

Overall Aspect-Oriented Analysis and Design Approach

ABSTRACT

The implementation of software has benefited from the advanced concern separation mechanisms the Aspect-Oriented paradigm enables, allowing for the textual localization of scattered and tangled concerns in source code. In this deliverable, we summarize an approach that shows how AOP can be made beneficial to activities taking place earlier in the software development life cycle. Specifically, we apply AO techniques to the requirements, architecture, and design development phases, and furthermore show how each phase can be suitably adapted to flow seamlessly into subsequent phases.

Document ID:	AOSD-Europe-ULANC-49
Deliverable/Milestone No:	D132
Workpackage No:	6
Type:	Research
Status:	FINAL
Version:	V4
Date:	September 1, 2008
Author(s):	Raffi Khatchadourian, Ruzanna Chitchyan, Phil Greenwood, Awais Rashid (Lancaster University, UK) Juan A. Valenzuela, Luis M. Fernández, Mónica Pinto, Lidia Fuentes (University of Malaga, Spain) Andrew Jackson, Siobhán Clarke (Trinity College, Dublin, Ireland)

History of Changes

Version	Date	Changes
1	2008-07-23	Initial version
2	2008-08-07	Initial inputs about the process
3	2008-08-22	Editing and revisions
4	2008-08-31	Final editing

Contents

Contents	3
List of Figures	4
1 Introduction	5
2 Aspect-Oriented Requirements Engineering Approach	6
2.1 Requirements: a Contract in Natural Language	6
2.2 Requirements Analysis: EA-Miner	7
2.3 The Requirements Description Language	7
2.4 Multi-dimensional Requirements Analysis Tool	9
3 From Requirements to Architecture	10
4 Aspect-Oriented Architecture Approach	12
4.1 Multiple Views Approach	12
4.2 The AO-ADL Architecture Description Language	12
4.3 The Visual Notation	13
4.4 The AO-ADL Tool Suite	13
4.5 Architectural Evaluation Process/Metrics	14
5 From Architecture to Design	16
6 Aspect-Oriented Design Approach	18
6.1 Aspect-Oriented Design Language	18
6.2 Aspect-Oriented Design Process	18
6.3 Aspect-Oriented Design Tooling	19
6.4 Aspect-Oriented Design Demonstrator	19
7 From Design to Implementation	20
7.1 Mappings to Implementation	20
7.2 Tools for Generation AO Implementation from AO Design	20
8 Towards a Traceability Framework	21
9 Conclusion and Future Work	23
References	24

List of Figures

1	Overview of the AORE approach	6
2	RDL meta-model.	8
3	Requirements to architecture mapping process	10
4	AO-ADL Tool Suite	13
5	AO-ADL Tool Suite flow of information	14
6	Architecture to design mapping process	16
7	Intra- and Inter-phase traceability information.	21

1 Introduction

Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) has emerged in order to localize the scattered and tangled implementations of crosscutting concerns in source code, allowing developers to characterize that certain actions should be taken at specific points during the execution of software. Since its inception, various authors (Dantas and Walker 2006; Deng et al. 2004; Laddad 2003; Lippert and Lopes 2002; Rashid and Chitchyan 2003) have shown how aspects may be used to write localized implementations of important, cross-cutting concerns such as process synchronization, event logging, exceptional situation handling, data persistence, and security, respectively.

The implementation of software has benefited from the advanced concern separation mechanisms the Aspect-Oriented paradigm enables. In this deliverable, we summarize an approach that shows how AOP can be made beneficial to activities taking place earlier in the software development life cycle. Specifically, we overview the AOSD-Europe Aspect-Oriented Analysis and Design approach developed within the Analysis and Design laboratory which allows software engineers to cleanly separate concerns contained within a multitude of software engineering artifacts, including requirements, architecture, and design. By separating concerns in this fashion, each can be studied and reasoned about in isolation, and also later *composed* in order to produce a complete system of artifacts.

This deliverable is organized as follows. Section 2 discusses the Aspect-Oriented Requirements Engineering (AORE) approach which applies AO techniques to the requirements phase of the software engineering life cycle. It also touches upon how a requirements document may be *mined* for potential aspects. Section 3 outlines how the information obtained from the AORE approach may be leveraged to build a suitable software architecture. Section 4 portrays the Aspect-Oriented Software Architecture (AOSA) approach, explaining how AO concepts have been adapted to this phase. Section 5 follows by detailing how an AO architecture may be translated into appropriate design artifacts. Section 6 confers about the Aspect-Oriented Design approach, while Section 7 considers how an AO design may be translated into an actual implementation skeleton. Finally, Section 8 discusses a future traceability framework which may be used to further reason about the information generated using our analysis and design approach. Section 9 concludes by summarizing our results and briefly deliberating about additional future endeavors.

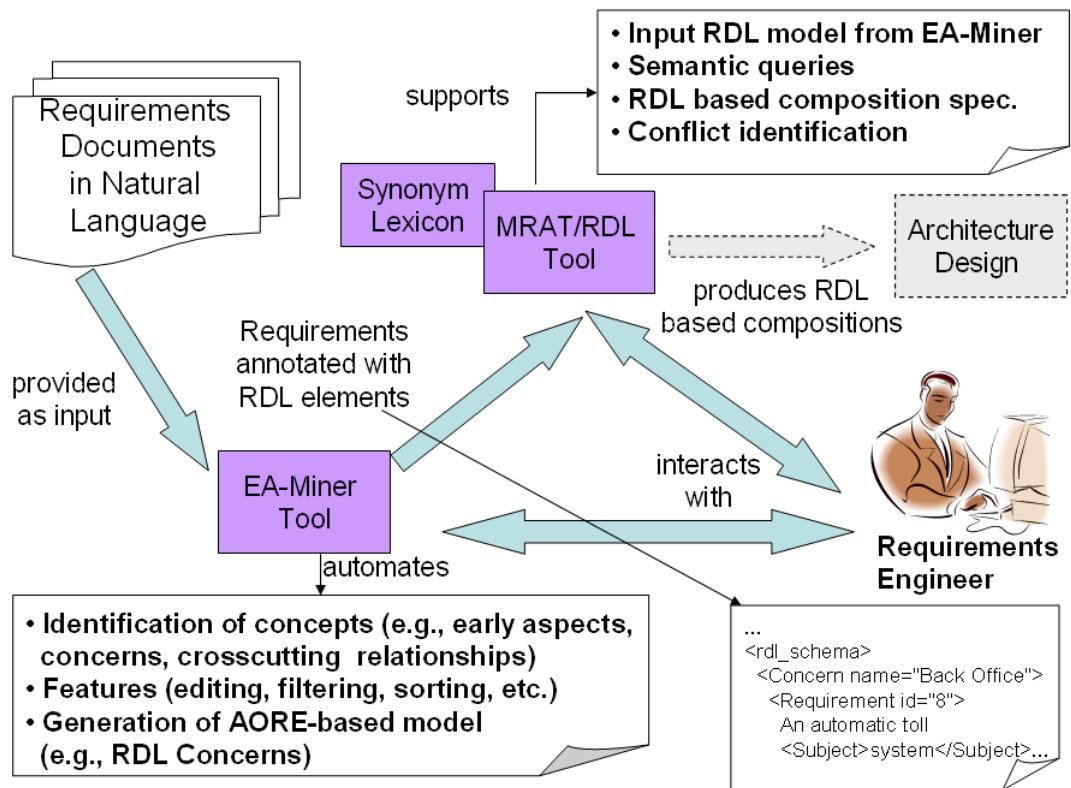


Figure 1: Overview of the AORE approach

2 Aspect-Oriented Requirements Engineering Approach

In this section, we overview the AOSD-Europe Aspect-Oriented Requirements Engineering (AORE) approach developed within the Analysis and Design laboratory. The techniques presented here enable software engineers to, among others, cleanly separate concerns contained within a requirements document and later *compose* these concerns in order to understand the complete set of system requirements as a cohesive whole. This composition allows engineers to be able to focus on concern in *isolation*, thus providing a vital level of requirement abstractness which is especially useful in large, complex systems. Figure 1 highlights some of the key elements of the approach, while the sections below elaborate on its constituent parts.

2.1 Requirements: a Contract in Natural Language

Software requirements documentation often serves as a binding *contract* between developers and the stakeholders. As such, it is a document that is intended to be understandable and comprehensible by *both* parties. Therefore, requirements are meant to be written in a *natural* language to accommodate the multiple parties regardless of their level, type, or lack of technical background.

Although a natural language is well-suited for this purpose, processing and analyzing natural language is, indeed, challenging. Natural language semantics are often vague, ambiguous, and occasionally implicit. Thus, *automated* analysis and processing of text at the abstraction level of requirements must be robust against such obstacles. However, the AORE approach presented here incorporates a unique user interaction-based feedback system to combat much of these difficulties.

2.2 Requirements Analysis: EA-Miner

The Early Aspects (EA) Miner tool, built using our EA Identification Method (Early-AIM), provides effective automated support for identifying and separating crosscutting (called early aspects) and core (i.e., non-crosscutting) concerns, as well as their crosscutting relationships at the requirements level (Chitchyan et al. 2006b). The Eclipse¹-based tool utilizes natural language processing techniques, e.g., part-of-speech and semantic tagging, word frequencies, to reason about the properties of the concerns and model their structure and crosscutting relationships.

Once EA-Miner has identified these concepts, it then enables developers and stakeholders a like to navigate concerns within the requirements documentation. These innovative features include, but are not limited to, editing, filtering, and sorting of concerns at the natural language level. Another key feature of EA-Miner in regards to the complete AORE process is its ability to export the analyzed data to a Requirements Description Language (RDL) format (see Section 2.3). The concerns manifested in the AORE-based RDL model are then provided as input into the remainder of the process.

2.3 The Requirements Description Language

The Requirements Description Language (RDL) (Chitchyan et al. 2006b) can be used to express concern representations in a very explicit and clear fashion, providing a rich representation schema of concern relationships. The RDL does so by *annotating* requirements, given in a natural language, with important, semantic information that makes processing and converting them to other representations in future steps of the software engineering process (e.g., architecture) easier. These annotations make relationships in the textual representation of the requirements clear and explicit, thus avoiding possible ambiguity in interpretation. Explicitly documented relationships within the requirements document also help to make the requirements more comprehensible.

Figure 2 portrays the generic RDL meta-model which shows how relationships amongst requirements are explicitly captured. A single requirement is aggregated of a textual description, a subject, an object, and a sequence number that constrains the ordering of the requirement. As the figure depicts, the

¹<http://eclipse.org>

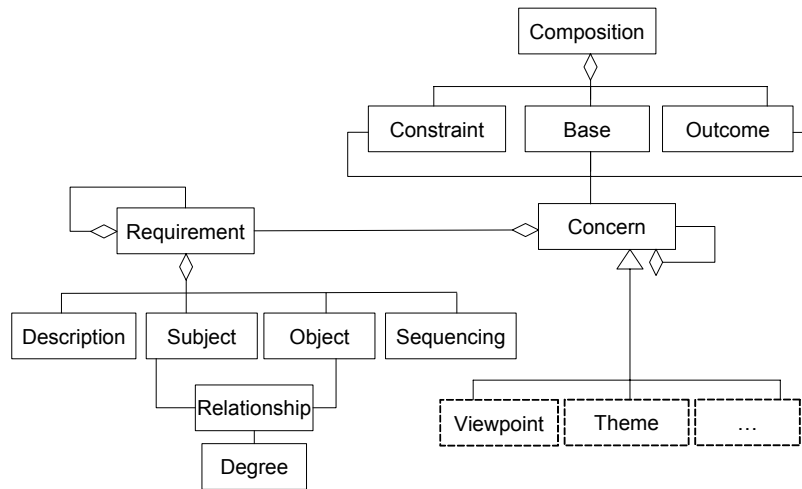


Figure 2: RDL meta-model.

aggregate subject and object may themselves be related with a particular degree. This pattern occurs rather often in RDL, and thus is coined the *Subject-Relationship-Object* (SRO) pattern. Moreover, a single requirement can be aggregated of multiple requirements, e.g., so-called sub-requirements. Each requirement is an aggregate of a particular concern; a concern may be aggregated of one or more requirements. Concerns may be represented as viewpoints, themes, UML use-cases, etc.

The novelty of the AORE approach is in the way concerns are *composed* with each other. It is quite natural and typical that individual concerns are not isolated, i.e., concerns are often dependent upon other concerns or involve them in some way, shape, or form. Unfortunately, however, this inherit property leads to concern descriptions which are both scattered and tangled (Kiczales et al. 1997). That is, an individual concern may manifest themselves in portions of other concerns, and an individual concern may contain textual information about other concerns. This makes reasoning about concerns in a complex system difficult; understanding a single concern requires understanding and analyzing multiple concerns.

For this reason, the RDL provides declarative *composition* semantics which allow the textual information regarding concerns to be localized, i.e., not scattered, but to also bind that information with a *composition specification*, indicating *where* the concern applies wrt to the entire system. Thus, concern representations are not tangled with the internal details of other concerns, thereby improving system comprehension. In terms of the meta-model described in Figure 2, a *composition* is explicitly denoted as a separate entity, aggregated of one or more concerns. The composition specification is notated as consisting of 3 aggregates, a constraint, a base, and an outcome. These aggregates are used to provide further details regarding the composition.

2.4 Multi-dimensional Requirements Analysis Tool

Once information is gathered and the RDL is generated, the RDL is then ready for further processing in order to serve as a basis for an architectural design. The Multi-dimensional Requirements Analysis Tool (MRAT) accepts RDL as input and assists developers with analyzing requirements in various ways. For instance, MRAT, along with the use of a synonym lexicon, allows the use of *semantic* queries, assistance in the creating and alteration of composition specifications, as well as the ability to detect various sorts of conflicts amongst the requirements. As Figure 1 depicts, the developer is able to interactively leverage the power of MRAT to refine the requirements, thereby forming a cycle until the requirements have reached a fixed-point, i.e., until no further changes are necessary prior to constructing an architecture. Thus, MRAT provides the RDL-based compositions for use as input into the architectural translation of the requirements.

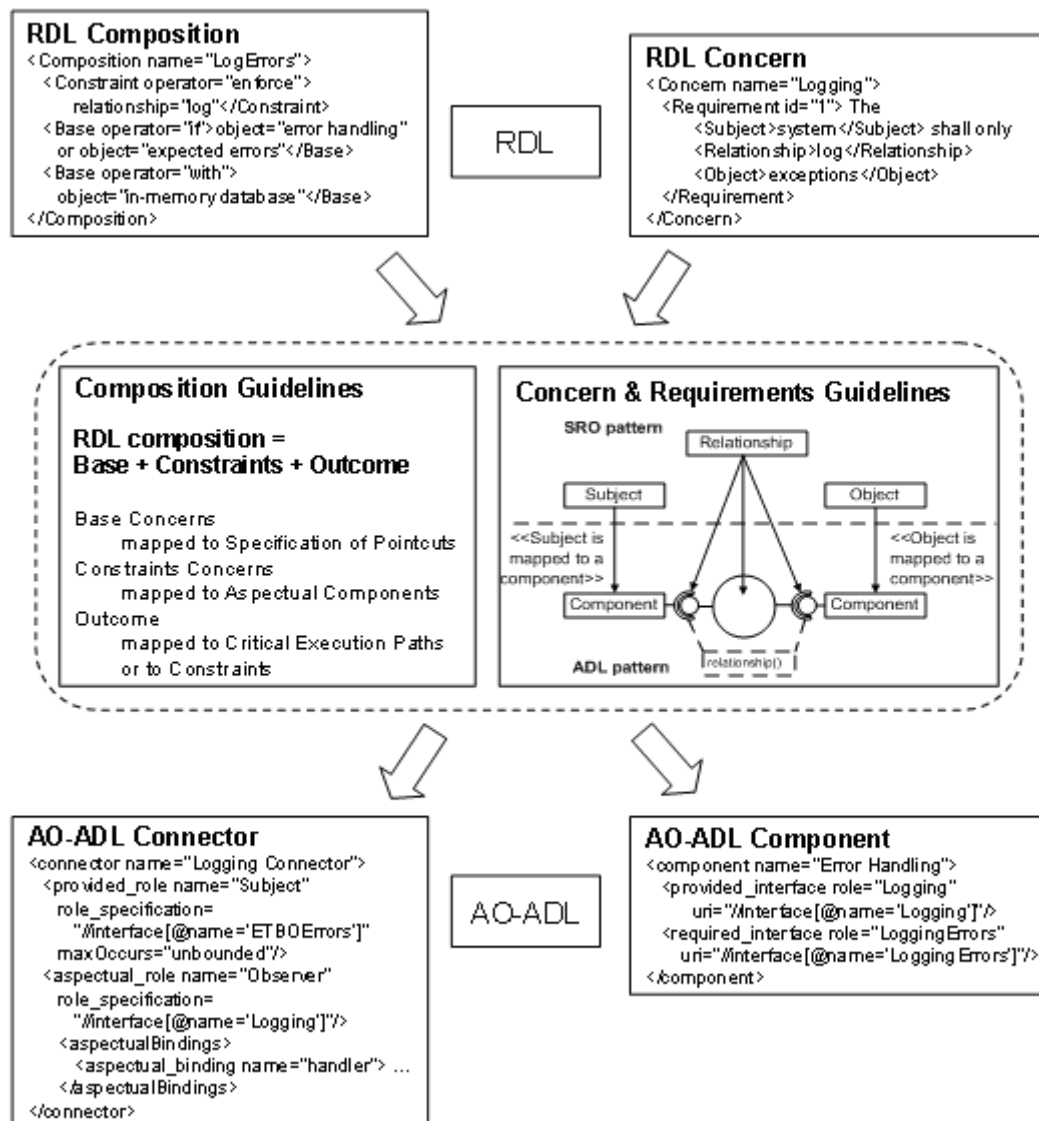


Figure 3: Requirements to architecture mapping process

3 From Requirements to Architecture

The RDL specification obtained in previous step, that is the RDL concerns and compositions, is the input to the mapping process from requirements to architecture defined in (Chitchyan et al. 2006a). The output of the mapping process is an AO-ADL specification of the aspect-oriented software architecture, concretely the component and connector view.

The details of the AO-ADL language and its tool support are provided in next section. In this section we briefly illustrate how the RDL concerns and compositions can be mapped to AO-ADL architectural elements, using the details about the mapping patterns and supporting guidelines described in D63. Figure 3 shows an example of the application of this mapping process. On the one hand, it illustrates the application of the guidelines that map the RDL *Subject*-

Relationship-Object (SRO) Pattern to AO-ADL. The identification of this pattern at the requirements level and the identification of the several candidate mappings to AO-ADL, depending on the semantic of the subjects, the relationships and the objects is one of the main contributions of the RDL to AO-ADL mapping. Concretely, we can observe that the *Logging* RDL concern (see the square at the upper right side of the figure) has a requirement that relates the subject *system* with the object *exceptions* by means of a *log* relationship. Following the *Concern & Requirements Guidelines* (see right part of the middle square in the figure), the RDL SRO pattern is mapped to the *Logging* interface (see number X) with the 'log' operation and to the *Logging Connector* (see number Y). Moreover, since there is an RDL composition where the *log* interface appears as part of the definition of a *constraint*, this means that the component providing the *Logging* interface would behave as an aspectual component. After the mapping from requirements to architecture and after a first refinement of the architecture, this component is the *Error Handling* component.

4 Aspect-Oriented Architecture Approach

In this section, we overview the AOSD-Europe Aspect-Oriented Software Architecture (AOSA) approach developed within the Analysis and Design lab. Our approach supports the identification, representation and evaluation of crosscutting concerns at the architectural level. The identification of aspects is mainly done using ASAAM as described in (Tekinerdogan et al. 2007b). For the representation of aspects we have defined a multiple views approach as defined in (Tekinerdogan et al. 2007a), mainly focusing on the module view, the component and connector view and the deployment view. For the particular case of the component and connector view, our approach defines both a textual model as an ADL and a visual model including graphical notations. We have also defined a set of aspect-oriented architectural metrics to evaluate the benefits of using an aspect-oriented approach in comparison with object-oriented or component-based approaches. The sections below elaborate on these constituent parts.

4.1 Multiple Views Approach

In (Tekinerdogan et al. 2007a) we have defined aspect-oriented metamodels of the module, the component and connector and the deployment view. On the one hand, the module and the deployment metamodels are in essence a simplified version of the UML metamodel, extended with aspect-oriented concepts such as aspects, crosscutting, pointcut and advice. On the other hand, the metamodel of the component and connector view, both the textual and the visual notations, is an extension of the metamodel of traditional ADLs, where components and connectors are the main architectural building block. We have specially extended the semantic of connectors to enrich them with additional composition semantics to cope with the crosscutting effect of aspectual components.

4.2 The AO-ADL Architecture Description Language

*AO-ADL*² is the AOSD-Europe approach for representing the component and connector view. This language is an XML-based ADL that considers a symmetric decomposition model – it uses components and connectors as the basic structural elements (similar to traditional ALDs) with aspects treated as specific types of components, increasing the possibilities of reusing components/aspects in different contexts. Also, it defines a new relationship within the connector which are enriched with additional composition semantics to cope with the crosscutting effect of aspectual components.

The initial architecture obtained via mapping is unlikely to be in line with the architect's final aim. It only provides the initial set of elements that should form a part of the architecture, as well as their relationships. Therefore, in order to produce a satisfactory architecture design of the system, it is necessary to carry

²<http://caosd.lcc.uma.es/aoadl>

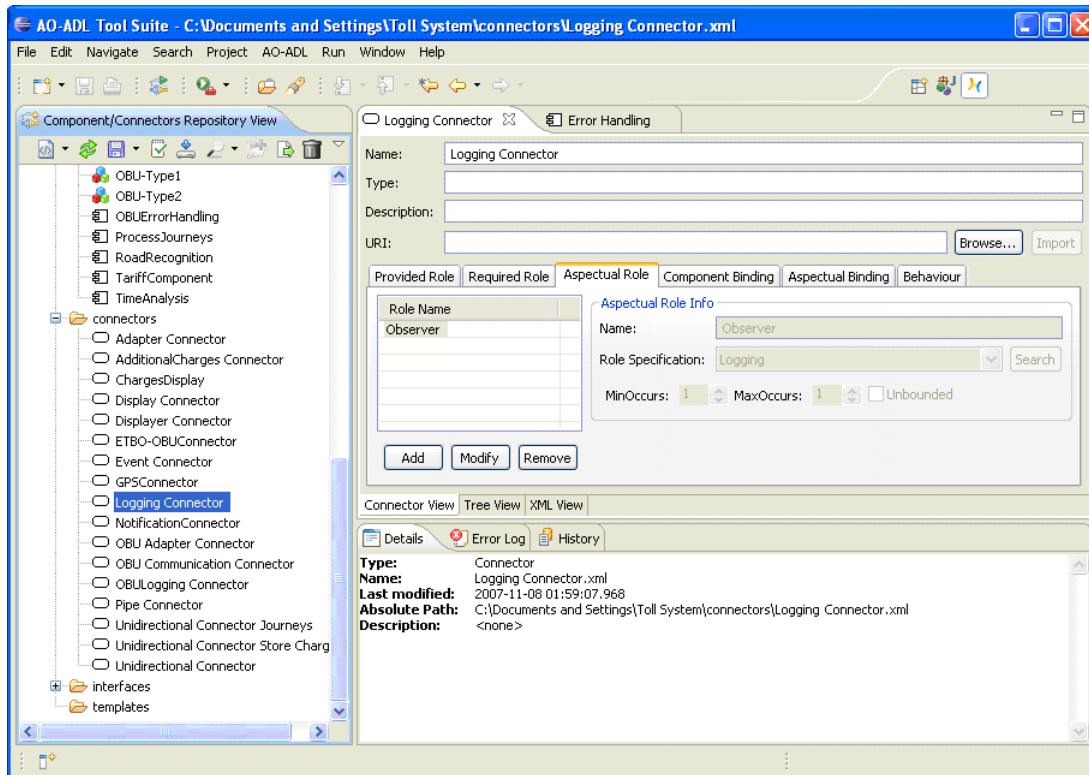


Figure 4: AO-ADL Tool Suite

out a refinement or rebuilding process of the initial architecture. Thus, in order to provide tool support for the application of the refinement guidelines described in (Chitchyan et al. 2006a), the initial version of an AO-ADL architecture, obtained after applying the mapping process, can be edited to be completed and refined using our tool support described below.

4.3 The Visual Notation

We have defined a 1-to-1 mapping between the metamodel of the AO-ADL language and the metamodel of the visual notation. This means that AO-ADL architectural descriptions can be visually represented in UML. This graphical notation includes symbols to represent Aspect-Oriented concepts, such as aspects and the different kinds of advice.

4.4 The AO-ADL Tool Suite

The *AO-ADL Tool Suite*³ (see Figure 4) is part of the AOSD-Europe Atelier IDE tools⁴ developed as an Eclipse plug-in that simplifies the architecture design process. This tool suite is composed by the following set of modules. The flow

³<http://caosd.lcc.uma.es/AO-ADLUpdates>

⁴<http://gateway.comp.lancs.ac.uk/computing/aosd-europe/atelier/updates>

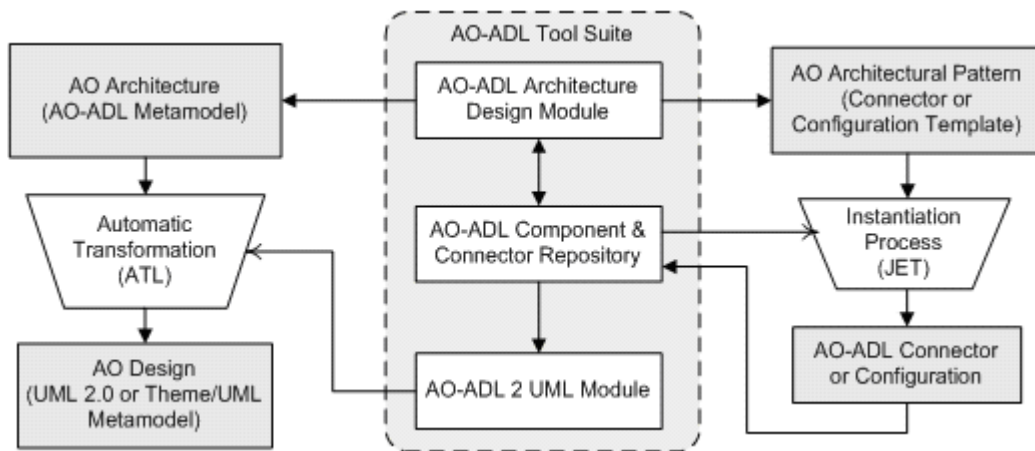


Figure 5: AO-ADL Tool Suite flow of information

of information between the AO-ADL Tool Suite modules is shown with more details in Figure 5.

Component & connector repository. The *Component & Connector Repository* module (C&C Repository) allows the AO-ADL specification of independent and reusable components and connectors. They will then be easily (re)used in different architecture specifications. Moreover, it also provides a module to support the specification of “connector templates”. These templates can be considered as connectors modeling architectural patterns. This will be especially useful to provide a library of connectors modeling well-known relationships among concerns at the architectural level, both crosscutting and non-crosscutting relationships.

AO-ADL architecture design module. Another module of the AO-ADL Tool Suite is the *AO-ADL Architecture Design Module*. This module supports the specification of AO-ADL software architectures. The flow of information between the C&C repository and this module is defined as that the components and the connectors in the C&C repository can be imported to be reused by the Architecture Design module.

AO-ADL2UML module. Finally, the AO-ADL Tool Suite provides support to go to the next stage in the software development life cycle, going from Architecture (using the AO-ADL language) to Design (using Theme/UML) automatically using the *AO-ADL2UML* module, as will be shown in Section 5.

4.5 Architectural Evaluation Process/Metrics

The metrics we have defined strictly focus on the evaluation of software architecture artifacts, since we are concerned with both: (i) understanding the suitability of aspect-oriented architecture-level solutions to address the modularity

problems associated with widely-scoped crosscutting concerns, and (ii) investigating to what extent the crosscutting nature of certain concerns entail design anomalies that are visible earlier at the architecture definition stage. The goal of this metrics suite is to support the assessment of structural attributes in the architecture description with a direct impact on architecture modularity.

The suite includes metrics for architectural separation of concerns (SoC), architectural coupling, component cohesion, and interface complexity. The application of the metrics aims at providing the architect with a fine-grained understanding of the overall architecture quality since modularity has a well-known impact on a vast number of architecturally-relevant non-functional requirements, such as re-usability, adaptability, flexibility, changeability and the like.

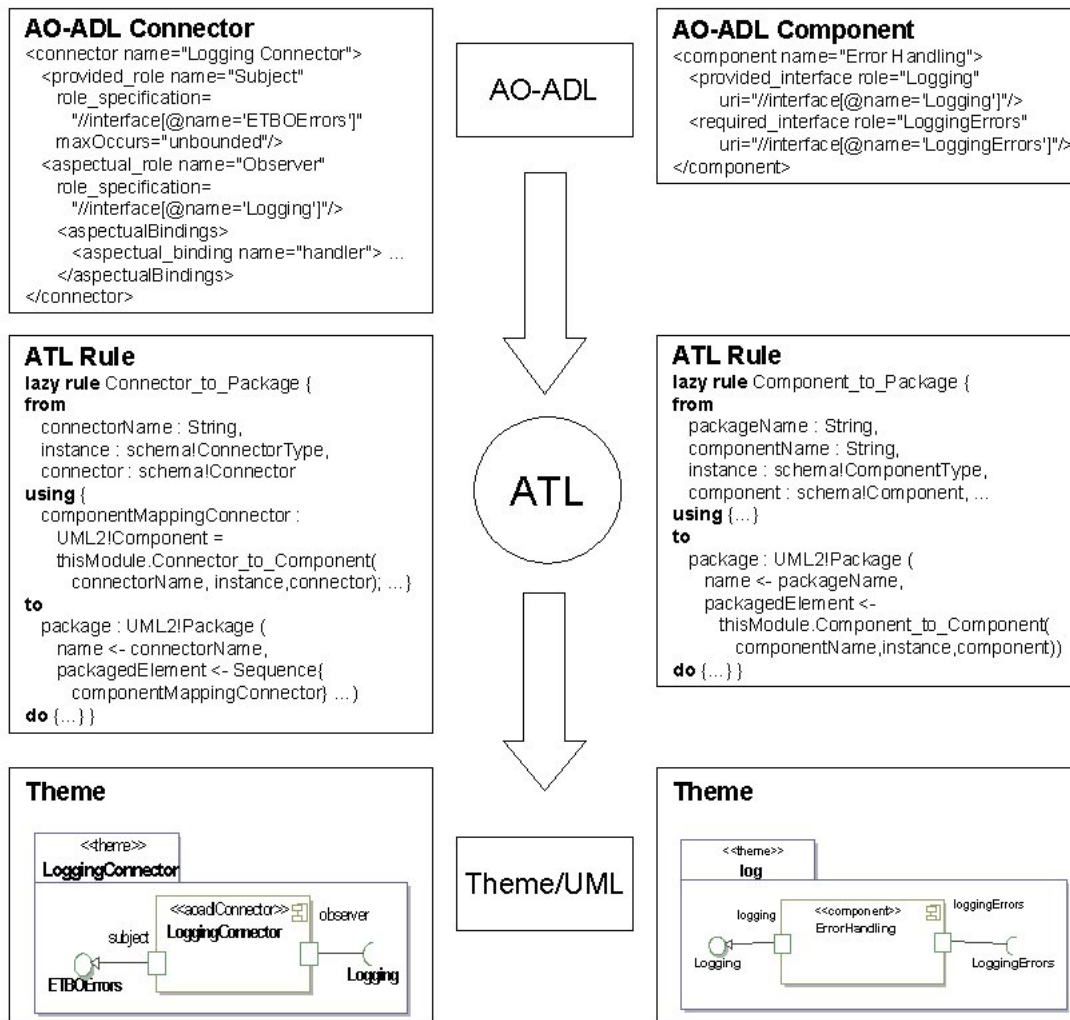


Figure 6: Architecture to design mapping process

5 From Architecture to Design

Once we are happy with the AO-ADL architecture, the AO-ADL specification is the input to the mapping process from architecture to detailed design as defined in D63 . The output of the mapping process is a Theme/UML representation of the software architecture, which is the starting point to continue with the detailed design of the system using the Theme/UML approach.

The mapping process from architecture to design defined in D63 has been automated using MDD/MDA technologies. Specifically we have used ATL (ATLAS Transformation Language) to write the transformation rules. In Figure 4 we can see an example of applying some ATL rules to transform architectural elements (i.e. a connector and a component) into its corresponding design elements (i.e. themes and parameterized themes).

As we have seen in the previous section, the AO-ADL Tool Suite provides a set of modules to easily manipulate AO-ADL software architectures. One of

these modules, the *AO-ADL2UML* module, allows us automatically realizing the mapping process. The complete mapping process can be resumed in two steps:

1. AO design model generation (Theme/UML model) using AO-ADL Tool Suite. We just have to select our architecture in the C&C Repository, pushing a button, selecting 'Transform into Theme/UML' and choosing a folder where the generated AO design model will be saved.
2. Generate a UML diagram from the generated UML models, so that the model can be properly opened in a UML editor. Concretely our tool provides support to export the UML models generated using the UML meta-model to the MagicDraw editor.

Thus, the output of this process is a MagicDraw project with the skeleton of an AO design model that will be completed in the following stage of the development, the detailed design.

6 Aspect-Oriented Design Approach

The AOSD-Europe design approach supports Aspect-Oriented design within an integrated AOSD-Europe development methodology. In this section we describe the design process, design language, backward and forward traceability paths and supporting tooling that comprise the approach. We also summarize the results of a study in which we compare Aspect- and Component-Oriented Designs.

The approach is an integration of languages, processes and concepts identified in a comprehensive state-of-the-art (SOTA) survey (Chitchyan et al. 2005). In this survey we gathered a very comprehensive snapshot of where research within and external to the network had been done and indicated integration opportunities. This survey also showed where the gaps in the existing research lay.

6.1 Aspect-Oriented Design Language

The design language (Jackson and Clarke 2007b) is predominantly based on an integration of Theme/UML, JPDDs and Composition Directives. The integration of these approaches brought the strong semantics of Theme/UML, the point-cut visualization of JPDDs and the composition meta-model of Composition Directives, identified in the SOTA survey, to form a new AOSD-Europe design language. The resulting design language was defined as a lightweight profile based extension of the standard UML 2.2 specification. Due to conformance to the UML, standard tooling can be used to create AOSD-Europe designs.

6.2 Aspect-Oriented Design Process

While the AOSD-Europe design language is based on the integration of existing languages, the process (Clarke and Jackson 2006) is more aspirational in nature. In the SOTA survey we found that the existing design processes typically recognised a narrow set of design tasks in the process and avoided issues such as evolution and testing. These processes were not detailed and did not provide design guidelines. Moreover, they were not integrated with the phases before and after design. The design approach that we developed was based on the integration and abstraction of existing design approaches. The resulting design process is a flexible and generic approach to design, following the Aspect-Oriented paradigm. This process provides detailed guidelines to aid the designer to understand the complexities and trade-offs that exist when designing with Aspects. The process provides advice on how to move from requirements and architecture to a detailed Aspect-Oriented design (Carton et al. 2008; Chitchyan et al. 2006a; Jackson and Clarke 2007a) and from design to implementation (Carton et al. 2008; Jackson and Clarke 2007a; Sánchez et al. 2007) and evolution. The AOSD-Europe design process described how the elements of requirements and architecture are linked to design elements and how

these in turn are linked to elements of implementation. In the SOTA survey we found that there was no existing work on traceability backwards or forwards for existing design approaches. To fill this gap in the research we defined a change and corresponding impact profiles to guide the designer in how to deal with changes propagated backwards from implementation and forwards from requirements and architecture phases of development.

6.3 Aspect-Oriented Design Tooling

A tool set supports the approach described above (Sánchez et al. 2007). This tool set consists of three major components – design, skeleton implementation generation and round trip engineering. The design component provides a service supporting the validation of composition specifications, as well as composition of design elements. The service provided by the skeleton implementation generation component supports the generation of Aspect-Oriented and Object-Oriented implementations from the design. Generation of Aspect-Oriented implementations can be done directly from Aspect-Oriented designs. Object-Oriented implementations can be generated only after the composition of Aspect-Oriented designs elements to create standard a UML design which is then used to generate the Object-Oriented implementation. The round trip engineering component is a work in progress but is based on the design traceability and design process. This component identifies changes in design and non-design artifacts and predicts the potential cost of change.

6.4 Aspect-Oriented Design Demonstrator

The design approach has been evaluated through a comparative study (Pinto et al. 2008). In this study a common set of requirements were selected from a large set of industrial requirements. Two design teams were then given the task of designing based on these requirements. One team followed the AOSD-Europe design approach. The other followed a component based design approach. The resulting designs were then compared based on measures of complexity, separation of concerns, size, cohesion, understandability and ease of change. Our findings were that there that the overall designs were relatively equivalent for most measures but that the Aspect-Oriented design did show much better separation of concerns over the Component based design.

7 From Design to Implementation

The design approach supports the creation of initial Aspect-Oriented designs from requirements and architectural designs. The design approach provides guidelines to refine these initial designs to a point that they can be used as a specification from which an implementation can be generated or derived. As we have described in Section 6, we have defined both mappings from Aspect-Oriented design to implementation. We detail these further in this section.

7.1 Mappings to Implementation

A plethora of Aspect-Oriented languages reported on in the literature. A subset of these are available for download and are suitable for real development. To support the generation or derivation of an Aspect-Oriented implementation from design we firstly needed to provide a mapping from design to implementation elements. A goal of the AOSD-Europe design approach was to maintain AOP language independence. To show that the design approach is independent of AOP language we created mappings to a selection of usable and diverse implementation platforms. This selection included, AspectJ, CaesarJ and CME. These mappings are detailed in (Carton et al. 2008; Pinto et al. 2008; Sánchez et al. 2007). The mappings are described in terms of mappings between elements and guidelines to help the practitioner to follow these mappings are provided. These mappings are further explained through examples.

7.2 Tools for Generation AO Implementation from AO Design

An implementation of these mappings and mappings from design to Object-Oriented implementation platforms are supported by the skeleton implementation generation component of our design tool set (Carton et al. 2008). This component takes as input Aspect-Oriented designs created using standard UML editors (we use Magic Draw) and can generate skeleton AspectJ code. The programmer can then refine this skeleton AspectJ code. The refinements can be checked to ensure the implementation conforms to the constraints specified in the design using this component. The component is extensible and enables the extension of the component to allow for other mappings to be implemented.

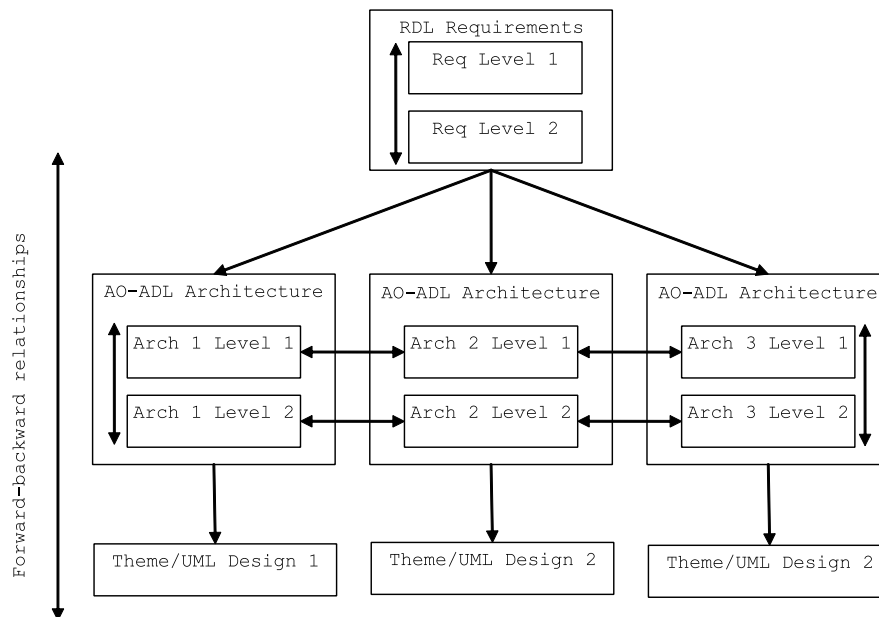


Figure 7: Intra- and Inter-phase traceability information.

8 Towards a Traceability Framework

The ability to follow and examine (trace) the life of a artifact throughout software evolution has proven to greatly improve overall system comprehension (Gotel and Finkelstein 1994; Ramesh et al. 1995). In this section, we overview our intent towards a future traceability framework for tracing between software engineering-related artifacts. The main intention of this framework is to provide support for analysis of cross-life-cycle concern representations and the assessment of the impact of their changes throughout system evolution.

Any non-trivial software system is likely to change over its life time. Thus, software maintainability and evolution may be difficult to manage, especially in large, complex systems. There are several related efforts (Hoffman et al. 2007; Perez-Toledano et al. 2007; Rashid et al. 2008; Stoerzer and Graf 2005; van den Berg et al. 2006; Wloka 2006; Zhang and Zhao 2007; Zhang et al. 2008; Zhao 2002; Zhao et al. 2002) that discuss the impact of change of software at specific phases of development. Unfortunately, few approaches have focused on the cost of evolution throughout the software development life-cycle as a whole.

In (Pinto et al. 2008), we provided a vision of our approach to identify and analyze changes to aspect constructs throughout the development life cycle using our Aspect-Oriented analysis and design approach. Specifically, we provided case studies and identification of relevant data to be incorporated into such an approach, as well as an initial discussion about support for change impact analysis. Also, in order to be able to utilize the traceability data, initial plans for a “knowledge-base” framework was presented, which is intended to assist in both querying the artifact dependency links.

The core effort within (Pinto et al. 2008) continues to build on the work previ-

ously documented in (Chitchyan et al. 2006a) by identifying the traceability information that needs to be recorded in order to constitute a traceability framework. Particularly, (Chitchyan et al. 2006a) discusses the refinement of architecture and provides a set of architecture refinement guidelines. From the perspective of a traceability framework, such refinements should be identifiable, i.e., a query should be supported to view the previous version (if available) of the given component in the architecture as well as its affect on the connector (compositions) produced by the given change. (Chitchyan et al. 2006a) also discusses both horizontal and vertical architecture refinement, and thus, as shown in Figure 7, traceability information can be across several candidate architectures at the same level of abstraction (traceability between Arch 1 Level 1 and Arch 2 Level 1 and Arch 3 level 1 in Figure 7), and several levels of abstraction of the same candidate architecture (traceability between Arch 1 Level 1 and Arch 1 Level 2 in Figure 7).

In conclusion, new data has been identified that summarizes the kind of traceability information that can be extracted by the application of our mapping process and guidelines as outlined in (Chitchyan et al. 2006a). By clearly identifying such information and allowing it to be able to be related to different stages of the software development process, we envision our planned traceability framework implementation to adequately ease the burden of tracing artifacts across the software development life-cycle, thus enabling effective change impact analysis and ultimately software maintenance.

9 Conclusion and Future Work

Using the approach summarized in this deliverable allows developers to localize the scattered and tangled representations of crosscutting concerns in a variety of software engineering-related artifacts, including requirements, architecture, and design. We have summarized how our approach adapts AO concepts to activities taking place earlier in the software development life cycle so that these stages may also benefit from the advanced separation of concerns AOP enables. By separating concerns in this fashion, each can be studied and reasoned about in isolation, and also later *composed* in order to produce a complete system of artifacts.

In the future, we plan to fully develop and implement a traceability framework which may be used to further reason about the information generated using our analysis and design approach. Specifically, we plan to concertize a full change impact model for AO artifacts spanning the entire software engineering life cycle, perhaps utilizing techniques found in (Wloka et al. 2008). Plans also entail utilizing a rules-based engine such as the Drools framework⁵ and incorporating our reasoning engine into an Eclipse IDE plugin, possibly building upon the well-established AspectJ Development Tools⁶.

⁵<http://www.jboss.org/drools>

⁶<http://www.eclipse.org/ajdt>

References

- A. Carton, A. Jackson, and S. Clarke. Model-driven theme/uml. *Transactions on Aspect-Oriented Software Development*, 2008.
- R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University, May 2005.
- R. Chitchyan, A. Rashid, A. Garcia, M. Pinto, P. Sanchez, L. Fuentes, A. Jackson, and I. Krechetov. Mapping and refinement of requirements level aspects. Technical Report AOSD-Europe Deliverable D63, AOSD-Europe-ULANC-24, Lancaster University, Lancaster, UK, Oct. 2006a.
- R. Chitchyan, A. Sampaio, A. Rashid, P. Sawyer, and S. S. Khan. Initial version of aspect-oriented requirements engineering model. Technical Report AOSD-Europe Deliverable D36, AOSD-Europe-ULANC-17, Lancaster University, Lancaster, UK, Feb. 2006b.
- S. Clarke and A. Jackson. Refined aspect-oriented design process. Technical Report AOSD-Europe Deliverable D57, AOSD-Europe-TCD-D57, Trinity College, Dublin, Ireland, Aug. 2006.
- D. Dantas and D. Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.
- X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. SyncGen: An AOP framework for synchronization. In *Int. Conf. on Tools and Alg. for Construction and Analysis of Sys.*, 2004.
- O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering*, 1994.
- K. Hoffman, M. K. Ramanathan, P. Eugster, and S. Jagannathan. Aspect-based introspection and change analysis for evolving programs. In *International Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2007.
- A. Jackson and S. Clarke. Report on evaluation of AOD in case study. Technical Report AOSD-Europe-TCD-D95, Trinity College, Dublin, Ireland, Aug. 2007a.
- A. Jackson and S. Clarke. Refined aspect-oriented design language. Technical Report AOSD-Europe Deliverable D75, AOSD-Europe-TCD-D75, Trinity College, Dublin, Ireland, Feb. 2007b.
- G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.
- R. Laddad. *AspectJ in Action*. Manning, 2003.
- M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *International Conference on Software Engineering*, 2002.
- M. A. Perez-Toledano, A. Navasa, J. M. Murillo, and C. Canal. Titan: a frame-

- work for aspect oriented system evolution. In *Int. Conf. Software Engineering Advances*, 2007.
- M. Pinto, L. Fuentes, R. Chitchyan, A. Rashid, A. Jackson, S. Clarke, B. Shishkov, B. Tekinerdogan, M. Aksit, P. Greenwood, and R. Khatchadourian. Traceability framework: From requirements through architecture and design. Technical Report AOSD-Europe Deliverable D126, AOSD-Europe-ULANC-43, Lancaster University, Lancaster, UK, July 2008.
- B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: a case study. In *Requirements Engineering*, 1995.
- A. Rashid and R. Chitchyan. Persistence as an aspect. In *Int. Conf. Aspect-Oriented Software Development*, 2003.
- S. O. Rashid, R. Chitchyan, A. Rashid, R. Khatchadourian, and P. Greenwood. Approach for change impact analysis of aspectual requirements. Technical Report AOSD-Europe Deliverable D110, AOSD-Europe-ULANC-40, European Network of Excellence on Aspect-Oriented Software Development, Jan. 2008.
- P. Sánchez, L. Fuentes, A. Jackson, and S. Clarke. Aspects at the right time. *Transactions on Aspect-Oriented Software Development*, 4(4640):54–113, Nov. 2007.
- M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Int. Conf. Software Maintenance*, 2005.
- B. Tekinerdogan, A. Garcia, C. Sant’Anna, E. Figueiredo, M. Pinto, and L. Fuentes. Approach for modeling aspects in architectural views. Technical Report AOSD-Europe Deliverable D77, AOSD-Europe-UT-D77, University of Twente, Enschede, The Netherlands, Feb. 2007a.
- B. Tekinerdogan, F. Scholten, M. Pinto, and L. Fuentes. Approach for identifying aspects in architectural views. Technical Report AOSD-Europe Deliverable D76, AOSD-Europe-UT-D76, University of Twente, Enschede, The Netherlands, Feb. 2007b.
- K. van den Berg, J. M. Conejero, and J. Hernandez. Analysis of crosscutting across software development phases based on traceability. In *Early Aspects at ICSE 2006*, 2006.
- J. Wloka. Towards tool-supported update of pointcuts in AO refactoring. In *International Workshop on Linking Aspect Technology and Evolution*, 2006.
- J. Wloka, R. Hirschfeld, and J. Hänsel. Tool-supported refactoring of aspect-oriented programs. In *Int. Conf. Aspect-Oriented Software Development*, 2008.
- S. Zhang and J. Zhao. Change impact analysis for aspectj programs. Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, Shanghai Jiao Tong University, Jan. 2007.
- S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Celadon: A change impact analysis tool for aspect-oriented programs. Demonstation at Int. Conf. Aspect-Oriented Software Development, Apr. 2008.

- J. Zhao. Change impact analysis for aspect-oriented software evolution. In *International Workshop on Principles of Software Evolution*, 2002.
- J. Zhao, H. Yang, L. Xiang, and B. Xu. Change impact analysis to support architectural evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 2002.