Exploiting Anti-Scenarios for the Non Realizability Problem *

Fernando Asteasuain[1,2], Federico Calonge[1], and Pablo Gamboa[2]

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

Abstract. Behavioral synthesis is a technique that automatically builds a controller for a system given its specification. This is achieved by obtaining a winning strategy in a game between the system and the environment. Behavioral synthesis has been successfully applied in modern modularization techniques such as Aspect Orientation to compose each particular view of the system in a single piece. When a controller can not be found the engineer faces the Non Realizability Problem. In this work we exploit FVS a distinguishable feature in the FVS specification language called anti-scenarios to address this problem. Several case studies are introduced to show our proposal in action. Additionally, a performance analysis is introduced to further validate our approach.

Keywords: Aspect Orientation, Behavioral Synthesis

1 Introduction

Modern modularization techniques have emerged in the past years aiming to improve how the different features supported by a system interact to provide its expected behavior. Among them, Aspect Orientation [18, 22] or Feature Oriented Programming [3, 29] earned a distinguished reputation since they have been applied successfully in several challenging domains such as software protocols, software architectures, hardware and robotic controllers or artificial intelligence [3, 11, 16, 22, 36], just to mention a few.

In essence what these techniques propose is to conceive a system as the interaction and combination of its individual features. These features are also called view or aspects, depending on the particular methodology to be employed. For example, in the classic ATM example the system is built combining diverse views such as security, availability, efficiency, data integrity or transaction management. Each view captures a portion of the behavior of the system. In this way the software engineer can focus on each one without being distracted with other irrelevant details that do not correspond to the view in turn being specified. This

* Corresponding Author: Fernando Asteasuain fasteasuain@undav.edu.ar ORCID: 0000-0002-5498-6878

kind of approach is also present in multiple ways in other software engineering tools and techniques. Actors when defining use cases or roles when defining user stories represent different ways (views) of interacting with the system. Similarly, UML's sequence diagrams aim to model localized behavior and also can be seen as a mechanism to specify a particular behavioral perspective of the system.

The critical point when developing system is to articulate and combine each particular view in order to produce the complete behavior of the system. This is indeed tricky since most of the times these views interfere with each other constituting a possible source of conflicts, problems and bugs. These conflicts should not be underestimated since the correct behavior of the system may depend on how they are addressed. For example, in the ATM system the encryption view must act (ie, must encrypt the message) before any message is sent from the ATM to the bank. Similarly, the encryption perspective may constitute a menace to achieve successfully the efficiency view, since delays are introduced when performing transactions.

In the Aspect Orientation approach each view is named an *Aspect* and the process of combining aspects behavior is named *Weaving*. The problem of detecting and resolving aspects interaction has been addressed by the software engineering community [7, 14, 26, 30, 33]. Nonetheless, most of them rely on syntactic mechanisms and the semantic application of aspects is hard to analyze, explore or reason about [6, 21, 32]. In this sense, some approaches [6, 28] have taken a different road to weave aspects employing a technique known as behavioral synthesis [8, 13]. Contrary to the usual development of systems, where the implementation of the system is analyzed whether its fulfills its specification or not, when employing behavioral synthesis the system is built in such a way that the specification is satisfied by construction. In general, the output of the synthesis procedure is an automaton called the controller of the system that receives input from the environment (typically this information is provided by sensors) and produces instructions to actuators and then waits to receive the new information from the environment to initiate the loop one more time. The controller is obtained employing algorithms from the artificial intelligence and game theory domains. The objective is to produce a winning strategy that no matters what action is taken by the environment the systems always find a way to achieve its goals. Efficient symbolic algorithms for controller synthesis have been presented in [8, 27].

A key issue when dealing with behavioral synthesis is how to proceed when a controller can not be derived from the specification. This is known as realizability checking. When the specification is non realizable the specifier must identify the source of the problem. Generally, this is achieved manually or employing sophisticated tools. Some approaches like [2] rely on inductive logics , work in [23] employs the concept of *counterstrategies* and work in [25] adds additional information into the specification as *conflict* and *recovery* sets of requirements. However, the specification language used in these approaches does not provide by itself a mechanism to help the specifier to identify the problem.

Given this context in this work we propose to exploit a distinguishable feature of the FVS language to address non realizability in the aspect oriented synthesis world. FVS can automatically build anti-scenarios from any property described in the language. This gives to the specifier an intuitive, visual and crucial information about all the possible ways things can go wrong and violate a property. This work illustrates how anti-scenarios can be employed to tackle non realizability in a friendly manner.

FVS is a declarative language based on graphical scenarios and features a flexible and expressive notation with clear and solid language semantics. In [6] we showed how FVS can be used to denote, compose and synthesize aspect oriented behavior, including dealing with properties that can not be expressed with Deterministic Büchi automata, which are excluded in other approaches like [27]. We now introduce a new edge on our methodology by exploiting FVS's anti-scenarios to alleviate the non realizability problem.

We also conducted a performance analysis to measure the effectiveness of our approach and explored a new tool named Acacia+ [9] to achieve our goals. The results show that FVS turns out to be a competitive framework among others known approaches.

To sum up, the contributions of this work can be summarized as follows:

- FVS anti-acenarios are employed to address the non realizability problem in behavioral synthesis.
- A performance analysis is shown to compare FVS efficiency against other tools.
- A new external tool is explored to obtain controllers.
- FVS is shown in action in new case studies in several applications and relevant domains

The rest of this work is structured as follows. Section 2 briefly introduces the FVS language, its usage in the Aspect Orientation domain and how the behavioral synthesis is achieved. Section 3 presents the case studies and also discusses related work. Section 4 introduces the performance analysis. Finally, Section 5 enumerates future work whereas Section 6 present the conclusions of this work.

2 Background

In this section we will informally describe the standing features of FVS [4, 5]. The reader is referred to [5] for a formal characterization of the language.

FVS is a graphical scenarios where scenarios and rules shape the expected behavior of the system. FVS scenarios consists of points and relationships between them. Points can be labeled with the events that occur in those points. In fact, a points' label consist of a logic formula that summarizes the events occurring on that moment. Two kind of relationships relate points: precedence, to indicate that an event occurs before another one, and constraining, which is

used to restrict behavior between points. Finally, two special points are introduced to indicate the beginning and the end of a trace. For example Figure 1 shows a typical FVS scenario modeling the behavior of a load-balancer server example. The big black point in the left extreme represents the beginning of the execution and the last point, which is double rounded, represents the end of the execution. In particular the scenario describes a situation where the server is ready once the execution started and afterwards a task is assigned such that the server never reaches its full capacity from the moment was ready until the end of the execution.

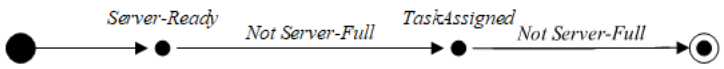

Fig. 1. A simple scenario in FVS

FVS rules can be seen as an implication relationship. They consist of an scenario playing the role of the antecedent and two or more scenarios playing the role of the consequent. The intuition is that whenever a trace matches a given antecedent scenario, then it must also match at least one of the consequents. When this happen we say that the rule is satisfied. The semantics of the language is given by the set of traces that fulfills every rule. Graphically, the antecedent is shown in black, and consequents in grey. Each item of consequents are labeled to indicate to whom consequent they belong. As an example, an FVS rule is shown in Figure 2. This rule says that if an *TaskAssigned* event occurs, then the *Server-Ready* event must occurred in the past (in other words, the server was in the *ready* state) and between these points the *Server-Full* event that indicates that the serves reached its maximum capacity did not occur.

Fig. 2. A simple FVS rule

We now introduce the concept of anti-scenarios. Anti-scenarios can be automatically generated from rule scenarios. Roughly speaking, they provide valuable information for the developer since they represent a sketch of how things could go wrong and violate the rule. The complete procedure is detailed in [10], but informally the algorithm obtains all the situations where the antecedent is found but none of the consequents is feasible. For example, for the previous rule in Figure 2 two anti-scenarios are found. In one situation a task is assigned, but the server was never ready before since the beginning of the execution. In the

other situation, the server was indeed ready, but it reaches its maximum capacity before the task is assigned. These two anti-scenarios are shown in Figure 3.

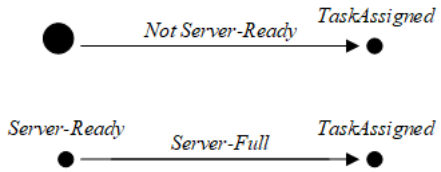

Fig. 3. Two anti-scenarios for the specified rule

2.1 FVS in Aspect Orientation and Behavioral Synthesis

Our previous work in [6] describes how FVS can be seen as an aspect oriented specification language and also how it can be connected to external tools to obtain a controller that satisfies the specification by construction. In few words, FVS rules can be seen as aspects, where the antecedent of the rule constitutes the *poincut* (those points where the aspect behavior is to be added) and the consequent represent the behavior to be introduced by the aspect.

FVS specifications are used to obtain a controller using different tools depending on the type of the property. Using the tableau algorithm detailed in [5] FVS scenarios are translated into Büchi automata. If the automata represent one of the specification patterns (excludings those that can not be represented by Deterministic Büchi automata) then we obtain a controller using a technique [27] based on the specification patterns [17] and the GR(1) subset of LTL. If that is not the case, but the automata is a Deterministic Büchi automata we either employ the GOAL [34] or the Acacia+ tool [9]. The performance analysis in Section 4 discusses the efficiency of these two tools. If the automaton is non deterministic we obtain a controller using only the GOAL tool using an intermediate translation from Büchi automata to *Rabin* automata.

3 Case Studies

In this section we show in action our approach to address non realizability in aspect oriented behavioral synthesis by modeling three different and challenging case studies. These examples are shown next in the following sub sections. Finally, Section 3.4 presents some considerations and conclusions and briefly discusses related work.

3.1 The Vessel example

This case of study is based on the functioning of several navigation transports introduced in [25]. In particular we consider the requirements from the vessel

example. We suppose that the basic specification describing the main behavior of a vessel denote the base system and then we introduce some security restrictions through the specification of an aspect named *SafeNavigation*. In particular, the *SafeNavigation* aspect considers three requirements:

1. The vessel can be in the *UnderWay* state either by starting on an engine or by sailing, but not on both modes at the same time. That is, the vessel navigates through the river either by using a engine or blowned by the wind.
2. A vessel can be constrained by her draught, but only when is navigating.
3. The vessel can be moored, but only when using the engine.

We modeled the *SafeNavigation* aspect in FVS. It contemplates three rules, one for each requirement of the aspect. The first rule says that if the *UnderWay* event occurs, then either the sailing mode was selected in the past and the engine has not been turned on since then, or viceversa, the engine was turned on in the past and the sailing mode has not occurred during that period. The FVS rule for this requirement is shown in Figure 4.

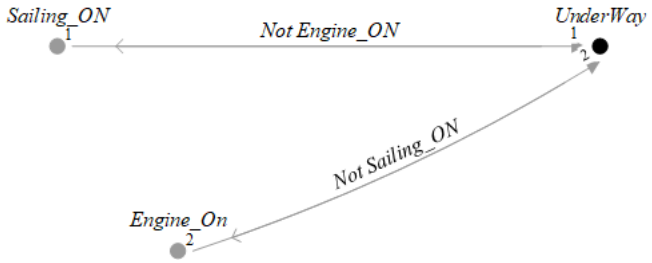

Fig. 4. FVS rule for requirement number one

The FVS rule in Figure 5 addresses the second and third requirements. The rule in the top of Figure corresponds to the second requirement whereas the rule in the botttom corresponds to the third one. In the former rule, the *Constrained by her draught* event (named *CbyDraught* to simplify) occurs, then the *UnderWay* event must had occurred in the past. The latter rule indicates that if the *Moored* event occurs then the engine was turned on in the past (consequent 1), or it was turned on later on (consequent 2).

When the *SafeNavigation* aspect is weaved with the base system a controller can not be found. This is because the environment can introduce the *Moored* event and then activate the sailing mode execution of the vessel. A trace like that requires the vessel to turn on the engine (because of requirement number three), but turning on the engine would be in conflict with requirement number one since the sailing mode was active. This behavior can be noted when exploring the anti-scenarios that the FVS tool generates from the rules shaping the *SafeNavigation* aspect easing the job for the specifier when faced to the non realizability problem. Three possible anti-scenarios are shown in Figure 6. The anti-scenarios in the

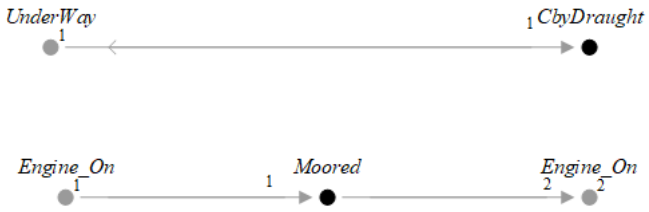

Fig. 5. FVS rule for requirements number two and three

top of the Figure corresponds to behavior that violate requirement number one: in the first one, the sailing mode is activated, afterwards the engine is turned on and finally the vessel starts to navigate. This is a behavior violation since both modes are active simultaneously. The second one is the analogous behavior but just considering the engine was turned on first. The anti-scenario at the bottom of Figure 6 violates requirement number three: the vessel is moored but the engine was neither turned on before or after its ocurrence.

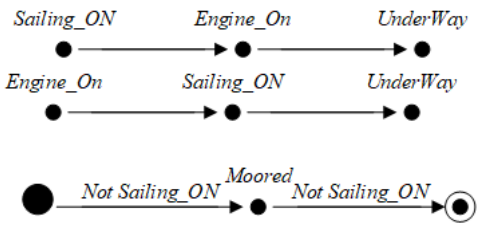

Fig. 6. Some anti-scenarios for the Safe Navigation Aspect

By just looking at the graphical information provided by these anti-scenarios the specifier can explore possible reasons that lead to the non realizability of the specification. In this case it can be noted that a trace like { *Moored*, *Sailing_ON* } would converge to a conflict since a occurrence of the *Engine_ON* event would match the first anti-scenario in Figure 6 (and therefore a violation to the specification) and at the same time the absence of the *Engine_ON* event would match the third anti-scenario in Figure 6, leading also to violation of the expected behavior. After this analysis the specifier can infer that the *Moored* event and the sailing mode are mutually exclusive. Introducing new rules for this extra requirement make the specification realizable and a controller con be found. These new rules are shown in Figure 7.

3.2 The Halt-Exception Example

This example is inspired in the case of study shown in [35]. It consists of a typical buffer implementation in a distributed environment. A producer puts data in the

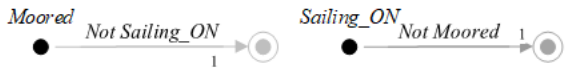

Fig. 7. New rules are introduced after exploring the anti-scenarios

buffer, it waits when the buffer is complete and sets a *Halt* flag on when it puts its last data. The consumer gets data from the buffer and waits when the buffer is empty. An exception is raised when the buffer is empty, the halt flag is on and the consumer attempts to get a data. This exception is specified with an aspect named *Halt Exception* aspect. Following the strategy presented in [35] we suppose that the specifier makes a mistake and the empty buffer condition is not included in the rule shaping this aspect. This incomplete rule is shown in Figure 8

Fig. 8. The incomplete FVS rule for the Halt Exception Aspect

This error makes the specification non realizable. As in the previous case the specifier can then look at the anti-scenario that FVS can automatically generate. The anti-scenario is shown in Figure 9. It shows how the aspect specification can be violated. In this case the sequence of events *Halt_ON* and *Get* on which the Halt Exception does not occur until the end of the execution is a violation of the rule.

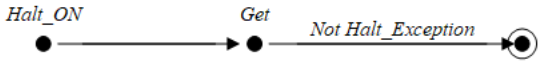

Fig. 9. Anti-Scenarios for the Halt Exception Aspect

By just looking at the sequence the specifier might detect that the empty buffer condition is not included and therefore spot the error in the specification. Adding this condition lead to the realizability of the specification and a controller is now found. The fixed rule is shown in Figure 10.

3.3 The Exam System

This example is analyzed in [6] based on the case study presented in [28]. It consists on a Exam system which monitors and regulates students taking exams.

Fig. 10. Fixed Rule for the Halt Exception Aspect

The system starts in a *Wait* state and when a students arrives it shows a welcome screen. Then, the student takes the exam. The student can fail or pass the exam, the systems shows an exit screen and enters the *Wait* state once again. Two aspects are added next: the *Tuition* aspect and the *Availability* aspect. The first one validates that only students that have paid their tuition take the exam and the second one addresses a classic liveness requirement: the welcome screen will eventually be showed for every student. The rules for these two aspects are shown in Figure 11.

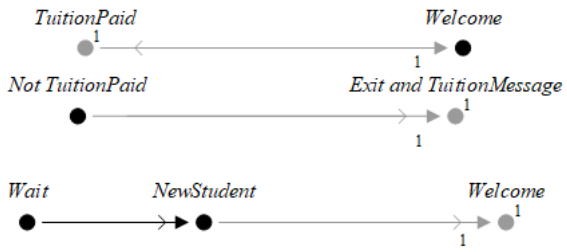

Fig. 11. Tuition and Availability aspects in the Exam System

As explained in [6] the specification containing both aspects in non realizable. If infinitely many students are received such that only a finite number of them have paid their tuition will lead the system not to enter the welcome state infinitely often, which violates its specification. Faced with this problem, the specifier might inspect anti-scenarios for both aspects. In particular, by analyzing the anti-scenario in Figure 12 the specifier could realize that not visiting the *Welcome* state infinitely often might be a too strong condition and observing that it is enough that the systems leaves the *Wait* state but not necessarily enters the *Welcome* state.

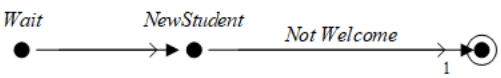

Fig. 12. An anti-scenario for the Availability aspect

The new rule introducing a more relaxed *Availability* aspect is shown in Figure 13. With this new rule a controller is found for the *Exam* system.

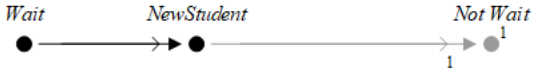

Fig. 13. A more general Availability Aspect solving non the realizability problem

3.4 Remarks, Observations and Related Work

In all of the examples anti-scenarios provide meaningful information to solve the non realizability problem. We believe that is an appealing feature since it is provided by the specification language itself and there is no need to interact with external tools. Others technique such as [23, 27] involves exchanging roles between environment and the system and try to discover in the winning strategy for the environment what behavior is obstructing the finding of a controller. Naturally, this can be achieved if anti-scenarios do not result in an effective weapon to solve the problem. Work in [25] provides a interesting solution to non realizability defining extra definitions such as *Conflict Sets* and *Recovery Sets* of Requirements. These definitions and the use of coloured automata allow the detection of possible conflicts between requirements that might lead to the non realizability of the specification. Finally, work in [2] proposes the us of inductive logic to solve non realizability. Inductive logics infers some missing information in the specification based on negative ans positive examples and its output could be used to solve non-realizability. We consider that our approach is orthogonal to these other works and could be easily combined. However, we affirm that taking the advantage of exploring anti-scenarios which are automatically obtained from the specification itself (win non costs or efforts for the specifier) is an captivating first setp to address the non realizability problem in behavioral synthesis.

4 Performance Analysis

In this section we describe some performance analysis regarding employing FVS as an aspect oriented behavioral synthesis. Since we relay on external tools to solve the game between the environment and the system to obtain a controller we are in particular interested on measuring if the automata produced by our approach is comparable and competitive with other approaches. To achieve this objective we compare the time taken to obtain a controller taken as input FVS specifications versus other approaches since the algorithms involved depend on the size of the automata.

Also, we compare the performance of the two tools used to obtain a controller in our work: *GOAL* and *Acacia+*. In previous work such as [6] we employed

the GOAL tool to resolve the behavioral synthesis problem. GOAL is a very powerful tool to rely on during the verification phase since it can handle several formalisms for automata and temporal logics and also, it implements several game solving algorithms such as [19, 20, 31, 37]. For this paper we also employ an attractive tool called *Acacia+*, an efficient tool for solving synthesis specifications implemented in Python and C. It can be easily downloaded from its site [1] and also can be executed online. Instead of employing BDDs (Binary Decision Diagrams) as the main structure of data is uses the concept of antichains [12, 15] with satisfactory results in some domains [9]. It takes as input LTL formulae as well as automata notations.

The rest of this section is structured as follows. Section 4.1 analyzes the performance of the external tools interacting with FVS, Section 4.2 compares FVS against other two approaches whereas Section 4.3 includes some final considerations regarding this performance experiment.

4.1 GOAL vs Acacia+ vs Specification Patterns

In this section we compare FVS performance using different external tools and kinds of specifications. As it is mentioned earlier in section 2 we rely on several techniques to realize the behavioral synthesis: using the GR(1) technique in [27] if the properties modeling the expected behavior of the system can be denoted with most of the specifications patterns [17], using GOAL and Acacia+ with Deterministic Büchi and finally, using GOAL with Non Deterministic Büchi.

Table 1 resumes this performance analysis. The time is measured in seconds taking the average time to obtain the controller for all the case of studies introduced in Section 3. For all the problems we divided the set of requirements as follows: set 1 contains only properties denoted by specification patterns (excepting those ones that can not be represented by Deterministic Büchi), set 2 contains properties beyond specification patterns but still represented by Deterministic Büchi automata and finally, set 3 contains properties expressible only by Non Deterministic Büchi automata.

Table 1. FVS Performance with External Tools

Set-Tool	FVS-GOAL	FVS-Acacia+	FVS-GR(1)
Set 1 sec	4 sec	2 sec	1 sec
Set 2 sec	7 sec	5 sec	-
Set 3 sec	20 sec	-	-

It can be observed that the technique introduced by [27] (denoted by the column FVS-GR(1) in Table 1) is the most efficient in time when the behavior can be expressed by most of the specification patterns. In the same row, *Acacia+* comes in second place while the *GOAL* tool ends at the bottom but close to *Acacia+*. *Acacia+* leverages the advantage moving on to the second

row, using Deterministic Büchi automata. Finally, only GOAL is able to handle requirements in set 3, which includes Non Deterministic Büchi automata. The GR(1) technique in [27] is only available for behavior denoted in the the first set of requirements. These results can definitely guide the specifier when choosing the external tool to perform the behavioral synthesis using as input FVS specifications.

4.2 Efficiency comparison

We now compare FVS against other approaches in those cases where performance times are available or can be obtained. We use only properties can could be expressed by Deterministic Büchi automata.

For the *Exam* System paper in [27] reports that 3.5 seconds is taken to obtain a controller using their approach based on the use of specification patterns. Employing FVS we obtained 4.5 seconds, since larger automata are used in our case. However, the difference is not a significative one. In [6] we add some extra requirement using specification patterns not covered in [27] since they can only be expressed with :the *Precedence pattern with After Q scope*, *the Response Chain pattern (with one stimuli and two responses) with After q until r scope* and *the Constrained Chain pattern with After q until r scope and the Precedence pattern with After Q scope*. In this case it took FVS-GOAL 33 seconds to obtain the controller.

For the *Vessel* System using *Acacia+* with its own specification language a controller is obtained in 7 seconds, whereas the tool took 11 seconds to obtain the controller when using FVS-Specifications as input. It is worth noticing that *Acacia+* specifications language is more complex since *Conflicting* and *Recovery* set of requirements need to be specified. This extra effort added to the specification was not measured for the performance experiment. In FVS no additional information need to be provided besides the rules shaping the behavior of the system.

For the *Halt-Exception* example no comparison can be made since in [35] the bug in the specification is detected using traditional model checking whereas in this work we employed a behavioral synthesis approach. Nonetheless, FVS specification can be used in a traditional model checking approach. In this domain we do not expect great differences in the elapsed time of execution since the size of the automata involved are similar [5]. Table 2 summarizes the performance analysis of this section.

Table 2. Comparison Against Other Approaches

Example-Tool	Acacia+	Patterns	FVS
Vessel Example	7 sec	-	11 sec
Exam System	-	3.5 sec	4.5 sec

4.3 Final Observations

From the performance analysis several interesting conclusions can be taken. For one side, FVS is behind in time execution against other approaches. However, the difference is not a significative one and barely has impact in the analyzed examples. On the other side, it is the only one that is able to handle all type of specifications, specially considering Non Deterministic Büchi automata. We believe this power expression enrichment of the specification language outweighs the loss in performance. Regarding anti-scenarios and the non realizability problem we believe the information provided by the anti-scenarios can solve faster the problem than feeding external tools. However, this must be concluded after an empiric study that is out of scope of this paper and addressed as future work.

5 Future Work

This work pinpoints several research line to further continue our proposal. In the first place we would like to measure the effectiveness of the use of anti-scenarios to solve the non realizability problem. This would imply the realization of an experiment that compares the results obtained by employing anti-scenarios against the usage of other techniques. The experiment should be designed to asses the percentage of non realizability problems solved and the time and resources taken to solve the problem. Secondly, we would like to employ our technique in the hardware verification and software architecture domain to take advantage of FVS's flexibility and expressive power. Finally, we would like to improve the efficiency of our tool. In order to achieve this goal the algorithm that translates FVS scenarios into Büchi automata must be analyzed aiming to reduce the size of the automata given as output.

6 Conclusions

In this work we exploit FVS's capability of automatically building anti-scenarios to address the non realizability problem when performing behavioral synthesis in aspect orientation. Anti-scenarios represent a meaningful graphical descriptions of situations and executions of the system that violates the specification and thus can be analyzed in order to solve the non realizability problem. The main advantage of this approach is that this information is provided by the specification language and there exists no need to interact with external tools that might make the process more complex. The proposal is validated with several and relevant case studies. Finally, a performance analysis is introduced to compare FVS interaction with synthesis tools and with other approaches as well.

References

1. Acacia+ web site: http://lit2.ulb.ac.be/acaciaplus/.

2. D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *2009 IEEE 31st International Conference on Software Engineering*, pages 265–275. IEEE, 2009.
3. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented software product lines*. Springer, 2016.
4. F. Asteasuain and V. Braberman. Specification patterns: formal and easy. *IJSEKE*, 25(04):669–700, 2015.
5. 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
6. F. Asteasuain, F. Calonge, and P. Gamboa. Aspect oriented behavioral synthesis. In *ISBN 978-987-688-377-1, pages 622-631, CACIC*, 2019.
7. L. M. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, 2003.
8. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar. Synthesis of reactive (1) designs. 2011.
9. A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia+, a tool for ltl synthesis. In *International Conference on Computer Aided Verification*, pages 652–657. Springer, 2012.
10. V. Braberman, D. Garbervestky, N. Kicillof, D. Monteverde, and A. Olivero. Speeding up model checking of timed-models by combining scenario specialization and live component analysis. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 58–72. Springer, 2009.
11. T. Cerny. Aspect-oriented challenges in system integration with microservices, soa and iot. *Enterprise Information Systems*, 13(4):467–489, 2019.
12. M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.
13. N. DIppolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran*, 22(9), 2013.
14. C. Disenfeld and S. Katz. A closer look at aspect interference and cooperation. In *AOSD*, pages 107–118. ACM, 2012.
15. L. Doyen and J.-F. Raskin. Improved algorithms for the automata-based approach to model-checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 451–465. Springer, 2007.
16. B. Duhoux, K. Mens, and B. Dumas. Implementation of a feature-based context-oriented programming language. In *Proceedings of the Workshop on Context-oriented Programming*, pages 9–16, 2019.
17. M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
18. R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.
19. 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.
20. M. Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.
21. S. Katz. Aspect categories and classes of temporal properties. In *Transactions on aspect-oriented software development I*, pages 106–134. Springer, 2006.

22. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.
23. R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *2009 Formal Methods in Computer-Aided Design*, pages 152–159. IEEE, 2009.
24. J. Krüger. Separation of concerns: experiences of the crowd. In *ACM Symposium on Applied Computing*, pages 2076–2077. ACM, 2018.
25. F. M. Maggi, M. Westergaard, M. Montali, and W. M. van der Aalst. Runtime verification of ltl-based declarative process models. In *International Conference on Runtime Verification*, pages 131–146. Springer, 2011.
26. S. Malakuti and M. Aksit. Event-based modularization: how emergent behavioral patterns must be modularized? *FOAL*, pages 7–12, 2014.
27. S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): A case study. *arXiv preprint arXiv:1602.01172*, 2016.
28. S. Maoz and Y. Sa'ar. Aspectltl: an aspect language for ltl specifications. In *AOSD*, pages 19–30. ACM, 2011.
29. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SEN*, volume 29, pages 127–136. ACM, 2004.
30. S. Sandra I. Casas, J. J. Baltasar García Perez-Schofield, and C. Claudia A. Marcos. MEDIATOR: an AOP Tool to Support Conflicts among Aspects. *International Journal of Software Engineering and Its Applications (IJSEIA)*, 3(3):33–44, 2009.
31. 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.
32. Y. Tahara, A. Ohsuga, and S. Honiden. Formal verification of dynamic evolution processes of uml models using aspects. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 152–162. IEEE Press, 2017.
33. T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. *SPLAT*, 2003.
34. 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.
35. N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154. ACM, 2002.
36. S. Velan. Introducing artificial intelligence agents to the empirical measurement of design properties for aspect oriented software development. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 80–85. IEEE, 2019.
37. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.