Aspect Oriented Behavioral Synthesis

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. Modern modularization techniques such as Aspect Orientation require powerful and expressive enough specification languages in order to conceive the development of a system as the combination of the different views it is composed of. In this work we present FVS as an aspect oriented language where the composition of individual aspects is achieved employing behavioral synthesis. As a distinctive feature, our approach can handle properties denoted by non deterministic Büchi automata. A case of study is introduced to show our approach in action.

Keywords: Aspect Orientation, Behavioral Synthesis

1 Introduction

In the recent years modern modularization techniques such as Aspect Orientation [19, 15] or Feature Oriented Programming [1, 25] have draw the attention of the Software Engineering Community and their application on several domains such as software protocols, software architectures and others keep growing and growing [1, 11, 21]. In few words, these approaches conceive systems as different views, perspectives or features that combined together produce the expected behavior of the system. For example, in the classic ATM example the system is built combining diverse views such as security, availability, efficiency, data integrity or transaction management, just to mention a few.

The main tasks to be addressed when dealing with these approaches are to specify individual views for one side, and for the other side, to specify how to compose and combine each view to produce the expected behavior of the system. This last item is clearly the ultimate challenge to be tackled because the different views rarely behave separately but rather interact with each other. In addition, feature or view interaction might not be trivially resolved. In some occasions one feature might depend on the behavior of another feature, sometimes two or more features can be in conflict with each other, or sometimes the behavior of the system might be different according to the order in which the views are combined. 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*. Several approaches aim to address aspects' interaction trying to identify and resolve possible conflicts between aspects [26, 8, 13, 22] tackling problems known as the Aspect Interference [8] problem and the Aspect Oriented Paradox [27]. However, most of them are based in operational notations inspired in finite state machines or labeled transition systems(e.g., statecharts) [18, 17]. Many aspect oriented approaches boil down into providing syntactic weaving mechanisms, usually with non-clear semantics counterpart [17]. Thus, unlike other well-established modularization mechanisms as procedures, parallel composition, or logical conjunction (in declarative approaches) aspect orientation, though attractive in principle, is still a second class citizen, holding just the status of a hacking or dynamic instrumentation mechanism where semantics impact is not neatly characterized.

An interesting and novel way of addressing aspect interaction is given in [24]. In this approach the system is built in a fashion that its behavior satisfies its specification by construction. This is achieved by employing behavioral synthesis [9, 12]. Behavioral synthesis can be seen as an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [23]. In the case of reactive synthesis, an implementation is typically given as an automaton that accepts input from the environment (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 [23]. In order to reduce the time complexity of the algorithms involved work in [9] 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 [12]. Efficient symbolic algorithms for GR(1) realizability checking and controller synthesis have been presented in [9].

However, the synthesis technique introduced in [24] only works if the behavior can be denoted using only Deterministic Büchi automata. This is because if an LTL formula con be expressed by Deterministic Büchi automaton then it can be expressed in the GR(1) fragment of LTL [12]. Therefore, there exists the need to further extend the expressive power of the specification language used to synthesize behavior.

Given this context in this work we present the FVS (Feather Weight Visual Scenarios) specification language [4, 3] as a powerful tool to denote, compose and synthesize aspect oriented behavior. 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 omega-regular properties, being for example, more expressive than LTL (Linear Temporal Logic) [4]. In addition, in [5] we introduced a version of FVS where behavioral synthesis it is not limited to behavior properties expressible by deterministic Büchi automata since properties which are not expressible by deterministic Büchi automata can also be handled. FVS was also thoroughly explored in previous work as an aspect oriented specification language [7, 2], and a tool named GTxFVS was developed giving support to all of the FVS features [6]. Taken all these points into consideration, *in this work we combine FVS expressive power with its aspect oriented flavour to provide an attractive tool to handle aspects interactions by employing behavioral synthesis as the weaving mechanism to build the system.*

The rest of the paper is structured as follows. Section 2 presents some background concepts for a more comprehensive reading of this work. Section 3 shows our approach in action whereas Section 4 presents related and future work. Finally, Section 5 enumerates the conclusions of the work.

2 Background

In this section we present some preliminaries concepts introducing our language, its usage on aspect orientation and the synthesis procedure which play the role of the weaver.

Feather weight Visual Scenarios: In this section we will informally describe the standing features of FVS [3, 4]. The reader is referred to [4] 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. Events labeling an arrow are interpreted as forbidden events between both points. In figure 1-c A-event precedes B-event such that C-event does not occur between them. Finally, FVS features aliasing between points. Scenario in 1-d indicates that a point labeled with A is also labeled with $A \wedge B$. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxis.

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.

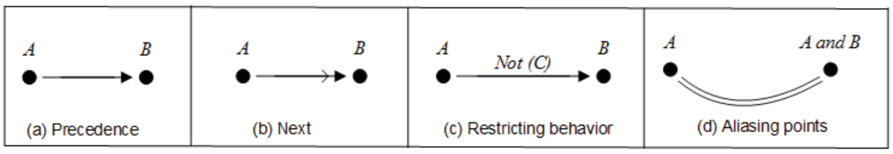

Fig. 1. Basic Elements in FVS

Since a rule can feature more than one consequent, elements which do not belong to the antecedent scenario are numbered to identify the consequent they belong to. An example is shown in Figure 2 describing the circumstances under which writing in a pipe is valid. For every write in the pipe it must be the case that either the pipe did not reach its maximum capacity since it was open(Consequent 1) or the pipe did reach its capacity, but another component performed a read over the pipe (making the pipe available again) afterwards and the pipe capacity did not reach again its maximum (Consequent 2).

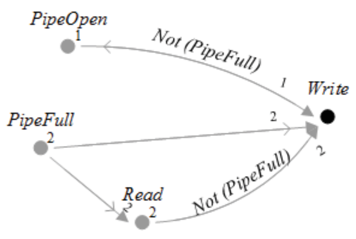

Fig. 2. An FVS rule example

FVS as an aspect oriented specification language: Aspect orientation can be described as a modularization technique which extends a base system at certain points of interest introducing new behavior specified in the so called *aspects*. Aspects are described as a twofold: a *pointcut*, which selects where the aspect's behavior is to be introduced, and an *advice*, which details what behavior in particular is to be added. FVS rules fits into the aspect oriented perspective: rules' antecedents play the role of *pointcuts*, whereas consequents play the role of *advices*. The reader is referred to [2] for a more comprehensive description of FVS as an aspect oriented language. Weaving is the process which inserts into the base system the behavior described by the aspects in those points of the base system that were indicated.

Behavioral Synthesis in FVS: FVS specifications can be used to automatically obtain a controller employing a classical behavioral synthesis procedure. We now briefly explains how this is achieved while the complete description is available in [5]. Using the tableau algorithm detailed in [4] FVS scenarios are translated into

Büchi automata. Then, if the obtained automata is deterministic, then we obtain a controller using a technique [23] based on the specification patterns [14] and the GR(1) subset of LTL. If the automaton is non deterministic, we can obtain a controller anyway. Employing an advanced tool for manipulating diverse kinds of automata named GOAL [28] we translate these automata into Deterministic Rabin automata. Since synthesis algorithms are also incorporated into the GOAL tool using Rabin automata as input, a controller can be obtained. Although this gain in expressiveness come with an cost in terms of performance due to the size of the involved automata we believe its crucial being able to express all type of behavioral properties. In the next section we illustrate the power of FVS's behavioral synthesis as a weaving process specifying an interesting case study.

3 Case of Study

In this section we analyze the case of study of this work. Section 3.1 models a first iteration of the system to be developed. Section 3.2 aggregates more sophisticated functionality whereas Section 3.3 enumerates some final considerations about the weaving process.

3.1 The Exam System

In order to show in action FVS's synthesis procedure as an aspect oriented weaving mechanism we studied the case study introduced in [24]. The system basically provides a service for student exams. Following the strategy in [24] we start off with a base system, and use it as a minimal basis on which to add aspects. This base system behaves as follows: when a new student comes, the service may leave the waiting state, show a welcome screen, and start the exam. The student may fail or pass the exam, after which the service moves to the exit state and back to waiting for a new student. The environment controls the input variables such as *evalExam* and *newStudent*. The system itself has another variable, *state*, specifying its current state (for example, *inExam*, *failed*, *passed* or *wait*). Note that the base system is not deterministic: when it is waiting and a new student comes, it can either stay in waiting state or move to the welcome state.

We start extending this base functionality adding new behavior in the shape of aspects. The first aspect to be introduced, called *Tuition*, handles a security concern: it prevents new students who have not paid their tuition from taking an exam and instead redirects them to the exit state where it shows a message with information about tuition payments. This behavior is shown in Figure 3. The FVS rule in the top of the figure dictates that a welcome screen will only be shown if and only if the tuition was paid before (that is, if an *PaidTuition* event occurred in the past). Similarly, the rule in the bottom of the figure says that in those cases where the student did not paid the tuition the system must show an allusive message and exit.

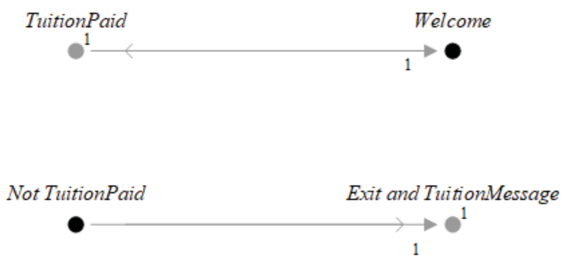

Fig. 3. Tuition Aspect in FVS

The second behavior to be introduced by an aspect handles the services availability concern: it specifies that whenever a new student is present when the system is waiting, eventually the system will welcome the student. As in [24], we named this aspect *Availability* which addresses a typical liveness constraint. This behavior is modeled in FVS as shown in Figure 4. The rule establishes that when the system is in the *wait* state and a new student arrives, then the welcome screen will be eventually displayed.

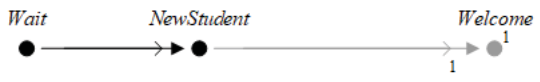

Fig. 4. A liveness property in the Exam System

Having specified these two aspect, we weave them into the base system by employing the synthesis procedure described earlier in Section 2. However, a winning strategy for the system is not found. As explained in [24], there is a winning move for the environment since the initial wait state is not winning for the system: an environment that sends infinitely many students such that only a finite number of them have paid their tuition, will force the system not to visit the welcome state infinitely often, and thus to violate its specification. We solve this functionality bug following the solution presented in [24]: forcing the system to leave the wait state but not necessarily visit the welcome state. This is shown in Figure 5.

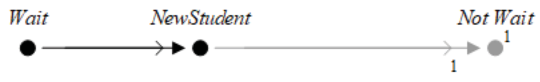

Fig. 5. A more general Availability Aspect

When performing the behavioral synthesis considering the *Tuition Aspect* and the revisited *Availability Aspect* a controller can be automatically obtained for the *Exam System*.

3.2 Adding new behavior

We now add some three extra features to the system. The first one gives the possibility to the student to retry the exam in the case she failed. Given this new feature, we consider an aspect that checks that every *retry* must occur while the system is in the *inExam* state. This is illustrated in Figure 6. In other words and considering the concrete events of the system, the aspects checks that once the system has presented the *Welcome* screen every occurrence of the event *retry* must be preceded by the occurrence of the *InExam* event.

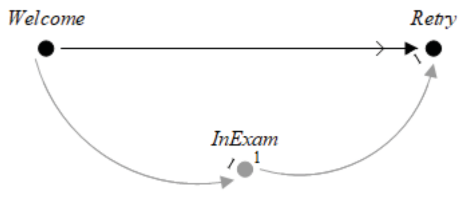

Fig. 6. Validating Retry occurrences

A security aspect is added next. It persists and logs every failed exam in case a student's complaint is raised in the future. The requirement for this aspect is the following: In each student' valid session (between the *Welcome* state and until the *Exit* state) every failed exam must be persisted and logged. This is reflected in Figure 7. During a valid session, the occurrence of the event *Fail* must trigger the occurrence of the events *Persist* and *Log*.

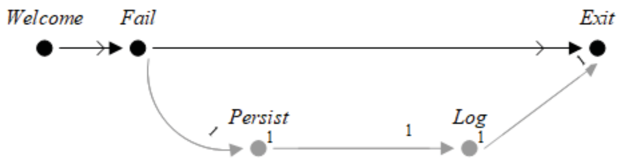

Fig. 7. Capturing information about failed exams

Finally, we revisit this last requirement (persisting and logging failed exams information) considering that a new feature is added: the student can quit the exam and leave the system. However, every failed exam must be persisted and logged anyway. Therefore, the *quit* event must not occur until the exam is persisted and logged. Taking this fact into consideration, we modified previous rule in Figure 7 as it is shown in Figure 8. The only difference is that quitting is not allowed until the failed exam is persisted and logged.

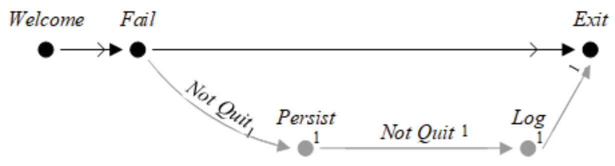

Fig. 8. Failed exams and the Quit feature

3.3 Case Study: Remarks and Observations

A controller was built upon FVS specifications for every version of the Exam System. This was achieved by employing the behavioral synthesis procedure described in Section 2. It is worth mentioning that the three aspects introduced in Section 3.2 can not be represented by Deterministic Büchi automata. As it was described in Section 2 FVS is still capable of obtaining a controller in those cases by translating non Deterministic Büchi automata into Deterministic Rabbin automata. In particular, the three aspects considered in Section 2 corresponds to three particular specification patterns [14]. The aspect that validates retry occurrences corresponds to the pattern *Precedence pattern with After Q scope*; the first version of the audition aspect for failed exams corresponds to the *Response Chain pattern (with one stimuli and two responses) with After q until r scope* and second version of this latter aspect corresponds to *the Constrained Chain pattern (responses s, t without z responds to the p stimuli event) with After q until r scope*. The time consumed to obtain this new controller (including these three new aspects) took eight times more than the previous one. However, we believe it is desirable being able to express these kind of properties when they arise despite the fact they are time consuming.

4 Related and Future Work

FVS was previously explored in the aspect oriented world [7, 2]. While this previous work was focused on modeling aspect's behavior this work provides the means to perform behavior synthesis. Other several approaches aim to formalize aspects specification and interaction. For example, work in [16] employs model checking techniques to prove the correctness of aspects application and weaving into the base system. Aspects are specified using state machines. Similarly, work in [20] validates CTL properties when weaving aspects into the base system. Also notations based on state machines are employed. We rely on a different weaving approach based on behavioral synthesis. In this sense, AspectLTL [24] also employs behavioral synthesis for weaving aspects. However, their approach is limited to the usage of Deterministic Büchi automata while our approach handles also Non Deterministic Büchi automata.

Work in [22] employs constructors based on implementations languages named *gummy* modules to denote aspect behavior and resolve aspects interaction in a modular way. Also more related to the implementation phase, work like [26] provides an interesting tool to detect aspect interference in the AspectJ language. Work in [10] extends aspects definition to tackle high level behavior with a Finite State Machine based notation denominated Dependency State Machines. The focus of this work is to improve automatic verification and optimization of aspects. An AspectJ extension is also presented in that work. We rely on a more declarative way of defining aspects and the implementation view is built using behavioral synthesis.

Regarding future work we would like to improve the performance of our approach. This line of research involves trying to optimize the size of the automata involved since the complexity of the algorithms involved are heavily influenced by the size of the automata. Other similar future direction is to analyze the trade off between expressivity and performance and provide stronger empirical evidence in this topic. Finally, we would like to explore automatic code generator tools using as input the controllers as in [23].

5 Conclusions

In this work we propose FVS a a powerful tool to denote, compose and synthesize aspect oriented behavior. Aspects are defined using graphical rules and the weaving process is achieved using behavioral synthesis. As a distinctive feature, our approach is not limited to Deterministic Büchi automata since Non Deterministic Büchi can also be input to the synthesis procedure. A case study is presented illustrating the main points of our approach.

6 Acknowledgements

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

References

1. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented software product lines*. Springer, 2016.
2. F. Asteasuain and V. Braberman. FVS: A declarative aspect oriented modeling language. *EJS - Electronic Journal SADIO*, 10(1):20–37, 2011.
3. F. Asteasuain and V. Braberman. Specification patterns: formal and easy. *IJSEKE*, 25(04):669–700, 2015.
4. F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2):239–274, 2017.
5. F. Asteasuain, F. Calonge, and M. Dubinsky. Exploring specification pattern based behavioral synthesis with scenario clauses. In *CACIC*, 2018.

6. F. Asteasuain and F. Tarulla. Exploring architectural model checking with declarative specifications. In *CACIC*, 2017.
7. F. Asteasuain, F. Tarulla, and P. Gamboa. Using the power of abstraction to express high-level behavior in aspect oriented approaches. In *CONAIISI*, 2017.
8. L. M. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, 2003.
9. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'Ar. Synthesis of reactive (1) designs. 2011.
10. E. Bodden. Specifying and exploiting advice-execution ordering using dependency state machines. In *FOAL*, volume 151, 2010.
11. T. Cerny. Aspect-oriented challenges in system integration with microservices, soa and iot. *Enterprise Information Systems*, 13(4):467–489, 2019.
12. N. DIppolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran*, 22(9), 2013.
13. C. Disenfeld and S. Katz. A closer look at aspect interference and cooperation. In *AOSD*, pages 107–118. ACM, 2012.
14. M. Dwyer, M. Avrunin, and M. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
15. R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented software development*. Addison-Wesley Professional, 2004.
16. M. Goldman, E. Katz, and S. Katz. Maven: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37(1):61–92, 2010.
17. S. Katz. Aspect categories and classes of temporal properties. In *Transactions on aspect-oriented software development I*, pages 106–134. Springer, 2006.
18. S. Katz and H. Israel. Diagnosis of harmful aspects using regression verification. *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, 2004.
19. 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.
20. S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT*, volume 29, pages 137–146. ACM, 2004.
21. J. Krüger. Separation of concerns: experiences of the crowd. In *ACM Symposium on Applied Computing*, pages 2076–2077. ACM, 2018.
22. S. Malakuti and M. Aksit. Event-based modularization: how emergent behavioral patterns must be modularized? *FOAL*, pages 7–12, 2014.
23. S. Maoz and J. O. Ringert. Synthesizing a lego forklift controller in gr (1): A case study. *arXiv preprint arXiv:1602.01172*, 2016.
24. S. Maoz and Y. Sa'ar. Aspectltl: an aspect language for ltl specifications. In *AOSD*, pages 19–30. ACM, 2011.
25. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SEN*, volume 29, pages 127–136. ACM, 2004.
26. 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.
27. T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. *SPLAT*, 2003.
28. 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.