

Validation of Context-Dependent Aspect-Oriented Adaptations to Components

Thomas Cottenier, Tzilla Elrad
Concurrent Programming Research Group
Illinois Institute of Technology, USA
{cotttho, elrad}@iit.edu

Abstract

All potential component non-functional properties needed by third parties for late composition can not be anticipated beforehand, as they tend to be context-dependent. Moreover, those quality properties tend to be hard to modularize and can hardly be factored out of components. Developers therefore tend to embed non-functional properties within domain-specific frameworks and components. We aim to take advantage of the expressive power of Aspect-Oriented Programming to modularize those concerns, so that both frameworks and components can be refined to satisfy specific requirements at deployment time, composition time as well as at runtime. We claim that the key to the integration of Aspect-Oriented Software Development to Component-Based Software Engineering lies in the ability to derive and validate the specification of a component that has been subject to aspect weaving. By explicitly specifying the desired properties of components and aspects, we aim to increase the capability to reason about aspect weaving, so that the correctness of a refined component can be verified with respect to a specification.

1. Introduction

In the context of late component composition by third parties, it may not be possible to anticipate all non-functional requirements to which a component may be subject. Those non-functional requirements tend to be dependent on the application domain, the deployment context, and the execution context. Therefore, in order to provide generic components for late composition, non-functional properties and context-dependent concerns imperatively need to be factored out of the components and their integration into the system needs to be an integral part of the composition process.

Unfortunately, by their nature, these concerns tend to crosscut the component structure, within components (intra-component crosscutting), as well as across the components boundaries (inter-component crosscutting). It is therefore very hard to cleanly modularize those concerns into regular components. Developers therefore tend to embed non-functional properties within domain-specific frameworks and components.

Inter-component crosscutting needs to be addressed by component frameworks. Generic frameworks should be customizable to support particular quality attributes. Intra-component crosscutting is a composition issue between the core functionality of a component and various additional concerns. In both cases, the various extensions needed cannot all be anticipated beforehand. We propose to take advantage of the expressive power of Aspect-Oriented Programming (AOP) to modularize those concerns so that both frameworks and components can be refined in a non-invasive way, at deployment time, composition time and runtime.

Aspect-Oriented Software Development (AOSD) [1][2] is an extension to other software development paradigms that allows capturing and modularizing concerns that crosscut a software system into modules called aspects. Aspect-Oriented Programming (AOP) makes very powerful program transformations possible, through a composition process where aspect *advice*s are *woven* into the core program at locations called *pointcuts*. Members and methods can also be inserted in classes through an aspect construct called *introduction*. Aspects have the ability to introduce functionality in a core program in a non-invasive way, making it possible to alter the behavior of a system a posteriori. Aspect weaving can be done either at compile time, load time, or even at runtime. We aim at taking advantage of dynamic, non-invasive AOP capabilities to achieve both framework customization and component adaptation. Aspects have a more general scope than non-functional concerns; they apply to all concern that has crosscutting semantics. However, in this paper, we will focus on the non-functional properties of components.

The rest of the paper is organized as follow. In Section 2, we discuss challenges to the integration of AOSD into CBSE. In section 3, we discuss some existing AOP approaches to CBSE, and introduce Aspect-Sensitive Components. Section 4 and 5 discuss the weaving process itself and component profile specifications. Section 6 illustrates the model by showing how it may be applied to Web Service composition, and finally, Section 7 concludes this paper.

2. Challenges to the integration of AOSD into CBSE

Research on integrating AOSD with Component-Based Software Engineering (CBSE) [3][4][9] has inspired component frameworks such as JBoss [11] to adopt AOP concepts. JBoss AOP primarily deals with inter-component crosscutting. The component framework can be extended transparently with system level aspects such as caching, persistence, transactions and security. As such, those aspects act on the interactions between the framework and the deployed components by intercepting invocations at the boundaries of components.

Addressing intra-component crosscutting with AOSD is trickier. It poses important challenges with respect to component encapsulation and certification, as well as composition predictability.

First, components are mostly provided as Black Boxes. By hiding implementation details that are likely to change, components are more evolvable, as the potential inter-component dependencies are specified by the interface. However, component interfaces are not well suited to express non-functional properties, for the very reason that those properties are not modularized inside the component. As clients may precisely rely on those quality attributes, dependencies that are not controlled by the component interface arise. AOP has the potential to cleanly modularize those concerns, and as such, to customize component quality attributes to the deployment context. To do so, intra-component aspects need access to the component implementation, which directly clashes with component opaqueness. Because intra-component aspects have to rely on implementation details, they create dependencies that impede system evolvability. This is known as the AOSD-evolution paradox [13]. Because current crosscut languages are too low level, intra-component aspects are tightly coupled to the component implementation. When swapping component implementation, aspect pointcuts generally need to be modified as well.

Second, the expressive power of AOP comes with a price. It can be hard to evaluate the potential consequences of letting aspects slip into a system, as they can have subtle side effects.

The use of wildcards in pointcut expressions makes it possible for aspects to be applied at unwanted locations. Aspect introductions can result in subtle modification of the system behavior through binding interferences. New members can be inserted and classes can be pushed down the class hierarchy, and thus, methods can be redefined on the way. Aspect advices can modify the runtime behavior of the system. The state and the control flow of the system can be radically affected, and the effects of advice may be difficult to foresee. At the code level, assessing those effects requires complex and expensive code-analysis. As composition predictability is an essential requirement for component-based systems, weaving predictability imperatively needs to be addressed.

Third, component certification is compromised. Certification provides guarantees about the essential properties of the component, such as quality attributes. As we aim to modularize those properties using aspects, there is an essential need for support that addresses reasoning on the composition of component and aspect specifications. Because of the lack of aspect weaving predictability, it may be very difficult to guarantee properties of components that have been subject to aspect weaving.

Aside from the AOSD-evolution paradox, the most serious brake to the integration of AOSD to CBSE lies in the lack of support for weaving compositional reasoning. The difficulty is inherent to the nature of aspects. While AOSD makes it possible to encapsulate crosscutting concerns syntactically, the semantics of aspects are still crosscutting.

3. Aspect-Sensitive Components

Several attempts have been made to integrate AOSD to CBSE. The proposed approaches vary in terms of aspect expressiveness, component obliviousness and component transparency.

Component obliviousness reflects the degree to which developers have to take potential aspects into account when developing components. It is generally acknowledged in the AOP community that core program developers cannot be totally oblivious to potential aspects. While concerns such as logging can be added to a component transparently, other concerns such as transaction management or concurrency require the core program to be aware of them. The use of annotations might be one way to explicitly make the core aware a particular aspect is weaved. Yet, this approach requires invasive changes. Once the core program exhibits appropriate aspect-awareness, those concerns can be modularized, and introduced in the core program non-invasively.

Component-based AOP frameworks such as JasCo [18] compromise aspect expressiveness to preserve component opaqueness. Aspects are only allowed to operate on the execution points that are exposed in the component interface and aspects are not allowed to extend a component interface through introduction. As such, they only allow superficial component adaptations. As component interfaces are unlikely to be modified, those approaches do not suffer from the AOSD-evolution paradox.

Open modules [15] make more expressive adaptations possible by providing the ability to explicitly expose component pointcuts in the component's interface. Component developers have therefore to give up some obliviousness regarding aspects, as they have to anticipate potential aspect so that the corresponding pointcuts can be exposed. However, the potential adaptations needed in a given context cannot all be anticipated during component development. Moreover, it doesn't solve the AOSD-evolution paradox, as the component developer has to commit to the fact that the exposed pointcuts are not subject to change.

We aim to provide an approach that preserves the expressive power of AOP, meaning that we don't want to restrict the range of potential component adaptations. We believe the key requirement to avoiding uncontrolled aspect weaving is that components have to compromise obliviousness at a higher level. Components have to provide a more detailed profile of their desired properties than what they usually offer, so that the consequences of aspect weaving can be assessed. Those properties need to be verifiable. The party supervising the weaving process then has enough information at its disposal to be able to make a decision on whether or not a given aspect is allowed to interfere with a component.

We call aspect-sensitive component a component that behaves as a black box with respect to other components, and whose potential pointcuts and desired properties are visible to Aspects. Correlating pointcuts and properties might also provide a way to improve the expressiveness of crosscut languages and reduce the syntactic coupling they introduce.

Manually providing a complete profile for each aspect and component might be an arduous task. However, with the advent of Model-Driven Architecture technologies, we envision that such profiles might be generated from component models. MDA requires models to be more complete, especially regarding program semantics. By taking different views of those models along various dimensions of concerns, we expect to be able to extract information on the essential properties of component.

4. Composition of Aspect-Sensitive Components with Non-Functional Aspects

The main goal of our work is to develop a model that would allow components to be dynamically customized with non-functional properties using AOP. The key requirement is that the resulting component should exhibit a predictable behavior, and that its correctness can be checked with respect to a specification. Predictability is conditioned by the ability to reason about weaving composition. We therefore associate a profile to both components and aspects that describes their essential properties.

Non-functional requirements of components tend to be context-dependent. The behavior to be inserted in components might depend on the deployment environment, on the runtime environment or on the composition process itself.

- At deployment time:
System-level concerns such as session, transaction or data access are added to the component at deployment time, through the deployment of interceptors at the component boundaries. This is also the model adopted by container-based component deployment models such as EJB or CCM. JBoss AOP treats those concerns as aspects.
At deployment time, aspect introductions can also be used to implement framework specific interfaces needed by the component for the framework to be able to manage its resources. Generic aspects can be developed for specific frameworks, making it possible to non-invasively enable components for various component frameworks. For example, we developed a “Gridifier” aspect that transparently enables Web Service implementations for deployment in the Globus Grid environment.
- At composition time:
Since aspects have the power to modify the semantics of components, they can be used both as semantic adapters for composition as well as interface adapters, making composition possible where component semantics and/or interfaces were initially not compatible.
In the area of pervasive computing, for example, the composition process can itself be highly context dependent. New components might be discovered and integrated to the system at run-time. Dynamic composition involves a negotiation process where trade-offs between components might have to be found. The ability to modify component non-functional properties during this process might be a key to achieve an agreement, and allowing new components to be integrated into the system. Aspects might play the role of interface adapters, making composition possible when component incompatibilities arise, for example, in heterogeneous environments.
- At run-time:
The system specification might evolve after system composition. A component can adapt its behavior to a new context, as long as it still satisfies the contracts it agreed on with other components and the framework it is deployed in. Moreover, contracts themselves can be adapted at runtime, as long as all contract participants agree on the changes. For example, a component and its client might come to interact in an unsecured environment, which requires triggering a communication encryption aspect.

5. Aspect-Sensitive Component Profile

To improve our ability to reason about weaving composition, we associate profiles to both components and aspects, describing their essential properties. We need to be able to derive the profile corresponding to the woven component in a reliable way, meaning that we need to be able to prove that the customized component satisfies the generated profile. The semantic interactions between aspect and core need therefore to be better understood.

Let $*$ be the weaving operator, and \circ the corresponding profile weaving operator, the model should guarantee that a component $C=M*A$ satisfies the profile $P_C=P_M \circ P_A$, given that the aspect-sensitive component M satisfies profile P_M and aspect A satisfies profile P_A .

We think linguistic support based on the weaving operational semantics might be needed for that purpose. [16][17]

At the code level, assessing those effects requires complex and expensive code-analysis. Because the desired properties of components and aspects are first specified during the analysis and design phases, we think the development of aspect and component profiles suitable to weaving reasoning should be an integral part of the development process.

Model-Driven Architecture technologies could be used to derive component and aspect profiles for compositional reasoning. MDA approaches further shift the focus of software development towards models,

from which implementations can be generated using mapping techniques. Special mappings could therefore be used to semi-automatically generate component and aspect profiles.

The fact that a composed profile violates the system specification would not necessarily be reverberated at the code level as a bug. It would signal that the Aspect interacts with sensible semantics of the core system, and that appropriate action may need to be taken.

Both Aspect and Aspect-Sensitive Components need to specify

- **Semantic contracts**
Those contracts specify the behavioral properties of the component methods and aspect advices in the form of pre and post-condition assertions and invariants. In a Model-Driven development environment, the semantic contracts might be expressed in OCL, using a UML profile supporting Aspect-Oriented modeling.
- **Temporal Properties**
As aspects can easily introduce deadlocks into a system, we need to be able to reason about safety and liveness properties. The practicability of completely specifying the temporal properties of all components and aspects in a system may, however, be limited. Note that speculative aspects, such as logging, can be classified as harmless without requiring model checking.
- **Quality attributes**
Aspects can refine components so that they address complex custom quality requirements. The final quality attributes of the woven component should, however, be predictable. QML-style [8] contracts could be used to qualify both components and aspects.
- **Access Control contracts**
Aspects have the power to completely break component encapsulation. Fields that are declared as being private can be accessed and exposed by aspects. We may want to prevent sensible variables from being accessed or modified by aspects. Dataflow analysis techniques, such as def-use relationships used for compiler optimization, might be used to detect encapsulation breaches.

To support dynamic component adaptation, component and aspects profiles need to keep existing at runtime. Moreover, profiles might be context-dependent, in the sense that we might have to query the environment before assertions can be checked for.

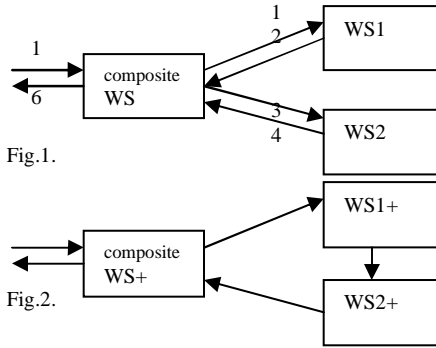
The system specification, with respect to which aspects are validated, might itself be subject to changes at runtime, in response to changes in the system environment. Each time the system specification changes, revalidation of the applied aspects has to be triggered, which may lead to the unweaving or weaving of aspects until the new specification is met.

6. Application to Web Service Composition

The following example illustrates the expressive power of dynamic AOP, and its potential use regarding web service composition. Web services aim at being building blocks for applications. Composite web services can be composed from multiple component web services given a workflow specification, in a language such as BPEL [11].

The web service composition process typically relies on a centralized engine that coordinates the composed services according to the workflow specification.

Fig. 1 shows a simple sequential composition of two web service. For some application, the centralized composition engine can become an important bottleneck. To address scalability issues, we may want to decentralize the composition, and alter the behavior of the web services so that synchronous RPC based calls are replaced by asynchronous messaging, such as depicted in Fig. 2.



In the case of web services, the life cycle of the component cannot be explicitly controlled through client interactions. We would like to be able to customize a web service to service a specific request, while maintaining the basic functionality of the service available to other users. We therefore adopt an object factory model, where the factory object is exposed as a web service. A customized version of a service can be deployed on a host on demand, with a new URI that is used as a component reference.

The composite web service host sets up the composite web service by sending weaving requests to the target web service environments, which deploy a customized version of the service, and return the endpoint URI for the new web service. We developed AspectWerkz aspects [10] that allow synchronous RPC-based request to be replaced by asynchronous messages to other services. Moreover, synchronization aspects can be woven directly into the web services to meet the workflow specification.

As the messaging aspect modify the web service interface, we have to simultaneously adapt the web service specification (WSDL) as well as the SOAP binding classes. Instead of generating a new WSDL from the customized web service, we generate the new WSDL from the old web service specification, and an aspect specification, which in this case, is an XSL file. The new WSDL file is then generated using a XSL transform. From the XSL file, we also automatically generated the aspects adapting the SOAP binding classes.

This example transforms a Web service WSDL file according to an XSL file that reflects the Web Service adaptation. We believe this approach could be generalized to compose component and aspect profiles.

7. Conclusions

We aim to take advantage of the expressive power of Aspect-Oriented Programming to adapt components to their deployment, composition, and run-time contexts. This expressive power has however to be controlled in one way or another, as aspects may have subtle side effects.

By explicitly specifying the desired properties of components and aspects, we aim to increase the capability to reason about aspect weaving, so that the correctness of the refined component can be verified with respect to a system specification.

Further work includes developing a formal model of profile composition based on the weaving operational semantics, so that it can be proved that components that have been subject to aspect weaving indeed satisfy the composed profile. The resulting profile can then be used to assess and validate the consequences aspect weaving may have on component properties.

References

- [1] Aspect-oriented programming, G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LCNS. Springer-Verlag, June 1997.
- [2] Aspect-oriented programming is quantification and obliviousness, R. Filman and D. Friedman. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

- [3] DAOP-ADL: An architecture Description Language for Dynamic Component and Aspect-Based Development, Monica Pinto, Lidia Fuentes, and Jose Maria Troya. In *Proceedings of the second international conference on Generative programming and component engineering (GPCE)*, number 2830 in LNCS. Springer-Verlag, 2003.
- [4] An implementation architecture for aspect-oriented component engineering Grundy, J.C. In *P, roceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications: Special Session on Aspect-oriented Programming*, Las Vegas, June 2000, CSREA Press
- [5] A semantics for advice and dynamic join points in aspect-oriented programming Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn.. In *Informal Workshop Record of FOOL 9 and FOAL (Workshop on Foundations of Aspect-Oriented Languages)*, AOSD 2002, 2002.
- [6] A Calculus of Untyped Aspect-Oriented Programs, R. Jagadeesan, A. Jeffrey, and J. Riely. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1853 in LNCS. Springer-Verlag, 2003.
- [7] Specification Transformers: a predicate transformer approach to composition, Michel Charpentier, K. Mani Chandy. In *Acta Informatica* 40. Springer-Verlag 2003.
- [8] QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Frolund, S., Koistinen, Hewlett-Packard Laboratories. 1998
- [9] The COMQUAD Component Model. Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects, Steffen Göbel, Christop Pohl, Simone Röttger and Steffen Zschaler. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, ACM, 2004.
- [10] AspectWerkz homepage <http://aspectwerkz.codehaus.org>
- [11] Aspect-oriented programming and JBoss, Bill Burke and Adrian Brock. O'Reily ONJava.com, May 2003. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html
- [12] Business Process Execution Language for Web Services Specification, BEA, IBM, Microsoft, SAP AG and Siebel Systems.
<ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf>
- [13] On the Existence of the AOSD-Evolution Paradox, Tom Tourwe, Johan Brichau, Kris Gybels In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003
- [14] Programming with Aspectual Components, Lieberherr, K., Lorenz, D. and Mezini, M. Technical Report, NU-CCS-99-01, March 1999
- [15] Open Modules: Reconciling Extensibility and Information Hiding, Jonathan Aldrich, Workshop on *Software-engineering Properties of Languages for Aspect Technologies* as part of AOSD 2004, 2004
- [16] Linguistic provisions for Aspect/Core semantic interactions, Thomas Cottenier, Tzilla Elrad, Workshop on *Software-engineering Properties of Languages for Aspect Technologies* as part of AOSD 2004, 2004
- [17] Diagnosis of Harmful Aspects Using Regression Verification, Shmuel Katz, *Foundations of Aspect-Oriented Languages Worksho* as part of AOSD 2004, 2004
- [18] JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development, Suvéé, D., Vanderperren, W., and Jonckers, V. In *Proc of the second international conference on aspect-oriented software development (AOSD)*, Boston, USA, march 2003.