Towards Generic Weaving of Adaptation Aspects for XML

Sven Karol[1], **Matthias Niederhausen**[1], **Uwe Aßmann**[1], **Klaus Meißner**[1], **Martin Steinfeldt**[1]
[1]Fakultät Informatik, Technische Universität Dresden, Dresden, Germany

Abstract— *XML is one of the most used languages in the Web and is frequently used to describe large parts of web applications. A common approach to reduce complexity of web applications relying on XML is to use a multistaged architecture in form of a transformation pipeline. These pipelines usually employ a fixed set of complex transformations to convert an XML source document into a displayable output format. A second common principle is to separate a web application's context-independent part from the context-dependent adaptation facet.*

In this paper, we present a generic approach that uses aspect-oriented programming (AOP) to separate and weave the adaptation facet of XML-based web applications using a multistaged architecture. We introduce an AOP-based terminology for adaptation aspects and present an existing aspect weaver prototype that realises our approach.

Keywords:aspect-oriented programming, xml, adaptive hypermedia, multi-staged weaving

1. Introduction

The eXtensible markup language (XML) is used in a wide range of applications in the World Wide Web. From websites (XHTML) over databases and repositories (RDF/XML), data exchange protocols (XMI, SOAP), documentation technologies (Docbook) to configuration files – few application domains renounce it nowadays.

Common to many XML applications is the principle that XML files are subject to multi-staged transformations, i.e., they undergo a number of transformations during the application's life-cycle (e.g., when fragments of XML are composed to a complex document). Frequently deployed stages of such a pipeline are, for instance, generation of data representations for visualization, conversion between different XML dialects or homogenization of different XML sources. In this process, typically, each stage generates its own, intermediate document.

A common approach to reduce the complexity of the design of such stages is to introduce a separation of concerns. In the field of web engineering, for example, a separation of core application and the facet of adaptation can be achieved [12], [10]. Furthermore, one can distinguish transformations that generate consistent results for the same input from transformations that additionally depend on context information (e.g., user preferences, interaction history, user location, device data or pipeline state). The latter ones are also called *adaptation transformations* since they do not change the basic execution path, but add transformations to the pipeline that adapt intermediate results to take a certain context into account.

However, current adaptation approaches like aspectWebML [12], aspectUWE [2] or OOHDM [11] are limited to a specific web application architecture and cannot be applied in other areas. Furthermore, designing adaptive transformation chains remains a complex task — with changing contexts, adaptation transformations may interact in unforeseen ways, causing headaches to developers of web application frameworks and to authors who use these frameworks to publish content.

Since adaptation transformations do not belong to the functional core of a web application [10], aspect-oriented programming (AOP) [8] can be used to specifiy adaptation transformations in separation from the actual core transformations. General AOP languages like AspectJ [7] can be used to weave adaptation transformations into an XML transformation pipeline, however, they require in-depth knowledge of the pipeline engine and provided APIs (e.g., DOM, the engine's public API or the context model) or even specific weaving hooks that have to be introduced manually for each pipeline. To support authors in specifying their own adaptation transformations, a more XML-specific approach is needed, including abstractions to access pipeline state and context model.

In this paper, we present our tool *PX-Weave* (Pipe-based XML Weaver), an aspect-oriented weaver for generic adaptation transformations in a multi-staged XML transformation environment. Being an offspring of our existing adaptation environment *HyperAdapt* in *Cocoon*[1], it provides an XML-based language for defining adaptation aspects that can be woven into multi-staged XML transformation pipelines. Aspects in PX-Weave can depend on a complex context and pipeline state. The design of the context model will not be deepened in this paper, as the focus of this paper lies more on transparently adding adaptation stages to the pipeline. To ease implementation and testing, the weaver manages an aspect depency graph for determining the weaving order and allows to integrate schema-based well-formedness checks at arbitrary execution states of the pipeline.

This paper is structured as follows: Section 2 introduces a sample multi-staged web application and shows how plain AOP, based on AspectJ, can be used to weave adaptation transformations. Section 3 explains the concepts behind our tool together with the achieved improvements and the general architecture of PX-Weave. Next, Section 5 compares PX-Weave to other approaches that use aspects for adaptating web applications. Section 4 then provides details on its implementation. Finally, Section 6 concludes the paper and presents our extension plans for PX-Weave.

[1]`http://cocoon.apache.org/`

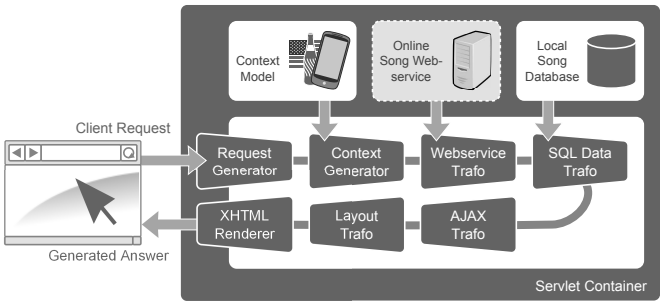

Fig. 1: SoundNexus portal architecture

2. Case Study

In this section, we will introduce SoundNexus, an example adaptive web application based on Cocoon. Afterwards, we will show how AspectJ can be used to weave adaptation transformations into the transformation chain that is executed.

Cocoon is a Java-based web application framework which allows to create XML transformation pipelines. To this end, the web application developer specifies several XML transformations and links them into one or more nested transformation pipelines. Cocoon then deploys them as a servlet on the web server. Though mainly geared towards transformations based on XSLT stylesheets, it also supports plain Java code using SAX or DOM APIs. The main advantages of Cocoon are its optimisation capabilities and the *Separation of Concerns* (SoC) principle, which is also at the heart of AOP. In contrast to a single, monolithic transformation, each pipeline stage handles a different concern. To achieve acceptable processing speeds in comparison to monolithic transformations, the framework supports transformation chaining by pushing a SAX stream through the pipeline instead of processing the stages successively.

2.1 A Simple Web Application

SoundNexus is an adaptive web application featuring an online music database which is aggregated from multiple sources, such as traditional SQL data sources or web services. It uses Cocoon's multi-staged transformations to transform a requested page from an abstract page description format into XHTML. This format lets web application authors specify single pages, data source configurations and adaptations. Further, an authoring toolkit and a set of graphical editors to control page structure, layout, linking and style of a page are provided.

Figure 1 gives an overview over the different pipeline stages and data sources of *SoundNexus*. A request is triggered by an arbitrary client device capable of processing HTTP requests and responses as well as rendering XHTML pages.

Every request is first sent to a Servlet container which contains the pipeline. A `RequestGenerator` selects and initialises the adequate pipeline resource from the Cocoon configuration[2]. In the second stage, the `ContextGenerator` initialises the context model by first evaluating standard HTTP request parameters (e.g., language or browser), and afterwards data collected by context sensors, for instance, Geo IP services or SoundNexus' client-side Javascript sensors. If a page requires additional data from a web service, e.g., to request the current top 10 charts, the `WebServiceTrafo` translates requests embedded within the document into standardized SOAP requests and sends them to the specified `Online Song Database`. The result is then added to the document by using a template specified by the author. Similar steps are taken by the `SQLDatabaseTrafo` which evaluates embedded SQL queries by delegating them to the local song database. After the content has been injected, the `AjaxTrafo` component derives a view from the transformed document. This way, only those parts of the page requested by the client will be delivered (local update in the browser). Finally, `LayoutTrafo` and `XHTMLRenderer` compile the page into a layouted XHTML document that can be rendered by standard browsers.

[2]Request Generators are built-in Cocoon components.

2.2 Adapting the Transformation Pipeline Using AspectJ

In order to provide SoundNexus users with a custom-tailored experience, the document traversing the pipeline must be adapted according to the user's context. Adaptation can be performed by inserting specialized *adaptation transformers* into the Cocoon pipeline. However, there is no optimal single point for adaptation, as this depends on the type of adaptation (e.g., adaptation of web service requests must be performed in the beginning while changing the page layout is ideally done at the end). Because inserting an adaptation transformer between every two stages would bloat the pipeline definition and is susceptible to pipeline changes, we propose to use aspect-oriented programming to weave in additional adaptation stages as needed. AOP is a well-established paradigm in software engineering. Aspect components allow to separate parts of an object-oriented program from its functional core. Hence, aspects separate so-called cross-cutting concerns from functional core concerns. Typical examples for cross-cutting concerns are logging, persistency or security. Such concerns would normally require programmers to insert identical or only slighty modified lines of code at multiple places throughout the whole program or to spread code which semantically belongs together. As a result, AOP increases maintainability and leads to more comprehensible and decomposed programs.

In AspectJ, aspects contain *advices* and *pointcut expressions*. Pointcuts capture sets of *joinpoints* during program execution that can be used to weave in additional behaviour and to change programs. Joinpoints mark the places where aspects can be woven and consist of different parts. Their static part is related to elements of a program's abstract syntax tree (AST), for example, a method in a class. Accordingly, the dynamic part is related to the actual program state (e.g., all method calls within a specific control flow). Advices specify what has to be woven at a joinpoint and how it has to be woven. The basic kinds of advices are *before*, *after* and *around*. The first two kinds of advices add program code directly before or after a joinpoint has been reached, while the third kind may replace a joinpoint completely.

In the following, we exemplify how plain AspectJ can be

```
1  pointcut webServiceTrafoHook(Document doc):
2    execution(* AmacontWSTransformer.transform(Document)) &&
3          args(doc);
```

Listing 1: Weaving hook `webSeriveTrafoHook`

used to specify parts of a "device independence" adaptation concern for *SoundNexus*. In this example, the concern is implemented by a single AspectJ aspect. We first need to specify the hooks where to weave. In our case, these are the pipeline stages presented in Section 2.1. Listing 1 shows the hook definition for the `WebServiceTrafo` component.

Note that it uses pointcut expressions for addressing the *execution* joinpoint of the *transform* method in the `AmacontWSTransformer` class. This method contains the actual implementation of the execution of embedded web service calls. The remaining stages' hooks can be specified in a similar way[3].

```
1  before(Document doc):
2   webServiceTrafoHook(doc)&&this(AmacontWSTransformer){
3
4    OntModel model = ContextOntology.getCurrentOntology();
5    injectRules(model);
6    log.info("Weaving␣Device␣Independence␣...");
7    if(model.getIndividual("...#image_Capability") == null) {
8
9    // 1. Setting optimised Query
10    Element queryNode = findElementNode(...);
11    computeNewQuery(queryNode);
12
13   // 2. Removing image components
14    Element[] imageComponents = findElementNodes(...);
15    for (Element e:imageComponents) removeComponent(e);
16
17   // 3. Removing references
18    Element[] refComponents = findElementNodes(...);
19    for (Element e:refComponents) removeComponent(e);
20 }}
```

Listing 2: Advice contributing to the device independence concern

For the sake of brevity, we only discuss the one advice shown in Listing 2, although there are numerous advices imaginable for adapting a *SoundNexus* page's content. Our advice's objective is to check whether the current device is capable of displaying images (e.g., the page may be displayed in a text browser or it may have a too small screen) and to safely remove the images otherwise. The first two lines indicate to weave the advice's body before the webService-TrafoHook, i.e., before the `AmacontWSTransformer`'s *transform* method is executed. Lines 4 and 5 access the current context model and inject rules to reason about the available context data in order to derive the device's image processing capabilities. If this data cannot be derived, all image components are removed by inserting a new database query which does no longer select the image data field (lines 9–11), deleting the actual images from the current page (lines 13–15) and removing the layout nodes which reference the image containers (lines 17–19).

[3]The `RequestGenerator` and XSLT-based transformations are built-in Cocoon components and thus may require to consider framework-internal call protocols and weaving of deployed classes.

2.3 Case Study Observations

Since current AOP languages like AspectJ are not designed w.r.t. to the specifics of XML documents and multi-staged XML transformations, handling such content is difficult. To specify an adaptation aspect, authors have to have a deep insight into the source code of the framework running the pipeline and need to know how and when each stage of interest is called by the host application. Furthermore, authors have to be aware of the context model's access routines and how to inject the rules they would like to query.

Additionally, general aspects may interact with each other and the base program in unpredictable ways [9]. Since the program code is changed invasively, general aspects can easily break implicit and explicit contracts of the program's base components, e.g., by changing a method's implementation through adding or removing some parts of its implementation code. Although changing a method's implementation is also possible with subclassing in plain object-oriented programming languages, subclassing neither touches the base class nor does it overwrite private methods. Approaches for statically or dynamically detecting aspect interactions have been extensively studied in literature [5], [9], [4]. While we are not aware of any general solution to aspect interactions, we have to consider this problem in our approach.

To improve over using general AOP languages in web applications, our approach has the following objectives, based on the above observations:

1) Transformation pipeline elements should be supported by the pointcut language.
2) Advices have to support XML adaptation transformations with precise semantics.
3) The weaver tooling must support intermediate, mixed content at any pipeline stage and should support weaving of schema-aware adaptation advices (e.g., for testing purposes).
4) A static aspect analyser should provide authoring support for aspect interactions/interference at shared joinpoints.
5) The context model should be accessible through an API and reasoning facilities should be available if an ontology is used.

3. Aspect-orientation for XML

Starting from the requirement of precise semantics for advices, we distinguish at least three kinds of concerns that may occur in XML documents. Content concerns are pieces of text related to a specific topic distributed over the document. Presentation concerns deal with machine-interpretable information on how to display and structure things (e.g., layout, order). And finally, adaptation concerns modify other concerns according to context-dependent objectives (for example, device independence, accessibility, personalisation).

In contrast to the pure programmatic approach in general AOP, we consider component-based transformation environments together with well-defined context models. In particular, we focus on adaptation concerns in general XML documents. In the following we present a terminology for

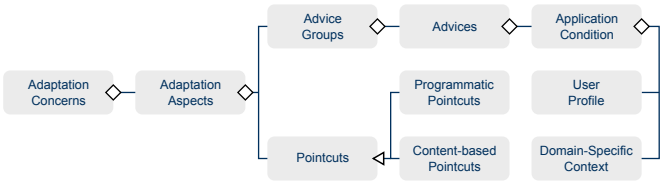

Fig. 2: A taxonomy for adaptation aspects

adaption aspects and a basic set of *adaptation advices*. A discussion about interaction of *adaptation aspects* concludes this section.

3.1 A Terminology for Adaptation Aspects

Our approach builds around a basic terminology for adaptation aspects, derived from terms used in AOP.

Figure 2 gives an overview in form of a concept hierarchy. *Adaptation aspects* are manifestations of *adaptation concerns*, which — at a very abstract level — group one or more aspects contributing to the concern's objectives. *Advices* in XML adaptation aspects do not only adapt documents at joinpoints, inserting or removing program code, but also have to consider other concerns within documents. *Advice groups* are introduced to limit the scope of a set of adaptation advices to a fragment of a document, which is essential to make advices on top of complex, nested documents manageable for developers. In our scenario, the advice's *application condition* is built on top of the context model, which can be split into two parts. In its general, *user profile* part, a context model provides information common to most application domains (like a basic user profile or location data). In contrast, the *domain-specific context* provides domain-specific data, depending on the concrete application (delivered by specific context information providers). Furthermore, context models may include support for reasoning about context. Adaptation aspects, however, do also have pointcuts. Given the XML nature of our approach, a pointcut can either address points in the pipeline execution — as a *programmatic pointcut* — or the XML content that flows through the pipeline — as a *content-based pointcut*. By combining both types of pointcuts, an aspect has the power to select a point in time and space of an XML document in transformation.

In the following section, we discuss the set of basic advices that we provide for adaptation aspects.

3.2 Classes of Adaptation Advices

Adaptation aspects can either be realized as arbitrary transformations or as a sequence of well-defined primitive operations. While the former offers the definite maximum in flexibility, it lacks an important property: ease of analysis. With two arbitrary adaptation aspects, the combined result is hard to predict. Therefore, we decided to use the latter approach, providing an extensible set of primitives. These primitives were conceived as to fulfill two opposed goals. On the one hand, they should be generic, independent of the application domain, so they can be utilized in any kind of XML application. On the other hand, they should form more than a kind of "assembler language" for XML

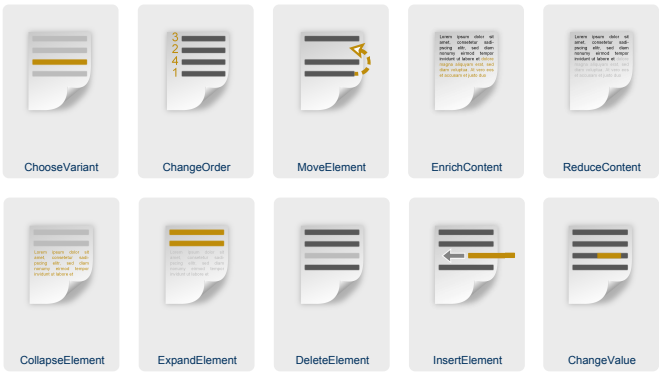

Fig. 3: Classes of adaptation advices

transformations. That is, they must be powerful enough to easily implement the more complex adaptation patterns that are widely recognized in the domain of web engineering [3]. The compromise we found, offering moderately powerful operations while still being generic to all XML domains, are the following ten advice classes.

Figure 3 gives a graphical impression of the different advice classes explained subsequently.

- *ChooseVariant* selects a variant from a set of exchangeable instances of an element node, according to the current context. The document author can define every variant individually in advance. A pointcut expression can then match them such that the most appropriate variant is selected at runtime. Example: A common use case for this kind of advice is language adaptation. Consider a multilingual document which provides different language versions of each paragraph. Language independence can be achieved by specifying an adaptation aspect that selects the correct variant according to the reader's language – ideally as early as possible in the pipeline.
- *ChangeOrder* changes the order of child nodes of an element node, according to a given permutation expression. Example: The order of elements is important when it comes to layout adaptations for different types of devices and their displays. Furthermore, elements may be ordered to reflect user preferences, e.g., after a transformation that aggregated data from multiple, inhomogenous sources.
- *MoveElement* removes an element at a joinpoint and inserts it at a different position in the document tree. Although this advice is quite simple, it is very useful for adaptations that perform restructuring. Example: Different display formats might require different document layouts, e.g., a generated XHTML page may use a table-based, vertical layout by default. From this default representation, a complex adaptation advice can create a horizontal layout by moving the elements into a horizontally arranged table if a widescreen display is used.
- *EnrichContent* adds or inserts additional text fragments in textual parts of a document, based on a text selection expression. This kind of advice allows to encapsulate

related parts of text in adaptation aspects at a very fine granularity. Example: An adaptive website may provide different kinds of access levels, e.g., for registered users and unregistered users. Registered users may see an adapted webpage with content annotated by user comments while unregistered users have only access to standard content.

- *ReduceContent* removes text fragments in plain textual parts of a document according to a text selection expression. This advice is the inverse of *EnrichContent*. Example: For advanced users, prerequisite steps of an installation tutorial can be omitted to increase readability.
- *CollapseElement* can be used to replace a complete subtree of a document with plain text. Example: In order to provide devices with small screens or little processing power with an appropriate representation of complex content elements (like Flash movies or JavaScript-intense containers), the author may decide to replace these elements with a short, textual abstract.
- *ExpandElement* substitutes a text fragment in the document with a more complex element or subtree. This advice is the inverse of *CollapseElement*. Example: A translation service can be extended to provide paying customers not only with the textual translation, but annotate the single words or phrases with the possibility to click them in order to find out how they are pronounced.
- *DeleteElement* is a basic advice that removes a subtree from a given XML document at a specific joinpoint. It can be compared to the *around* advice in AOP. Example: Administrative links on a blog can be removed for unregistered users.
- *InsertElement* is a basic advice that adds an element before or after a specific joinpoint. This kind of advice can be considered the closest relative of the *before* and *after* advices in AOP. Example: Additional information can be embedded in popup layers when users request help.
- *ChangeValue* is a basic advice that can change attribute values at a certain joinpoint. Example: According to the user's interests, an adaptation concern may require the author to highlight different concerns within the document by changing text colors or adding boxes around affected text paragraphs.

3.3 Interaction of Adaptation Advices

Similar to aspects in AOP, there is a risk for aspect interaction, i.e., that aspects influence each other. For example, an aspect `A` which is applied sequentially before an aspect `B` in the transformation chain may influence or even hinder the application of `B`. By default, aspect interaction is not a problem, since influencing another aspect's execution is a common case. However, problems arise if interactions cannot be foreseen, e.g., when too many aspects or blackbox aspects (i.e., from third parties) are deployed. Many approaches investigate semantic interactions, e.g., by monitoring program state [4] or simulating state spaces through graph rewriting [1]. In contrast to that, Kniesel [9] analyses weaving interferences, which occur when woven aspects show an unintended behaviour. In the following, we transfer terms coined in [9] to our domain. We use doc for the document to which adaptation advices are applied and L_{doc} for the set of possible documents, which may be given by an XML schema. The set of nodes in doc is denoted by N_{doc}. Furthermore, we use $ppc(a)$ and $cpc(a, doc)$ to refer to the programmatic pointcut and content-based pointcut of an advice a w.r.t. doc. The application of an advice to a document is denoted by $a \bullet doc$. The following equation defines when adaptation advices directly affect other deployed advices by influencing their content-based pointcuts[4].

$$\forall a, b \in Advices : \exists doc \in L_{doc} : ppc(a) \cap ppc(b) \neq \emptyset \wedge cpc(a, doc) \neq \emptyset \wedge cpc(b, doc) \neq cpc(b, a \bullet doc) \Rightarrow affects(a, b) \quad (1)$$

Based on affection, two special kinds of interactions can be derived (see equation 2). *Triggering* applies when one aspect modifies the set of joinpoints such that another one's pointcut now matches additional joinpoints. For example, consider an aspect that uses *CollapseElement* to replace images with a textual abstract. If there is another aspect responsible for internationalisation of text content, this aspect will now also match the newly created abstract.

$$\forall a, b \in Advices : \exists doc \in L_{doc} : \exists n \in N_{doc} : affects(a, b) \wedge n \notin cpc(b, doc) \wedge n \in cpc(b, a \bullet doc) \Rightarrow triggers(a, b) \quad (2)$$

The opposite interaction type is *inhibition*, preventing another aspect from applying (see equation 3). For example, one aspect could use *ExpandElement* to replace a piece of text with an image. In this case, our internationalisation aspect from above would no longer match the intended piece of text in its pointcut.

$$\forall a, b \in Advices : \exists doc \in L_{doc} : \exists n \in N_{doc} : affects(a, b) \wedge n \in cpc(b, doc) \wedge n \notin cpc(b, a \bullet doc) \Rightarrow inhibits(a, b) \quad (3)$$

Until now we considered interaction of adaptation advices in a general way. However, there are subtle differences w.r.t. the specific advices of Section 3.2. Advices such as *ChangeOrder* and *MoveElement* can be regarded as harmless or passive since they do not add or remove content. Hence, if sufficiently robust poincuts w.r.t. the XML tree structure are used, they should never affect advices of other adaptation aspects. The same holds for *ChangeValue*, *EnrichContent* and *ReduceContent* if other adaptation advices do not rely on the values manipulated by these advices. In contrast to that, *InsertElement*, *ExpandElement*, *DeleteElement* and *CollapseElement* add new nodes or remove nodes from the tree and thus actively trigger or inhibit other advices. Hence, as a rule of thumb, harmless advices should be applied subsequently to active ones, such that their effects are also applied to newly introduced parts of the document. A special

[4]For now, we only consider advices applied to the same programmatic joinpoints and do not consider context-dependent application conditions.

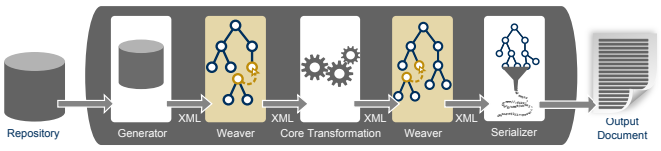

Fig. 4: Weaver in context of a pipeline

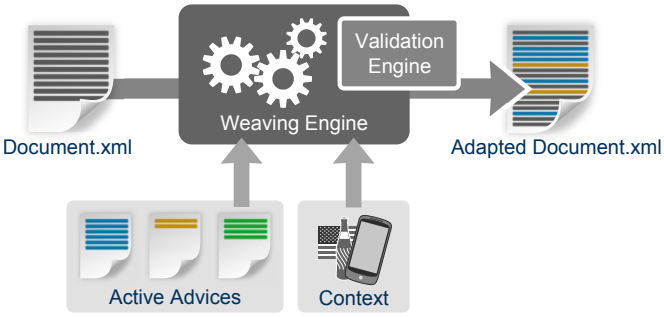

Fig. 5: Execution of weaving at a single stage

```
<aspect name="DeviceIndependence">                                    1
<interface> <!— context parameters—> </interface>                     2
<adviceGroup>                                                         3
  <depends> <!— application condition —> </depends>                   4
  <scope>                                                             5
    <xpath>//aco:AmaSetComponent[@id='a56zuaa']/</xpath>              6
    <before>wstransformer</before>                                    7
  </scope>                                                            8
  <advices>                                                           9
    <changeValue>                                                     10
      <pointcut> <!— query location —> </pointcut>                    11
      <value> .... </value>                                          12
    </changeValue>                                                    13
    <delete>                                                          14
      <pointcut> <!— images location —> </pointcut>                   15
    </delete>                                                         16
    <delete>                                                          17
      <pointcut> <!— references location —> </pointcut>               18
    </delete>                                                         19
  </advices>                                                          20
</adviceGroup>                                                        21
</aspect>                                                             22
```

Listing 3: PX-Weave advice for device independence

kind of advice is *ChooseVariant,* which may inhibit other advices by choosing a remaining node and removing the others. Since these inhibitions are intended (only the selected variant should remain in the document), they may be ignored during analyses.

4. Initial Tool Architecture and Prototype Implementation

In this section we present a first prototype implementation of our tool, PX-Weave. While our approach is generic in the sense that it can be applied to arbitrary XML transformation pipeline environments, PX-Weave is a Java-based, Cocoon-specific realisation. Its main components are a *weaving engine*, an XML-based aspect language[5], a *validation engine* and an *aspect analyser*.

Figure 4 shows the weaving engine embedded in a short cocoon pipeline consisting of a single, complex core transformation transformer, which is the only programmatic joinpoint in this pipeline. A close-up view of the weaving engine is presented in Figure 5. It takes an ordered set of adaptation advices at a certain programmatic joinpoint (i.e., before or after a pipeline stage is executed) and applies them to the specified content-based joinpoints in the current intermediate XML document (i.e., which is input or output of the current pipeline element). The set of active adaptation advices is determined by the aspect analyser which maintains a set of deployed adaptation aspects. Active advices can be identified by monitoring the pipeline state and selecting the currently applicable advices. The default advice order is derived from the order of aspects in the filesystem and by the advice order in the aspect specification. To avoid non-deterministic behaviour, authors can specify their own orderings over aspects. Even though such an ordering is sufficient to resolve aspect interactions unintended by the author, we intend to extend our *aspect analyser* component to provide static analyses for discovering such unintended interactions.

[5]In the future, we intend to include sophisticated editing support for such aspects.

A *validation engine* is provided for two reasons: Invalid intermediate XML documents and errors during weaving may yield unexpected results. However, since validation of XML files is a computationally expensive process, it should only be used to a minimal extent (or for testing purposes) in a specific weaving scenario. The *validation engine* can be run in different validation modes. The first mode allows to enable validation before and/or after all active advices of a pipeline element have been applied. A second, more expensive validation mode is offered for checking the validity of single XML transformations of an advice. In the prototype, this procedure is restricted to globally declared elements in the XML schema. Given that any XML schema can be transformed into a form that only incorporates such elements, this does not pose a far-reaching limitation of the validation mode's applicability.

Finally, Listing 3 shows an example PX-Weave adaptation aspect for the concern of device independence in SoundNexus, which we introduced in Section 2.1. The aspect `interface` in line 2 declares context parameters, the core document format to be adapted and imports additional XML fragments. The `adviceGroup` contains a context-based application condition declared by the `depends` element in line 4. The `scope` element declares a common context for all adaption advices within the advice group. Note that currently, we simply use XPath expressions for content-based pointcuts. The actual programmatic joinpoint (i.e., the pipeline stage where the advice should be applied) is declared by the `before` element. As in Section 2, the aspect is hooked in before the webservice transformation component.

In the following section, we compare our method and prototype to existing approaches that support modeling of context-dependent adaptation in web applications.

5. Related Work

GAC: The Generic Adaptation Component (GAC) [6] is an approach to rule-based adaptation of XML documents. Although it has been discontinued, it shares a number of similarities with our approach. GAC and PX-Weave

both apply pre-defined transformation patterns to XML documents. These pre-defined patterns are instantiated and parametrized by an application developer and executed by a transformation pipeline. The execution of rules is bound to conditions that refer to a context model holding information on the user, his device and the environment. However, GAC does not include support for selecting programmatic joinpoints. Furthermore, GAC does not offer any support for ensuring validity of intermediate transformation results. Especially if there are multiple GAC rules applied, their combined effect cannot be predicted. While the author has the chance to supply an order over his rules, he is left on his own on how to determine this order and whether it results in a valid document. Additionally, our approach also aims at providing support for static analyses of aspect interactions while GAC only supports ordering of adaptation rules and analyses at run-time.

UWE: [2] propose an extension of the UWE framework, treating adaptivity in web applications as a crosscutting concern. Similar to PX-Weave, they introduce a number of transformation templates which an author can apply to parts of his web application. However, their approach is based on a specific model (UWE), in contrast to our generic XML weaver. Accordingly, the transformation templates are also targeted on a higher level, resembling Brusilovsky's adaptation techniques [3].

AspectWebML: Similar to UWE, aspectWebML [12] is an effort to add aspect-orientation to the development of web applications, namely to the prominent language WebML. However, just like UWE, aspectWebML is concerned with the model level. By introducing an additional aspect layer, it becomes possible to extend or replace parts of an existing WebML model. While this is useful in the domain of web engineering, it is not as universally applicable as our approach.

Doxpects: Doxpects are an AOP approach for XML-based web services [13], which uses XPath expressions for content-based pointcuts. A doxpect contains one ore more *request* or *response* advices to transform a web service message before sending or receiving it. Transformations are based on XML Beans, i.e., statically typed objects representing XML fragments. Weaving is realised by compiling doxpects into web service handlers. In contrast to our approach, doxpects are applied to web service messages, but not to multi-staged XML transformations. Furthermore, XML Beans are less adequate for multi-staged transformation environments, since they would be required not just for transformation input and output but also for intermediate results.

6. Conclusion

In this paper, we presented a generic approach to context-specific adaptation of XML-based web applications. The approach combines previous work on adaptation aspects and aspect-oriented software development. In comparison to plain AspectJ, our tool allows to specify adaptation aspects in a declarative and easily understandable way, not requiring authors to learn complex technologies. The approach has the potential to help web application authors to organise authoring of adaptive XML documents, however this still has to be proven in a more complex case study. Furthermore, while PX-Weave is in its early prototype stage, authoring support for adaptation aspects is still missing. In the future, this will be provided through a user interface or a feature-rich textual editor with a user-friendly syntax.

In Section 3, we introduced a novel basic terminology for adaptation aspects and adaptation advices and discussed the problem of aspect interaction for adaptation advices. It remains future work to extend the aspect analyser component of PX-Weave to achieve automatic detection of such interactions. We also need to investigate to which extent static analyses can help authors with understanding their adaptation interdependencies and finding good strategies for handling them.

Acknowledgment

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) within the project "HyperAdapt".

References

[1] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of AOSD*, pages 39–50, Charlottesville, Virginia, USA, 2009. ACM.

[2] Hubert Baumeister, Alexander Knapp, Nora Koch, and Gefei Zhang. Modelling adaptivity with aspects. In *Proceedings of ICWE*, number 3579 in LNCS, pages 406–416. Springer, 2005.

[3] Peter Brusilovsky. Adaptive hypermedia. *User Modeling and User Adapted Interaction*, 11(1-2):87–110, 2001.

[4] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE*, volume 2487 of *LNCS*, pages 173–188, London, UK, 2002. Springer-Verlag.

[5] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Static and dynamic detection of behavioral conflicts between aspects. In *Proceedings of RV*, volume 4839 of *LNCS*, pages 38–50. Springer-Verlag, 2007.

[6] Zoltán Fiala and Geert-Jan Houben. A generic transcoding tool for making web applications adaptive. In *Proceedings of CAiSE*, pages 15–20. FEUP, June 2005.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP*, volume 2072 of *LNCS*, pages 327–353, London, UK, 2001. Springer-Verlag.

[8] Gregor Kiczales, Anurag Mendhekar, John Lamping, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented programming. In *proceedings of ECOOP*, volume 1241 of *LNCS*, Finland, June 1997. Springer-Verlag.

[9] Günter Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development (Special issue 'Dependencies and Interactions with Aspects')*, LNCS, April 2006.

[10] M. Niederhausen, Z. Fiala, N. Kopcsek, and K. Meissner. Web software evolution by aspect-oriented adaptation engineering. In *Proceedings of WSE*, pages 3–7. IEEE Computer Society, Oct. 2007.

[11] Gustavo Rossi, Daniel Schwabe, and R.M. Guimaraes. Designing personalized web applications. In *Proceedings of WWW*, pages 275–284. ACM, 2001.

[12] Andrea Schauerhuber, Manuel Wimmer, Wieland Schwinger, Elisabeth Kapsammer, and Werner Retschitzegger. Aspect-oriented modeling of ubiquitous web applications: The aspectwebml approach. In *Proceddings of ECBS*, pages 569–576. IEEE Computer Society, 2007.

[13] Eric Wohlstadter and Kris De Volder. Doxpects: aspects supporting XML transformation interfaces. In *Proceedings of AOSD*, page 108. ACM, 2006.