVS specification can be used as noted in the examples to feed model checking tools.

In order to validate the results, we took two different and orthogonal directions: *correctness and performance*. Regarding correctness, work in [3] exhibits that the tableau algorithm translating FVS scenarios to Büchi automata is sound and complete. For the purpose of synthesizing behavior, we employed widely known tools and techniques, such as specification pattern-based synthesis in [30] or the GOAL tool [38], which have proved their correctness as well. Since we connected FVS with these tools using FVS specifications as input, the correctness of the approach was demonstrated. We performed a second validation covering the correctness topic. In all the cases, the language accepted by the automata obtained with our approach was the very same language that the automata obtained from the examples in the literature. In other words, they turn out to be equivalent to strengthen the correctness of our approach.

Taking performance and efficiency as parameters to validate the method, we compared not only the time consumed to obtain the controllers of the systems, but also the complexity of the algorithms involved and the size of the automata. It can be stated that these three points are extremely relevant. Although FVS tableau algorithm holds worst times (factorial [3] vs exponential in linear time logics or polynomial in branching logics [39]), considering the complexity of the algorithm, we can state the execution time is similar in all the case studies. Regarding the size of the automata, once again although FVS automata are slightly bigger [3], the difference does not resonate in the final results as the execution time to achieve need to produce them are similar enough. All these considerations are fully explained in each case of the study section.

A particular point to address and analyze is why our synthesis scheme conducts to more states when compared to the other approaches. This is a compelling future line of research to improve the efficiency of our work.

The settings of the experiments were the following. We ran our experiments in a Bangho Inspiron5458, with a Dual Core i5-5200U and 8GB RAM memory. We replicate the exact examples found in the literature([30, 32]), comparing the reported results with ours. To conclude, all the source files and our tool can be inspected at the FVS website: https://gitlab.com/fernando.asteasuain/fvsweb .

8 Related Work

We divided this section taking into consideration Open System and graphical notations related work (Section 8.1) and Branching Behavior related work in Section 8.2.

8.1 Open Systems and Graphical notations

In this category, several approaches can be mentioned. To begin with, there exist several graphical specifications languages based on events like FVS. For example, TimeEdit [37] or Graphical Interval Logic (GIL) [14]. An well grounded survey on graphical and animated languages for formal verification is introduced in [49]. Different case studies are used as illustrative support, and a set of characteristics are introduced in order to define a graphical language as a proper one. However, these languages are not focused on modeling behavior in Open Systems.

Work in [20] uses the concept of fluent to relate the occurrence of events and predicate about the expected behavior of the system. A fluent represents an ongoing behavior, with a set of starting and ending events. We believe there is a possible contribution combining fluents and FVS scenarios for specifying behavior in Open Systems. Finally, GR(1) synthesis has been used and extended to different contexts and for different application domains like e-commerce applications, web services systems, assume-Guarantee Scenarios, and also systems considering controllers handling failures [15, 31]. However, we consider that FVS expressivity can be a distinguishable feature among these and other similar approaches.

Work in [30, 45] presents a procedure to synthesize behavior using only expressible behavior employing the specification patterns [16]. As it was earlier mentioned in this work in Section 3, we combine FVS with this technique to synthesise behavior. In addition, we include some patterns that were left out in these works aside from specifying all types of properties, not only those included in the specification patterns.

In [47] an interesting approach is presented where specification models are executed and latter on verified employing on the fly model checking tools. It would be interesting to combine this technique with synthesis behavior as studied in this work. An advanced technique employing Petri Nets is presented in [48]. In this approach, Petri Nets acts as a intermediary language between models so that a system combining different behavior specifications sources can be formally verified with an unique semantics.

8.2 Branching Reasoning

The work introduced in [42] presents an extension of GR(1) with branching operators denominated GR(1)*. This extension is specially appealing because it is more expressive than GR(1)and the increase of complexity in the algorithms involved is negligible. FVS holds more expressive power at the expense of more complexity and performance costs. Other interesting line of research is introduced in [43]. It presents a captivating scheme employing Conditional Live Sequence Charts (LSC) [46] using a statistical data-mining algorithm. It would be interesting to explore the combination of FVS with conditional LSC.

The language PSL [17, 23] 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 direction of our work, they acknowledge the need and value of reviewing and validating properties specification. However, this feature is achieved by introducing tools built around PSL [8, 9]. So, contrary to Branching-FVS, validation capabilities require tool support and cannot be obtained directly from PSL specifications.

On top of SPL, work in [29] 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 language 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 [27], which make easier the behavioral exploration and specification of systems.

In [32] a tool for embedded control software synthesis named PESSOA is presented. Systems are defined using a domain specific language based on a classic logical calculus and requirements are described as trajectories over open sets. The tool provides an interesting visualization of the execution of the system easing the behavioral monitoring process. However, behavior of the system can only be described by using four template properties. In Branching FVS, all type of properties can be stated. In PESSOA the templates are implemented in an operational way combining logical calculus and constructors resembling programming language instructions. In our approach, we adopted a more declarative way to specify behavior since we aim to capture requirements nearer to the way they are described in natural language instead of "programming" them [27].

Real-time monitoring of the timed LTL (TLTL) logic is studied in [7]. 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 [13]. Other known realtime extensions of

LTL are Temporal logics MTL [25] and MITL [1].

Work in [21] 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 in comparing usability, flexibility and expressivity of Petri Nets, temporal logic and other approaches like Branching FVS constitute an appealing line of research to address in the mid-term future. Other approach employing Petri Nets is [50]. The Maude model checker is presented in [51]. Both linear and temporal logics can be used as input in this model checker. Although FVS also features graphical representations of behavior a comparison between both approaches expressive power could trigger interesting points to expand our research line.

Among branching logics extensions with formal verification purposes in other domains it is worth mentioning works like [28] or MCK [19]. 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 there is an opportunity for Branching FVS in this domain given its flexibility and expressive power.

We now briefly describe some possible research lines to continue this work. First of all, performance and complexity of the algorithms used must be addressed. One way to optimize this problem is trying to reduce the overhead introduced in the pipe of tools used to obtain a controller. Another captivating future research line is to analyze why FVS scheme conducts to bigger automata considering the number of states. Analyzing the reasons behind this fact might lead to significant improvements of our executions times.

We would like to compare Branching FVS with other known notations such as Petri Nets or other CTL extensions like [28, 29] taking into account issues like usability, flexibility, and expressive power. We also would like to continue exploring Branching-FVS in the hardware verification domain.

Finally, inspired in the work of [28] we would like to explore Branching-FVS and formal verification in the Artificial Intelligence domain.

9 Conclusions and Future Work

In this work, we showed how FVS is able to handle behavior in Open Systems as well as denoting branching type properties. We showed how FVS specification can be used as input to automatically build a controller from its specification, consolidating a crucial giant step from specification to implementation. Moving from the specification phase to the implementation phase by obtaining controllers for the system under study constitute an important milestone for our approach, since now our specification can now be "executed". Syntaxes and Semantics of the language are revisited introducing new features. Finally, complex, complete and industrial relevant case studies are introduced exhibiting and validating the flexibility and expressive power of FVS.

To conclude, we now briefly describe some possible research lines to continue this work. First of all, performance and complexity of the algorithms used must be addressed. One way to optimize this problem is trying to reduce the overhead introduced in the pipe of tools used to obtain a controller. Another captivating future research line is to analyze why FVS scheme conducts to bigger automata considering the number of states. Analyzing the reasons behind this fact might lead to significant improvements of our executions times.

We would like to compare Branching FVS with other known notations such as Petri Nets or other CTL extensions like [29, 28] taking into account issues like usability, flexibility, and expressive power. We also would like to continue exploring Branching-FVS in the hardware verification domain.

Finally, inspired in the work of [28] we would like to explore Branching-FVS and formal verification in the Artificial Intelligence domain.

10 Acknowledgments

This work was partially funded by UNDAVCYT 2014 and UAI-CAETI.

References

[1] R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

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

[3] F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2):239–274, 2017. doi:10.1007/s00766-015-0242-2

[4] F. Asteasuain, F. Calonge, F. Diaz, F. Dangiolo, and P. Gamboa. Expressing early behavior specifications with branching visual scenarios. In *CONAIISI, Argentina*, ISSN 2347-0372, 2018.

[5] F. Asteasuain, F. Calonge, and M. Dubinsky. Exploring specification pattern based behavioral synthesis with scenario clauses. In *2018 CACIC ISBN 978-950-658-472-6*, pages 22,34. CACIC, Argentina, 2018.

[6] F. Asteasuain and F. Tarulla. Exploring architectural model checking with declarative specifications. In *CACIC*, 2017.

[7] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.

[8] S. Ben-David and A. Orni. Property-by-example guide: a handbook of psl/sugar examples. *PROSYD deliverable D*, 1(3), 2005.

[9] R. Bloem, R. Cavada, C. Esiner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and assurance tool. Technical report, Technical Report Deliverable, 2005.

[10] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar. Synthesis of reactive (1) designs. 2011.

[11] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model checking for extended timed temporal logics. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326. Springer, 1996.

[12] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[13] B. d'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.

[14] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.

[15] N. DIppolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran*, 22(9), 2013.

[16] M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[17] C. Eisner and D. Fisman. *A practical introduction to PSL*. Springer Science & Business Media, 2007.

[18] O. Friedmann and M. Lange. Solving parity games in practice. In *International Symposium on Automated Technology for Verification and Analysis*, pages 182–196. Springer, 2009.

[19] P. Gammie and R. Van Der Meyden. Mck: Model checking the logic of knowledge. In *International Conference on Computer Aided Verification*, pages 479–483. Springer, 2004.

[20] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT*, volume 28, pages 257–266. ACM, 2003.

[21] I. Grobelna, R. Wiśniewski, M. Grobelny, and M. Wiśniewska. Design and verification of real-life processes with application of petri nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(11):2856–2869, 2016.

[22] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[23] IEEE-Commission et al. Ieee standard for property specification language (psl). *IEEE Std 1850-2005*, 2005.

[24] M. Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[25] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[26] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *ESEC-FSE*, pages 305–314, 2009.

[27] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering*, pages 249–262, IEEE,2001.

[28] A. Lomuscio, C. Pecheur, and F. Raimondi. Automatic verification of knowledge and time with nusmv. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1384–1389. IJCAI/AAAI Press, 2007.

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

[30] S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): a case study. *arXiv preprint arXiv:1602.01172*, 2016.

[31] S. Maoz and Y. Saar. Assume-guarantee scenarios: Semantics and synthesis. In *MODELS*, pages 335–351. Springer, 2012.

[32] M. Mazo, A. Davitian, and P. Tabuada. Pessoa: A tool for embedded controller synthesis. In *International Conference on Computer Aided Verification*, pages 566–569. Springer, 2010.

[33] P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for designing and verifying architectural specifications. *IEEE TSE*, 35(3):325–346, 2009.

[34] V. Raman, N. Piterman, and H. Kress-Gazit. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *2013 IEEE International Conference on Robotics and Automation*, pages 4075–4081. IEEE, 2013.

[35] S. Schewe. Solving parity games in big steps. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 449–460. Springer, 2007.

[36] G. E. Sibay, S. Uchitel, V. Braberman, and J. Kramer. Distribution of modal transition systems. In *International Symposium on Formal Methods*, pages 403–417. Springer, 2012.

[37] M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Requirements Engineering*, pages 14–22, IEEE, 2001.

[38] 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.

[39] M. Y. Vardi. Branching vs. linear time: Final showdown. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22. Springer, 2001.

[40] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

[41] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

[42] G. Amram, S. Maoz, and O. Pistiner. Gr (1)*: Gr (1) specifications extended with existential guarantees. In *International Symposium on Formal Methods*, pages 83–100. Springer, 2019.

[43] D. Fahland, D. Lo, and S. Maoz. Mining branching-time scenarios. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–453. IEEE, 2013.

[44] E. Firman, S. Maoz, and J. O. Ringert. Performance heuristics for gr (1) synthesis and related algorithms. *Acta Informatica*, 57(1):37–79, 2020.

[45] S. Maoz and J. O. Ringert. Gr (1) synthesis for ltl specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 96–106, 2015.

[46] G. Sibay, S. Uchitel, and V. Braberman. Existential live sequence charts revisited. In *Proceedings of the 30th international conference on Software engineering*, pages 41–50, 2008.

[47] B. Horváth, B. Graics,H.Ákos, Z.Micskei, V. Molnár, I.Ráth, L.Andolfato, I.Gomes, and R.Karban Model checking as a service: towards pragmatic hidden formal methods In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, 2020.

[48] M.Haustermann, D.Mosteller, and D.Moldt Model Checking of Synchronized Domain-Specific Multi-formalism Models Using High-Level Petri Nets In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 230–249, Springer, 2021.

[49] C.Ponsard,and J.Deprez Survey and Consistency Checking of Formal Requirements Animations In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 364–370, IEEE, 2021.

[50] B.Finkbeiner,M.Gieseking, J.Hecking-Harbusch, and R. Olderog Model checking branching properties on Petri nets with transits In *International Symposium on Automated Technology for Verification and Analysis*, pages 394–410, Springer, 2020.

[51] R.Rubio, N.Martí-Oliet, I.Pita, and A.Verdejo Strategies, model checking and branching-time properties in Maude. In *Journal of Logical and Algebraic Methods in Programming*, v123, Elsevier, 2021.