Using AOSD and MDD to Enhance the Architectural Design Phase*

Mónica Pinto, Lidia Fuentes, Luis Fernández, and Juan A. Valenzuela

Dept. Lenguajes y Ciencias de la Computación, University of Málaga, Spain
{pinto,lff,lfernandez,valenzuela}@lcc.uma.es
http://caosd.lcc.uma.es

Abstract. This paper describes an MDD process that enhances the architectural design phase by closing the gap between ADLs and the notations used at the detailed design phase. We have defined model-to-model transformation rules to automatically generate either aspect-oriented or object-oriented UML 2.0 models from high-level architectural specifications specified using AO-ADL. These rules have been integrated in the AO-ADL Tool Suite, providing support to automatically generate a skeleton of the detailed design that preserves the crosscutting and the non-crosscutting functionalities identified at the architecture level.

1 Introduction

Architecture description languages (ADLs) deal with the description, analysis and reuse of software architectures, providing an appropriate high-level view of complex software systems. However, in spite of the importance of describing the software architecture of a system, ADLs are not extensively used. This is not due to the lack of ADLs, since there is an important number of them, not only those based on formalisms (ACME [1], RAPIDE [2] and C2 [3]), but more modern ones based on XML (xADL [4]) or based on aspect-oriented (AO) separation of concerns (PRISMA [5] and AO-ADL [6]). We consider that the reasons are instead related to the lack of appropriate architectural tool support and the lack of connection with next phases of the software development.

On the one hand, there is a clear gap between ADLs and the notations used in later phases of the software development – while most ADLs are based either on formalisms, XML or specifically defined notations, the most widely used modelling language for detailed design is UML. The consequence is that software engineers do not usually take advantage of the power of ADLs. When a high-level architecture is provided it is usually modelled using the UML component diagrams, which define a very restricted version of the concept of ADLs connectors. On the other hand, tool support is needed to model, reuse and reason about software architectures. Moreover, tool support to be able to generate a skeleton of the detailed design from information available at the software architecture,

* Supported by Spanish Project TIN2008-01942 and European Project AMPLE IST-033710.

R. Meersman, P. Herrero, and T. Dillon (Eds.): OTM 2009 Workshops, LNCS 5872, pp. 360–369, 2009.

without loss of relevant information, is also required. Finally, it would be desirable that the skeletons were automatically generated. These shortcomings can be solved by defining mappings between ADL and UML 2.0 metamodels, and by implementing the required tool support. Both things can be appropriately done using current Model-Driven Development (MDD) [7] technologies.

In this paper we show our experience using MDD to enhance the architectural design phase by closing the gap with the detailed design. After this introduction, an overview of our MDD process is provided in Section 2, while the details are provided in Section 3 using a running example. Section 4 describes the AO-ADL Tool Suite, Section 5 the related work and Section 6 our conclusions.

2 Our MDD Process

Using MDD, a software system is created through the definition of different models at different abstraction layers, where models of a certain abstraction layer – i.e. output models, are automatically derived from models of the upper abstraction layer – i.e. input models, by means of model transformations. In order to make a model transformation feasible, both the input and the output models must conform to a metamodel, which specifies their precise abstract syntax [8]. The model transformation specifies the rules for automatically generating the corresponding output model from an input model. These transformations are specified using a model transformation language, such as QVT or ATL [9].

In this section we provide an overview of the MDD process that we have defined to automatically transform a software architecture, described using an aspect-oriented ADL (in this case the AO-ADL language [6]), into a skeleton of its detailed design, using either plain UML 2.0 or an AO extension of UML, like Theme/UML [10]. Figure 1 shows the main models and transformations of our process. Although the starting point is always an AO version of the software architecture (box (1)), an important contribution of this process is that in the lower levels it allows choosing between following an AO development methodology (left side of Figure 1) or an OO development methodology (right side of Figure 1). Following this approach, our main goal is the definition of an automatic process that can be useful for both AO and OO designers and implementers.

Taking as input an AO-ADL software architecture, the following step in our MDD process is the automatic generation of a skeleton of its detailed design. Two alternative outputs are possible. The first transformation (box (2)) automatically generates an AO version of the detailed design using *Theme/UML*, an AO extension of UML. In this case we would be following an AO development methodology, so the separation of concerns identified and modelled at the architectural level is explicitly maintained at the design level. The second transformation (box (6)) generates a skeleton of the detailed design of such architecture in *plain UML 2.0*. In this case, we would be using an OO development methodology, so the crosscutting concerns identified at the architectural level are not explicitly separated at the design level. Instead, they are weaved at the design level with the non-crosscutting concerns. The generated design models will conform to the Theme/UML metamodel (box (3)), or to the UML

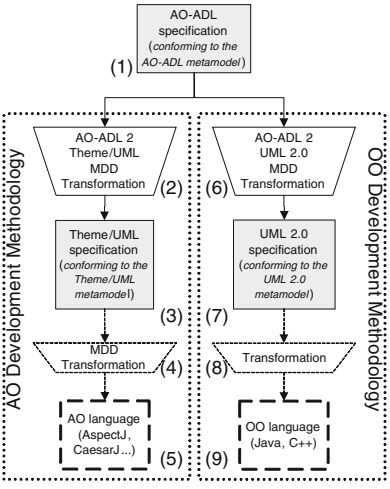

Fig. 1. MDD Process in the Software Development Lifecycle using AO-ADL

2.0 metamodel (box (7)), respectively. These designs need to be completed using any existing UML editor. Once the detailed design is completed, the last step is the automatic generation of code (boxes (4) and (8)), either AO code (box (5)) or OO code (box (9)). These are third-party transformations [11] that can be integrated into our MDD process, although they are not included in this paper.

3 An Example Step by Step

In this section our MDD process is followed step by step using an auction system running example. Basically, in an auction system *"sellers initiates auctions offering items to be sold, and buyers bid for them. The sellers set the initial and minimum bid and initiate the auction. Buyers join auctions, bid for items and increase their previous bids"*. Additionally to these functional requirements, several extra-functional requirements are identified: (1) *Encryption*: the communication between users and the auction system must be encrypted, (2) *Consistency*: minimum bids, bid prices and bid increments must be valid, and (3) *Synchronization*: more than one buyer can simultaneously bid for the same item.

3.1 AO-ADL Specification

The main building blocks in AO-ADL[1] are components and connectors. *Components* model both non-crosscutting and crosscutting concerns, and are described by their provided/required *ports*. The composition (e.g. weaving in AO) between components modelling crosscutting and non-crosscutting concerns is specified as part of connectors. Thus, the semantic of connectors is extended with the definition of *aspectual bindings*. Also, *aspectual* roles are a new kind of role in AO-ADL.

[1] The AO-ADL metamodel [6] is not shown here due to the lack of space.

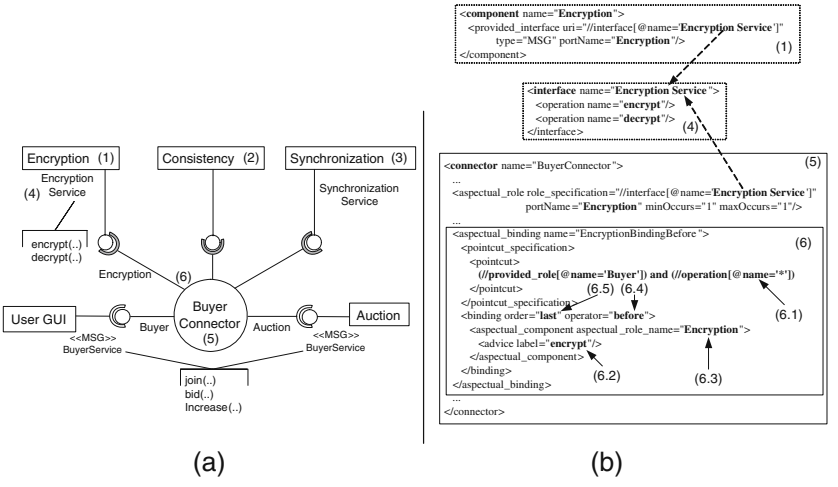

Fig. 2. AO-ADL Architecture of the Auction System

Components connected to *aspectual* roles behave as *aspectual components*, while those connected to *provided/required* roles behave as *base components*.

Figure 2.(a) shows five different components connected through a connector. On the one hand, there is an interaction between the *User GUI* and the *Auction* base components, which model the user and the auction non-crosscutting concerns. On the other hand, the *Encryption*, *Consistency* and *Synchronization* aspectual components model crosscutting concerns. The auction system shown in Figure 2.(a) is specified by a set of XML files in Figure 2.(b), where XML code partially describing the *Encryption* component and the *BuyerConnector* connector is shown. The code (4) describes the *Encryption Service* interface with two operations: *encrypt* and *decrypt*. This interface is used in the description of the *Encryption* component (code (1)) that *provides* the *Encryption Service* interface by means of its port *Encryption*. The code (5) specifies the aspectual part of the *BuyerConnector* connector. On the one hand, the *Encryption Service* interface is associated with an aspectual role of the connector, named *Encryption*. This means that a component connected to this role will behave as an aspect affecting the interaction between base components, according to the information specified in the aspectual binding part of the connector (code (6)). Concretely, the pointcut specification (code (6.1)) describes that the *encrypt* method (code (6.2)) of the aspectual component connected to the *Encryption* role (code (6.3)) will affect to any operation of the provided role *Buyer*. Moreover, it will be injected *before* (code (6.4)) any call intercepted by the pointcut.

3.2 Transformation to Theme/UML

Theme/UML is an UML profile that extends the UML metamodel to define some new elements specifically created to aspect-oriented design. The most relevant

elements are the *themes*. A *theme* is a UML packet that contains all the UML elements that are needed to model a particular concern. It also defines two new relationships that can be established between themes: the *merge* and the *bind* relationships. The *merge* relationship models the relationships between concerns and the *bind* relationship models the aspectual bindings of crosscutting concerns.

In order to implement this transformation, the first step is the definition of the mapping between the metamodels of AO-ADL and Theme/UML. This mapping was defined as a collaboration with the Trinity College of Dublin (creators of Theme/UML) [12]. A summary of this mapping is shown in table 1. We have then implemented it using MDD, specifically the ATL language. The transformation is carried out by the application of a set of ATL rules that describe how elements of the source model (one or several elements of AO-ADL) are transformed into elements of the target model (one or several elements of Theme/UML).

Three different possibilities are offered in our approach to transform AO-ADL elements into Theme/UML elements. They are motivated by the fact that in Theme/UML a *theme* represents a single *concern*. However, in AO-ADL, components of low granularity can be associated with a single *concern*, but more complex components and, in particular, composite components are usually modelling more than one concern. In order to cope with this difference, our MDD process includes a manual step, previous to the automatic transformation, in which an expert in the domain of the architecture to be transformed must choose the kind of transformation to be performed: (1) An AO-ADL *component* models a single concern and is mapped to only one *theme*; (2) Each *interface* in the *component* models a different concern and thus each interface is mapped to a different *theme*, or (3) Each *operation* in the *interfaces* of the *component* models a different concern and thus each operation is mapped to different *themes*. Notice that these options can be combined so the designer can tune the design model.

Following with our example, Figure 3 partly shows the Theme/UML model automatically generated from Figure 2. In this particular case, each architectural component is modelling a single concern and thus we chosen to transform

Table 1. Theme/UML Mapping Rules

AO-ADL Element	Theme/UML Element	Rationale
Interface and Operations	Interface and Operations. Optionally, a theme can be generated from each of them.	Basically a 1:1 translation process. A theme is generated for each of them depending on the selected mapping option.
Component (base or aspectual components, there is no distinction in AO-ADL)	- A set of themes for *AO-ADL base components*. - A set of parameterized themes for *AO-ADL aspectual components*. - Both themes and parameterized themes composed by means of a merge operator.	In AO-ADL the same component can play a base or aspectual role depending on how it is attached to connectors. Theme/UML differentiates between two kinds of themes, distinguishing the representation of aspectual themes.
Connector	One single theme, containing a set of UML components.	This rule transforms a connector in the context of a particular architectural configuration, considering the AO-ADL components attached to the roles of AO-ADL connectors.
Configuration	A Theme/UML model.	All the elements transformed by previous rules are part of the Theme/UML model, creating a UML representation of an AO-ADL software architecture.

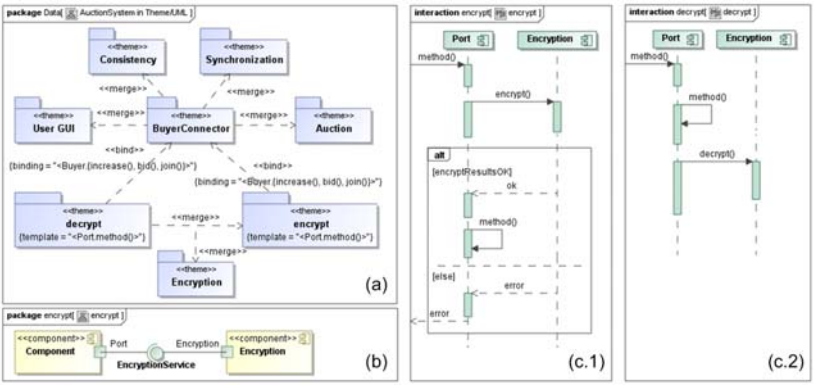

Fig. 3. Architecture of the Auction System in Theme/UML

each AO-ADL component into a theme. Thus, according to table 1, a *theme* is generated for each component in the software architecture. Moreover, an additional transformation is performed for those components that play the role of aspectual components in AO-ADL, consisting on two steps: (1) the previously generated *theme* is transformed into a *parameterized theme*, and (2) a new *theme* is generated for each operation in the interface of the aspectual component. Those additional themes model the aspect advice. In our example, the *Encryption* component is an aspectual component connected to an aspectual role of the *BuyerConnector* connector with two advice: *encrypt* and *decrypt*. Thus, two new themes are created modelling the advice. According to the Theme/UML metamodel, these new themes are related to the *Encryption* theme by means of *merge* relationships. The union of these three themes represents the whole behaviour of the Encryption component. Notice that Figure 3.(a) is a simplification of the real transformation since more themes should appear modelling the advice of the rest of aspectual components in the software architecture.

While, in AO-ADL, the aspectual information – i.e. pointcuts and aspectual bindings, is encapsulated as part of the connectors, in Theme/UML, pointcuts and aspectual bindings are specified by means of *bind* relationships between themes. In our example, the *encrypt* and *decrypt* themes interfere with the behaviour of *BuyerConnector* by means of two *bind* relationships, which specify the pointcuts where the aspectual theme affects to the behaviour of the base theme. In our example these points are the *increase()*, *bid()* and *join()* operations of the *Buyer* provided role shown in Figure 2.(a), as we can see in the *binding* attribute of the bind relationship (*binding="<Buyer.increase(), bid(), join()>"*).

In addition to Figure 3.(a), Figure 3.(b) shows the internal design of the *encrypt* theme, in which an *Encryption* component is related with a generic component *Component* through the *EncryptionService* interface. Moreover, the sequence diagram in Figure 3.(c.1) shows the precise way in which the *encrypt* advice behaves in relation to the interfered method. Concretely, the *encrypt()* advice is executed when a *method* comes to *Port* before transmit it to the inside

Table 2. UML 2.0 Mapping Rules

AO-ADL Element	UML 2.0 Element	Rationale
Interface/Operations	Interface and Operations.	Basically a 1:1 translation process.
Component	A UML 2.0 component.	Basically a 1:1 translation process.
Connector	- Base components related with the connector, connect themselves directly through the interfaces. - Aspectual components related with the connector, connect through the new crosscuts relationship.	All the aspectual information contained in the connector is translated to sequence diagrams where it is represented the aspectual interaction between the components connected to the connector.
Configuration	A UML 2.0 model.	All the elements transformed by previous rules are included in the new UML 2.0 model.

of the *Component*. The diagram shown in Figure 3.(b) is a template to be instantiated with the binding information of the *bind* relationship shown in Figure 3.(a). In our example, the *Component* would be instantiated to the *BuyerConnector* connector, the *Port* to the provided role *Buyer* and the *method()* is instantiated to the *increase()*, *bid()* and *join()* operations of the *Buyer* role.

Notice that by means of these diagrams the Theme/UML model maintains all the aspectual information included in the AO-ADL architecture.

3.3 Transformation to UML 2.0

UML is the *de facto* standard language at the design stage. In this transformation the main goal is to generate a UML 2.0 design that contain all the relevant information expressed in AO-ADL, but without the requirement of learning an specific aspect-oriented design approach to be able to continue the software development process. In order to do that we have defined a mapping between the AO-ADL and the UML 2.0 metamodels. We have only introduced a new stereotype named *crosscuts* that allows us to make explicit that the source of the relationship was identified as an aspectual component at the architectural level, and the target was identified as a base component at the architectural level.

Notice however, that the use of this stereotyped relationship is optional and can be omitted without changing the semantic of the model. This is possible because the information contained in an AO-ADL architecture is transformed into two kind of UML diagrams: statics and dynamics. As indicated in table 2, components in an AO-ADL configuration are always mapped to a single UML static diagram, a component diagram, where every AO-ADL component is transformed into a UML component. The information encapsulated in AO-ADL connectors is represented partly in component diagrams and partly in sequence diagrams. Thus, even if the *crosscuts* relationships are not specified in the component diagrams, the aspectual interactions between components encapsulated in the AO-ADL connectors are always transformed into a set of sequence diagrams. Notice that sequence diagrams have to be seen as the result of applying the weaving process between AO-ADL base and aspectual components, according to the aspectual binding information specified in AO-ADL connectors.

Following with our example, the output model generated by the transformation from AO-ADL to UML 2.0 is shown in Figure 4. Figure 4.(a) shows a

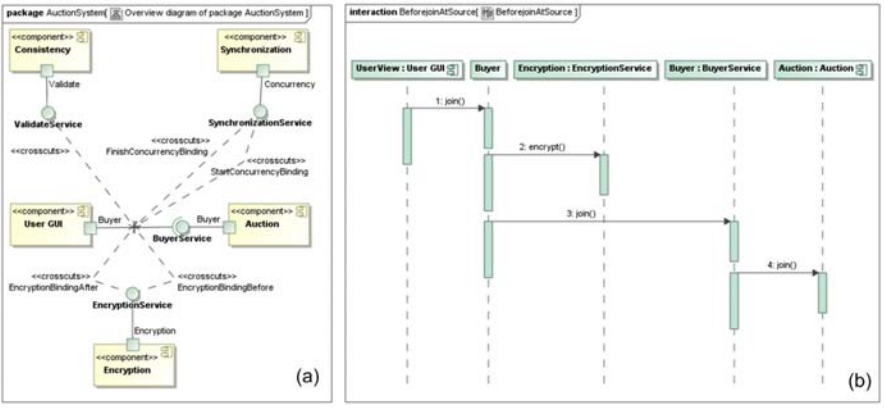

Fig. 4. Architecture of the Auction System in UML 2.0

component diagram (the static diagram) where we can observe that the components that were joined by the *BuyerConnector* at the architectural level are now directly connected by means of the *BuyerService* interface. Meanwhile, the components that had an aspectual behaviour at the architecture level (*Encryption*, *Consistency* and *Synchronization*) interfere with this relation by means of the *crosscuts* relationship. All the aspectual information that was contained in the *BuyerConnector* is transformed into dynamic diagrams like the sequence diagram we can see in Figure 4.(b). This figure shows how the *User GUI* component invokes the *join()* method through its *Buyer* port and that *before* transmitting the message to the corresponding port of the *Auction* component the *encrypt()* method is invoked, just as it was specified in the *BuyerConnector*.

3.4 Comparison

The output of the transformation to Theme/UML is an aspect-oriented design. This means that both the separation between crosscutting and non-crosscutting concerns and all the aspectual information identified at the architectural level is preserved in the design model. The design in Theme/UML enhances the aspectual vision of the architecture, since what you see in the design is a group of themes (e.g. concerns) and the interactions among them. The aspectual information is also well delimit into: (a) the themes that represent aspectual behaviour, and (b) the bind relationships. Finally, the AO-ADL connectors are explicitly represented in the design by using themes. On the other hand, the result of the transformation to UML 2.0 weaves the aspectual information into the model, making easier to implement the model using a general purpose programming language. The configuration of the architecture (the components and their relationships) is clearly visible in the static part of the design, but the aspectual information is translated from the AO-ADL connectors into the dynamic sequence diagrams, so the separation disappears from the design model.

4 The AO-ADL Tool Suite

Our MDD process is supported by the AO-ADL Tool Suite[2], an Eclipse plug-in that provides: (1) graphical support to describe and manipulate AO-ADL software architectures, covering the first step of our MDD process, and (2) integration of the architecture to design MDD transformations previously described, covering the second step of our MDD process. Finally, the designs generated with ATL do not contain graphical information, basically because this information is specific of each particular UML editor. Thus, our tool prepares the output of ATL transformations in order to automatically generate graphical information required to visualize the design in MagicDraw UML. This makes easier to continue the design of the system using an existing UML editor.

5 Related Work

AOSD and MDD can be considered complementary technologies. the main motivation for applying AOSD to MDD is to solve the problem that MDD presents regarding the lack of specific mechanisms for the separation of crosscutting concerns within each level of abstraction [13]. If all concerns can be modeled separately at a certain level, that level will become more manageable, extensible and maintainable. Most approaches combining MDD and AOSD follow this approach [13,14] where their main goal is to obtain "separation of aspects" in each MDA level. An alternative approach, closer to ours, consists of applying the MDD philosophy to the development of AO applications, by proposing MDD generation processes to transform AO models at different phases of the software life cycle. An example of this approach is our work in [15], in which MDA helped us to identify that there were different models in our aspect-component proposal [16]. Other examples are [17,18] that use MDD to transform AO requirement specifications into AO software architectures using a CAM profile and PRISMA respectively. The proposal in [11] transforms an Theme/UML AO detailed design into an AspectJ implementation. Our approach in this paper complements this approach in the sense that we use MDD to transform AO software architectures into Theme/UML AO detailed design. Finally, another example is the AOMDF (AO Model Driven Framework) proposal [19]. AOMDF combines both approaches, proposing an MDD framework that provides mechanisms supporting both vertical and horizontal separation of concerns.

6 Conclusions and Future Work

In this paper we have shown how the use of MDD can be used to bridge the gap between between AO architecture and (AO) detailed design. We have completely implemented the model-to-model transformations using ATL, integrating them into the AO-ADL tool suite. The complete mapping information and the main ATL rules are available in the AO-ADL web site. They have not been included in

[2] Downloadable from our website (http://caosd.lcc.uma.es/aoadl/download.htm).

the paper due to the lack of space. Interested readers can download the running example used in the paper, as well as other examples, from our web page, in order to use our tool to reproduce the MDD process presented in the paper. One current limitation of our approach is that pointcuts in UML do not capture all the expressiveness of AO-ADL pointcuts. In order to preserve the expressiveness of AO-ADL pointcuts at design, our next step is the definition of new transformations to the Join Point Designation Diagrams (JPDDs), which allow the specification of contextual static and dynamic join point selection criteria.

References

1. Garlan, D., et al.: ACME: An architecture description interchange language. In: Johnson, J.H. (ed.) Proc. of CASCON, Toronto, Ontario, pp. 169–183. IBM Press (1997)
2. Luckham, D.C., et al.: Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering 21(4), 336–355 (1995)
3. Medvidovic, N., et al.: Using object-oriented typing to support architectural design in the C2 style. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 24–32. Springer, Heidelberg (1996)
4. Dashofy, E.M., et al.: A highly-extensible, xml-based architecture description language. In: Proc. of WICSA, Amsterdam, The Netherlands, pp. 103–112 (2001)
5. Pérez, J., et al.: PRISMA:Towards Quality, Aspect-Oriented and Dynamic Software Architectures. In: QSIC 2003, pp. 59–66 (2003)
6. Pinto, M., Fuentes, L.: AO-ADL: An ADL for Describing Aspect-Oriented Architectures. In: Moreira, A., Grundy, J. (eds.) Early Aspects Workshop 2007 and EACSL 2007. LNCS, vol. 4765, pp. 94–114. Springer, Heidelberg (2007)
7. Beydeda, S., et al.: Model-driven development. Springer, Heidelberg (2005)
8. Kühne, T.: Matters of (meta-)modeling. Software and Systems Modeling (SoSyM) 5(4), 369–385 (2006)
9. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
10. Clarke, S., et al.: Aspect Oriented Analysis and Design. The Theme Approach. Aw Professional (2005)
11. Jackson, A., et al.: Mapping design to implementation. Technical Report AOSD-Europe Deliverable D111
12. Chitchyan, R., et al.: Mapping and refinement of requirements level aspects. Technical Report AOSD-Europe Deliverable D63
13. Amaya, P., et al.: MDA and separation of aspects: An approach based on multiple views and subject oriented design. In: AOM Workshop (2005)
14. Atkinson, C., et al.: Aspect-oriented development with stratified frameworks. IEEE Software 20(1), 81–89 (2003)
15. Fuentes, L., et al.: How MDA can help designing component- and aspect-based applications. In: Proc. of EDOC, pp. 124–135 (2003)
16. Pinto, M., Fuentes, L., Troya, J.M.: A Dynamic Component and Aspect-Oriented Platform. The Computer Journal 48(4), 401–420 (2005)
17. Sánchez, P., Magno, J., Fuentes, L., Moreira, A., Araújo, J.: Towards MDD transformations from AO requirements into AO architecture. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 159–174. Springer, Heidelberg (2006)
18. Navarro, E.: ATRIUM: Architecture traced from requirements applying a unified methodology, Ph.D thesis (2007)
19. Simmonds, D., et al.: An Aspect Oriented model driven framework. In: EDOC 2005, pp. 119–130 (2005)