2011 Fifth IEEE International Conference on Theoretical Aspects of Software Engineering

Undesirable Aspect Interactions: a Prevention Policy

Arsène Sabas
* Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, Canada,
{sabasars,boyer}@iro.umontreal.ca

Subash Shankar
Department of Computer Science
Hunter College and the Graduate Center
City University of New York
subash.shankar@hunter.cuny.edu

Virginie Wiels and Michel Boyer*
Onera / DTIM 2 avenue E. Belin
BP 74025, 31055 Toulouse cedex, France
Virginie.Wiels@onera.fr

***Abstract*—Aspect-oriented software development (AOSD) has emerged in recent years as a new paradigm for software development, providing mechanisms to localize cross-cutting concerns (i.e. scattered in many locations) during the software development process. Aspect interaction problems (due to their integration into the base components) is an important issue in AOSD; verification is most often based on a detection and correction strategy. This paper presents an ongoing work which goal is to built a prevention mechanism at the specification phase for aspect-oriented systems. This prevention mechanism will allow to avoid undesirable interactions in a aspect-oriented system. By acting at the specification phase, we believe that verification will be made timeless and costless.**

I. Introduction

One of the main goals of software engineering is to enable the construction of large, complex and reliable software in timely fashion. Aspect Oriented Programming (AOP) is a new technology for improving comprehension and maintainability of complex programs by localizing behaviors that would otherwise be scattered and tangled. These behaviors are referred to as crosscutting concerns. Scattering is the condition where a concern is implemented in several non-contiguous places in the program. Tangling, the dual of scattering, occurs when several concerns overlap at a region in the program text. The traditional Object Oriented Programming (OOP) cannot control theses crosscutting concerns. AOP targets the separation of crosscutting concerns by isolating each crosscutting concern from the core concerns (the rest of code) into modules called aspects.

After the decomposition step, a system engineer composes all the aspects into the base components to get the whole system. Aspect Oriented Technology (AOT) provides weaving mechanisms to integrate each aspect into the base components (core concerns). The result of the weaving is the complete system. Without the aspect components, the base system will be more maintainable. It is believed that AOP will increase modularity and cohesion, thus increasing overall software quality. AOP, giving rise to programming languages such that AspectJ , evolved from a programming activity to a full-blown software engineering process, having goal to preserve modularity and traceability, which are two important properties of high-quality software.

An increase in software quality does not, however, imply that programmers will stop making mistakes and that Aspect Oriented software will be bug free. There also remains many challenges in AOT. Aspect interaction is one of the main concerns in the aspect-oriented community [3]. Detection and resolution of undesirable aspect interactions is an important opened research field. Most of the aspect-oriented verification approaches are based on a detection and correction strategy. Although these detection approaches are relevant for aspect-oriented software reliability, we believe that they are time and cost consuming.

It is a good thing to detect and correct system failures, but it is better to first prevent them and it is a prevention policy, to be integrated at the specification phase, that we will be advocating. We believe indeed that this will make the verification phase timeless and costless. An aspect faults model developed in [1] lists the main fault types that can arise in aspect-oriented applications due to the aspect integration into the base systems. The existing verification approaches can detect only one or two of these fault types. There are no existing methods or tools capable to take into account all of them [7]. There is a lack of efficient mechanism for dealing with aspect interactions, and which can take into account most of these aspect fault types. The logic $\mathscr{L}_a$ (see section III) will help us to specify system components with a prevention mechanism, which will prevent most of the undesirable aspect interactions characterized by the fault classes developed in [1]. In our approach, to deal with aspect interaction problems, we adopt the corrective and preventive strategies. Corrective strategy is based on a non-conformance event that has happened in the past. Preventive strategy is based on preventing a non-conformance event in the future. Our prevention mechanism play the role of the prevention action. Corrective action will be taken at the verification phase when it is necessary, i.e., after an interaction problem has occurred.

II. A fault model for AOP

Alexander and Bieman [1] propose an initial fault model for AOP. This fault model is based on the nature of faults and the failures in AOP. This initial model has been refined

978-0-7695-4506-6/11 $26.00 © 2011 IEEE
DOI 10.1109/TASE.2011.27

in [4], [2]. In this section, we present the fault model on which our prevention policy is based.

A. Incorrect strength in pointcut patterns

For example, the pointcut pointcut creditOps1(): call(*Account.credit(..)); captures the calls to all methods of the Account class named credit. In this example, call is the type of joinpoints, and *Account.credit(..) is the signature pattern of the joinpoints. Now, consider another example of pointcut pointcut creditOps2(): call(*Account+.credit(..));. The pointcut creditOps2() captures the calls of all methods named credit, of the Account class and all its sub-types. The pattern in the pointcut creditOps1() is said to be stronger than the pattern in second pattern creditOps2(), because the first one is more restrictive than the second one. If the pattern is too strong, some necessary joinpoints will not be selected. If the pattern is too weak, additional join points will be selected that should be ignored. That is one of the main type of aspect interaction problem.

B. Incorrect aspect precedence

In an Aspect Oriented Program, it is possible for more than one advice to be affecting the same join point. In such situations it may be important to control the order in which the advice is applied to the joinpoint. The order in which the advice is applied can be controlled by assigning precedence to the various Aspects whose advice is affecting the joinpoint. If no precedence order is explicitly assigned to the aspects, undesirable behaviors can emerge especially when there are mutual interactions between these aspects.

C. Failure to establish expected post-conditions

Aspects can cause changes in the flow of control of class's code (classes of the base program). Such a change in flow of control can result in a class (core concern) not being able to fulfill the post-conditions of its class contract. The clients of core concerns expect those concerns to behave according to their contracts. Clients expect method post-conditions to be satisfied regardless of whether or not aspects are woven into the concern. Hence the behavioral contracts of the concern should hold after the weaving process. Thus, for correct behavior, woven advice must allow methods in core concerns to satisfy their post-conditions.

D. Failure to preserve state invariants

A concern's behavior is defined in terms of a physical representation of its state, and methods that act on that state. The integration of an aspect into the base program can introduce new methods and instance variables in to the core concerns (classes). Thus, this integration can introduce new states and can cause the classes to violate their state invariants. In addition to establishing their post-conditions, methods must also ensure that state invariants are satisfied.

E. Incorrect focus of control flow

A pointcut designator selects which of a method's join points to capture. This selection is determined at weave time. However, there are often cases where the information needed to correctly make such a decision is available only at run time. Sometimes join points should only be selected in a particular execution context. This context could be within the control structure of a particular object, or within the control flow that occurs below a point in the execution

F. Incorrect changes in control dependencies

The advice type "around" can alter the behavioral semantics of a method upon which it is applied. New code is inserted, new branches appear that alter the dependencies among statements, and new data may also be inserted. These faults can occur when for example, an around advice and a matched join-point have different control flows. These faults will affect core concern behavior. In [1], they propose to apply regression testing strategies in this case.

III. LOGIC $\mathscr{L}_a$

The logic $\mathscr{L}_a$ includes modalities of three other logics: (1) linear temporal logic (LTL) [9] to reason on the time (2) (first-order) dynamic logic (FDL) [6] to reason explicitly on actions or computer programs and properties (LTL cannot explicitly reason on actions); (3) deontic logic [15] to specify the social life of system components. Others have shown that such a combination (LTL + Dynamic Logic + Deontic Logic) is adequate for systems specification [8] because it can distinguish between description and prescription of behavior. Action prescriptions are meant to convey when actions may or must occur, via the deontic concepts of obligatory and permissible action. The description is achieved by the traditional pre and post condition style description of actions. This then allows us to state when actions may and must happen as opposed to just describing the effects of such actions [8].

We give an informal meaning of a few example formulae of $\mathscr{L}_a$ (because of space limitation). $[\alpha]\beta$ is an action which means that immediately after each execution of the action α, the action β must be executed. $<\alpha>\beta$ is an action which means that it is possible to execute the action α and reach a state in which the action β has to be executed. $I\alpha$ means that it is forbidden to execute the action α, otherwise there will be a punishment represented by a constant action V. $O\alpha$ means that it is obligatory to execute the action α, otherwise there will be a reparation represented by a constant action V. $(B_e\phi)\alpha$ means that just before each state i where the formula ϕ is true, the action α is executed. $(A_f\phi)\alpha$ means that immediately after each state i where the formula ϕ is true, the action α is executed. $(A_r\phi)\alpha_1\alpha_2$ means that in each state i where the formula ϕ is true, α_1 is executed immediately before and α_2 immediately after the action that should be executed in i. $(I_o\phi)\alpha$ means that in each state i where the formula ϕ is true, the action α is executed instead of the action that should be executed in i.

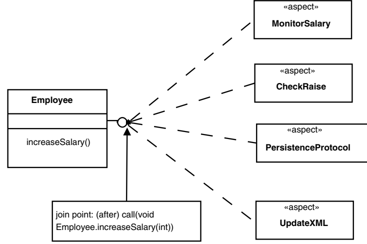

Fig. 1. The Employee Class and its superimposed aspects

The complete description of this logic can be found in the paper [12].

IV. Aspect Prevention Mechanism

The best strategy of handling conflicts is to prevent conflicts from happening. Conflicts can be prevented by adding arbitrators to arbitrate the conflict between components or seeking alternative components, based on the causes of conflicts. We will define a model for the prevention policy of each of the fault types presented in the previous section. The collection of these models represents our prevention mechanism for a given aspect application. In this paper, we only present a model for *Incorrect Aspect Precedence* fault type. We are building the other models, which will be the core of another paper.

We use the example shown in Figure 1 to describe how to define the prevention policy for this kind of fault in aspect-oriented systems. This example is taken from [11]. The example consists of a simple personnel management system. The *Employee* class forms an important part of the system. In particular, we will focus on the method *increaseSalary()*, which uses its argument to compute a new salary. This example has been constructed as a scenario that introduces new requirements at each step. Applying the principle of separation of concerns, each of these requirements is represented by aspects that will be woven on the same join point (as well as others), after the execution of the method *increaseSalary()* of the *Employee* class.

As a first step, the company introduces a logging system to monitor the change of salaries. This feature is represented by the *MonitorSalary* aspect. This aspect prints a notification whenever a salary has been changed. This could include information about the employee and the type of salary change.

The second requirement states that certain classes of the application should store their state in a database. The database should be updated, as soon as possible, after each state changes in the object. To keep persistence separate from the application model, an aspect is used to realize this requirement. The *PersistenceProtocol* aspect contains the advice that performs the update of a persistent object. If the data of a persistent object changes, the corresponding information should be updated in the database too.

Because the database needs to be updated as soon as possible after the state changes in the object, the advice of the *PersistenceProtocol* aspect has to be executed before the advice of the *MonitorSalary* aspect.

The next requirement states that an employee's salary cannot be higher than his/her manager's salary. Thus, a raise is not accepted if it violates this criterion. This is enforced by the *CheckRaise* aspect. The advice of this aspect checks the new salary after the *increaseSalary()* method has executed. If the rule is violated, a warning message is printed and the salary is set back to its original value.

Adding the *CheckRaise* aspect affects the composition; if this aspect fails the *PersistenceProtocol* aspect should not be executed because the employee's data has not changed. That is, the execution of the *PersistenceProtocol* aspects depends on the outcome of *CheckRaise*.

The fourth requirement states that if the database is not available, persistence must be implemented with XML files. For each instance of *Employee*, an XML file is generated. If the regular persistence does not take place (e.g. because of database connection problems), the file must be updated after each state change of the *Employee* object. This is realized by the *UpdateXML* aspect. This aspect has one advice that calls the method that rewrites the XML file if the salary (or other data) changes.

In this example, XML files should be updated only if the *PersistenceProtocol* aspect was not able to update the database. This means that *UpdateXML* should also execute conditionally; only if *PersistenceProtocol* failed. The execution of an aspect may depend on the outcome of other aspects. Only if the outcome of these other aspects satisfies a certain criterion, the dependent aspect is allowed to execute.

To define the prevention policy of this fault type, we can follow these three steps:

1) Define the aspect precedence requirements or constraints for a given application.
2) Formalize each constraint by using the logic $\mathscr{L}_a$.
3) Define a coordination aspect module which will contains mainly these formalized constraints under the field Prescription Axioms. This field defines properties describing what the system should do, or proscribing the violation of desired properties, i.e., undesirable interactions. It is a field of our component abstraction.

a) Step 1: The constraints of the management system of the section II-B are: (1) PersistenceProtocol aspect should be executed before MonitorSalary aspect. (2) CheckRaise aspect should be executed before PersistenceProtocol aspect. If CheckRaise aspect returns an error message (i.e., the method check returns an error message), PersistenceProtocol aspect should not be executed (that is, it is forbidden to perform update action). (1) If PersistenceProtocol aspect is not executed, then UpdateXML aspect should be performed.

b) Step 2: The formalization of the constraints are done on instances of the aspects and classes. Let em, mo, pe, ch, up, be instances of Employee class, MonitorSalary, PersistentProtocol, CheckRaise, UpdateXML aspects,

respectively. x, y are variables of sort int, s, s' are variables of sort string, and f is a variable of sort file. A sort is like a type in a programming language. The translation of the first constraint into the logic $\mathscr{L}_a$ is as follows:

$$(A_f pr(salaryChange(em.salary)))(I(mo.print(s);pe.update \vee mo.print(s);pe.print(s')))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point (the part $A_f pr(salaryChange(em.salary))$), we should never perform the action mo.print(s);pe.update or mo.print(s);pe.print(s'). Recall that I is the symbol of the forbidden operator of the logic $\mathscr{L}_a$ and ; the one of the sequential actions composition operator. The translation of the second constraint into the logic $\mathscr{L}_a$ is as follows:

$$(A_f pr(salaryChange(em.salary)))(I(pe.update;ch.check(x,y) \vee pe.print(s');ch.check(x,y)) \wedge ch.check(x,y) = error \rightarrow I(pe.update \vee pe.print(s')))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point, it is forbidden to perform the action pe.update;ch.check(x,y) or pe.print(s');ch.check(x,y), and if the check method returns an error message, then it is forbidden to execute the action pe.update or pe.print(s'). The translation of the third constraint into the logic $\mathscr{L}_a$ is as follows:

$$(A_f pr(salaryChange(em.salary)))([\neg(pe.update \wedge pe.print(s'))]up.updateFile(f))$$

This formula means that each time the join point call(em.increaseSalary(int)) is selected by the pointcut salaryChange(em.salary) and immediately after this join point, if the action $pe.update \wedge pe.print(s')$ is not executed, then perform the action up.updateFile(f). Note that [] is the necessity operator of the logic $\mathscr{L}_a$.

c) Step 3: The coordination module aspect will mainly contain the above properties under the field Prescription Axioms. It records information about all aspect modules which are present in the application and also the classes. It can contain other elements such as mechanisms that add and remove dynamically an aspect to and from the application, respectively. This coordination aspect module is responsible for the management of the aspect scheduling or ordering at the shared join points. This module concept reinforces the separation of concerns principle of aspect technology and therefore improves the modularity principle.

V. Conclusion and Future Work

The few verification approaches, [18], [17], [10], [14], [16], [5], we found in the literature that we studied focuss on one or two specific undesirable aspect interactions; nevertheless there exist many kinds of undesirable aspect interactions. These verification approaches are based on conflict detection, contrary to our approach which is based firstly on a prevention strategy and then on a correction strategy if it is necessary. In this paper, we present our ongoing work on the development of the prevention mechanism. We are currently working on the specification in the logic $\mathscr{L}_a$ of all aspect fault types presented above.

References

[1] R. Alexander, J. Bieman, and A. Andrews. "Towards the systematic testing of aspect oriented programs". Technical Report CS-4-105, Colorado State University, Mar. 2004.

[2] J. S. Bækken, R. T. Alexander. "A Candidate Fault Model for AspectJ Pointcuts". Proc. of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006). pages:169–178, Washington, DC, USA, 2006, IEEE Computer Society.

[3] R. Douence, P. Fradet, and M. Südholt. "A framework for the detection and resolution of aspect interactions". In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE '02)*, pages 173–188, London, UK, 2002. Springer-Verlag.

[4] F. C. Ferrari, J. C. Maldonado, and A. Rashid. "Mutation Testing for Aspect-Oriented Programs". Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST'08). pages 52–61, Washington, DC, USA, 2008. IEEE Computer Society.

[5] M. Goldman and S. Katz. "Modular generic verification of ltl properties for aspects". In *Proc. of Foundations of Aspect Languages Workshop (FOAL06)*, pages 17–24, Mar. 2006.

[6] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* The MIT Press Cambridge, Massachusetts London, England, 2000.

[7] S. Katz. "A survey of verification and static analysis for aspects". Technical Report Part of Milestone M8.1, Formal Methods Laboratory of AOSD-Europe, July 2005.

[8] S. Khosla and T. Maibaum. "The prescription and description of state based systems". In *B.Banieqbal, H.Barringer and A.Pnueli (eds) Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 243–294. Springer-Verlag, 1987.

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, Berlin, Jan. 1992.

[10] F. Mostefaoui. *Un cadre formel pour le développement orienté aspect: modélisation et vérification des interactions dues aux aspects.* PhD thesis, Université de Montréal, Aug. 2008.

[11] I. Nagy and L. Bergmans and M. Aksit. "Declarative Aspect Composition". Software-engineering Properties of Languages for Aspect Technologies, SPLAT. Held in conjunction with the Third International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster UK, March 22-26 2004.

[12] A. Sabas and M. Boyer M, "La, an Aspect-Oriented Multi-Modal Logic". Post-NASSLLI Workshop on New Directions in Dynamic Epistemic Logic, Bloomington, Indiana, USA, June 20-26, 2010.

[13] P. Shaker and D. K. Peters. "An introduction to aspect oriented software development". In *Proc. Newfoundland Electrical and Computer Engineering Conference, IEEE*, November 2005.

[14] M. Storzer, J. Krinke, and S. Breu. "Trace analysis for aspect application". In *Analysis of Aspect Oriented Software (AAOS)*, volume 398, pages 243–294. Springer-Verlag, July 2003. in conjunction with ECOOP 2003.

[15] G. H. V. Wright. "Deontic logic". *Mind, JSTOR*, 60(237):1–15, 1951.

[16] D. Xu, I. Alsmadi, and W. Xu. "Model checking aspect oriented design specification". In *Proceedings of 31st Annual International Computer Software and Applications Conference*, volume 1, pages 491–500. compsac, July 2007.

[17] D. Xu, W. Xu, and K. Nygard. "A state-based approach to testing aspect oriented programs". In *Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering*, pages 188–197, July 2005.

[18] J. Zhao. "Data-flow-based unit testing of aspect oriented programs". In *Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pages 188–197. IEEE Computer Society, Dec. 2003.