On the Need of Architectural Patterns in AOSD for Software Evolution

M. Pinto, L. Fuentes, J. A. Valenzuela, P. F. Pires, F. C. Delicato, E. Marinho
Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain
DIMAp, Federal University of Rio Grande do Norte, Brazil
{pinto, lff, valenzuela}@lcc.uma.es; {paulo.f.pires, fdelicato, ebertonsm}@gmail.com

Abstract

One promising approach to tackle software evolution in AOSD is model-based pointcuts, where pointcuts are defined in terms of elements of a conceptual model, which are less susceptible to evolution than elements of the base model. We propose the definition of model-based pointcuts at the architectural level and identify three layers in the definition of our conceptual model: the system, the domain-specific and the application-specific layer. An MDD process drives the definition of conceptual and aspect models, their instantiation and composition. AO-ADL is used to implement it.

1. Introduction

As it occurs in other software engineering approaches, Aspect-Oriented Software Development (AOSD) is susceptible to software evolution, basically because the points in the base model intercepted by the aspects may change as the system evolves. In AOSD, the compositions between base elements and aspects are specified as pointcut expressions (PCE), which contain information about where to compose aspects with base elements. PCEs refer to collections of join points, which are specific points in the base elements candidate to be intercepted by aspects. As a consequence of defining such expressions, often pointcut definitions depend on a previous knowledge of the system design, resulting in a high coupling between aspects and base elements [1]. Due to this dependence, problems occur when the system evolves and a pointcut unintentionally captures or misses a given join point, as a result of apparently harmless modifications of base elements [2].

One promising approach to tackle the evolvability problem in AO systems, by minimizing the coupling generated by PCEs, is model-based pointcuts [2], which is based on building a conceptual model that describes structural and behavioural concepts shared by an application domain and classifying elements of the base model according to these concepts. Thus, PCEs refer to elements of the conceptual model instead of the base model. The main advantage of this approach lays on the premise that the conceptual model is less susceptible to evolution in the base code than the base model is. However, this and other approaches do not investigate if (and how) this problem can be solved at the architectural level, and postpone their solutions to lower stages of the development (detailed design or implementation).

In this paper, we propose a solution to minimize the coupling between base and aspect models at the architectural level. Since architectural and design patterns make explicit the structure and behaviour that are commonly implicit in the application code, we argue that these patterns could be the base of a methodology to build an abstract representation of a software system, which may be used as reference for the pointcuts creation. In our work, these patterns are used as the underpinning for building the conceptual model of a system. However, we recognize that not all concepts in a system are fully classified only using well-known architectural patterns. There are domain specific and application specific concepts that, besides being classified as belonging to an architectural pattern, need further attributes that are essential to represent their meaning in the context of the domain/application. Thus, we propose to organize the conceptual model in three layers, encompassing widely used generic patterns (system layer), domain specific concepts (domain layer), and application specific concepts (application layer).

An MDD process drives the definition of our conceptual and aspect models, as well as their instantiation and composition to generate the base model architecture. Since we are proposing an architecture oriented approach to define the conceptual model, we propose the use of AO-ADL (Aspect-Oriented Architecture Description Language [3]) to define both our conceptual model and the PCE used to intercept such conceptual model.

2. Our Approach: The Models

In the current version, our solution is instantiated using the AO-ADL language and its tool support. The complete description of AO-ADL is not included.

1 Supported by Spanish Project TIN2008-01942, European STREP Project AMPLE IST-033710 and Brazilian Funding Agency CNPq.
here due to lack of space, and can be found in [3,6].

In AO-ADL the same architectural building block is used to model both components (named base components) and aspects (aspectual components). AO-ADL components are specified in terms of their provided and required ports. The interactions among components are defined within AO-ADL connectors.

The semantic of connectors is extended with the definition of aspectual roles and aspectual binding information. Thus, components playing the role of a base component are connected to the connector provided and required roles, while components playing the role of aspectual components are connected to the connector aspectual roles. The aspectual binding information defines the weaving between base and aspectual components, including the definition of PCE and advices (before, after, concurrent-with, around). Figure 2 shows AO-ADL components and connectors. Figure 3 shows examples of aspectual roles and aspectual binding information. AO-ADL provides the expression power requested to model concepts in our proposal through the following features. (1) Semantics-based composition: a classifier can be associated to any element of AO-ADL; pointcut expressions can be specified in terms of these classifiers, and not in terms of the name of components, interfaces or operations, (2) Semantics of connector roles: the roles of connectors are defined specifying a query that identifies the component(s) that may be connected to these roles, instead of just binding a particular interface to a role.

### 2.2 The Conceptual Model

Our proposed conceptual model is organised in three complementary layers: the system, the domain-specific and the application-specific layer. The system layer includes well-known architecture or design patterns. Usually, any software architect will be able to define a new system layer pattern. Examples at this level are the MVC and the Visitor patterns. Figure 2 shows an example of the MVC pattern in AO-ADL. There are components representing the Model, View and Control concepts of the MVC pattern, connectors modelling the connections between them, and the roles played by each interface. The ports (interfaces) provided and/or required by components model the typical behaviors of components that have the semantics of a Model, a View or a Control: set represents a port that assigns or updates a value in a concept of Model; get denotes a port which provides some information about entities in the Model; show represents ports which display a graphic interface to the user; command represents a command issued by View, and commandHandler handles a specific command.

The domain-specific layer includes patterns of behaviour and interaction including concepts specific of a particular application domain. Thus, domain-specific patterns should be included by software architects that are expert on that domain. An example at this level could be concepts specific from the multimedia domain. Most of multimedia applications require processing of data streams in order to play media files. So, we can define the ManagerStream concept which represents the functionality of transforming a media file in a data stream. ManagerStream has interfaces named inputStream, responsible to receive data, and outputStream, which provides streams of data. The Player concept has an interface named inputStream that represents a data stream which will be displayed to the user.

Finally, the models in the application-specific layer are defined to accommodate particular requirements of an application that do not fit appropriately in the roles defined in generic architectural or domain-specific patterns. They are patterns of behaviour and interaction that, being specific of a single application, are recurrent in different places of the application and are subject to be affected by different scattered and/or tangled behaviours represented by aspects. They will be normally defined by software architects modelling the software architecture of a single system.

It is worth noting that both system and domain patterns are not dependent on the specificities of an application. Thus, base elements classified as concepts of these layers can evolve while their pattern classification will remain unchanged. In this case, once the PCE references pattern elements, instead of base elements, the coupling between base elements and aspects is minimized since the crosscutting concerns will intercept much more stable constructs (patterns) comparing to base elements. Moreover, the PCEs referencing concepts in these layers are prone to be reused, since they reference
patterns that are shared by a large number of applications, as illustrated in the rest of the paper.

2.3 The Aspect Model

In this Section we illustrate the definition of aspect models using our model-based pointcut approach. We use a case study [4, 5] based on the MobilePhoto and MobileMedia, where several aspects are applied to applications that store, manipulate and visualize photos. Figure 3 shows how the Consistency and the Formatting aspects intercept the interaction between the Control and the View components in the MVC pattern (defined using a query in the specification of the connector roles). This aspectual interaction indicates that the output of the application needs to be checked for consistency, and also that it needs to be formatted according to the kind of view (e.g. display in a resource-constraint device screen). Consistency and Formatting are behaving as aspectual components since they are connected to the connector aspctual roles (ConsistencyRole and FormattingRole). The aspectual binding information indicates that the aspects are applied before (advice) the show operation is received in the component connected to the Show role of the connector.

![Figure 3. Aspect Model in AO-ADL](image)

2.4 The Base Model

As an example of a base model, Figure 4 shows the conceptual model of Figure 2 (along with the multimedia domain specific pattern) instantiated several times to generate part of the software architecture of the MobilePhoto application. Notice that the instantiated components, interfaces, roles, connectors, etc. are classified, using the AO-ADL classifier attribute, according to the patterns from which they were instantiated. Notice also how the aspect model defined in Figure 3, in terms of the elements of the conceptual model, intercepts here the elements of the base model. For instance, the Consistency and Formatting aspectual components intercept the interaction between the BaseController (classified as Control) and the PhotoView components (classified as View), according to the information provided in the aspect model of Figure 3. The figure also shows the Persistence aspectual component intercepting interfaces classified as set in Model components.

3. Our Approach: The MDD Process

Using Model-Driven Development (MDD) [7], a software system is created through the definition of different models at different abstraction layers, where models of a given abstraction layer are automatically derived from models of the upper abstraction layer by means of model transformations. In order to define model-based pointcuts, we have specified an MDD process that distinguishes two levels in the definition of software architectures: (1) An M1 level, where the conceptual models and reusable aspect models are defined, and (2) An M0 level, where patterns defined in level M1 are instantiated and composed using MDD technologies to automatically generate the software architecture of a particular system. All the steps in this process are fully supported by the AO-ADL tool.

- **M1 Level (Conceptual Model).** In level M1 most of the identified patterns should be classified as part of the system or the domain-specific levels. We have even considered the possibility of moving the application-specific level patterns to level M0. Finally, we decided to keep them in level M1 since, even for application-specific patterns, when the same behaviour is recurrent in different parts of a same system, it is worthy to define reusable aspect models based on the definition of model-based pointcuts.

- **M1-M0 Transformation and Composition.** This MDD transformation receives as inputs two or more instances of M1 elements (possibly from different models – system and/or domain and/or application) and generates an instance of a M0 element. The elements of such M0 architecture can be traced up to the different M1 models, since during the instantiation process, the role played by an M1 architectural element (interface, component, connector) is used to classify the generated M0 element. Several classifiers will be used to classify a M0 architectural element when the M0 architecture is generated throughout the composition of different M1 models – i.e. when the same component plays different roles in each different model. For instance, Figure 4 shows how a given component (e.g. BaseController) can behave both as a Control (from the MVC pattern in the system layer) and a ManagerStream (a concept of the domain layer). PCE are also transformed at this point. Concretely, pointcuts expressed in terms of the names of components, interfaces, etc. in level M1 are defined in level M0 in terms of AO-ADL classifiers. This is aligned with the fact that elements in level M1 are used to classify the elements generated in level M0.

- **M0 Level (Base Model).** The output of the M1-M0 transformation process is a single vision of the actual system architecture, without aspects. Notice that aspects crosscutting such architecture where
already defined in level M1. This is an important benefit of using model-based pointcuts, since PCEs that link aspect models with the base behaviour of software architectures are defined at the conceptual level (level M1). Then, their effects are automatically propagated to the base model when M1 elements are instantiated and composed to generate a M0 element. At this point, the reader may question the reason for not including new aspect models at level M0 (e.g. they were not identified at level M1 because they can only be specified in terms of elements of the instantiated model, and not in terms of the roles played by those elements in the conceptual model). The reason for not considering this possibility is that although new aspect model may be defined at level M0, they are out of the scope of our proposal since a PCE defined directly in level M0 will suffer from the pointcut fragility problem that we are trying to solve.

- **Architecture to Design Transformation and Weaving Process.** This MDD transformation allows us to close the gap between the notations used at the architectural level (an ADL in our case) and the notations used at the design level (UML 2.0 is accepted as the standard in most cases). We have used ATL as the MDD tool to automatically generate a UML 2.0 skeleton of the system design based on the information provided at the architectural level. The weaving process of aspects and components, according to the aspectual bindings provided by AO-ADL, is also automatically done at this point. The output of the previously defined transformation is a UML 2.0 model. ATL rules and generated UML models are not shown due to the lack of space. Videos showing them are available in [6].

4. **Final Remarks**

In this paper we have motivated and shown the usefulness of defining a conceptual model at the early stage of software development to minimize the high coupling between aspects and base elements which often occurs in AOSD, hindering the software evolution. We also depicted the benefits of adopting an architectural approach to build such model and propose organizing the model in three layers with different degrees of abstraction and reuse. Besides this, the layered organization allows some sort of reasoning on the advantages of employing AOSD in a given system. If most of the concepts to be affected by crosscutting behavior represented by aspects lay on the application-specific layer, such system tends to be more susceptible to evolution problems generated by the use of aspects. On the other hand, aspects affecting system and domain layer present a high robustness in the presence of software evolution. Moreover, such organization provides a hint on the advantages of modeling concerns as aspects. Aspects affecting concepts of upper layers (system and domain) are more prone to crosscut several elements in a given application architecture, thus achieving the real benefits of AOSD.

5. **References**