Doxpects: Aspects Supporting XML Transformation Interfaces

Eric Wohlstadter
University of British Columbia
Vancouver, BC Canada
wohlstad@cs.ubc.ca

Kris De Volder
University of British Columbia
Vancouver, BC Canada
kdvolder@cs.ubc.ca

ABSTRACT
In the web services environment software development can involve writing both object-oriented programs and XML transformations. This can be seen in the popular Web Services architecture. In this architecture, crosscutting concerns are often manifest as transformations on XML messages; encrypting messages, adapting between schemas types or adding extra-functional elements such as transaction contexts can be seen as prime examples. Some existing middleware platforms provide support for *Handlers* where crosscutting message transformation concerns can be addressed. Although handlers localize some concerns, they do not support the sound software-engineering principle of "programming to an interface". This prevents a clean mapping from design to implementation and inhibits useful static checking which could take advantage of a well specified interface. To address this and similar design challenges, we have developed Doxpects, which solve many problems with the existing handler approach to implementing these new crosscutting concerns. We describe an AOP abstraction called the *content-based pointcut* which integrates support for XML transformation to enable implementation of crosscutting concerns with good modularity properties. We present examples based on XML encryption and service interoperability implemented on top of the Apache Axis Web Services middleware.

1. INTRODUCTION
In the web services environment software development can involve writing both object-oriented programs and XML transformations. XML-based protocols such as SOAP provide the substrate on which Web Services are built. Now, programmers are often confronted with mitigating conceptual mismatches between two different primary decompositions in their code: the object-oriented design and the XML document structure. As pointed out in the literature this causes frustration [2,11].

In a Web Service implementation many concerns tend to *crosscut* the document-oriented decomposition. As one particular example, a programmer may be concerned with protecting sensitive credit card information which is used in several operations of a web service. The programmer might want to protect just this element *no matter where it appeared* in *any* web service message, even if it wasn't explicitly given as a method argument (this is called *element-wise encryption* [27]).

The existing approach to modularizing document-oriented concerns in Web Services is through the use of specialized APIs. For example, the Java XML Remote Procedure Call (JAX-RPC)[1] standard allows *Handlers* to be installed into the flow of XML message processing. These handlers may intercept and also transform messages. Although handlers localize some concerns, they do not support the sound software-engineering principle of "programming to an interface". They provide only a very coarse-grained interface to applications, making the design of concerns difficult to map to implementation and preventing useful static checking by a language compiler (e.g. checking whether two advice may interact [20]).

```
interface Handler {
 boolean handleRequest(MessageContext mc);
 boolean handleResponse(MessageContext mc);
 boolean handleFault(MessageContext mc);
}
```

Figure 1 : The JAX-RPC Handler interface.

In Figure 1, we see the `Handler` interface which is implemented by a handler implementation. Each handler exports the same interface with one method to intercept incoming messages, one method to intercept outgoing messages, and one method to intercept exceptions (other infrastructure methods not shown). Each method takes an argument which gives access to the message content in a standard dynamically typed format called the Document Object Model (DOM). Although this allows programmers to localize crosscutting XML transformation concerns, we argue it does not adequately address many of the other criteria for good modularity, as described by Kiczales and Mezini [10], including: "a well-defined interface that describes how it interacts with the rest of the system" and "an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules". We propose to bring these properties to modules which implement crosscutting document-oriented concerns.

Unlike traditional distributed object middleware, web services middleware uses a high-level XML representation of messages. In this sense, web services middleware goes beyond traditional middleware, which has a much more static, fixed view of message structure. Features such as element-wise encryption (section 2.1) or schema transformation (section 2.2) simply cannot be handled by traditional OO middleware. It is not possible to implement these types of features using

[1] JAX-RPC: http://java.sun.com/webservices/jaxrpc/index.jsp.

traditional distributed object meta-programming [16,21] (Interceptors) and would require changes to the lowest layers of standard middleware. Aspect-oriented middleware, such as JBoss [8] or our own earlier work on DADO[23], is firmly ensconced in the world of fixed message structures, and does not address, for example, the aforementioned encryption feature. We have therefore developed a new aspect-oriented programming model, called Doxpects, which specially deals with crosscutting features in the world of dynamic, distributed, flexible applications enabled by web services middleware.

In this paper, we show that the implementation of aspects to address document-oriented concerns is improved by first-class support for pointcuts defined on document content. *Content-based pointcuts* provide a statically-declared fine-grained interface between aspects and applications. These pointcuts build on XPath to express properties of document structures which define when aspects are activated. Our implementation integrates XPath with XMLBeans [26] (an XML to OO mapping) to provide a natural Java based programming model for document-oriented aspects. Transformations are achieved simply by replacing some bean instances with others. In this paper we focus specifically on the use of content-based pointcuts in the Web Services setting.

The contribution of this paper is the formulation of an aspect language for XML transformations which promotes this fine–grained interface between aspects and applications. The interface is expressed through the explicit binding between content-based pointcuts and AOP style advice. The interface provides the following concrete benefits:

1. Enhanced traceability of design level requirements to implementation (Section 2).

2. Statically checked XML to Java argument binding from content-based pointcuts to XMLBeans (Section 4.2.2).

3. *Replacement types* provide a simple model for transformation, by replacement of bean instances (Section 4.3.1).

4. Certain advice interactions are discovered statically (Section 4.3.2).

This paper is organized as follows: in Section 2 we further motivate the problem space with an example, in Section 3 we describe background and delve into details in Section 4; in Section 5 we continue with a further example, in Section 6 we present related work and conclude in Section 7.

2. MOTIVATING EXAMPLE

Consider a component of a parcel tracking and shipping dispatch service as presented in Figure 2. This service would be used by various business partners that took part in the shipment process and would also be used from some brick-and-mortar outlets to schedule shipments. In this paper we consider two different aspects which may be needed as part of message processing between these partners.

2.1 Encryption Aspect

Deploying services in an open environment such as the web requires careful consideration for the implementation of crosscutting non-functional concerns such as security. Here we are concerned with protecting some sensitive pieces of documents exchanged between the shipping application web service partners. Using *element-wise encryption*, documents are protected from both malicious tampering and unauthorized disclosure. Specific components (called *elements* in XML) of a message may be hidden, instead of or in addition to, the entire document. This feature is especially useful when messages may traverse several intermediate services. An intermediate service can make use of the unencrypted information for routing, logging, access control, etc. before the message arrives at the service endpoint that actually needs to use the sensitive information.

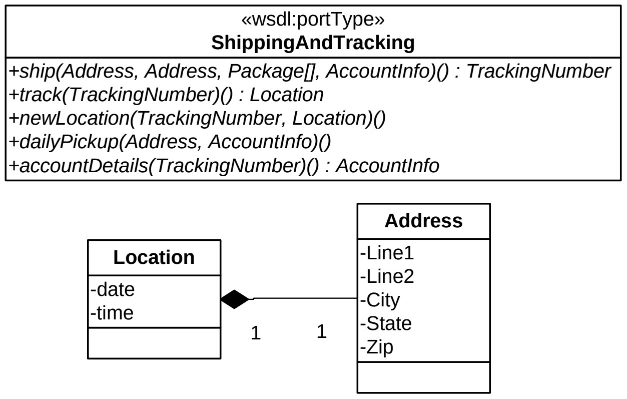
«wsdl:portType»
ShippingAndTracking
+ship(Address, Address, Package[], AccountInfo)() : TrackingNumber
+track(TrackingNumber)() : Location
+newLocation(TrackingNumber, Location)()
+dailyPickup(Address, AccountInfo)()
+accountDetails(TrackingNumber)() : AccountInfo
Location
-date
-time
1
1
Address
-Line1
-Line2
-City
-State
-Zip

Figure 2: UML Diagram mock-up showing part of the original shipping and tracking service. Actual design is specified in WSDL and XML Schema.

The implementation of software to address this concern requires additional functions to be executed *along with* the original web service functions. Three different pieces of behavior are required. First, a function is required which provides decryption of incoming document elements. Second, a function is required which provides encryption of sensitive document elements. For example, we may want to encrypt just the `PaymentInfo` information (definition not shown) which is `part-of` the `AccountInfo` type (for example in the `ship` operation). Finally, a function is necessary to set up certain state, called a *security context*, to properly parameterize the encryption of messages to the client. *Different behavior* will be appropriate at *different times* and the behavior to execute *depends upon the content* of the messages being exchanged.

If we are able to implement each function as a different advice and use pointcuts to identify when each advice is applicable, then these high-level design requirements can seamlessly be captured in our implementation. Still, there are some issues which could easily be overlooked. How can our programming model provide an equally seamless integration between XML transformation concerns and AOP? In this particular example there are other more subtle implementation issues. For example, it may be natural to identify encrypted messages using a pointcut style abstraction and use advice to implement decryption. But, what if there are encrypted elements inside of encrypted elements (called *super-encryption* [27])? For example, a client may encrypt all the `PaymentInfo` types to ensure credit card information is only available to services with which it has established trust. However, later the *entire message* might be encrypted at a firewall. Will the decryption advice be activated just once or should decryption advice be managed *recursively* in this case? Using a language based approach we are able to warn programmers about these subtle *advice interactions* (this particular interaction is addressed in Section 5).

2.2 Interoperability Aspect

Now, consider an interoperability example inspired by [19]. To bridge syntactic mismatches between otherwise semantically compatible clients and web services, an adapter can be written to transform SOAP requests. This type of packaging mismatch [6] is common whenever interfaces are evolved independently [17] by disparate organizations.

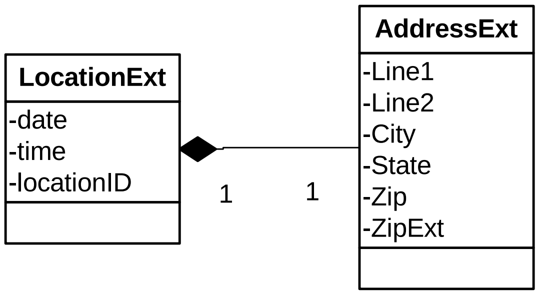

Figure 3 : UML Diagram mock-up showing part of the evolved shipping and tracking service types.

Looking at Figure 2, as part of the XML data comprising the application, a `Location` is composed of an `Address` where a package was processed and also the time and date of processing. At a high-level there are two transformation concerns to be addressed to ensure interoperability which should each be traceable to implementation. First, by comparing Figures 2 and 3 we need to transform *all* the `Address` data types to a new format, `AddressExt;` looking at Figure 2, `Address` appears as an argument in more than one operation. The upgraded version includes a separate element for two postal code (so-called Zip code) components. Second, we need to transform all the `Location` information types to include a `locationID`. This information was added to easily identify package handling and outlet locations. By implementing both of these transformations as advice (which we refer to as the `Location` and `Address` advice) in a single aspect we are able to map both concerns into implementation, making the code look like the design we are trying to capture.

Although, on further inspection we see that the `Address` type is `part-of` the `Location` type (or equivalently, `Location has-a Address`). So, the `Location` advice requires that the `Address` advice be executed as part of the overall `Location` transformation. This makes it *dependent* on the other advice, another form of advice interaction (addressed in Section 4.3.3.2). By using a language based approach we can perform automatic checking in the compiler to ensure a proper nesting of advice.

2.3 Discussion

Considering the encryption example, the current approach to encryption/decryption would be based on handlers. Using, the Apache Axis[2] middleware, these handlers can be installed on a per-service basis (or globally). Handlers do not support specific, explicit deployment of a feature for specific document contents. Thus, handlers do not allow the programmer to clearly commit to a document-based interface between the handler and the application. This a) reduces design transparency b) precludes any type of static checking and c) inhibits maintainability.

The design and implementation of an aspect-like way to express the same concern would be improved in the following ways. First, decomposition would be based on advice associated with pointcuts which expressed crosscutting properties of program execution, not limited to service deployment, and enriched with the expression of document content properties. Second, information bound from the pointcuts would be made available through a statically checked interface and would automatically map XML based content into their Java equivalents. Finally, declaring the types of elements to be replaced and the types of the replacement elements in the advice signature (we call these replacement types) allows lightweight checking for interaction between advice.

3. BACKGROUND

The collection of supporting XML technologies for Java and Web service programming continues to expand rapidly. Here we begin by reviewing the existing technologies on top of which our solution is built.

3.1 Web Services

Nowhere is XML message processing more prevalent than in the area of Web Services, built on SOAP (Simple Object Access Protocol) messages. Programmers describe the interface of a service using WSDL (Web Service Definition Language) specifications. These specifications may reference XML Schemas which specify the types of XML data elements passed to and from services. SOAP provides a standard envelope in which these custom messages are transported. We have implemented Doxpects in the context of the Apache Axis Web Service middleware which supports the JAX-RPC standard.

3.2 XPath

XPath is a domain specific language used primarily to drive traversal over document structures. At its core, an XPath expression consists of a list of element names (separated by "/") specifying a path from the root document element to

[2] Apache Axis: http://ws.apache.org/axis/.

```
[1] doxpect ShippingInterop
[2] {
[3]    private static final LOCATION_ID = 13546343;
[4]
[5]    request(Location->LocationExt location, AddressExt<-Address addr) :
[6]     body(/newLocation/Location, location) && body(/newLocation/Location/Address, addr)
[7]    {
[8]           locationPrime.setLine1(location.getLine1());
[9]           /*Elided, similar to above, copy over identical fields*/
[10]          locationPrime.setLocationID(LOCATION_ID); /*Fill in missing locationID*/
[11]          locationPrime.setAddress(addr); /*addr is of type AddressExt*/
[12]   }
[13]
[14]   request each(Address->AddressExt address) :
[15]     body(//Address, address)
[16]   {
[17]          String[] zipParts = address.getZip().split("-");
[18]          addressPrime.setZip(zipParts[0]);
[19]          addressPrime.setZipExt(zipParts[1]);
[20]          /*Elided, copy over identical fields*/
[21]   }
[22] }
```

Figure 4 : Doxpect implementation for Interoperability example. XMLBean types shown in bold. XPath shown in italics. Advice for reverse transforming incoming content not shown.

specific elements of interest. Two special operators are available to quantify over multiple possible elements, these are the "*" and "//" operators. A "*" fills in as a wildcard for any possible document element while the "//" fills in as a wildcard for any possible document sub-tree (essentially a closure over the "*" operator). We use XPath in a novel way, to separate the specification of *when* documents are processed and *what* the function does which processes the documents.

3.3 XMLBeans

As described above, manipulating documents using the dynamically typed DOM representation has distinct disadvantages including decreased readability of code and no support for static type-checking. To deal with these problems, a number of static XML to object structure mappings have been proposed, including XMLBeans. In our programming model, document elements which are captured in a content-based pointcut are automatically made available to programmers using a statically typed XMLBean representation. XMLBean data is accessed or mutated using the standard "get" and "set" JavaBean pattern. This feature provides the bridge necessary where pointcuts are expressed in one language (XPath) and behavior is expressed in another language, Java. Another bean, called the *replacement*, is made automatically available to advice so that transformation is achieved simply by filling in the values of the new replacement bean.

4. DETAILS

Here we present the details of our Doxpect language. To ground the discussion we showcase concepts as applied in the implementation of the `ShippingInterop` doxpect as discussed in Section 2.2.

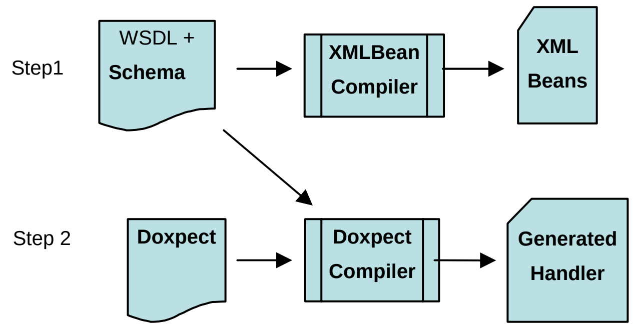

Figure 5: Doxpect Tool Process (From left to right, programming artifacts, tools, binary output).

4.1 Doxpects

A programmer who needs to address a crosscutting XML transformation concern can write a doxpect as in Figure 4, line 1. An doxpect can include pointcuts, advice, fields and methods just like a standard AspectJ aspect. However, the pointcuts we use are different (Section 4.2) and the semantics of executing advice (Section 4.3) is tailored for the special case of XML transformations. Here we see the `ShippingInterop` doxpect consisting of two advice (lines 5 and 14) for transforming outgoing content and one field (line 3) used to fill in for the missing `locationID`.

As shown in figure 5, the development of a doxpect begins in step 1 by compiling the WSDL service definitions and the XML schema types referenced in the service interface. In our example, this would include the definitions for both the original and the evolved shipping services. The definitions are used by the off-the-shelf XMLBean compiler to generate equivalent Java class definitions. Next, in step 2, a programmer writes a doxpect which may crosscut some of the service and schema definitions. The Doxpect compiler performs three tasks. First, it ensures type compatibility of variables matched by content-based pointcuts and generates the code to deserialize message fragments into equivalent XMLBeans. Second, it performs interaction checking to ensure that doxpect advice do not conflict. Third, it generates pointcut matching and advice dispatching code in a standard JAX-RPC handler which can now be deployed safely.

4.2 Doxpect Pointcuts

We expand upon the standard AspectJ joinpoint model by including first class support for document content properties. Here we describe the semantics in terms of AspectJs joinpoint model.

4.2.1 Joinpoint Semantics

AspectJ provides support for specifying the execution of advice in terms of properties on joinpoints. During program execution different joinpoints are executed in sequence. Pointcuts specify properties of joinpoints so that when a joinpoint matching that property is encountered any associated advice will be executed. AspectJ provides extensive support for the specification of program execution properties such as field access, method calls, and control-flow. AspectJ is not limited however to the expression of these "program execution" properties. Using the `if` pointcut, advice may be predicated upon the *data values* of variables captured in the pointcut. The `if` pointcut can include any Java code, using these variables, which evaluates to a boolean expression. Our model is similar to using the `if` pointcut, capturing a Java representation of some XML document of interest, and then predicating on its content. Although that approach provides the right semantics for our intended model, the cross-language nature of XML and Java programming demands a specially tailored approach. Essentially, AspectJ's `if` pointcuts do not respect the specification of XML documents as described in their schema. This prevents a satisfactory integration providing our desired benefits.

4.2.2 Content-Based Pointcuts

The content-based pointcut provides the programmer a convenient way to express properties of joinpoints where SOAP messages are being processed. A property of a message is expressed in the pointcut using XPath expressions which may be joined together using the standard pointcut logical operators. An example can be seen in Figure 4, line 6. Here we want to match the `Location` data type *and* the associated `Address` which are passed in a call to the `newLocation` web service operation. Another example is shown on line 15. This pointcut matches all `Address` types as part of any web service operation. Notice that both advice will match an `Address` in the `newLocation` operation. In section 4.3.1, we explain how the first advice expresses a dependency that another advice will perform transformation on the `Address` element the first advice matches.

XPath wildcard quantifiers provide the means to express properties which are *structure shy*[13,14]; enabling transformation to be performed across many different service operations. Additionally, only certain pieces of those messages may be of interest to the advice. This allows a concise means to implement those aspects which crosscut the WSDL service and schema type definitions.

As a convenience two pointcut designators are available: `header` and `body`, which predicate on properties of either the SOAP message header or body. The SOAP message header is often used to carry invocation context information to be processed by middleware components implementing non-functional concerns. For example in the encryption example, the header carries a shared-key identifier.

The SOAP message body includes the data which is passed in the message including the name of a Web service operation to be invoked and the XML data for each of the arguments. Doxpects may to need to monitor or transform both the header and body elements across different web service invocations.

A distinct advantage of the content-based pointcut is that elements matched in the XPath expression can be made directly available to doxpect advice, providing a fine-grained interface between documents and doxpects. Each pointcut designator can include the name of an argument from the advice signature which binds the result of XPath evaluation to that argument. Naturally, pointcuts joined using the OR operator (not shown) are all required to have bindings for the same arguments. Arguments are made available automatically to advice as equivalent XMLBeans through generated code which deserializes the result of XPath evaluation. For example, on line 14 an `Address` in any message is made available as the variable `address` on line 15 in the advice signature.

4.3 Doxpect Advice

Doxpect advice provides a strongly typed environment in which XML transformations take place. This helps to ensures that the implementation adheres to the contract of the interface. In this section we discuss our model for transformation using XMLBean replacement, static checking for advice interactions, and the semantics for Doxpect advice execution.

Currently, each advice is executed `before` the sending or receiving of a particular asynchronous SOAP message (i.e. synchronous invocations are not treated specially). This is implemented by a programmer using either the `request` or `response` advice forms which replace the familiar `before`, `after`, and `around` advice forms.

XML transformations are particularly sensitive to the order in which advice are executed because each advice is allowed to mutate the current message. As in AspectJ, advice in a single doxpect execute in the order they appear textually. We say an advice which is declared before another advice is *dominating* and the second advice is *dominated*. We provide special compiler support (Section 4.3.3.) to help ensure that the programmer specified order makes sense.

4.3.1 Replacement Types

Although doxpect advice may behave just like standard AspectJ advice by executing extra code, doxpect advice are specially suited to perform transformation over captured document content. Each argument captured in the advice signature can also include a *replacement type.* Three kinds of replacement types are the *provided*, *optional*, and *required* replacement.

A provided replacement specifies a contract that an element *matched* by a pointcut will be *replaced* by another element of a different type in the advice. For the type of the argument a programmer includes two XMLBean types, separated by an arrow "→", called the *matched type* and *replacement type* respectively. For example, in Figure 4 , line 5 we see that `Location` will be replaced by `LocationExt` (*note:* the second argument with the arrow reversed is explained below). The advice interface along with the pointcut tells us not only how the types will be converted but also, from the pointcut, where in the document this will take place.

A special variable representing the replacement is made available in the scope of the advice. This variable is given the same name as the matched argument with the suffix "`Prime`". For example, on line 8, the `Line1` field from the matched `location` argument is copied over to the `locationPrime` replacement. Programmers can fill in the data of the special variable and the transformation is automatically executed on the underlying document when the advice exits.

An *optional replacement* provides similar semantics except transformation does not occur if the special variable is set to `null`. In this case, the replacement mechanism provides a convenience to the programmer, but is not a contract. This is specified syntactically by including a question mark after the matched and replaced type. Optional replacements are used when the advice must make a run-time decision whether transformation will occur (demonstrated in Section 5). They cannot be used to *provide* a replacement for a *required* replacement type, discussed next.

Required replacements are used to delegate transformation of specific elements to other advice which provides transformation. The semantics can be viewed as the execution of a `proceed` statement where some guarantee is made on the new structure of the document when the executing `proceed` has returned. This expresses a dependency from a dominating advice to some advice it dominates. The syntax is a reverse of the provided replacement. Here, only the named argument variable (matching the type on the left side of "←") is made available in the context of the advice. For example, on line 5 a variable named `addr` of type `AddressExt` is made available. The value of this variable is the result of transformation by another advice. The element to transform is expressed by the pointcut and matches the type of the right hand side of the "←". So, in the example the value of `addr` is derived from an element of type `Address` which is delegated to and transformed by some other advice. An analysis of the pointcuts is used to match up required and provided replacements, so dependent advice do not need to explicitly refer to advice providing replacement, but the compiler statically ensures that one exists. Further motivation for the use of required and provided replacements and this matching is explained in the section on *advice inhibition* (Section 4.3.2.2).

4.3.2 Advice Interaction

Sometimes different advice will match the same document, however since advice can mutate the document, this could interfere with the other advice. Programmers will need to be sensitive to the order that advice executes. We have identified three types of content-based advice interactions which we call *corruption*, *inhibition*, and *activation.* Each explanation is given by describing a particular use of replacement types in a pair of dominating and dominated advice that interact. Note that we have currently only implemented interaction checking in the context of a single aspect and not yet interaction between advice across aspects, although we feel that an extension will be straightforward.

4.3.2.1 Advice Corruption

In this situation a dominating advice replaces an element type, `A`, with another `B` (i.e. `A`→`B`). The dominated advice matches some type, `C` , which contains as one of its parts an element of type `A` (i.e. `C has-a A`, transitively). Now, when the dominated advice executes, `C` is no longer properly typed (because it should not contain an element of type `B`) and cannot be properly represented as a static Java type. When the two advice match the same document, this is always an error and it is straightforward to detect when this may occur (simply by using the rules above). Since the pointcuts of two advice might never match the same document, our analysis is overly conservative because it uses only type information in detection. This is achieved by a traversal of the WSDL and XML Schema definitions.

An example of corruption would occur in the `ShippingInterop` example if the two advice were switched. If all `Address` were replaced with `AddressExt`, then a match of `Location` would create a type error. We provide a type-safe way for advice to cooperate using required and provided replacements which are discussed next in the context of advice inhibition.

4.3.2.2 Advice Inhibition

In this situation a dominating advice replaces an element type, `A`, with another, `B`, (i.e. `A`→`B`) so that a dominated advice is prevented from matching `A` or one of `A`'s parts. The doxpect compiler warns programmers in case this behavior is an error.

This exact situation could occur in the `ShippingInterop` aspect. When `Location` is replaced by `LocationExt`, the second advice will be prevented from matching the `Address` type that is part of `Location`. If we switched the order of the advice, we would be back in the corruption situation.

Now, the problem solved by our use of required and provided replacements should become apparent. In order to allow this cooperation of advice in a type-safe, statically

```
doxpect EncryptionDecryption per(request) /*Server-side implementation*/
{
   Identifier contextID;

   request(Identifier id) : header(/SecureConversation/Identifier, id)
   {
          contextID = id; /*Set request scoped security context identifier*/
   }

   request each*(EncryptedData->XmlObject? encrypted): body(//EncryptedData, encrypted)
   {
      byte[] cipherValue = encrypted.getCipherData().getCipherValue();  /*Get raw data*/
      String keyName = encrypted.getKeyInfo().getKeyName();               /*Lookup key*/
      Key key = KeyStore.getInstance("JKS").getKey(keyName, Constants.PASSWORD));
      if(key != null) {                                  /*Check if key exists*/
         encryptedPrime = decrypt(cipherValue,key);  /*Decrypt data using key*/
      } else
         encryptedPrime = null; /*If not a known key, abort transformation*/
   }

   response(PaymentInfo->EncryptedData? payment) : body(//PaymentInfo, payment)
   {
          /*Lookup key associated with the security context*/
          String keyName = SecurityContext.getKeyName(contextID);
          Key key = KeyStore.getInstance("JKS").getKey(keyName, Constants.PASSWORD);
          if(key != null) {
                 /*If key is valid, fill in replacement with encrypted data*/
                 paymentPrime.getCipherData().setCipherValue(encrypt(payment,key));
          } else {
                 paymentPrime = null; /*Abort transformation*/
                 throw new SOAPFaultException(...);
          }
   }
}
```

Figure 6 : Doxpect implementation for Encryption/Decryption example. XMLBean types shown in bold. XPath shown in italics. Library functions and encrypt/decrypt helper functions not shown

checked, manner the first advice can use a required replacement. The first advice will delegate an element matched by its pointcut to be transformed by some other dominated advice. The result of the transformation is then made available to the first advice. To ensure that a dominated advice is always available to provide the transformation we perform two checks. First, some dominated advice must contain a provided type with exactly the reverse signature of the required type. Second, we must ensure that the pointcut for the provided type always matches in the context of the dominating advice. This is achieved by checking *containment* of XPath expressions.

One XPath expression, A, is said to contain another, B, if, for all documents the set of elements matched by A is always a superset of the elements matched by B. The doxpect compiler checks for containment of XPath expressions to ensure that there is always a match for a required replacment[3]. For example, in Figure 4, the pointcut on line 15 *contains* the pointcut on line 6, so it can safely provide the required replacement (we can be sure it will execute at all the joinpoints of the dominating advice).

By providing a checked mechanism for advice to cooperate, doxpect programmers are free to decompose pointcuts and advice directly to high-level transformation concerns.

4.3.2.3 Inter(Intra)-Advice Missed Activation

In this situation a dominating advice matches some element, $\mathcal{B}$, but that element doesn't appear until after the advice has finished executing. Perhaps another dominated advice transforms A to B (A→B) subsequently. We would like to warn the programmer in case she would like to invert the ordering. Otherwise, an advice may not match when it should have since it executed before certain content appeared.

[3] Checking of containment is achieved by a call to an external tool called `xviz` [9].

An interesting situation arises when a single advice performs a transformation which causes the same advice to again become active. We can detect this by checking whether the replacement type of any argument in the advice signature is part-of (transitively) a matched type in the same advice signature. This situation arises in the context of the encryption example (when using super-encryption) which we will elaborate in Section 5.

4.3.3 Advice Execution

Using content-based pointcuts a single pointcut may match a single joinpoint in more than one way (i.e. it may match multiple places in a document). For example, one example pointcut matches an `Address` element anywhere in the document. From our service definition, we know that `Address` might appear in multiple argument positions or as part of a `Location`. By default, if multiple matches are possible (determined by the service and schema definitions) any matched argument is required to be of an array type, to hold all matches. An optional advice modifier called `each` (shown in Figure 4, line 14), provides a simple means of iterating over all such matches automatically.

Although this phenomenon may seem peculiar and limited to our content-based scenario, a similar issue arises in standard aspect languages such as AspectJ. Consider a programmer who wants to write an aspect which converts all Strings passed to a particular API to a specific format. One might write the syntactically correct pointcut:

```
around(String x):
    call(* apiName.*(.., String, ..)) &&
    args(.., x, ..)
```

Here the semantics are similar but the AspectJ compiler will issue an error about a compiler limitation because it can't match argument ``x" in more than one context. Since this scenario is more common in a document-oriented decomposition we provided special support through array types and the `each` advice modifier.

5. Encryption Revisited

Here we expand on the encryption doxpect presented in Figure 6. This example further demonstrates the use of advice interaction checking and optional replacements in the context of the encryption crosscutting document concern.

We have refactored most of the actual encryption and decryption functionality into helper functions (not shown) in order to highlight the novel features of our language. We define a doxpect called `ElementWiseEncryption` on line 1. This doxpect is declared to be `per(request)`which provides a mechanism for `request` and `response` advice to share state. Each doxpect instance will be used to store the security context identifier information for one request, as defined by the field in line 3. This is achieved on lines 5-8; an advice is declared which picks out messages containing a security context element and stores the information in the field. This is an element standardized by the WS-SecureConversation [22] specification. It provides the server a way to identify a shared-key to use in encrypting messages back to the client. This doxpect applies only to the server-side of message exchange because the semantics for the security context will be different on the client.

Two more advice are declared on lines 10 and 21. On line 10 an advice is used to pick out messages with any encrypted elements tagged by the `EncryptedData` element. This is a standardized element defined by the XML Encryption [27] specification. First, on line 12 the XMLBean named `encrypted` which was captured by the advice is used to access the raw encrypted bytes (called a `CipherValue`). Next, on line 13 the name of the key for the encrypted data is retrieved. Recall, that encrypted elements may be destined for services other than the current service processing the message. So, on lines 14 and 15, we check to see if the key is known to our system. If it is (line 16) we fill in the replacement bean, `encryptedPrime`, with the decrypted element through a call to the helper function `decrypt` passing in the encrypted bytes and the key.

Unfortunately, the advice implementation is not so simple. Recall, the advice *missed activation* interaction described in Section (4.3.2.3): after an advice has completed execution, it may match again at the same joinpoint. As described, this is checked for statically, and can even occur within a single advice. Considering, the optional replacement on line 10, `EncryptedData→XmlObject?`. Here `XmlObject` is the root of the XMLBean hierarchy so the replacement may actual be (or contain) an `EncryptedData` element. We would like programmers be able to mitigate these types of interactions easily rather than requiring them to program an explicit traversal of the document. This is achieved on line 10, using our `each*` advice modifier. Using `each*` an advice is executed recursively on the output replacement elements until it fails to match. Now, these types of advice interactions can be checked for statically and resolved through simple domain-specific abstractions.

On line 23 we see the advice used to encrypt outgoing elements. The pointcut picks outgoing messages (i.e. return values) with a `PaymentInfo` element. Looking back at Figure 1, this actually only occurs in the `accountDetails` operation since `PaymentInfo` is `part-of` `AccountInfo`. A client-side implementation of the encryption/decryption concern would match `PaymentInfo` in multiple places of different message types. Also, this pointcut should really be defined as abstract to allow deployment specialists to specialize the doxpect for different applications. We are currently still implementing support for abstract pointcuts.

The encryption work is done in lines 23-31 and follows similarly to the decryption advice. An important difference is shown on line 24. Here the advice can use `contextID` bean which was saved from the incoming request. This shows how doxpects are useful in implementing coordinated transformations across multiple XML messages.

6. RELATED WORK

6.1 XML Transformation Languages

The standard language for XML transformation is the eXtensible Stylesheet Language Transformation (XSLT). Similar to Doxpect, XSLT provides support for modular transformations through the use of the `template`. Templates encapsulate only certain transformation sub-tasks and delegate processing of other tasks using the `apply-templates` operator. Unlike, Doxpects, XSLTs transform XML documents into any type of content (e.g. LateX or plaintext). So, there is no way to express the interface of a template in terms of its output.

Using replacements types we are able to perform useful static checking to warn against certain common transformation ordering mistakes and also to ensure replacements are provided when required. As a pragmatic benefit, Doxpects integrate more naturally with a standard OO language to provide management of state (through fields) and easier access to standard Java libraries (such as encryption). XSLT programmers are required to learn an entirely new purely functional programming language.

One design choice we made was to allow each advice to individually mutate each message. This required support to ensure advice did not accidentally interact. To avoid such problems, the XSLT specification originally did not allow the manipulation of transformation output. However, this made programming many transformations extremely cumbersome [8] and so support was provided in most popular XSLT processors (using something called the `node-set` operator). Following this demand we chose to allow mutation on document instances.

Several other projects focus on statically guaranteeing that the implementation of an XML transformation with an input document from one schema type always produces a document conforming to another specific schema type. A variety of approaches have been taken including type-inference [2] and whole program data-flow analysis [11]. Although the checking they do is more sophisticated than in Doxpect, they make no contribution to the design, traceability, and maintenance of crosscutting concerns. All information about the transformation is simply part of the implementation. We provide an explicit, lightweight approach by promoting a transformation interface.

6.2 Aspects and XPath

Previous work [3,5] has leveraged XPath as a pointcut designator for adapting business process workflow descriptions which are written in XML. This work is closer to standard AOP, where pointcuts match program execution events and not data content. The BAT project [7] uses XQuery, a functional programming language built on XPath, to implement pointcuts on an XML representation of Java byte-code. We have proposed using XPath to match properties of actual XML message instances and not descriptions of program structures. This requires a completely different solution to deal with messages which are transformed at run-time.

6.3 Middleware Meta-programming

Middleware platforms have a long tradition of enabling flexible customizations [4] in various middleware layers. The use of the Interceptor [16,21] became popular as a way to capture a reflective representation of base program execution. Similar to the Handler approach, these approaches are completely generic and do not promote an interface which is typed in the environment of the base program. In our work, we have followed a model closer to AspectJ where each advice specifies a narrow interface using a concrete syntax at the level of the base program and not at the meta-level. This allows programmers to express crosscutting properties explicitly and more naturally.

6.4 Web Service Middleware

Our shipping interoperability doxpect is inspired by previous work on service interoperability in Web Services. In an on-line article, Provost [19] implemented a schema transformation with a similar motivation to the example we presented using XSLT.

Ponnekanti [17] presented a taxonomy of web service interface mismatches that can occur when interfaces are allowed to evolve independently as well as an analysis to discover mismatches using WSDL.

In previous work, we extended the handler mechanism of Apache Axis with support for dynamically negotiated policies. At run-time clients and servers would decide which handlers should be activated. However, in that work we made no contribution to the actual programming of handlers. A similar approach is taken by [1]. WSS4J [25] is a publicly available library of handlers for web services security, but not does make any contributions to the design of handlers.

7. CONCLUSION

We have explored the concept of modularity in crosscutting document concerns, provided an approach using content-based pointcuts and demonstrated some illustrative examples. Our research was motivated to address new concerns which appear uniquely in the web service setting. The previous approach using handlers inadequately addressed important software engineering practices such as “programming to an interface”.

In the future our Doxpect language could be integrated with AspectJ. Currently, Doxpect is implemented separately as a source to source translator on top of the Apache Axis Web Service middleware without the full power of AspectJ pointcuts or inter-type declarations.

Our experience with Doxpects has been primarily example driven. We have focused our attention on building those pieces of the language which addressed the programming difficulties encountered for those examples. We believe the current version to be suitable for addressing a wide variety of concerns encountered in the web services area which we intend to investigate such as reliable messaging, batch processing, caching, and transactions.

8. REFERENCES

[1] Baligand, F. and Monfort, V. A Concrete Solution for Web Services Adaptability using Policies and Aspects. In *Proc. of the International Conference on Service-oriented Computing*, 2004.

[2] Benzaken, V., Castagna, G., and Frisch, A. CDuce: An XML-Centric General Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[3] Charfi, A. and Mezini, M. Aspect-Oriented Web Service Composition with AO4BPEL. In *Proc. of the European Conference on Web Services,* 2004.

[4] Clarke, M., Blair, G., Coulson, G. and Parlavantzas, N. An efficient component model for the construction of adaptive middleware. In *Proc. of the International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[5] Courbis, C. and Finkelstein, A. Towards Aspect Weaving Applications. In *Proc. of the International Conference on Software Engineering*, 2005.

[6] De Line, R. Avoid packaging mismatch with flexible packaging. In *Proc. of the International Conference on Software Engineering*, 1999.

[7] Eichberg, M., Mezini, M, and Ostermann, K. Pointcuts as functional queries. In *Proc. of the Asian Symposium on Programming Languages and Systems*, 2004.

[8] Fleury, M. and Reverbel, F. The JBoss Extensible Server. In *Proc. of the International Middleware Conference*, 2003.

[9] Handy, B. and Suciu, D. XViz: a tool for visualizing XPath expressions. In *Proc. of the XML database symposium*, 2003.

[10] Kiczales, G. and Mezini, M. Aspect-Oriented Programming and Modular Reasoning. In *Proc. of the International Conference on Software Engineering*, 2005.

[11] Kirkegaard, C., Moller, A. and Scwartzback, M. Static Analysis of XML Transformations in Java. IEEE Transactions on Software Engineering.

[12] Kosek, Jirka. Understanding the node-set() Function. <www.xml.com/pub/a/2003/07/16/nodeset.html>

[13] Laemmel, R. and Visser, E. Strategic Programming Meets Adaptive Programming. In *Proc. of the International Conference on Aspect-Oriented Programming*, 2003.

[14] Lieberherr, K., Patt-Shamir, B. and Orleans, D. Traversals of object structures: Specification and Efficient Implementation. ACM Transactions on Programming Languages and Systems, 2004.

[15] Miklau, G. and Suciu, D. Containment and equivalence for a fragment of XPath. Journal of the ACM, 51:1, 2004.

[16] Narasimhan, P. Moser, L.E. and Melliar-Smith, P.M. Using interceptors to enhance CORBA. IEEE Computer, 32(7): 62-68, 1999.

[17] Ponnekanti, S. and Fox, A. Interoperability among Independently Evolving Web Services. In *Proc. of the International Conference on Middleware*, 2004.

[18] Popovici, A. Gross, T. Alonso, G. Dynamic Weaving for Aspect Oriented Programming. *In Proc. of the International Conference on Aspect-Oriented Software Development*, 2002.

[19] Provost, Will. Integrating Services with XSLT. O'Reilly WebServices.XML. <http://webservices.xml.com/pub/a/ws/2003/09/30/integrating.html>.

[20] Rinard, M., Salcianu, A., Bugrara, S. A Classification System and Analysis for Aspect-Oriented Programs. In *Proc. of the Symposium on Foundations of Software Engineering*, 2004.

[21] Wang, N., Parameswaran, K., and Schmidt, D. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware.

[22] Web Services Secure Conversation Language. 2005. <http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf>.

[23] Wohlstadter E., Jackson S., and Devanbu P. DADO: Enhancing Middleware to Support Cross-Cutting Features in Distributed, Heterogeneous Systems. In *Proc. of the International Conference of Software Engineering*, 2003.

[24] Wohlstadter, E. Tai, S. Mikalsen, T. Rouvellou, I. and Devanbu, P. GlueQoS: Middleware to sweeten quality services policy interactions. In *Proc. of the International Conference on Software Engineeering*, 2004.

[25] WSS4J <http://ws.apache.org/wss4j/>.

[26] XMLBeans. Version 2. <http://xmlbeans.apache.org/>

[27] XML Encryption. W3C. 2001. <http://www.w3.org/Encryption/2001/>