Exploring Specification Pattern based Behavioral Synthesis with Scenario Clauses ⋆

Asteasuain Fernando[1,2], Federico Calonge[1], and Manuel Dubinsky[1]

[1] Universidad Nacional de Avellaneda, BsAs, Argentina,
fasteasuain,fcalonge,mdubinsky@undav.edu.ar,
www.undav.edu.ar
[2] Universidad Abierta Interamericana, CAETI
BsAs, Argentina

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. In this work we present an extension of the FVS language to denote behavior in open systems. By relying on an existing behavioral synthesis technique based on the specification patterns we show how FVS specification can be used as input to automatically build a controller from its specification.

Keywords: Open Systems, Behavioral Specifications, Synthesis

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 as model checking [5] from the academic to the industrial world [11, 10]. On the other side, the increasing demand of Open Systems [7, 10, 12] calls for the creation of tools that assists the software engineer in the complex task of specifying and describing the expected behavior of the system. There is an natural challenge involved when dealing with Open Systems since actions beyond the control of the system must be considered, in contrast to systems known as *closed* where all the events to occur are handled entirely by the system. Open Systems interact with an environment which generates events (non controllable by the system) which may impact in its behavior, which constitutes an affect 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 [12]. In the case of reactive synthesis, an implementation is typically given as an automaton that accepts input from the environment

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

(e.g., from sensors) and produces the system's output (e.g., on actuators). By construction the input and output assignments of every infinite run of the automaton satisfy the specification it was synthesized from [12]. In order to reduce the time complexity of the algorithms involved work in [4] suggested the General Reactivity of Rank 1 (GR(1)) fragment of LTL, which has an efficient polynomial time symbolic synthesis algorithm. GR(1) is a strict assume/guarantee subset of LTL, comprised of constraints for initial states, safety propositions over the current and successor state and assertions about what should hold infinitely often also known as justice constraints. A GR(1) synthesis problem is defined as a game between a system player and an environment player [4]. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis have been presented in [4].

However, these approaches have some limitations regarding the specification language used. Most of them are based on temporal logics such as LTL (Linear Temporal Logic) and some extensions as fluents [9]. The expressivity of these notations has been challenged by the community [9, 2, 14, 6, 15] so there is a need to develop more expressive specification languages.

Given this context in this work we explore the FVS (Feather Weight Visual Scenarios) [2, 1] specification language in the context of Open Systems and behavior synthesis. 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) [2]. In [1] all the specification patterns [8] 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 [3].

The first step of FVS into the world of Open Systems and Behavior Synthesis is based on a technique introduced in [12], which is guided by the usage of specification patterns [8]. The mentioned paper presents an automated, sound and complete translation of most of the the specification patterns [8] to the GR(1) form. Although at this point this open flavour of FVS works only for systems whose behavior can be described using exclusively the specification patterns, the kind of properties covered by these patterns is useful enough to denote most of the common behavior [8]. What is more, the technique described in [12] does not include three specification patterns which are addressed in this work.

Summing up, the contributions of this work can be stated as:

- We introduce a simple extension of FVS to consider Open Systems.
- We rely on an existing technique based on the specification patterns so that FVS specifications can be used to synthesise behavior and automatically build a controller from its specification.
- Our approach is shown in action by modeling an attractive case of study.
- We present a version of our tool GTxFVS including these new features.
- We incorporate three specifications patterns that were left out in the original technique.

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 while section 4 shows our approach in action by modeling a case of study. Section 5 details how we incorporate the three missing patterns. Finally, section 6 introduces some threats to validity and also discuss future and related work whereas section 7 presents the conclusions of this work.

2 Background

In this section we will informally describe the standing features of FVS [2]. The reader is referred to [2] for a formal characterization of the language. FVS is a graphical language based on scenarios. Scenarios are partial order of events, consisting of points, which are labeled with a logic formula expressing the possible events occurring at that point, and arrows connecting them. An arrow between two points indicates precedence 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 and B*. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxes [2] . 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.

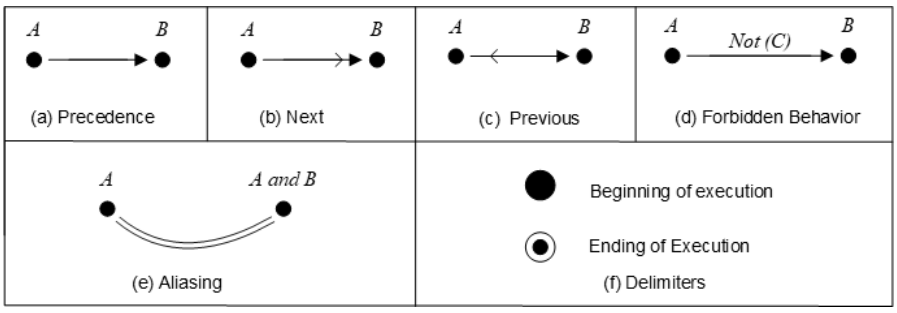

Fig. 1. FVS Basic Features

We now introduce the concept of **FVS rules**, a core concept in the language. Roughly speaking, a rule is divided into two parts: a scenario playing the role of an antecedent and at least one scenario playing the role of a consequent. The intuition is that whenever a trace matches" a given antecedent scenario, then it must also match at least one of the consequents. In other words, rules take the form of an implication: an antecedent scenario and one or more consequent scenarios. Graphically, the antecedent is shown in black, and consequents in grey. Since a rule can feature more than one consequent, elements which do not belong to the antecedent scenario are numbered to identify the consequent they belong to. Two examples are shown in Figure 2 modeling the behavior of a client-server system. The rule in 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 at the right margin of Figure 2 dictates that every granted request must be logged due to auditing requirements.

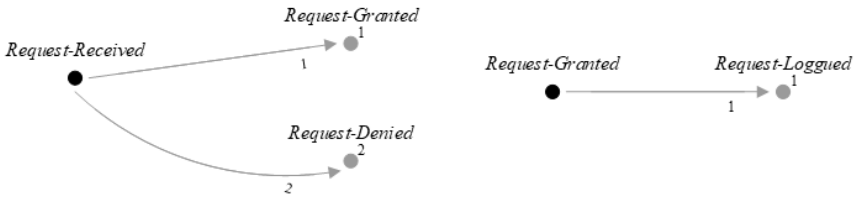

Fig. 2. FVS Rules examples

3 Open FVS and Behavior Synthesis

We now briefly introduce a simple extension to the FVS language to handle open systems. This is achieved by introducing a new type of events, which are considered as environment events not controlled by the system. Semantics of FVS (see [2]) is based on the notion of morphisms between scenarios. In particular morphisms between scenarios allow to determine whether a system trace satisfies a rule and semantics of the language is defined by the set of traces that fulfills all the specified rules. We first define the concept of scenarios considering non controllable events. The set of events is divided into controllable events and non controllable events. An FVS scenario including non controllable events in their alphabet is described by the following definition

Definition 31 (FVS Scenario) *An FVS* scenario *is a tuple* $\langle \Sigma, P, \ell, \equiv, \not\equiv, <, \gamma \rangle$, *where:*
S1: *Σ is a finite set of propositional variables standing for types of events such that $\Sigma = \Sigma_c \bigcup \Sigma_{uc}$ where Σ_c represents controllable events and Σ_{uc} non controllable events*

S2: *P is a finite set of points;*
S3: *$\ell : P \to \mathcal{PL}(\Sigma)$ is a function that labels each point with a given formula;*
S4: *$\equiv \ \subseteq P \times P$ is an equivalence relation;*
S5: *$\not\equiv \ \subseteq P \times P$ is an asymmetric relation among points;*
S6: *$<\subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \setminus \{\langle \mathbf{0}, \infty \rangle\}$ is a precedence relation between points, where $\mathbf{0}$ and ∞ represent the beginning and the end of execution, respectively;*
S7: *$\gamma : (\not\equiv \cup <) \to \mathcal{PL}(\Sigma)$ assigns to each pair of points, related by precedence or separation, a formula which constrains the set of events occurrences that may occur between the pair.*

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.Given that non controllable events values are considered as inputs to the system the usual morphism definition specified in [2] can be applied.

Definition 32 (Morphism) *Given two scenarios $\mathcal{S}_1$, $\mathcal{S}_2$ (assuming a common universe of event propositions), and f a total function between P_1 and P_2 we say that f is a morphism from $\mathcal{S}_1$ to $\mathcal{S}_2$ (denoted $f : \mathcal{S}_1 \to \mathcal{S}_2$) iff*
M1: *$\ell_2(a) \Rightarrow \ell_1(\mathsf{p})$ is a tautology for all $\mathsf{p} \in P_1$ and all $a \in P_2$ such that $a \equiv_2 f(p)$;*
M2: *$\gamma_2(f(\mathsf{p}), f(\mathsf{q})) \Rightarrow \gamma_1(\mathsf{p}, \mathsf{q})$ is a tautology for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M3: *if $\mathsf{p} \equiv_1 \mathsf{q}$ then $f(\mathsf{p}) \equiv_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M4: *if $\mathsf{p} \not\equiv_1 \mathsf{q}$ then $f(\mathsf{p}) \not\equiv_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$;*
M5: *if $\mathsf{p} <_1 \mathsf{q}$ then $f(\mathsf{p}) <_2 f(\mathsf{q})$ for all $\mathsf{p}, \mathsf{q} \in P_1$.*

3.1 FVS specifications as input to a synthesis scheme

Work in [2] describes a tableau algorithm which translates FVS scenarios into Büchi automata. Also related to FVS, work in [1] shows how FVS is expressive enough to denote all the specification patterns introduced by [8]. On the other hand, work in [12] proposes a synthesis scheme for a set of LTL (linear temporal logic) formulas, the set of formulas describing the behavior of all the specification patterns. These formulas were translated into General Reactivity of Rank 1 (GR(1)) fragment of LTL (due to performance and complexity reasons) and then synthesised following an usual game based strategy.

Given this context, we translated all the FVS scenarios describing all the specification patterns. The resulting Büchi automata were then used to feed the synthesis scheme described earlier. In this way, a controller can be automatically built synthesizing the behavior of a system using as input FVS specifications. Although this open flavour of FVS works only for system whose behavior can be described using only the specification patterns, the kind of properties covered by these patterns is an attractive portion of the cake [8]. It is worth mentioning that this version of FVS is fully available in the current status of GTxFVS[3], the software tool which implements all the features of the language.

4 Case of Study

In this section we analyze the case of study introduced in [12], the "Lego Forklift" example. The behavior of the model is given in section 4.1 while the synthesis procedure is detailed in section 4.2.

4.1 The "Lego Forklift" example

The behavior of the system is described in [12]. In few words, the forklift has three sensors: one sensor to determine whether it is at a station and two distance sensors to detect obstacles and cargo. It also has three motors, to turn the left and right wheels and to lift the fork. Values read by the sensors are provided as inputs to component ForkliftController and its outputs are commands controlling the motors. The properties modeling the behavior of the system is described using assumptions and guarantees based on the scheme introduced in [12]. In particular, three guarantees and two assumptions properties are specified. The first guarantee property is a simple safety one: *if the forklift detects an obstacle, both motors are stopped*. The second one is a liveness property: *cargo will always eventually be delivered*. Finally, the third guarantee property establishes that *the forklift has to leave its pick-up station between lifting and dropping cargo*. Rules in figure 3 denote the behavior of these three properties.

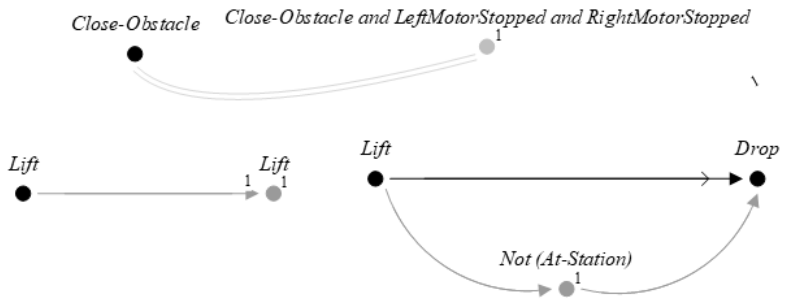

Fig. 3. Guarantees properties for the Lift Controller

The rule in the upper part of figure 3 says that every time the *Close-Obstacle* event occurs, then motors left and right should be stopped (denoted by the occurrence of the events *LeftMotorStopped* and *RightMottorStopped*). Note that these events are modeled as occurring simultaneously as it is indicated in the systems requirements specified in [12]. The second rule (in the left margin of the lower part of figure 3) simply states that *Lift* event will always eventually occurs, a classical liveness property. Finally, the rule in the right margin of the lower part of figure 3 demands the forklift to leave the station (modeled as the

negation of the *At-Station* event) between the occurrence of the *Lift* event (the forklift lifted an item) and the occurrence of the *Drop* Event (the forklift dropped an item).

The forklift specification is completed with two assumptions on the environment. One assumption is that going forward with both motors will lead to reaching a station unless the motors are not going forward anymore. As it is noted in [12] in order to satisfy this assumption an adversary environment can prevent the forklift from reaching a cargo station by presenting obstacles forcing the forklift to stop. Hence, an additional assumption for a well-behaved environment is added: in the given setting it is reasonable to expect that between two stations, the forklift may be blocked by obstacles at most twice.

Rules in figure 4 model these two assumptions. The rule in the top of the figure tackles the first assumption: once left and right motors are on, then the forklift will reach an station if *MotorStopped* event do not occur. The second rule focuses on the two times blocked maximum restriction. The rule contains four possible consequents once the forklift has left the station and the first obstacle occurs. Consequent 1 deals with the situation that no other obstacle is detected until the forklift arrives at the station. Consequent 2 do consider the occurrence of a second obstacle, but no other obstacle must occur until the station is reached. Consequents three and four follows a similar analysis but considering that the system may be stopped for some reason before reaching an station. This is why the *End* point of traces is included. So, before reaching a new station or the system is stopped the forklift may be blocked by obstacles at most twice.

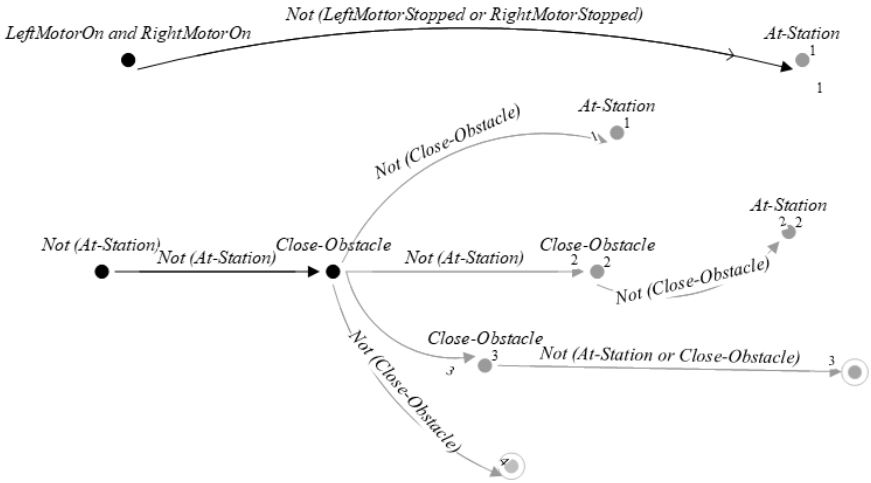

Fig. 4. Environment assumptions properties for the Lift Controller

4.2 Synthesising behavior from FVS scenarios

By relying on the technique introduced in [12] as detailed in section 3.1 we build a controller for the *ForkLift* case study taking as input FVS scenarios. In particular, three properties of the *Forklift* example matches with three specification pattern: the third guarantee property corresponds to the Existence pattern with *Between Q and R* scope, the first assumption with the Response event with *Global* scope and and second assumption is an instance of the Bounded Existence pattern with *After Q until R* scope.

5 The three missing patterns

Work in [12] proposes a synthesis scheme based on the GR(1) fragment of LTL (Linear Temporal Logic) for most of the specification patterns and scopes introduced by [8]. However, three patterns are not currently supported by their technique. These patterns are the followings ones: *Precedence pattern with After Q scope*, *Response Chain pattern (with one stimuli and two responses) with After q until r scope* and *the Constrained Chain pattern (responses s, t without z responds to the p stimuli event) with After q until r scope*. These patterns are not included since no Deterministic Büchi automata exists to describe their behavior and the mentioned technique requires this kind of automaton as input.

We propose an alternative approach to circumvent this issue. The FVS scenarios for these patterns (the FVS scenarios for all the specification patterns are shown in [1]) can be later translated into non determinist Büchi automata. By employing an advanced tool for manipulating diverse kinds of automata named GOAL [16] we translated these automata into *Deterministic Rabin automata*. Since synthesis algorithms are also incorporated into the GOAL tool using Rabin automata as input, we could incorporate these patterns so as to cover all the specification patterns. Furthermore, since our tool GTxFVS [3] can interact with the GOAL tool, all the patterns and their synthesized behavior are available for the software engineer.

Being able to handle all the specification patterns including these three patterns is important in terms of completeness and expressiveness but it also has its drawbacks. Most of the algorithms involved in automata manipulation and synthesis depend on the size of the automata and have performance and complexity issues. In addition, the synthesis procedure proposed in [12] introduces extra variables and therefore the the costs are higher.

The problem is exacerbated with these patterns using *Deterministic Rabin* automata since their size is really important. For example, the Rabin automaton for the *Response Chain pattern (with one stimuli and two responses) with After q until r scope* consists of 94 states and 1003 transitions. These automata can be simplified, but their size remains important. The simplified automaton for the latter pattern consists of 63 states and 611 transitions. Table 1 describes the automata size for these three patterns. The abbreviations s and t stands for states and transitions respectively.

Table 1. Automata size Complexity for the missing patterns

Pattern	Not Det. Buchi	Det. Rabin	Simplified Rabbin
Precedence after	5s and 13t	11s and 41t	11s and 41t
Response Chain after until	9s and 37t	94s and 1003s	63s and 611t
Constrained Chain after until	8s and 34t	47s and 611t	35t and 411t

Using the proposed technique to incorporate these patterns (using Rabin automata and the GOAL tool) we add an extra assumption for the ForkLift example and obtained a new controller that includes this new behavior: *after leaving an station, all the Drop events must be preceded in time with an Lift event*. This behavior is an instance of the *Precedence pattern with After Q scope*. This gain in expressiveness come with an cost in terms of performance due to the size of the involved automata. The time consumed to obtain this new controller took eight times more than the previous one. As it is mentioned in [8, 12] these patterns are far from being the most used ones. However, we believe it is desirable being able to express them despite the fact they are time consuming.

6 Some Threats to Validity, Future and Related Work

In this section we briefly discuss some issues that can be seen as threats to validity to the results introduced in 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 using the mentioned synthesis scheme, which we aim to cover in future work. A second issue is related to FVS as an open system language specification. Synthesis scheme is built upon the work introduced in [12], which only covers properties included in the specification patterns. In this sense, to obtain a complete synthesis scheme for properties beyond the patterns is clearly an appealing challenge regarding future work. Despite these facts, we believe that the obtained results are promising enough to consider FVS as an attractive alternative to describe behavior in Open Systems.

Regarding related work several approaches can be mentioned. To begin with, there exist several specifications graphical languages based on events like FVS. For example, TimeEdit [15] or Graphical Interval Logic (GIL) [6]. However, these languages are not focused on modeling behavior in Open Systems.

Work in [9] uses the concept of fluent to relate occurrence of events and predicate about systems behavior. 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 in different contexts and for different application domains [13, 7]. However, we consider that FVS expressivity can be a distinguishable feature among these and other similar approaches.

7 Conclusions

In this work we present an open system flavour of FVS. By relying on a synthesis technique proposed by [12] we showed how FVS specification can be used as input to automatically build a controller from its specification. This work constitutes an early first step for FVS in the open systems specifications since our approach currently applies for those properties denoted by the specification patterns. As it was mentioned in Section 6 we would like to augment the kind of properties that can be expressed in future work. We also incorporated in our approach three specification patterns that were not included in [12]. This gain in expressibility come with a cost in terms of complexity and computing time. In this sense, we believe expressivity is a key factor when denoting behavior in early stages.

References

1. F. Asteasuain and V. Braberman. Specification patterns: formal and easy. *IJSEKE*, 25(04):669–700, 2015.
2. F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2):239–274, 2017.
3. F. Asteasuain and F. Tarulla. Exploring architectural model checking with declarative specifications. In *CACIC*, 2017.
4. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar. Synthesis of reactive (1) designs. 2011.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
6. 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.
7. N. DIppolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran*, 22(9), 2013.
8. M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
9. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT*, volume 28, pages 257–266. ACM, 2003.
10. I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *ESEC-FSE*.
11. A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE*, 2001.
12. S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): a case study. *arXiv preprint arXiv:1602.01172*, 2016.
13. S. Maoz and Y. Saar. Assume-guarantee scenarios: Semantics and synthesis. In *MODELS*, pages 335–351. Springer, 2012.
14. P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for designing and verifying architectural specifications. *IEEE TSE*, 35(3):325–346, 2009.
15. M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Re*.
16. 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.