Vigilant usage of Aspects

Freddy Muñoz, Olivier Barais, Benoit Baudry
IRISA
Campus de Beaulieu, F-35042 Rennes Cedex, France
{fmunoz, barais, bbaudry}(at)irisa.fr

Abstract—**In the last 10 years the Aspect-Oriented Software Development (AOSD) has gradually become a concern stone in Software Engineering as an engine to reduce complexity and increase reuse by providing modularization of concerns that tend to crosscut. Nevertheless, its use in certain situations can presents some problems that can not only discourage its mainstream adoption, but also hinder the realization of software quality goals. The first problem, the AOSD-Evolution paradox, encompasses the difficulties with evolving software developed using AOSD. The second arises as a result of the invasive nature of aspects. The use of aspects without any control can result in a harmful practice. This work describes these problems and exposes the strength and limitations of the current approaches to solve them. Thus allowing us to reason in a clear fashion about the problems and their solutions, then justifying a contract base approach, which aims to control the usage of aspect without constraining the power of AOSD.**

I. Introduction

The current mechanisms to implement aspects such AspectJ [8], allow sophisticated ways to express pointcuts and advice. But fail to (1) define pointcuts abstracting from syntactical properties of the base code, and to (2) control the aspects invasiveness. These deficiencies lead to the following problems:

1) Evolution difficulties, reflected in the called *AOSD-Evolution paradox* [29], this means that software built using AOSD is more modular but less reliable. Then, as the base code evolves aspects pointcuts can miss the desired join points or capture undesired join points.
2) Invasiveness problems, caused by the ability of aspects to access unrestrictedly the base code. These aspects can invalidate some important properties of the system by modifying the program flow or leaving protected data structures in an inconsistent state, then becoming harmful to the base code.

Both problems are exacerbated by the *obliviousness property* [5], which requires aspects to be transparent for the base code, hence allowing it to evolve independently from aspects. Then, as base code evolves, seemingly safe modifications can break pointcuts turning aspects into harmful entities. Furthermore, as base code designers are oblivious about aspects, they have no means to protect their code from them.

The current approaches to solve these problems offer improved mechanisms to AOSD; but fail to either (1) control the aspects invasiveness without constraining its power, e.g. specifying where aspects are not allowed to be invasive, or (2) to fit aspects with an adequate mechanism to support evolution, e.g. a mechanism that tells when an aspect can do harm.

This lack of invasiveness control and evolution support, impacts directly on evolution and reutilization of software built using AOSD, therefore discouraging its mainstream adoption. These deficiencies as the proposed solutions impose a strong need to bring solutions to the AOSD problems, not only encouraging its adoption, but also leading to software that is mode modular and reliable.

Our interest is to analyze the strengths and limitations of the current approaches for solving the AOSD problems. This allows us to reason in a clear fashion about them and propose the beginning of a contract based approach to solve them.

The reminder of this work is organized as follows. Section 2 presents the AOSD problems. Section 3 exposes the current approaches to solve these problems and discusses their strengths and limitations. Section 4 presents the beginning of our approach to solve AOSD problems by controlling its invasiveness. Finally, section 5 concludes and presents the future work.

II. Problems from Aspects

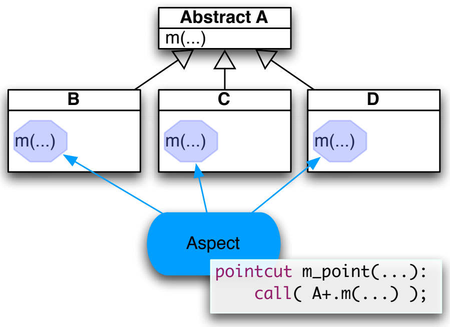
Aspects make a great contribution to improve system modularity. Unfortunately, in many situations aspects can carry some problems that can hinder the achievement of a reliable software system. This because, aspects can negatively affect the behavior and evolvability by introducing side effects into the advised base code.

This section exposes the problems that present a major threat for AOSD adoption: *AOSD-Evolution Paradox* and *Invasive aspects*.

A. AOSD Evolution Paradox

An intuitive notion is that software built using AOSD is more modular, evolvable and reusable. But contrarily, software evolution and reutilization can be negatively affected by the presence of aspects.

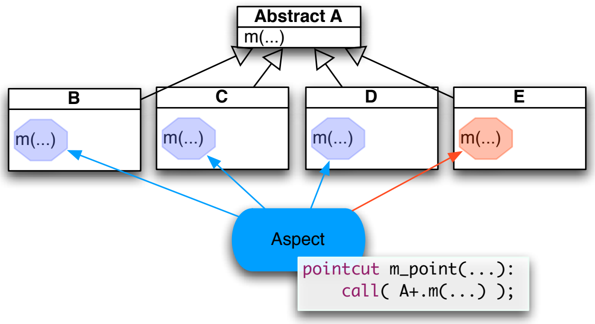
The *AOSD-Evolution Paradox* [29], encompasses the difficulties that arise when an application created using AOSD tries to evolve. It arises on the insufficiency of the current languages for defining pointcuts (crosscutting languages), to abstract over structural properties of the base code. Pointcut specification tends to be very concrete and make explicit assumptions about program structures. Then, as software evolves, pointcuts can lose correct join points or capture wrong join points leading to unexpected behavior or even make the software to break down. This means that AOSD leads to software that is more modular but with a reduced evolvability.



```
pointcut m_point(...):
    call( A+.m(...) );
```

Fig. 1. Aspect applied to three different classes

Figure 1 depicts an example of the AOSD-Evolution paradox. In the figure, an abstract class `A` containing a method `m(..)`, is extended by the classes `B`, `C` and `D` with an aspect targeting all occurrences of the method `m(..)`. If the base code is not changed, the aspect would behave as expected. But, what happens when a new class `E` extending `A` is added? The aspect will also apply to the method `m(..)` in `E`, no matter if it is needed or not. In the worst case, the class `E` has no need of the aspect and it application will lead to an incorrect behavior (Figure 2). In this case, the pattern matching over names allows developers to abstract over syntax, and thus avoid enumerating on join points. This provides some flexibility, but in some situations it is not enough.



```
pointcut m_point(...):
    call( A+.m(...) );
```

Fig. 2. Aspect applied to a class that does not need it

However, if the software is small enough, aspect and base code changes will be easy to coordinate when required. In real-world projects, size and complexity increase very fast making aspects management tricky and error-prone.

B. Invasive aspects

In order to perform separation of concerns and provide security, each paradigm promotes its own way of encapsulating concerns. For example the object-oriented paradigm encapsulates concerns in classes, fields and methods, defining a protection level for each one of them, ensuring that properties access will be respected.

Aspects can break the encapsulation that each paradigm provides, this because they have the ability to interject functionalities at almost any point in the base code, then leaving properties vulnerable to any change from the aspect side. This rupture can be fruitfully used in several situations resulting in a proper introduction of functionalities and flexibility; invalidating or twisting properties that can be considered undesirable.

However, aspects can invalidate some of the already existing desirable properties of the system, therefore becoming harmful to the base code. They can introduce side effects, by partially replacing procedures, modifying the control flow, assigning new values on protected fields, etc. Moreover, aspects can add a tight coupling to the base code, and consequence of the obliviousness property, any seemly safe modification in protected properties can break aspects leading to data inconsistencies or incorrect behavior. Hence, as they can compromise sensitive system properties, they represent a threat for software security [4] and reliability.

The real problem with aspects breaking encapsulation, is the absence of mechanisms to prevent aspects to invalidate desirable and important properties of the system without constraining the power of AOSD.

```
public class SecuredClass {
    private String Username;
    private double accountID;
    private double credit;
    public SecuredClass(String username){
      //obtain higly confidential data
      //using the username
    }
    private boolean canBuy(double amount){
      if(this.credit>amount){
         return true;
      }
      else return false;
    }
    public void withDraw(double amount){
      if(this.canBuy(amount)){
         this.credit=this.credit-amount;
      }
    }
    ...
}
```

Base Code

```
public privileged aspect Leak {
    public String SecuredClass.publicUsername;
    public double SecuredClass.publicAccountID;
    public double SecuredClass.publicCredit;
    after(SecuredClass secured):
     execution(* SecuredClass.*(..))&&target(secured){
     secured.publicUsername=secured.Username;
     secured.publicAccountID=secured.accountID;
     secured.publicCredit=secured.credit;
    }
    after(SecuredClass secured):
     execution(SecuredClass.new(..))&&target(secured){
     secured.publicUsername=secured.Username;
     secured.publicAccountID=secured.accountID;
     secured.publicCredit=secured.credit;
    }
}
```

Aspect Code

Fig. 3. Security violation using aspects

Figure 3 presents an example of security problems introduced by invasive aspects. The java code on top corresponds to a secured class with private fields and operations. On bottom, an aspect that adds unprotected (public access) fields to the secured class, then assigning them the values of the protected fields and keeping synchronized their values during the execution. This aspect expose protected values maliciously, and as the base code is oblivious about it there is no mechanism to prevent this kind of situation.

III. Current Approaches

The major difficulty for AOSD is the behavior assurance, specially when base code tends to evolve. In this scenario the obliviousness property and the tight coupling of aspects with the base code become a two edged sword. By one side increasing modularity, but representing a threat to software reliability on the other.

This section presents the current approaches to solve or reduce AOSD problems, categorized according to the means they use.

A. Guildelines

These approaches are intended to offer guidelines about good practices to avoid the existing problems related with the use of current AOSD technologies.

- An analysis of the limitation in the current AOSD mechanisms is carried out in [18], thus leading to a set of guidelines for a good usage of those mechanisms in order avoid their limitations, hence overcoming evolution and invasiveness problems.
- The defects of the current crosscutting languages are studied in [29], then generating a set of facts about the languages gaps that can be used as a guideline to pass over them.
- Kiczales and Mezini in [12] perform an study of the different mechanism for concern separation. They emphasize the fashion and the flexibility that those mechanisms offer to localize concerns. By distilling a set of guidelines, they provide to developers an orientation about which mechanism is better suited for concern separation in a given scenario.
- Rinard et al. focused on the interaction between advice and methods propose a classification system for Aspect-Oriented Software [23]. This enable developers to reason modularly about aspects behavior when interacting with the base code, and to focus on the aspects causing non-modular interaction.
- Dean Wampler explores the translation of design principles from Object Orientation to Aspect Orientation [32]. Intended to avoid the problems related to invasive aspects, those principles should tell what type of restrictions and coupling between aspects and software entities are appropriate according to a given scenario in order to allow noninvasiveness.

Guidelines make a contribution telling developers and designers (aspects and base code) what to do in a given situation when aspects are used, or how to use the current mechanism to be better prepared to support evolution and limit aspects invasiveness. For language designers, they tell which are the points that need more effort and development in order to carry the current mechanism to a mature state. But, as with any guideline there is no concrete solution, and to follow a guideline gives no confidence that the result will perform as expected.

B. Code Based

Code based approaches are intended to deliver new languages or extend an existing language in order to provide the functionality and flexibility necessary to avoid AOSD problems.

- Kiczales and Mezini in [11] proposes the foundations for Aspect-aware interfaces. Modules interfaces that are aware about the existence of an aspect advising them. They describe how aspects will crosscut modules and how modules will interact between them. This enable modular reason by exposing how aspects interact with the base code, therefore leading to a better evolution support by propagating aspects and base code changes across interfaces.
- Sullivan et al., based on the information hiding principle [21] present a new kind of interfaces called XPI [27], [6]. Abstracting the crosscutting behavior, they impose several documented rules in a design pattern fashion that have no explicit representation in either aspect or base code. This approach constrain the manner in which code is written, making the base code and aspect implementation to rely on the XPI definition. Hence, evolution will be controlled through the XPI.
- J. Aldrich proposes Open Modules [1] , a modular system for aspects, which focuses on the exposure of join pointss such that modules constructs export pointcuts as a part of their specifications. This leads to aspects that depends only on the details exposed by modules specifications and not on the internal details of each module. For this modular system, he proposes a language called "Tiny Aspects", which tries to assure that aspects will not change the program behavior in an unexpected way. This enable programmers to separate concerns and reason about them in a modular way abstracting from hard implementation details.
- Gyble and Brichau propose a prolog-like crosscutting language [7]. This language is intended to describe pointcuts as conditions on properties, then allowing to describe more expressive join points, based more on semantic than structural properties.
- In [20], a prolog like crosscutting language called ALPHA is proposed. Intended to provide a rich model of programming semantics jointly with abstraction mechanism, it allows to write more expressive pointcuts, targeting join pointss by its semantical properties.
- Dantas and Walker proposes harmless advice. A piece of computation as standard advice, but being constrained to prevent it from interfering with the base code underlying computation. This unable the advice to affect data structures or change the base control flow. In [3], they present the core language as well as the operational semantics and type system for this advice system.

- Clifton and Leavens propose *Spectators and Assistants* for AspectJ [2]. In order to enable AspectJ's modular reasoning, they propose to classify aspect according to their effect on program flow. *Spectators*, that do not modify the behavior of the underlying program and *assistants*, that modify the behavior of the underlying program affecting its flow. Then, this classification allows the program modules to accept explicitly the assistance of assistant aspects.
- In [31], an approach to constraint aspect usage is proposed. The underlying idea is to restrict aspect usage by imposing several constraint on their application. Such restriction range from restrict the data that aspects can modify to the actions that advice code can perform when reaching a join points, the moments and places in which aspects are allowed to intervene.
- Recebeli proposes the notion of aspects purity [33]. A pure aspect is an aspect that promises not to alter the behavior of an specified set of base code pieces. He present a prototype of this notion, called *Pure Aspects*, an extension to AspectJ compiler intended to reduce the harm that aspects can do.
- In [24], the concept of Superimpositions is imported from distributed systems to aspects. Then they are a collection of generic aspects and singleton classes that are super imposed with the base code to generate an augmented final program. Superimpositions augment the semantics of AOP by allowing to express interactions and relations among generic aspects, combine collections of them.

All these code based approaches are intended to overcome the AOSD problems, by the proposition of new interfaces [27], [6], [1], [11], languages [7], [20], [3], [31] and extensions to existing languages [33], [2], [24]. But each approach presents its own limitations.

In order to provide harmless advice, Open Modules [1] and Harmless advice [3] constrain the power of AOSD by forbidden aspects to be invasive. XPIs [27], [6] do not define a concrete interface, instead they define a coordinate coding style between aspects and base code. Logic based crosscutting languages [7], [20] can be computationally too expensive like to scale real world applications. Pure Aspects [33] only assure harmless when aspects have pure intentions, but giving no assurance with other aspects.*Spectators and Assistants* [2] offer just a general granularity level about aspect affecting the behavior leaving more fine grain level unexplored (i.e: aspect modifying data fields). Finally, the constraint language presented in [31] appears to be a promising idea, but is just a notion and no details are given.

C. Analysis

Analysis approaches are characterized by the analysis of properties and behavior present in software built using AOSD mechanisms. Most than a solutions, they are an orientations and a tools to predict when AOSD can become a threat to evolution and reliability.

- Koppen and Storzer propose a pointcut delta analysis [14], [26]. This delta analysis operates by comparing the changes in the set of matched join points for two different version of the base code. It serves as a mean to diagnose and help programmers to find bugs and reveal unexpected changes in the behavior of broken pointcuts.
- Guided by the goal of specification and verification of aspect-oriented systems, [9] proposes a regression test for AOSD in order to diagnose if a particular aspect is harmful or not for the base code. Harmful aspects analysis allows a weakening of obliviousness while maintaining extensibility, diagnosing malicious or inadvertent corruption of the desired properties of the underlying system.
- Based on algebraic foundations, a program analysis is proposed [16]. Here aspects are seen as a "program transformation function", or a function that maps programs to programs, and the effects of the weaving process can be understood in terms of algebraic transformations. Around this definition, theoretical properties (commutativity, associativity and identity) and rules are associated to aspect compositions (in e.g. precedence rules for compositions). This allows to reason about composition, exposing its problems.

Aspect analysis appears to be promising to support the evolution of software built using AOSD, by alerting when an aspect may have undesired effects on the system. But those approaches can be either too inefficient [14], [26], [9] (too expensive in computation time) or too abstract [16] (may be impossible to implement without expensive computations) like to scale to real systems.

D. Model based

Model based approaches make use of a high abstraction level to deliver a solution that relies in modeling or meta modeling facilities to avoid AOSD problems.

- In [10], a Model-based pointcut definition is proposed. These pointcuts are defined in terms of a conceptual model of the base program, rather than referring directly to the implementation structure. This results in joint points based on conceptual properties instead of structural properties of the base program, hence leading to a low coupling of the pointcut definition and base code. These pointcuts are called *view-based pointcuts*, because they use the formalism of intentional views to express a conceptual model of a program and to keep it synchronized with the source code of that program.
- In [28], the *Motorola WEAVR* is proposed. This is a tool that can weave aspects defined at model level, by using Specific Domain Languages and UML 2.0 . These aspects are defined as abstract entities based on a transitions-oriented state machine, and the way in which they affect the base code is defined in terms of a novel join point model. This model is composed of call expressions, timer set actions and state transitions. The representations of pointcuts and advises (called *connectors*) is a finite state machine, where *connectors* always contains a start and

an end state. This approach enables aspects to be defined in terms of a system specification without requiring a complete knowledge of its implementation, then giving additional robustness to aspects.

The model based approaches abstract from code properties (eliminating the tight coupling between aspects and code) and bring generality to the aspect representation. Then, being promising to overcome evolution problems. Nevertheless, these approaches need a deeper study to determine the concrete impact of using models to define aspects and overcome the AOSD problems.

E. Contracts for Aspects

These approaches make use of Design by Contract [19] in order to avoid AOSD problems and increase the confidence in aspects usage.

- Skotiniotis and Lorenz propose some notions for Design by Contract applied to aspects [25]. They emphasize in the help that Design by contract can give to ensure, that aspects capture crosscutting concerns adequately, and aspects do not interfere with some other parts of the program. Later in [17] they study the combination of aspects and contracts, then generating a classification of aspects according to its interaction with contracts and determining who must be blamed in the case of contract violation. An implementation of this classification has been made in CONA an aspect based Design by Contract tool for Java.
- In [13] assertions are used as a means to validate aspect composition, verifying if a class or an application contains a suitable sets of aspects for whom the weavage can be validated. This validation is performed in respect to a set of specifications included as a contract that assess the correctness of the composition according to the design. Those specification can, for example, prohibit the application of two aspects in the same place, or enforce the interdependence of two aspects.
- In [15], *Aspect integration contracts* (AiC) are presented. Based on the idea of contracts for components, these contracts specify the permitted interference between an aspect and the base code. AiC are composed of the *aspect requirements* specifications (what aspects require from the base code), *aspect functionalities and effects* specifications (what aspect do with the base code), and the *permitted interference* specification (what aspect can do with the base code). Then, these contracts state that all the specifications are respected in order to weave aspects with the base code.

In [25], [17], the interaction between contracts and aspects is studied. It gives an idea of how could be the blaming process in the case of assertions violation. But leaves several questions on the table like "How aspects can be contractualized to restrict their application to base program?" or "How contracts can assure that aspects adequately capture crosscutting concerns?". The approach of [13], allows to reason about inter-aspects interaction and aspects-methods interaction by defining weaving specification in a Design by Contract fashion. Finally, AiC [15] have a lot of specifications about specific aspects and the interference that the base code allows from them. Then, may being too complex and specific to a software version like to be practical and scale to the real world.

We think that the current approaches to solve AOSD problems do well a part of the work, being partial solutions. But instead, we hope that a complete solution will be one that controls and does not constrain aspects invasiveness and suits aspect to better support evolution in a concrete fashion.

IV. Towards Contracts

Design by Contract [19] has proved its value in the object-oriented world, by specifying what a program is intended to be through precondition, postcondition and invariants. The presence of contracts in an object-oriented application increase its reliability and the confidence that its components have [30].

We think that contracts applied to aspects will result in a good way to reduce the AOSD problems and increase the confidence that developers have in aspects. Those contracts will bind responsibilities to aspects by specifying what the base code expects from them, thus ensuring that only aspects that satisfy contracts will have the right to interact with the base code.

Our aim is not to constrain the power of AOSD, but control aspects invasiveness and fit aspects to better support evolution. By being vigilant about the usage of AOSD, developers and designers reduce the risk of unexpected side effects and the harm that aspects can do.

A. Two sides contracts

In our approach, contracts are represented by a two-side specification. One side belongs to the base code, and specify what kind of aspects are allowed to interact with a given piece of the base code. The other, belongs to aspects and define which kind of advices it contains by placing them through a classification base on their incidence over the base code.

These contracts are roughly similar to their defined by Design by Contract. But instead of preconditions,postconditions and invariants, we have just preconditions and data invariants. While preconditions and data invariants are represented by the base contract side; the aspects side contract serves as a mean to determine if a given aspect can satisfy the conditions stated in the base code contract side. Aspect side classification can be determined automatically by performing static analysis over the aspect code. However, we think that the manual determination of that classification can report benefits by specifying what the aspect behavior must be, instead of rely on hard code deduction.

B. Aspects Classification

In order to allow contract to specify what kind of aspect are allowed to interact with the base code, we propose an aspect classification based on their incidence over the base code. For this classification we first sight three major groups of aspects:

1) Pure behavioral aspects, which only intervene the protected control flow of the base code.
2) Pure data aspects, which only manipulates protected data structures of the base code.
3) Hybrid aspects, which not only intervene the protected control flow of the base code, but also manipulate its protected data structures.

Inside each one of this groups we have identified different kind of aspect advice that vary on harm level according on the actions they perform. For example, an aspect which modifies protected data structures and override protected methods will be harmful than an aspect that just read values from protected data structures.

This classification will allow us to specify in the contract, which kind of aspects is desired to advise the base code, or to protect the base code against aspects that can do unintentional harm.

C. *Contracts violation and evolution*

If a contract is violated, it may mean that aspects are not well suited to advise the base code, or base code is not well suited to be advised by aspects. Consequence of this, base code may needs to be refactored in order to be well suited to aspects or aspects might need to be reformulated in order to be well suited to advised the base code.

In the case of evolution, contracts will inform when aspects capture a wrong join point or do not hold the behavior expected by the base code, because it will violates base code specifications. Hence, when the source code evolves independently from aspect, we have a mechanism to know if aspects will perform harmful.

D. *No more obliviousness*

The obliviousness property of AOSD requires aspects to be transparent for the base code, thus meaning that the base program must be ignorant about the aspects that advice it. The placement of contract between the aspect and the base code breaks the obliviousness property of aspects and make the base code aware about the existence of aspects that can be weaved with it. We think that the obliviousness property is just desirable, and as exposed by Rashid and Moreira in [22], abstraction, modularity and composability are more fundamental properties.

E. *Close approaches*

Some of the previously presented approaches are close to our proposition. The regression test proposed in [9] have similarities because it tries to state when an aspect can be harmful by making use of structural properties of the aspect and the base code. Our aspect classification is similar to the classification proposed in [23], but instead of focus our attention on the interaction aspect-base code, we focus our classification on the behavioral incidence of aspects over the base code. Pure Aspects [33] only assures that aspects presenting purity will perform harmless. Contracts not only assure that aspects with "purity" will perform harmless but also inform about the harm that non pure aspects can do.

The classification of *Assistants and Spectators* [2] is quite similar to our flow classification of aspects, but we try to address a more fine granularity level and the case when aspects affect data. While this approach breaks the obliviousness property in a such way that the base code is aware about specific aspect advising it, then adding coupling between the base code and the aspects, we make the base code aware about a special kind of aspects (defined in our classification) that can potentially advise it.

Aspect Integration contracts [15] are very closer to our idea. But, instead of make contract assertions on specific aspects (referencing them by their names), we make assertions on our proposed classification in order to give more generality and simplicity to our contracts.

While the approach proposed by Open Modules [1] and Harmless advice [3] constraint the power of AOSD in order to achieve less invasiveness, we do not constrain the power of AOSD, instead we try to save its benefits by controlling it and telling when it can be harmful. Interfaces approaches like XPI [27], [6] and Aspect-aware interfaces [11], propose to control the evolution by making aspects and the base code dependent on a single interface. This abandoning the obliviousness property by making the base code aware about aspects. Instead, we abandon the obliviousness property, by establishing a mechanism that do not synchronize the aspect - base code evolution, but informing when aspects will break the base code specifications.

Current contract for aspects approaches [25], [17], [13] are focused on the establishment of pre and post conditions in the Design by Contract fashion, then trying to specify what the weaved application must be. Instead we believe that it is more adequate to adopt contracts that make assertions on what how aspects can influence the base code, then specifying what the base code expect from aspects. The aspect usage constraint approach [31] is similar to our notion, but instead of constrain single aspects according to given conditions, our aim is to specify what well classified aspects are allowed to interact with.

We think that the guidelines proposed by Wampler in [32], can be used with our approach of contracts in order to reduce the coupling of aspect with the base code and encouraging the use of specifications. Approaches like [20], [3], [31] that propose new crosscutting languages or guidelines like [18], [29], [12] are completely compatible with our approach because they try to change, augment or advise the way in which join point are selected, while we try to specify what aspect advise can do with the advised join points.

V. Conclusions and Future work

The aim of this paper was to show the current problems that without a solution presents a major threat against a mainstream adoption of AOSD. By exposing the current approaches to solve them, we were able to reason about their strengths and limitations, then proposing a novel approach based on

Design by contract principles. This approach will increase the confidence in aspects, allows to be vigilant about their usage by controlling their incidence over the base code and give earlier information when facing evolution.

Our current work is focused on validate this approach by the development of a prototype to suit the AspectJ compiler with our notions of contracts.

References

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[2] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, March 2002.

[3] Daniel S. Dantas. Harmless advice. *ACM SIGPLAN Notices*, 41(1):383–396, January 2006.

[4] Bart De, Win Frank, Piessens Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering, June 03 2002.

[5] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[6] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, pages 51–60, January/February 2006.

[7] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, pages 60–69, 2003.

[8] http://www.aspectj.org. Aspectj.

[9] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, March 2004.

[10] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 501–525. Springer, 2006.

[11] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of the 27th International Conference on Software Engineering*, pages 49–58. ACM, 2005.

[12] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[13] Herbert Klaeren, Elke Pulvermueller, Awais Rashid, and Andreas Speck. Aspect composition applying the design by contract principle. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 57–69, London, UK, 2001. Springer-Verlag.

[14] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[15] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing semantic interference with aspect integration contracts. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect*, March 2004.

[16] Roberto E. Lopez-Herrejon, Don S. Batory, and Christian Lengauer. A disciplined approach to aspect composition. In John Hatcliff and Frank Tip, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 68–77. ACM, 2006.

[17] David H. Lorenz and Therapon Skotiniotis. Extending design by contract for aspect-oriented programming, January 24 2005.

[18] Nathan McEachen and Roger Alexander. Distributing classes with woven concerns—an exploration of potential fault scenarios. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 192–200. ACM Press, March 2005.

[19] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[20] Klaus Ostermann, Mira Mezini, and Christophe Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of ECOOP 2005*, 2005.

[21] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[22] Awais Rashid and Ana Moreira. Domain models are NOT aspect free. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2006.

[23] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, October 2004.

[24] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[25] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 196–197, New York, NY, USA, 2004. ACM Press.

[26] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*, pages 653–656. IEEE Computer Society, 2005.

[27] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, 2005.

[28] T. Elrad T. Cottenier, A. van den Berg. Motorola weavr: Model weaving in a large industrial context. In *In Proc. of the 6th Int. Conference on Aspect-Oriented Software Development, Industry Track (AOSD)*. Motorola, ACM, March 2007.

[29] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the AOSD-evolution paradox. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.

[30] Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng*, 32(8):571–586, 2006.

[31] Shmuel Tyszberowicz. Constraining aspect usage. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2005.

[32] Dean Wampler. Noninvasiveness and aspect-oriented design: Lessons from object-oriented design principles.

[33] Elcin Recebli Wolfson. Pure aspects. Master's thesis, Oxford University, September 2005.