

Doxpects: XML Transformation Aspects

by

Ivan Minevskiy

B.Sc. Hon., University of British Columbia, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

August 2006

© Ivan Minevskiy, 2006

Abstract

Web services in the context of an Enterprise Service Bus provide an architectural approach to decomposing IT solutions into reusable pieces readily responsive to changing business demands. Developing a web service involves working with two different primary decompositions: the object-oriented design and the XML document structure. XML message transformations, such as encryption and adaptation, often manifest as crosscutting concerns. Some existing middleware standards address such document-oriented concerns by allowing custom Handlers to intercept and transform XML messages at various times during a service invocation. However, handlers do not support the sound software-engineering principle of “programming to an interface”. Their coarse-grained interface makes it difficult to map the design of concerns to the implementation and prevents useful static checking by a language compiler (e.g. checking whether two handlers may perform conflicting actions). To address this and similar design challenges, we have developed a new programming model, called Doxpects, which simplifies the development and maintenance of Handlers. The main benefit of Doxpects is the ability to perform XML document transformation tasks (insertion, deletion, and replacement) in a familiar object-oriented way. Doxpects enable fine-grained statically-typed handler interfaces and take advantage of them.

Contents

Abstract	ii
Contents	iii
List of Tables	v
List of Figures	vi
Acknowledgements	vii
1 Introduction	1
1.1 Contributions of this Research.....	3
1.2 Outline	4
2 Background	6
2.1 Messaging Protocols.....	6
2.2 JAX-RPC	7
2.3 XML Schema.....	8
2.4 XPath	11
2.5 XMLBeans.....	11
3 Message Processing.....	12
3.1 Message Filters	12
3.2 Content Capture	12
3.2.1 Automatic Iteration and Recursion	14
3.3 Content Modification.....	16
3.3.1 Replacement	16
3.3.2 Insertion.....	19
3.3.3 References to the Input Element Name.....	22

3.3.4 Deletion	24
3.3.5 Advice Interaction	24
3.4 Response Initiation	27
4 Type Checking and Dynamic Types	29
4.1 Modification Type Checking	29
4.1.1 Simple Type Comparison	32
4.1.2 Complex Type Comparison	33
4.1.3 Tests of the Algorithm	38
4.2 Dynamic Types	39
5 Implementation Details	42
5.1 Compilation Process	42
5.2 Doxpect Handler	43
5.3 Doxpect Compilers	44
5.3.1 Reflection API Compiler	44
5.3.2 Command-Line Interface	46
5.4 Connected XMLBeans and the <code>DoxpectBase</code> Class	46
6 Related Work	49
7 Conclusion	51
Bibliography	53
Appendix A: Filtering Messages by Service Endpoints	56
Appendix B: Graph Comparison Algorithm	58

List of Tables

Table 3.1: Sample <code>Pointcut</code> annotations.....	13
Table 3.2: The type restrictions for input and output parameters.....	17
Table 3.3: The insertion actions taken for different types of parameters.	19
Table 4.1: A summary of the facet support levels.....	33
Table 4.2: An overview of XML Schema language features used in eBay, FedEx, Amazon and Google Search schemas.	37
Table 4.3: A summary of comparison results of several eBay schema versions.....	38
Table 5.1: The possible options for the command-line Doxpects compiler.	45

List of Figures

Figure 1.1: The Enterprise Service Bus architecture.....	2
Figure 2.1: SOAP intermediaries.....	7
Figure 2.2: The JAX-RPC Handler interface.	8
Figure 2.3: The containment hierarchy of essential XML Schema elements.....	9
Figure 2.4: Sample XML Schema.	10
Figure 2.5: A sample instance document of the schema from Figure 2.4.	10
Figure 3.1: A decryption advice.	15
Figure 3.2: A dimension units validation handler.	18
Figure 3.3: A directions insertion handler.	21
Figure 3.4: An address validation advice.	23
Figure 3.5: An example of two request advices which are deemed conflicting by the interaction analysis in our previous work.	25
Figure 3.6: An XML Schema used by advices in Figure 3.5.	26
Figure 3.7: A response initiation example.....	28
Figure 4.1: An interoperability handler with type checking annotations.	31
Figure 4.2: A recursive type definition found in an eBay schema.	34
Figure 4.3: The effective occurrence interval.....	36
Figure 4.4: A case where our algorithm computes a wrong occurrence interval.	36
Figure 4.5: An advice with a <code>DynPointcut</code> annotation and a dynamic type.	41
Figure 4.6: An advice with a <code>DynUnionPointcut</code> annotation and a dynamic type.	41
Figure 5.1: A sample deployment descriptor of the <code>DoXpectHandler</code> for an Apache Axis server.....	43
Figure 5.2: A sample XML instance not conforming to its XML Schema.	47
Figure 5.3: An encryption advice.	47

Acknowledgements

I would like to thank Eric Wohlstadter for being my supervisor. His support, guidance, and encouragement made this work possible. Working with him was an amazing learning experience.

I would also like to thank Kris De Volder for being my second reader.

Finally, thanks to my family for all of their love and support.

IVAN MINEVSKIY

*The University of British Columbia
August 2006*

Chapter 1

Introduction

The Enterprise Service Bus (ESB) is an architectural pattern that is increasingly gaining the attention of architects and developers, as it provides an effective approach to optimize the distribution of information between different types of applications across multiple locations [23]. The architecture, depicted in Figure 1.1, is centered on a bus which provides a common backbone through which applications can interoperate. The bus provides message delivery services, based on standards such as HTTP and SOAP. Many enterprises find ESB attractive because it combines features from previous technologies with new services, such as message validation, transformation, content-based routing, security, load balancing, etc [11,21]. ESB makes applications oblivious of the routing services and transport protocols details, and allows substitution of service implementations as needed. The ESB architecture can be seen as a distributed version of the mediator pattern: the bus serves as a mediator that has detailed knowledge of all applications; applications send messages to the bus when needed and the bus passes them on to any other applications that need to be informed.

Web services in an ESB context provide an architectural approach to decomposing IT solutions into reusable pieces readily responsive to changing business demands. Developing a web service involves working with two different primary decompositions: the object-oriented design and the XML document structure. As pointed out in the literature, developers often get frustrated when mitigating conceptual mismatches between the decompositions [2,26].

The existing approach to modularizing document-oriented concerns in Web Services is through the use of specialized APIs. Some existing middleware standards, such as the Java XML Remote Procedure Call (JAX-RPC), allow custom *Handlers* to intercept and transform a SOAP message at various times during a service invocation [17]. Handlers can manage various

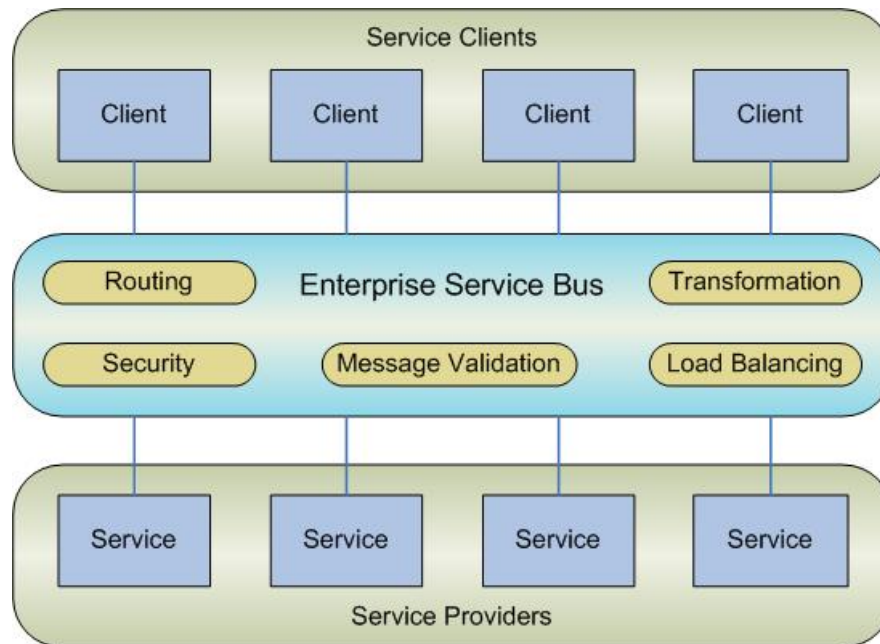


Figure 1.1: The Enterprise Service Bus architecture.

crosscutting concerns such as logging and auditing, encryption and decryption, and so on. However, they do not support the sound software-engineering principle of “programming to an interface”. Their coarse-grained interface makes it difficult to map the design of concerns to the implementation and prevents useful static checking by a language compiler (e.g. checking whether two handlers may perform conflicting actions).

Aspect-Oriented Programming (AOP) helps to localize code related to a crosscutting concern (aspect) in a single logical unit (aspect definition), and to define points (joinpoints) in the dynamic call graph of a program where aspect-related functionality should be inserted (woven) and executed [24]. With aspects, concerns can be kept separate and be woven together with other concerns during compilation. Moreover, aspects improve modularization and make the system oblivious of the new concerns added into it.

In our previous work, we have introduced a new aspect-oriented programming model, called Doxpects, which simplifies the development and maintenance of Handlers [30]. The main benefit of Doxpects is the ability to perform XML document transformation tasks (insertion, deletion, replacement) in a familiar object-oriented way. We expand upon the standard AspectJ joinpoint model by including first class support for document content properties. A *content-*

based pointcut provides the programmer a convenient way to express properties of joinpoints where XML 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.

The document content matched by a pointcut is parsed into object-oriented entities with XMLBeans [32] to provide a natural Java-based programming model for document-oriented aspects. An advice does not have to perform the actual transformation, but its signature must contain Java annotations specifying where and how the transformation should be performed.

Doxpects can perform useful compile-time checking of pointcuts to detect situations where a parameter type is incompatible with the content matched by a pointcut. The compiler can analyze advices to detect ones with conflicting transformations and to verify that the transformation output will be valid at runtime or that the transformations will succeed without runtime errors.

For situations when a pointcut matches elements with different types, but with similar content, Doxpects provide a simple way to access and modify content *common* to the matched elements. The doxpect compiler can create a new, *dynamic*, XML type containing the common child elements of the matched elements. The schema of the new type is compiled into XMLBeans which provide `get` and `set` methods. Alternatively, the compiler can be instructed to generate a dynamic type corresponding to a union of child elements.

1.1 Contributions of this Research

This thesis contributes a further step towards providing a clean mapping from design to implementation of handlers with a useful static checking which takes advantage of a well specified interface. We provide a detailed description of old and new Doxpects features and show how they can help to develop and customize web-services. In particular, we offer the following contributions:

- ❖ Reimplemented the Doxpects framework to be able to write handlers in Java 5 language using annotations (Chapter 3).

- ❖ Enabled handlers to filter messages by service endpoint URIs (Section 8).
- ❖ Developed a compile-time checking of pointcuts to detect incompatibilities between a parameter type and the content matched by a pointcut (Section 3.2).
- ❖ Added insertion and deletion transformations on top of the previously introduced replacement transformation (Sections 3.3.2 and 3.3.4).
- ❖ Improved the compile-time analysis of advice interactions by taking advice pointcuts into account. Before, two advices could be deemed conflicting even if the elements they matched were in different and non-overlapping parts of a message (Section 3.3.5).
- ❖ Enabled any request advice to terminate a request flow and initiate a response (Section 3.4).
- ❖ Improved the compile-time analysis of message content transformations by using pointcuts to get an approximate context in which the transformations might happen (Section 4.1).
- ❖ Introduced dynamic types, which provide a simple way to access and mutate content common to matched elements of different types (Section 4.2).

1.2 Outline

We begin by presenting the background in Chapter 2. In Chapter 3, we delve into details and explain how a handler in the Doxpects framework can intercept, filter, and process messages. First, in Section 3.1, we present message filtering by service endpoint URI and by message flow. We then describe how message content can be captured and transformed (Sections 3.2 and 3.3) and present our compile-time analysis of advice interactions (Section 3.3.5). At the end of Chapter 3, we show how a request advice can initiate a response (Section 3.4). In the first section of Chapter 4, we describe the compile-time analysis of message content transformations. In Section 4.2, we present dynamic types. In Chapter 5 we discuss the implementation details. In Section 5.1 we outline the compilation process and present the two available compilers: one based on Java Mirror API and the Annotation Processing Tool, and one based on the Java

Reflection API. Then, in Section 5.2, we explain how to write a deployment descriptor for a doxpect handler. We conclude Chapter 5 with more details on the Reflection API compiler and a summary of the command-line interface. The related work in the areas of XML transformation languages is presented in Chapter 6. Finally, in Chapter 7, we conclude and offer suggestions for future work.

Most of the examples throughout the thesis are based on the XML Schemas of the FedEx¹ web-service. The service is used worldwide to ship and track items like letters and parcels.

¹ FedEx: <http://www.fedex.com/>

Chapter 2

Background

2.1 Messaging Protocols

Many web services use Simple Object Access Protocol (SOAP) messages to communicate with clients [28]. SOAP is an XML-based protocol that enables a completely interoperable exchange between clients and web services. SOAP messages are transmitted over HTTP, which is a commonly used request-and-response standard for sending messages over the Internet.

Remote Method Invocation (RMI) is the biggest competitor of SOAP for communication in the Java space [20]. Under RMI, entire objects can be passed and returned as parameters. RMI has been available since JDK 1.02, and it is still popular with its use in technologies such as Enterprise JavaBeans (EJB). However, RMI is limited to Java Virtual Machines and cannot interface with other languages.

Common Object Request Broker Architecture (CORBA) is a competing distributed systems technology that offers greater portability than RMI [9]. Unlike RMI, CORBA is not tied to one language. Its only limitation is that a language must have a CORBA implementation written for it. However, for Java developers, CORBA offers less flexibility, because it does not allow executable code to be sent to remote systems.

We choose to base the Doxpects framework on SOAP because it offers a much easier means of messaging and communication than do Java RMI and CORBA. First, it is completely platform-independent, which enables greater interoperability between the client and server. Unlike CORBA and RMI, it is human readable, very lightweight, and it is not necessary to have stub objects on client machines. It is great for asynchronous communication and for loosely



Figure 2.1: SOAP intermediaries.

coupled clients and servers. Also, unlike XML-based SOAP messages, RMI and CORBA messages have binary content, which makes it hard to perform low-level message transformations. Finally, SOAP is designed with *intermediaries* in mind

An intermediary is an entity that provides additional functionality and value-added services by processing parts of a SOAP message as it travels between a client and service provider. There can be many intermediaries between a client and service, and each of them can both accept and forward messages (Figure 2.1). An intermediary can be dynamically added and removed, providing a flexible way to extend the functionality of the client, service, and other intermediaries. SOAP intermediaries are widely used and provide functions like message filtering, transformation, and caching.

2.2 JAX-RPC

JAX-RPC handlers define an architectural means for an application to access a raw SOAP message of a web service request or response [17]. The Handler API defines three methods for message handling, as shown in Figure 2.2. They handle SOAP *requests*, *responses* and *faults*, respectively.

On the client side, the handler's `handleRequest` method is invoked right after the proxy marshals the message. The `handleResponse` or `handleFault` method is invoked just before the proxy demarshals the message and returns it to the client. On a service endpoint, the `handleRequest` method is called before the message is dispatched to the target service endpoint; the `handleResponse` or `handleFault` method is called just before the container marshals the message and returns it to the client. Generally, the `handleFault` method may be called in the event of errors in endpoints, containers, or handlers themselves.

```
package javax.xml.rpc.handler;
public interface Handler {
    boolean handleRequest(MessageContext context);
    boolean handleResponse(MessageContext context);
    boolean handleFault(MessageContext context);
}
```

Figure 2.2: The JAX-RPC Handler interface.

On both client and server, a handler may be configured either programmatically or in a deployment descriptor. A deployment descriptor associates a handler with a particular web service endpoint and may specify any handler-specific configuration information. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code.

Handlers are permitted to modify the messages passed to them. If multiple handlers that are involved in one service invocation need to share information, they can do so by adding properties to the message context as it is passed from handler to handler.

2.3 XML Schema

The syntax of an XML-based language is defined using a dictionary of elements and attributes together with rules constraining their use. There exist several schema languages, such as DTD [3], XML Schema [31], or DSD2 [27], which allow the syntax to be formalized. An XML document is valid relative to a given schema if all the syntactic requirements specified by the schema are satisfied in the document.

The W3C XML Schema Definition Language is an XML language for describing and constraining the content of XML documents. The purpose of a schema is to define a class of XML documents, and so the term "*instance document*" is often used to describe an XML document that conforms to a particular schema.

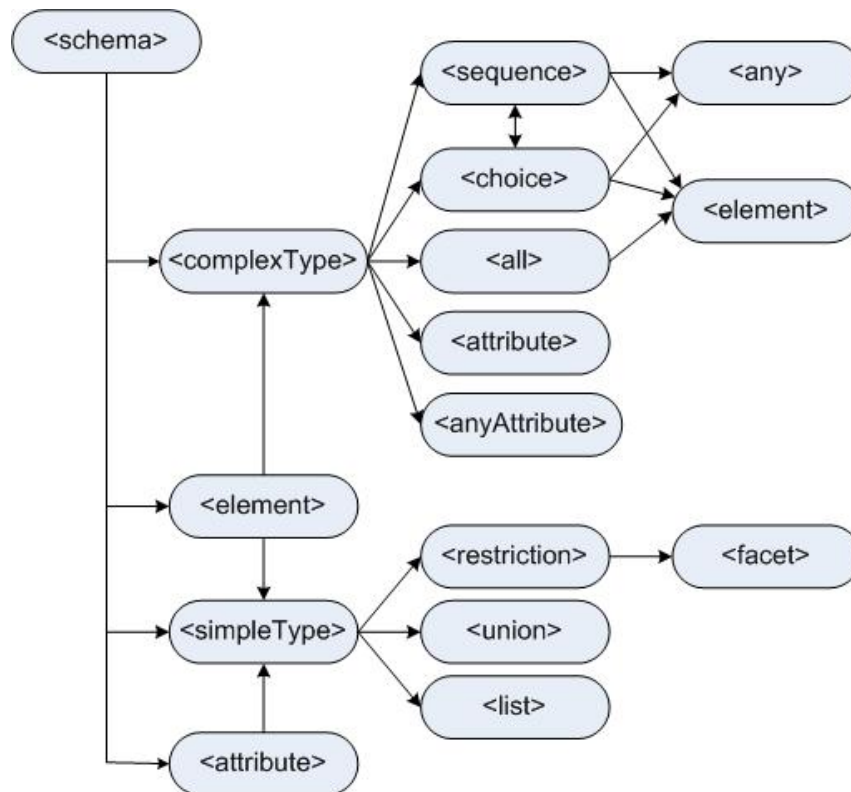


Figure 2.3: The containment hierarchy of essential XML Schema elements.

An XML Schema is composed of the top-level *schema* element. The *schema* element contains type definitions (*simpleType* and *complexType* elements) and *attribute* and *element* declarations. In addition to its built-in data types (such as integer, string, and so on), XML Schema also allows for the definition of new data types using the *simpleType* and *complexType* elements. Figure 2.3 depicts the containment hierarchy of essential Schema elements.

For example, the schema in Figure 2.4 has one top-level element *book* of type *bookType*, which has three child elements: *title*, *pages*, and *character*. A possible instance document of the schema is shown in Figure 2.5.

Unlike other schema definition languages, W3C XML Schema lets us define the cardinality of an element (i.e. the number of its possible occurrences). We can specify both *minOccurs* (the minimum number of occurrences) and *maxOccurs* (the maximum number of occurrences).

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="book" type="bookType"/>
  <complexType name="bookType">
    <sequence>
      <element name="title" type="string"/>
      <element name="pages" type="decimal"/>
      <element name="character" type="string" minOccurs="0"
        maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</schema>
```

Figure 2.4: Sample XML Schema.

```
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <title>Romeo and Juliet</title>
  <pages>100</pages>
  <character>Romeo</character>
  <character>Juliet</character>
</book>
```

Figure 2.5: A sample instance document of the schema from Figure 2.4.

For example, element `character` has `maxOccurs` set to `unbounded`, which means it may occur any number of times. Both attributes have a default value of one.

In an instance document, elements within a `sequence` structure must appear in the same order as in the schema. On the other hand, elements within an `all` structure may appear in any order. The choice structure allows only one of its child elements to appear in an instance document. However, note that `sequence` and `choice` structures may have `minOccurs` and/or `maxOccurs` attributes which effectively allows their content to occur multiple times in an instance document. The `any` element allows any elements from specific namespaces to appear in an instance document in place of it. Similarly, `anyAttribute` has the same meaning for attributes.

2.4 XPath

XPath is a domain specific language used primarily to address portions of an XML document [5]. An XPath expression is written as a sequence of slash-separated steps to get from one XML node (the current 'context node') to another node or set of nodes. The simplest XPath takes a form such as `/A/B/C`, which selects `C` elements that are children of `B` elements that are children of the top-level `A` element in a XML document. Two special operators are available to quantify over multiple possible elements: “*” and “//”. 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). For example, the expression `/A//B/*` selects all child elements of a `B` element that itself is a descendant ('//') of a top-level `A` element. Also, predicate expressions can be specified in square brackets after any step. For example, `//x[@id='Foo']` will match all `x` elements with an `id` attribute which value is `Foo`.

2.5 XMLBeans

XMLBeans is a Java-to-XML binding framework which is part of the Apache Software Foundation XML project [32]. XMLBeans map XML and XML Schema features as naturally as possible to the equivalent Java language and typing constructs. From a schema written in XML Schema language it can generate a collection of Java interfaces and classes representing an object model of the corresponding XML documents. XML data may then be processed as Java objects using the standard “get” and “set” JavaBeans pattern. Methods for marshalling and unmarshalling are automatically generated, and the mapping between XML and Java can be controlled by specifying explicit bindings.

In our programming model, document elements which are captured in a content-based pointcut are automatically made available to programmers using a statically typed XMLBeans representation. 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. This feature provides the bridge necessary where pointcuts are expressed in one language (XPath) and behaviour is expressed in another language (Java).

Chapter 3

Message Processing

In this chapter we describe how a handler in the Doxpects framework (a *doxpect* for short) can intercept, filter, and process messages. We first present message filtering by service endpoint URI and by message flow (request, response, and fault). We then describe operations with message content, which can be grouped into two categories: capture and modification. We continue by describing our compile-time analysis of advice interactions. In conclusion, we show how a request advice can terminate a request flow and initiate a response.

3.1 Message Filters

Doxpects can filter messages by a flow direction and by a service endpoint. The flow direction filters are implemented by the annotations *Request*, *Response*, and *Fault*. One of these three annotations must be defined on every method that should be considered as an advice. Advices with *Request* are applied to request messages, with *Response* – to response messages, and with *Fault* – to fault messages. A doxpect may have more than one advice of each type. Advices of the same type are executed in the order of their appearance in the doxpect source code.

To filter messages by a service endpoint, one has to use the class-level annotation *Target*. This annotation is described in details in Appendix A.

3.2 Content Capture

All operations with message content can be grouped into two categories: capture and modification. Correspondingly, there are two groups of annotations designed to help with these tasks. No parameter can have annotations from both groups and, therefore, for simplicity, we

Pointcut	Matched items
<code>@Pointcut("//fdx:Address")</code>	all <code>fdx:Address</code> elements in the SOAP Body element
<code>@Pointcut(root = RootType.HEADER, value = "wsse:Security/xenc:EncryptedKey")</code>	the <code>xenc:EncryptedKey</code> element from the <code>wsse:Security</code> header element in the SOAP Header element
<code>@Pointcut(".')</code>	the SOAP Body element

Table 3.1: Sample `Pointcut` annotations.

will call a parameter with a capture annotation an *input parameter* and a parameter with a modification annotation – an *output parameter*.

The only required annotation for an input parameter is `Pointcut`, which matches content within a message. It has two parameters: `value` and `root`. The `value` parameter defines an XPath expression; the `root` parameter defines a context for the XPath expression. The context can be the SOAP Body (default) or Header element of the current message, or an element matched by the `Pointcut` of another parameter in the same advice. Table 3.1 has some examples of the `Pointcut` annotation.

You may have noticed that XPath expressions use prefixes, but do not bind them to namespaces. The binding is done in a class-level `Include` annotation. Its value is a list of URI-to-prefix mappings of the form "URI as prefix", where URI and prefix stand for themselves. Both URI and prefix must have no spaces inside them. For example:

```
@Include({ "http://www.w3.org/2001/04/xmlenc# as xenc",
          "http://webservice.fedex.com/v20060404/ as fdx" })
```

The content matched by a pointcut is made directly available to a doxpect advice through the value of the corresponding advice parameter, providing a fine-grained interface between documents and doxpects. An input parameter must have a type which is either a subtype of the `xmlObject` class, which is the root of the XMLBeans type hierarchy, or a subtype of the `xmlObject[]`. In the former case, the parameter captures only the first matched element. If the compiler detects that the pointcut may match multiple element, it gives a warning. An array

parameter captures all matched elements. In both cases, the content to be captured is required to have the same type as the XML Schema type from which the parameter type was generated by an XMLBeans compiler. If the compiler detects that the parameter type is incompatible with the content matched by the pointcut, it gives a warning. To get an approximate content matched by a pointcut, the compiler evaluates the pointcut in the context of schema documents. The approximations we employ in evaluating a pointcut at compile-time are described in Section 4.1.

Consider a situation when one pointcut has to be evaluated in the context of elements matched by another pointcut. For example, one might want to have separate input parameters for several descendants of an element captured by another input parameter. To avoid repeating the XPath to the parent element in pointcuts of its descendants, one could use the annotation *Mark*. This pointcut can make the content captured by an advice parameter to be the context for a pointcut of another parameter. This annotation has one integer argument whose value may range from 1 (default) to 9 inclusive. The value of a *Mark* annotation becomes a numeric index of its advice parameter, which can be referred from a *Pointcut* annotation of another parameter in the same advice. In a method signature, a context parameter does not have to be defined before the parameters which use it. Mutual or looped context dependencies are not allowed. For example, to match all `fdx:Address` elements in the context of a parameter with an annotation `@Mark(3)`, one would have to write an advice with a signature similar to the following one:

```
public void checkAddr(  
    @Pointcut("/fdx:ShipRequest") @Mark(3)  
    ShipRequestXmlBean request,  
    @Pointcut(root=Pointcut.RootType.MARK3, value="//fdx:Address")  
    AddressXmlBean[] addresses)
```

3.2.1 Automatic Iteration and Recursion

Often, a pointcut might match more than one element. To get access to all matches through an advice parameter, the parameter must have an array type. The most usual use case in such situations is to iterate over the array and do something with each matched element on its own. The method-level *Each* annotation provides a simple means of iterating over all such matches automatically. When an advice has this annotation, it is executed once for each element matched by the pointcut of the first parameter. Moreover, a pointcut query, which has the value of the first

```

@Request @Each @Recursive
public void decrypt(
    @Pointcut(value = "//xenc:EncryptedData")
    @In EncryptedDataTypeXmlBean encData,
    @Out Vector<XmlObject> decData,
    @Pointcut(root = RootType.HEADER,
        value = "./wsse:Security[@soapenv:actor='wss4j']")
    SecurityHeaderTypeXmlBean secHeader)
{
    WSHandlerResult decResult = null;
    try {
        DoxpectSecurity wss4j = new DoxpectSecurity(secProps);
        wss4j.prepareReceiver(getMessageContext(), secHeader);
        decResult = wss4j.decrypt(encData);
    } catch (WSecurityException se) { /* Details elided */ }

    /* Elided, copy data from decResult to decData */
}

```

Figure 3.1: A decryption advice recursively decrypts all `xenc:EncryptedData` body elements.

parameter as its context, is re-executed at each step of an automatic iteration and, therefore, usually has results which vary between steps. A pointcut of a parameter other than the first one, which does not have the first parameter as a context, is executed once and, therefore, has the same value at each step of an automatic iteration.

Consider a situation where an advice modifies a message in such a way that this same advice has to be applied to the message again. For example, this happens in cryptography when some data is encrypted twice (super-encryption) [15]. In this case, a decryption advice replaces an encrypted element with a decrypted content, which itself turns out to be an encrypted element. Since the decryption advice most likely does not know how many layers of encryption an element has, it is useful to be able to automatically apply the decryption advice to the element until it is no longer encrypted. The method-level *Recursive* annotation provides just that (Figure 3.1). It continues to execute an advice until the pointcut of one of the parameters matches nothing.

3.3 Content Modification

3.3.1 Replacement

Although a doxpect advice may behave just like a standard AspectJ advice by executing extra code, doxpect advice are specially suited to perform transformation over captured document content. In an advice signature, any input parameter can be paired with an output parameter with the help of parameter annotations *In* and *Out*.

The annotation `In` must be declared on every input parameter to be replaced. The annotation has one parameter - pair index, which is equal to one by default. The corresponding output parameter must have an `Out` annotation with the same pair index. Initially, on advice invocation, an output parameter is an empty placeholder bean (or an empty `Vector` of beans) of the type specified in the advice signature. Upon advice termination, the content captured by a parameter with the corresponding `In` annotation is replaced by the content stored in the output parameter. The annotation `Out`, in addition to the pair index parameter, takes an XML name of the replacement element. This name is required because the Java type of a replacement parameter only indicates the XML type of the replacement element, which is often different from the desired name of the XML element.

A replacement specifies a contract that an element *matched* by a pointcut will be *replaced* by another element of the same or a different type in the advice. 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. Programmers can fill in the data of the replacement variable and the transformation is automatically executed on the underlying document when the advice exits. The contract information is used for checking pointcut types and advice conflicts.

A replacement can be either one-to-one or one-to-many. In the former case, each captured element is replaced with exactly one output element. When an input parameter is capturing a single element and, therefore, is a subtype of `XmlObject`, the corresponding output parameter must store exactly one output element and, therefore, must be a subtype of `XmlObject` as well. On advice invocation, the output parameter contains an empty bean. Similarly, an input parameter which is a subtype of `XmlObject[]` requires the corresponding output parameter to be

	Possible types	
Input parameter	a subtype of <code>XmlObject</code>	a subtype of <code>XmlObject[]</code>
Output parameter	a subtype of <code>XmlObject</code> , <code>Vector<? extends XmlObject></code>	a subtype of <code>XmlObject[]</code> , <code>Vector[]<? extends XmlObject></code>

Table 3.2: The type restrictions for input and output parameters.

an array of an `XmlObject` subtype as well. On advice invocation, the output array contains empty beans and has the same size as the input array.

In the one-to-many case, each captured element is replaced with any number of output elements (zero output elements is equivalent to the removal of the input element). When an input parameter is capturing a single element, the corresponding output parameter must have a `Vector<X extends XmlObject>` type. On advice invocation, the output parameter contains an empty `Vector` with a component type `x`. Similarly, an input parameter which is a subtype of `XmlObject[]` requires the corresponding output parameter to be a `Vector[]<X extends XmlObject>`. On advice invocation, the output parameter contains an empty array of empty `Vector` objects with component type `x`, and the output array has the same size as the input array. Table 3.2 summarizes the type restrictions for replacement parameter pairs.

The annotation `Each` can be used to automatically iterate over an array of replacement parameter pairs. As with simple iteration in Section 2.1.1, the input parameter must be the first advice argument. The corresponding output parameter is assigned an empty value at the beginning of each iteration, and, therefore, must be either a subtype of `XmlObject` or a `Vector<? extends XmlObject>`.

For example, the FedEx web-service only supports centimetres (CM) and inches (IN) as dimension units. The `ValidationHandler` in Figure 3.2 has the `validateUnits` advice which converts dimensions in other units, such as millimeters (MM), to CM or IN. The handler intercepts all messages having the FedEx service as an endpoint. The advice further filters out the request messages. The annotation `Each` instructs the advice to iteratively apply itself to each `fdx:Dimensions` element matched by the pointcut of the first advice parameter. At the i^{th} iteration step, the advice is invoked with the following parameter values: the parameter `inDims`

```
@Doxpect    // This annotation identifies the class as a doxpect handler
@Include({"http://webservice.fedex.com/v20060404/ as fdx"})
@Target({"http://[^\]*/axis/services/fedex"})
public class ValidationHandler extends DoxpectBase
{
    @Request
    @Each
    public void validateUnits(@Pointcut("//fdx:Dimensions")
                               @In DimensionsXmlBean inDims,
                               @Out DimensionsXmlBean outDims,
                               @Pointcut("//fdx:RequestHeader")
                               RequestHeaderXmlBean reqHeader)
    {
        String inUnits = inDims.getUnits();
        // CM and IN units do not need to be converted
        if (inUnits.equals("CM") || inUnits.equals("IN")) {
            outDims.set(inDims);
            return;
        }
        // Convert MM to CM
        } else if (inUnits.equals("MM")) {
            outDims.setLength(inDims.getLength()/10);
            outDims.setWidth(inDims.getWidth()/10);
            outDims.setHeight(inDims.getHeight()/10);
            outDims.setUnits("CM");
        }
        /* Other conversions are elided */

        // Respond with an error if units are not supported
        respondWithError(reqHeader, "ER-0125",
                        "Dimension units "+ inUnits +" are not supported");
    }
}
```

Figure 3.2: A dimension units validation handler.

Possible output type	Action
a subtype of <code>XmlObject</code>	The output element is inserted before or after the first matched element.
a subtype of <code>XmlObject[]</code>	One output element is inserted before or after each matched element.
<code>Vector<? extends XmlObject></code>	All output elements are inserted before or after the first matched element.
<code>Vector[]<? extends XmlObject></code>	All output elements in a <code>Vector</code> are inserted before or after each matched element.

Table 3.3: The insertion actions taken for different types of parameters.

contains the i^{th} `fdx:Dimensions` element found in the message body; the parameter `outDims` is an empty placeholder; and the parameter `reqHeader` contains the `fdx:RequestHeader` element, which is unique in a message. When the advice is invoked, it uses `get` methods of the `inDims` parameter and `set` methods of the `outDims` parameter to get and convert dimension units. If the units cannot be converted, the advice uses the information in the `reqHeader` to create and send back to the requestor a response with an error message (more details on response initiation can be found in Section 3.4).

3.3.2 Insertion

Dopects provide two parameter annotations for new element insertions. A parameter annotation `InsertBefore` indicates that, upon advice termination, the value of the parameter will be inserted before the content matched by the `pointcut`. Similarly, an annotation `InsertAfter` indicates that the value of the parameter will be inserted after the matched content. In both cases, the content captured by the `pointcut` is not available to the advice.

The annotations `InsertBefore` and `InsertAfter` have one parameter which, similar to the `name` parameter of the annotation `Out`, defines the XML name of the element to be inserted.

A parameter with one of the insertion annotations is an output parameter and, therefore, can have any type suitable for an output parameter as defined in Table 3.2. The specific insertion action taken for different parameter types are outlined in Table 3.3.

The annotation `Each` can be used to automatically iterate over all elements matched by a pointcut of an insertion parameter. As with simple iteration in Section 3.2.1, the parameter must be the first advice argument.

For example, the handler in Figure 3.3 has two advices which insert FedEx outlet addresses and directions to response messages. The first advice, `insertDirsPerLocation`, inserts a `fdx:Directions` element with driving directions into each `fdx:Location` element found in `fdx:FDX FedExLocatorReply` messages. The first parameter, `location`, provides access to the content of `fdx:Location` element from which we obtain the ID of the FedEx outlet. The second parameter provides a placeholder for the driving directions; its pointcut defines the element after which the `fdx:Directions` element should be inserted. In the advice body, the ID is mapped to the driving directions which are saved in the `directions` object. Upon advice termination, the value of the `directions` parameter is inserted as the last child of the `fdx:Location` element (the `fdx:Location` schema allows any elements at the end of its child sequence). The `Mark` annotation is used to correlate the pointcut values: the pointcut of the parameter `directions` should be evaluated in the context of the element matched by the pointcut of the parameter `location`.

Similarly, the second advice inserts `fdx:StationAddress` and `fdx:Directions` elements into the parent element of each `fdx:DestinationStationID` found in any FedEx response message. The first parameter, `stationID`, provides access to the ID of the FedEx station. The second parameter, `address`, provides a placeholder for the address. The pointcut of `address` defines the element after which the `fdx:StationAddress` element should be inserted – after the last child of the `stationID`'s parent element. The last parameter, `directions`, provides a placeholder for the driving directions. Its pointcut says that the `fdx:Directions` element should be inserted at the same place as the `fdx:StationAddress` element.

```
@Doxpect
@Target({"http://[^/]*axis/services/fedex"})
@Include({"http://webservice.fedex.com/v20060404/ as fdx"})
public class DirectionsHandler extends DoxpectBase
{
    @Response
    @Each
    public void insertDirsPerLocation(
        @Pointcut("fdx:FDXFedExLocatorReply/fdx:Location") @Mark
        LocationXmlBean location,
        @Pointcut(root=RootType.MARK1, value="../*[last()]")
        @InsertAfter("fdx:Directions")
        XmlString directions)
    {
        directions.setStringValue(
            Locations.getLocationDirections(location.getBusinessID()));
    }

    @Response
    @Each
    public void insertDirsPerID(
        @Pointcut("//fdx:DestinationStationID") @Mark
        XmlString stationID,
        @Pointcut(root=Pointcut.RootType.MARK1,value="../*[last()]")
        @Mark(2)
        @InsertAfter("fdx:StationAddress")
        XmlString address,
        @Pointcut(root=Pointcut.RootType.MARK2, value=".")
        @InsertAfter("fdx:Directions")
        XmlString directions)
    {
        address.setStringValue(
            Locations.getLocationAddress(stationID.getStringValue()));
        directions.setStringValue(
            Locations.getLocationDirections(stationID.getStringValue()));
    }
}
```

Figure 3.3: A directions insertion handler.

3.3.3 References to the Input Element Name

Often the name of the output XML element should be the same as or similar to the name of the matched element. A special construct `{In}` can be used to refer to the name of the original element from the `name` parameter of the `Out`, `InsertBefore`, or `InsertAfter` annotation. For example, the following `Out` annotation defines the name of the replacement element as the name of the original element with suffix “2”:

```
@Out (name=" {In}2" )
```

It is also possible to get a substring of the original name. For example, to get a substring `[3, length-2]`, one would write

```
 ${In[3,L-2]},
```

where `L` identifies the length of the string captured by `{In}`.

For instance, consider the advice in Figure 3.4. The advice validates addresses present in a FedEx request message. Depending on the message type (e.g. `FDXShipRequest`, `FDXRateRequest`, etc), an address element might have one the following names: `fdx:Address`, `fdx:OriginAddress`, or `fdx:DestinationAddress`. The advice cleans up the address fields (e.g. removes unnecessary spaces, resolves country name abbreviations, etc) and ensures they have the valid syntax. If the validation fails, the advice creates and sends back to the requestor a response with an error message (see Section 3.4 for more details on response initiation). Otherwise, if all fields are valid, we want to replace the original address element by a cleaned address element with the same name. However, since we do not know the name of the address element at compile-time, we cannot specify the output element name in the `Out` annotation. In this case, the `{In}` construct comes in handy: setting the `name` parameter of the `Out` annotation to `"fdx:{In}"` makes the name of the replacement element equal to the name of the matched element.

```
@Request
@Each
public void validateAddress(
    @Pointcut("//fdx:Address | " +
        "//fdx:OriginAddress | " +
        "//fdx:DestinationAddress")
    @In AddressXmlBean inAddr,
    @Out(name="fdx:${In}") AddressXmlBean outAddr,
    @Pointcut("//fdx:RequestHeader")
    RequestHeaderXmlBean reqHeader)
{
    StringBuffer errorMsg = new StringBuffer(128);

    String line1 = inAddr.getLine1().trim().replaceAll(" ", " ");
    if (line1.length() == 0)
        errorMsg.append("Address/Line1 is a required element and " +
            "cannot be empty.");
    else if (line1.length() > MAX_LINE1)
        errorMsg.append("Address/Line1 cannot be longer than " + MAX_LINE1 +
            " characters.\n");

    /* The validation of Line2, City, State, Country, and Postal Code
       * is similar to the above and is elided */

    if (errorMsg.length() > 0)
        respondWithError(reqHeader, "ER-0057",
            "Validation errors: \n"+ errorMsg.toString());
    else {
        outAddr.setLine1(line1);
        /* Elided: copy other address components */
    }
}
```

Figure 3.4: An address validation advice.

3.3.4 Deletion

To remove the content matched by a pointcut from the underlying message, one has to annotate the corresponding parameter with a *Remove* annotation. Similarly to input parameters discussed before, the parameter captures the content matched by the pointcut, but this content is deleted from the message upon advice termination.

3.3.5 Advice Interaction

When several advices mutate the same part of the message, they could interfere with each other. In such situations, the order in which the advices are executed may be very important. In our earlier work, three types of content-based advice interactions were identified: *corruption*, *inhibition*, and *activation* [30].

Consider a situation where a dominating advice replaces an element A with an element B of a different type. The dominated advice matches an element C which contains a child element of type A. Now, when the dominated advice executes, C is no longer properly typed because it should not contain an element of type B. Such a conflict is called an *advice corruption*.

An *advice inhibition* occurs when a dominating advice replaces an element A with an element B of a different type, so that a dominated advice is prevented from matching A or one of A's children.

A *missed inter-advice activation* occurs when an advice does not match what it should have since it executed before certain content appeared. For example, a dominating advice matches an element B, but that element only appears in the document when another dominated advice transforms A to B subsequently.

A *missed intra-advice activation* occurs when a single advice performs a transformation which causes the same advice to again become active. For example, this situation arises when the decryption advice from Section 3.2.1 decrypts a super-encrypted element.

The interaction analysis in our previous work was overly conservative because it used only type information in detection. Since advice pointcuts were not taken into account, two advices could be deemed conflicting even if the elements they matched were in different and non-overlapping parts of a message.

```
@Request @Each
public void transformClientAddr(@Pointcut("/del:Client/del:Address")
                               @In AddressTypeXmlBean inAddr,
                               @Out Address2TypeXmlBean outAddr)
{ /* Details elided */ }

@Request @Each
public void validateDeliveryAddr(@Pointcut("/del:Delivery/del:Address")
                                 AddressTypeXmlBean addr)
{ /* Details elided */ }
```

Figure 3.5: An example of two request advices which are deemed conflicting by the interaction analysis in our previous work.

For example, consider an XML Schema in Figure 3.6. The `Address` element occurs in three places within the schema: as a top-level element, as a child of the element `Delivery` and a child of the element `Client`. Now observe the two request advices in Figure 3.5. The first advice, `transformClientAddr`, performs an interoperability transformation of all `Client` elements: the `Address` child element is replaced with its newer version – `Address2`. The second advice, `validateDeliveryAddr`, validates the `Address` elements within all `Delivery` elements. Note that the two advices operate on different parts of a message and, therefore, cannot perform any conflicting actions with respect to each other. However, the interaction analysis in our previous work did not take pointcuts into account, and, in this case, it would conclude that both advices operate on `Address` elements in all three locations shown in Figure 3.6. Then, since `validateDeliveryAddr` is executed after `transformClientAddr`, the analysis would report an inhibition conflict. To avoid such false positives, we have improved the analysis by using pointcuts to get an approximate context in which the interactions might happen.

To get the context at compile-time, we evaluate advice pointcuts in the context of schema documents. If an advice interaction is detected, it is reported as a compile-time warning. The approximations we employ in computing the context are described in Section 4.1. Note that we have currently only implemented interaction checking in the context of a single doxpect and not yet across doxpects, although we feel that an extension will be straightforward.

```
<schema elementFormDefault="qualified"
  targetNamespace="http://delivery"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:del="http://delivery">

  <element name="Address" type="del:AddressType"/>
  <complexType name="AddressType">
    <!-- Details elided -->
  </complexType>

  <element name="Address2" type="del:Address2Type"/>
  <complexType name="Address2Type">
    <!-- Details elided -->
  </complexType>

  <element name="Delivery">
    <complexType>
      <sequence>
        <element name="Date" type="xsd:date"/>
        <element name="Time" type="xsd:time"/>
        <element name="Address" type="del:AddressType"/>
      </sequence>
    </complexType>
  </element>

  <element name="Client">
    <complexType>
      <sequence>
        <element name="Name" type="xsd:string" />
        <element name="Address" type="del:AddressType" />
      </sequence>
    </complexType>
  </element>
</schema>
```

Figure 3.6: An XML Schema used by advices in Figure 3.5. The `Address` element is in bold and occurs in three places within the schema.

3.4 Response Initiation

Consider a situation where a request advice performs some validation of message content. If the content is invalid the advice should report the error to the client right away without propagating the message further down the request flow. In other words, the advice should construct a response message with some customized error description and send it back to the client. Doxpects may help to achieve this behaviour when executed in the Apache Axis environment.

A utility class `AxisUtils` has methods which help to initiate a response. The methods `appendToResponseHeader(QName, XmlObject)` and `appendToResponseBody(QName, XmlObject)` append a new element with the given name and content to the response message SOAP Header and Body correspondingly. If the response message does not exist yet, it is created. Note that these methods can be called from different advices and even different doxpects.

The method `startResponse` makes the current advice the pivot point. This effectively starts the response flow right after the current advice terminates. Note that the pivot point has the granularity of an advice and not of a doxpect. The response message constructed with the help of the `AxisUtils` methods becomes the initial response message which can be altered by response advices. If no content is added to the response message, it remains empty unless response advices modify it. This method has no effect if called from a response advice or if the current doxpect is not deployed in a `HandlerInfoChain`.

For example, Figure 3.7 shows the `respondWithError` method used by several request advices in previous sections. The method receives three parameters: request message header, error code, and error message. The top-level element of a request or response FedEx message corresponds to an operation, and its name starts with the operation name followed by the word "Request" or "Reply" correspondingly. To construct the name of the response top-level element, the method gets the name of the request top-level element and replaces its suffix "Request" with "Reply". Then, the top-level element is created with the help of the `appendToResponseBody` method. Note that the second argument of the call is `null` – this instructs the `appendToResponseBody` method to create an empty element in the message body. If a valid `XmlObject` was given rather than `null`, the content of the provided `XmlObject` would be copied to the new top-level element. Next, a `fdx:ReplyHeader` element is inserted into the empty top-level element. The reply header contains a transaction identifier, which is set to the value stored

```
private void respondWithError(RequestHeaderXmlBean reqHeader,
                             String errorCode, String errorMsg)
{
    // Get the name of the operation
    XmlObject reqTopElem = DoxpectUtils.getParent(reqHeader);
    String reqName = reqTopElem.newCursor().getName().getLocalPart();

    // Replace the name suffix "Request" with "Reply"
    String respName = reqName.substring(0, reqName.length()-7) + "Reply";
    QName name = new QName(FDX_NS_URI, respName, "fdx");

    // Create the root message element
    XmlObject reply = AxisUtils.appendToResponseBody(name, null);

    // Insert a fdx:ReplyHeader element and set its TransactionID to be
    // the same as in the reqHeader parameter
    QName replyHeaderName = new QName(FDX_NS_URI, "ReplyHeader", "fdx");
    XmlObject child = DoxpectUtils.appendChild(reply, replyHeaderName);
    ReplyHeaderXmlBean replyHeader = (ReplyHeaderXmlBean)child;
    replyHeader.setCustomerTransactionIdentifier(
        reqHeader.getCustomerTransactionIdentifier());

    // Insert a fdx:Error element and set its code and message
    QName errorName = new QName(FDX_NS_URI, "Error", "fdx");
    child = DoxpectUtils.appendChild(reply, errorName);
    ErrorXmlBean error = (ErrorXmlBean)child;
    error.setCode(errorCode);
    error.setMessage(errorMsg);

    // Make this advice the pivot point
    AxisUtils.startResponse();
}
```

Figure 3.7: A response initiation example.

in the request header. Next, a `fdx:Error` element is appended after the reply header, and its code and message are set. Finally, the `startResponse` method is invoked to indicate that the current advice should be the last advice in the request flow.

Chapter 4

Type Checking and Dynamic Types

In the first section of this chapter we describe the compile-time analysis of message content transformations which verifies that the transformation output will be valid at runtime and that the transformations will succeed without runtime errors. In the second section of the chapter, we present *dynamic types*, which provide a simple way to access and mutate content common to matched elements of different types.

4.1 Modification Type Checking

Consider a situation when a service has updated its schemas, but a client is still using old schemas. To remain backwards-compatible, the service must be able to receive and send messages in the old format. As shown in previous sections, doxpects can help to automatically transform both request and response messages to the new format. However, how can we check that the set of modifications performed by advices is enough to completely convert any message into the new format? We could do that by generating all possible message instances, transforming them according to advice signatures, and validating them against the new set of schemas. However, this is impossible because the set of instances described by a schema is potentially infinite. Instead, we apply the modifications to schema documents. Obviously, the modifications are emulated and may be incomplete. For example, a schema might have particles `any` which, in an instance document, allow any number of any elements in place of them. We choose to assume that none of the XPath steps match particles `any`. Also, it is hard and often even impossible to evaluate XPath predicates on values known only at runtime. For emulation purposes, we choose to ignore predicates altogether. However, the set of schemas altered by emulated modifications can still represent well the set of transformed messages.

It might seem that the modification analysis could be based on XMLBeans classes generated from schemas rather than the schemas themselves. However, it cannot be done because an XMLBeans class lacks a lot of important information about the XML type it corresponds to. First of all, the class lacks information about the order in which the child elements appear in the type. Secondly, the class does not know the structure of the type. For example, it does not model how the child elements are organized by `<sequence>`, `<choice>`, and `<all>` tags. Also, the class does not store values of `minOccurs` and `maxOccurs` attributes. Only indirectly, by analysing field types and method signatures, the attribute value can be inferred to be zero, one, or greater than one. Finally, we perform the modification type checking to improve the external service interface, which is defined in XML Schema language. Therefore, to achieve the best results, we have to base our analysis on schemas and cannot use shortcuts like XMLBeans classes.

To automatically perform a modification type checking of request messages, one should use the class-level annotation `TypeCheckRequest`. Similarly, to check response message transformations, one should use the class-level annotation `TypeCheckResponse`. Each of these annotations takes a paired list of schemas as a parameter. Each pair defines a schema to which the transformed message should conform (a *reference schema*) and a schema to which the original message conforms. The doxpect compiler transforms the later schema into a schema to be compared against the former schema. We call the transformed schema the *validated schema*.

The exact syntax of a schema pair is "OrigSchema vs RefSchema", where `OrigSchema` is the target namespace URI of the original schema to which the message conforms, and `RefSchema` is the URI of the reference schema. URIs of server-side schemas defined in an `Include` annotation (Section 3.2) may be replaced by the corresponding prefixes.

For example, an interoperability handler in Figure 4.1 bridges syntactic mismatches between two otherwise semantically compatible versions of the eBay XML Schema: an old one used by a client (`http://webservice.ebay.com/v419/`) and a new one used by the service (`http://webservice.ebay.com/v421/`). To ensure that the set of modifications performed by advices is enough to completely convert any message between the old and new format, the handler performs a modification type checking. The `TypeCheckRequest` annotation instructs the

```
@Doxpect
@Include({"http://webservice.ebay.com/v421/ as ebay"})
@Target({"http://[^/]*/axis/services/ebay "})
@TypeCheckRequest({"http://webservice.ebay.com/v419/ vs ebay"})
@TypeCheckResponse({"ebay vs http://webservice.ebay.com/v419/"})
public class InteropHandler extends DoxpectBase {
    /* Elided */
}
```

Figure 4.1: An interoperability handler with type checking annotations.

handler to check that, after the set of transformations is applied to a client request, the message conforms to the new schema. Vice versa, the `TypeCheckResponse` checks that, after the transformations, any response message is compatible with the old schema.

During a request type checking, a comparison is performed to ensure that, after all request advices of a doxpect are executed, all instances of a remote schema are valid instances of a local schema. Hence, in a request comparison, the reference schemas are the local ones and the validated schemas are the remote ones. Conversely, during a response type checking, a comparison ensures that, upon execution of all response advices within a doxpect, all instances of a local, validated, schema are valid instances of a remote, reference, schema. This naming convention recursively applies to elements, attributes and types defined in a schema.

To compare two schemas, the set of global elements in the validated schema is compared to the set of global elements in the reference schema. A comparison succeeds if the reference schema has all global elements of the validated schema.

Two elements or two attributes are equal if they have the same name and equivalent types. A pair of *any* or *anyAttribute* wildcards are compared by their namespace constraints. To compare an element to an *any* wildcard, we check whether the wildcard namespace constraint allows the element namespace. Similarly, to compare an attribute to an *anyAttribute* wildcard, we check whether the wildcard namespace constraint allows the attribute namespace.

When an element definition has no `type` attribute, the element type is effectively a wildcard allowing the element to have any type. If a reference element has the wildcard type then any validated element with the same name matches it. On the contrary, a validated element with the

wildcard type may only match a reference element with the wildcard type. When a type schema cannot be loaded, the type remains *unresolved*. Unresolved types can only be compared by names and we choose to do so when both compared types are unresolved. We report a mismatch when only one type is unresolved.

When present, the value of the `type` attribute of an element resolves either to a complex type or a simple type. While a simple type can have neither element children nor attributes, a complex type may have both. Complex types are further divided into two groups: those with simple content and those with complex content. While both forms of complex types allow attributes, only those with complex content allow child elements; those with simple content only allow character content. A complex type with a simple content is an extension or restriction of either a simple type or another complex type with a simple content. Therefore, we view a complex type with a simple content as a simple type with attributes and handle it together with simple types. The comparison algorithm of simple types and complex types with simple contents is outlined in Section 4.1.1. In Section 4.1.2, we describe the comparison algorithm of complex types with complex contents.

4.1.1 Simple Type Comparison

In this section, we outline the comparison process of simple types and complex types with simple contents. To compare two types, we compare their primitive base types and facets that constrain the range of their values. For complex types, we also compare attributes. A comparison succeeds if the validated type is a supertype of the reference type or, in other words, if the set of possible element instances of the validated type is a subset of the one of the reference type. To meet this condition, the types must have the same primitive base type and the validated type must be constrained no less than the reference type. For complex types, the set of attributes of the validated type must be a subset of attributes of the reference type. For a simple type restriction and a complex type restriction, the constraints of the base type are taken into account as well. For a complex type extension, we take into account attributes of its base type.

The comparison algorithm supports atomic, list and union simple types. It supports all facets except for `pattern` and `whiteSpace`. Both numeric and date or time related values of the facets `minInclusive`, `maxInclusive`, `minExclusive`, and `maxExclusive` are supported.

Facet	Level of support	Facet	Level of support
enumeration	Full	minInclusive	Full
length	Full	maxInclusive	Full
minLength	Full	minExclusive	Full
maxLength	Full	maxExclusive	Full
totalDigits	Full	pattern	None
fractionDigits	Full	whiteSpace	None

Table 4.1: A summary of the facet support levels.

The `pattern` facet constraints the lexical space of an attribute value to literals which match a specific pattern. The pattern value must be a regular expression. The `pattern` facets could be compared using the standard finite state machine technique.

The `whiteSpace` facet specifies how occurrences of space, tab, line feed, and carriage return characters should be handled: preserved, replaced by space characters, or, for contiguous sequences, collapsed into a single space character. We see little benefit in comparing values of `whiteSpace` facets and therefore choose to ignore it. Table 4.1 provides a summary of the facet support levels.

4.1.2 Complex Type Comparison

In this section, we focus on the comparison of two complex types with complex contents. A comparison of a validated complex type and a reference complex type is done in two steps: attribute comparison and element comparison. To pass the attribute comparison, the attribute set of the validated type must be a subset of the attribute set of the reference type.

To simplify the element comparison, the elements of each complex type are converted into a graph which models their relative ordering. If two elements may directly follow each other in a message, their graph nodes are connected. A type extension is handled by appending the elements defined in the extension to the graph of the base type. A type restriction is handled like a complete redefinition of the base type and, therefore, the base type is ignored. A *validated graph* is a graph corresponding to a type from a validated schema, and, similarly, a *reference graph* corresponds to a type from a reference schema.

```
<xs:complexType name="StoreCustomCategoryType">
  <xs:sequence>
    <xs:element name="CategoryID" type="xs:int" />
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Order" type="xs:int" />
    <xs:element name="ChildrenCategories"
      type="ns:StoreCustomCategoryType" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

Figure 4.2: A recursive type definition found in an eBay schema. The type describes a configuration of a store custom category: category ID and name, order in which the custom category appears in the list of store categories, and an optional sub-category.

A graph *wildcard node* stores namespace constraints and occurrence parameters of an *any* wildcard. A graph *element node* stores the element itself and its two *occurrence constraints*, which indicate how many times the element *must* and *may* occur in an instance document. As the comparison progresses and nodes from two graphs get matched to each other, their occurrence constraint values decrease, reflecting the remaining number of unmatched element instances.

Two element nodes are equal if the schema elements they correspond to have the same names and types. If the types are not simple, they are themselves compared. Recursive and mutually recursive type definitions are rarely used in the real life schemas that we have analysed: Amazon, eBay, FedEx, and Google Search. A sample of a recursive type definition from an eBay schema is shown in Figure 4.2. However, the algorithm is able to detect such definitions by maintaining a stack of type pairs that are currently being compared. When a pair of types, which is already in the process of being compared, is attempted to be compared again, the types in the last pair are assumed to be equal. The result of a comparison of a pair of types is cached to improve performance.

Along with nodes corresponding to elements and wildcards, a graph has nodes which model the structural tags in a complex type: `<sequence>`, `<choice>`, `<group>`, `<all>`, and corresponding terminal tags. Similarly to the element nodes, the nodes corresponding to the

above start tags store occurrence parameters which indicate how many times the whole structure must and may appear in an instance document. A node that must occur in the graph at least once is called *required*; a node that may not appear in the graph – *optional*.

A comparison is successful if all instances of the validated graph are also instances of the reference graph. An instance of a graph is a *spanning path* of its nodes. A spanning path covers all required nodes and any number of optional nodes in the order they are defined in the type. Note that instances of the reference graph do not have to be instances of the validated graph.

Different paths can be created by *forks* like alternative children of a `choice` structure, different permutations of elements within an `all` structure, or different final values of node occurrence parameters. The number of different paths is exponential to the number of forks, and, therefore, it is infeasible to analyse all of them. While each child element of a `choice` structure results in a new graph instance (a new spanning path), there are several other fork cases when it is very expensive to construct all instances of a graph and we choose to construct only a subset of them. For example, we do not compute permutations of elements in an `all` structure, but we plan to address this in the future work. Also, we do not create a separate graph instance for each possible value of an occurrence parameter. We only ensure that the validated element can occur no less and no more than the reference element. We take into account that a schema may have two or more elements with the same name and type directly following each other in a graph. For example, in Figure 4.3, the effective occurrence interval of any of the three `<foo>` elements is $[2,4] + [0,1] + [0,3] = [2,8]$. Due to the simplified way the occurrence interval is computed, the interval is always considered to be continuous, which is not true in practice. For example, Figure 4.4 shows a case where our algorithm computes a wrong occurrence interval: it reports $[0,4]$ for `<foo>` elements, but the correct interval is $[0,1] \cup [3,4]$. However, for our purposes, we find this approximation acceptable.

The number of spanning paths in our analysis is never infinite. The two potential infinite sources of spanning paths are occurrence parameters and type recursion. We choose to create only one spanning path per element regardless of its occurrence parameters. Also, we restrict the maximal occurrence values of structural tags. Type recursion is guarded by element labels and, therefore, cannot result in additional spanning paths. If recursion could be used without enclosing the type reference in an element label, the schema language would be context-free.

```
<sequence>
  <sequence>
    <element name="bar" type="nsl:barType">
    <element name="foo" type="nsl:fooType" minOccurs="2" maxOccurs="4">
  </sequence>
  <element name="foo" type="nsl:fooType" minOccurs="0" maxOccurs="1">
  <choice>
    <element name="bar" type="nsl:barType">
    <element name="foo" type="nsl:fooType" minOccurs="1" maxOccurs="3">
  </choice>
</sequence>
```

Figure 4.3: The effective occurrence interval of any of the three `<foo>` elements is $[2,8]$.

```
<sequence>
  <element name="foo" type="nsl:fooType" minOccurs="0" maxOccurs="1">
  <choice>
    <element name="bar" type="nsl:barType">
    <element name="foo" type="nsl:fooType" minOccurs="3" maxOccurs="3">
  </choice>
</sequence>
```

Figure 4.4: A case where our algorithm computes a wrong occurrence interval.

For context-free languages, the sub-type problem is undecidable, so it would not be possible to build an algorithm for that.

Fortunately, most of the hard cases do not occur often in practice: according to our study of Amazon, eBay, Google Search, and FedEx XML Schemas, the structure `all` and occurrence parameter values other than zero, one, and unbounded are very rarely used. In fact, no `minOccurs` attribute had a value other than zero or one, and only about two percents of `maxOccurs` attributes had a value different from one or unbounded. Table 4.2 provides a summary of extents to which major XML Schema language features are used in the above four schemas.

Feature	Number of occurrences in a schema			
	eBay	FedEx	Amazon	Google
element	2193	531	1053	22
any	0	86	0	0
attribute	25	0	12	2
anyAttribute	0	0	0	0
sequence	461	96	160	0
choice	0	14	0	0
group	0	0	0	0
all	0	0	0	3
minOccurs	1721	445	876	0
minOccurs="0"	1718	444	876	0
minOccurs (excl. "0", "1")	0	0	0	0
maxOccurs	194	445	134	0
maxOccurs="unbounded"	187	94	129	0
maxOccurs (excl. "1", "unbounded")	4	8	0	0
complexType	496	86	167	5
anonymous complexType	0	26	142	0
simpleType	190	144	14	0
anonymous simpleType	0	120	14	0
schema size (lines)	60000	6700	2000	200

Table 4.2: An overview of XML Schema language features used in eBay, FedEx, Amazon and Google Search schemas.

Also, it has been shown that the problem of numerical occurrence indicator comparison in regular expressions is NP-hard [25]. Since the XML Schema language subsumes regular expressions, this fact also applies to the comparison of its `minOccurs` and `maxOccurs` occurrence constraints.

The graph comparison algorithm is implemented as iteration with backtracking. It concurrently iterates over the nodes in both graphs trying to match them against each other. The fork points in the validated graph, which occur during a comparison, are saved together with the associated states of node iterators. A join path of matched nodes is complete when the ends of both graphs are reached. Then, the state of the most recent fork point is restored and the iterators

		Reference schema version							
		421	419	417	415	413	411	409	407
Validated schema version	407	x	x	x	x	x	x	✓	✓
	409	x	x	x	x	x	x	✓	
	411	✓	✓	✓	✓	✓	✓		
	413	✓	✓	✓	✓	✓			
	415	✓	✓	✓	✓				
	417	✓	✓	✓					
	419	✓	✓						
	421	✓							

Table 4.3: A summary of comparison results of several eBay schema versions. Pairs with a subtype relationship are marked by ✓. The versions 407 and 409 are not compatible with the rest of the versions because the child element order in several types of the version 411 has been changed and some child elements were removed altogether.

proceed from it. If, at some step of a comparison, no match can be found due to a wrong subpath previously taken at a fork in the reference graph, the algorithm tries to step back by discarding the latest pair of matched nodes. It continues to step back until either a fork with a yet unmatched alternative subpath is found in the reference graph or all matches of the current join path are discarded. In the later case, the comparison terminates reporting that an instance of the validated graph is not an instance of the reference graph. The pseudo-code of the comparison algorithm is provided in Appendix B. The algorithm is similar to the one in the XDuce language [13], which is described in the Related Work chapter.

4.1.3 Tests of the Algorithm

We have tested the schema comparison algorithm on eight versions of the eBay schema: 407, 409, 411, 413, 415, 417, 419 and 421. For any pair of versions, the newer one has all global elements of the older one, but a couple of types are slightly changed in each version. All eBay types are based on the `<sequence>` structure, and, therefore, a newer type is a subtype of an older one when it just inserts an optional element into its sequence or adds a required element to the end of the sequence. But when an element is removed, or the element order is changed, the

newer version is no longer a subtype of the older version. The algorithm had correctly identified all pairs of schemas that have a subtype relationship (Table 4.3). Recall that a reference schema is said to be a subtype of a validated schema if all possible instance documents of the validated schema are valid instances of the reference schema.

4.2 Dynamic Types

Consider a situation when a pointcut matches elements with different types, but with similar content. The author of the pointcut is only interested in a subset of the content common to all matched elements. One way to define the corresponding advice parameter is with the generic `xmlObject` type. However, in this case, the content of matched elements would not be accessible through `get` and `set` methods, and the developer would have to use the generic `selectChildren(QName)` method and/or the `XmlCursor` interface.

Doxpects provide a simple way to access and modify content common to matched elements of different types. The doxpect compiler can create a new, *dynamic*, XML type containing the common content of the matched elements. Effectively, the schema of the new type contains an intersection of the child element definitions according to the XML schemas of the matched elements. The schema of the new type is compiled into XMLBeans classes which have `get` and `set` methods for all common child elements of the matched elements.

The child elements are compared by names and types. We ignore the relative order of child elements and, therefore, provide an object-oriented subtyping for schema types. We choose to do it this way because dynamic types are only used in the object-oriented environment, where the ordering is not important. Moreover, enforcing the relative order would significantly restrict the applicability of dynamic types.

In fact, rather than using schemas to compute the list of elements to be included in a dynamic type, we could be using XMLBeans classes corresponding to the XML types of the matched elements. This is different from the modification type checking discussed in Section 4.1, where we cannot rely on the schema representation provided by XMLBeans classes because it is not sufficiently detailed to deal with an external service interface defined in XML Schema language. A dynamic type is an object-oriented paradigm used internally by a service, and, thus, we could take a shortcut by using XMLBeans classes and the Java Reflection API to compute a

list of dynamic type elements. However, it seems that this shortcut would not improve the performance. Therefore, we choose to reuse parts of the modification type checking framework in dynamic type generation.

To define a parameter with a dynamic type, the annotation `DynPointcut` must be used in place of the usual `Pointcut` annotation. The `DynPointcut` annotation has the same parameters and behaves similarly to the `Pointcut` annotation, but it also gives a hint to the compiler that a dynamic type should be created. The name of the dynamic type is arbitrary. With the `DynPointcut` annotation, the dynamic type is regenerated at each doxpect compilation. If one wishes to keep the current version of the generated type, the `DynPointcut` annotation should be replaced back with the `Pointcut` annotation.

For example, the advice `checkPackageRestrictions` in Figure 4.5 intercepts three types of request messages: `fdx:FDXRateAvailableServicesRequest`, `fdx:FDXRateRequest` and `fdx:FDXShipRequest`. At compile-time, the common elements of the three types are combined into a dynamic type `ShipOrRateRequestXmlBean`. At runtime, the content matched by the pointcut is parsed into a `ShipOrRateRequestXmlBean` object and passed to the advice as the `request` parameter. The advice uses the `request` object to check whether a package of the requested type may have the requested weight and dimensions. The advice initiates a response with an error if some restriction is not satisfied.

For situation when a dynamic type must have a union of child elements, doxpects provide the `DynUnionPointcut` annotation. It is similar to the `DynPointcut` annotations, but the schema of the new type contains a union of the child element definitions.

For instance, Figure 4.6 has an advice with a `DynUnionPointcut` annotation. This advice is similar to the advice in Figure 4.5, but its `ShipOrRateRequestXmlBean` class will provide `get` and `set` methods for all elements of the three types matched by the pointcut. At runtime, if the advice calls a `get` method of an element not present in the matched message, `null` is returned.

```
@Request
public void checkPackageRestrictions(
    @DynPointcut("fdx:FDXRateAvailableServicesRequest | " +
                "fdx:FDXRateRequest | " +
                "fdx:FDXShipRequest")
    ShipOrRateRequestXmlBean request)
{
    int weight = request.getWeight().intValue();
    int length = request.getDimensions().getLength().intValue();
    int width = request.getDimensions().getWidth().intValue();
    int height = request.getDimensions().getHeight().intValue();

    if (!checkPackageWeight(request.getPackaging(), weight))
        respondWithError( /* Details elided */ );

    if (!checkPackageDims(request.getPackaging(), length, height, width))
        respondWithError( /* Details elided */ );
}
```

Figure 4.5: An advice with a `DynPointcut` annotation and a dynamic type.

```
@Request
public void checkPackageRestrictions(
    @DynUnionPointcut("fdx:FDXRateAvailableServicesRequest | " +
                      "fdx:FDXRateRequest | " +
                      "fdx:FDXShipRequest")
    ShipOrRateRequestXmlBean request)
{
    /* Details elided */
}
```

Figure 4.6: An advice with a `DynUnionPointcut` annotation and a dynamic type.

Chapter 5

Implementation Details

In this chapter we discuss the implementation details. We start by presenting the two available compilers: one based on Java Mirror API and the Annotation Processing Tool and one based on the Java Reflection API. We then outline the compilation process and explain how to write a deployment descriptor for a doxpect handler. We follow with more details on the Reflection API compiler and a summary of a command-line interface. Finally, we conclude with some noteworthy details of the XMLBeans parser and describe the `DoxpectBase` class, which is a superclass of every doxpect.

5.1 Compilation Process

Doxpects have two compilers: one based on Java Mirror API and the Annotation Processing Tool and one based on the Java Reflection API. All prior discussions were in the context of the Mirror API compiler because it does not have many of the limitations of the Reflection API compiler, which will be described in Section 5.3.1.

The Mirror API compiler takes a doxpect source file and generates a *wrapper class* source file. Both source files are then compiled. The wrapper class does all the dirty work: it parses messages into XMLBeans, evaluates pointcuts, iterates over matched content (if the annotation `Each` is present), invokes doxpect methods, and performs message modifications. The Reflection API compiler takes a doxpect class file and generates a wrapper class source file, which is then compiled. Both compilers may be invoked either automatically by the `DoxpectHandler` class (Section 5.2) or manually from a command line (Section 5.3.2).

```
<handlerInfoChain>
  <handlerInfo classname="doxpect.handler.DoxpectHandler">
    <parameter name="requestClassNames"
      value="com.fedex.webservice.v20060404.doxpect.EncryptionHandler,
        com.fedex.webservice.v20060404.doxpect.ValidationHandler"/>
    <parameter name="responseClassNames"
      value="com.fedex.webservice.v20060404.doxpect.PostprocessHandler,
        com.fedex.webservice.v20060404.doxpect.EncryptionHandler"/>
    <parameter name="faultClassNames"
      value="com.fedex.webservice.v20060404.doxpect.FaultHandler" />
    <parameter name="outPackage"
      value="com.fedex.webservice.v20060404.doxpect" />
    <parameter name="srcInputDir"
      value="/apache/Tomcat5.5/webapps/axis/src/" />
    <parameter name="binOutputDir"
      value="/apache/Tomcat5.5/webapps/axis/WEB-INF/classes/" />
  </handlerInfo>
</handlerInfoChain>
```

Figure 5.1: A sample deployment descriptor of the `DoxpectHandler` for an Apache Axis server.

A `doxpect` compiler needs schema definition files in order to perform type checking, detect advice conflicts, generate dynamic types, etc. A `doxpect` specifies schema namespaces in its `include` annotation, but a schema is not always accessible at the location indicated in its namespace URI. The actual schema locations should be given in a `schema-locations.properties` file stored in a directory specified in the system `CLASSPATH` variable. The file should be formatted as a Java Properties file [19].

5.2 Doxpect Handler

The `doxpect` library provides a special JAX-RPC handler, `Doxpect Handler`, which may wrap any number of `doxpect` classes. Certain compilation parameters can be configured when `Doxpect Handler` is installed, or *deployed*, to the web container. The configuration information

is maintained in an XML file called a *deployment descriptor*. A sample descriptor of the `DoxpectHandler` for an Apache Axis server is shown in Figure 5.1.

The fully-qualified names of the doxpects to be executed for request, response, and fault messages are given in the handler parameters `requestClassNames`, `responseClassNames`, and `faultClassNames` correspondingly. If a doxpect wrapper class is older than the corresponding doxpect source file or does not exist, it is compiled dynamically by the `DoxpectHandler` class. The package to which the generated doxpect wrapper classes should belong must be given in the `outPackage` handler parameter. If the `outPackage` parameter is missing, the wrappers are generated in the same package as the doxpect classes.

The parameter `srcInputDir` specifies where to find input source files. This parameter is only used by the Mirror API compiler (to find doxpect source file), and, therefore, its presence indicates that this compiler should be used rather than the Reflection API one. The parameters `srcOutputDir` and `binOutputDir` specify where to place generated source and binary files correspondingly. Source files include Java source files and dynamic type schema definition files. The possible binary files are Java class files and schema binary files (.xsb) generated by the XMLBeans compiler. For the Mirror API compiler, both of these parameters are optional and default to the value of the `srcInputDir` parameter. For the Reflection API compiler, only the `srcOutputDir` is optional – it defaults to the value of the `binOutputDir` parameter.

5.3 Doxpect Compilers

5.3.1 Reflection API Compiler

The Reflection API compiler takes a compiled doxpect class as input, which misses some information about the original doxpect source. For example, the order of advices inside a doxpect is unknown. Therefore, to enable a relative ordering of advices of the same type, the annotations `Request`, `Response` and `Fault` should have a positive integer parameter which indicates the ordinal value of their advices. By default, this value is equal to one. Advices are invoked in the ascending order of their ordinal values. If two or more advices have the same number, their relative invocation order is unspecified.

Option	Meaning
<code>-r, -reflect</code>	Use the compiler based on Reflection API
<code>-srcInDir <directory></code>	Specify where to find input source files (required without <code>-reflect</code> , otherwise ignored)
<code>-binOutDir <directory></code>	Specify where to place generated class files (required with <code>-reflect</code> , otherwise defaults to the value of the <code>-srcInDir</code> parameter)
<code>-srcOutDir <directory></code>	Specify where to place generated source files. An optional parameter. With <code>-reflect</code> , defaults to the value of <code>-binOutDir</code> ; otherwise defaults to the value of <code>-srcInDir</code> .
<code>-p <package name></code>	Specify package for generated files
<code>-package <package name></code>	
<code>-f, -forceRecompile</code>	Force recompilation even when the wrapper class files are up-to-date
<code>-noXA, -noXPathAnalysis</code>	Do not compare the types of potential XPath matches against the actual advice parameter type
<code>-noCA, -noConfAnalysis</code>	Do not perform advice conflict checking
<code>-noRA, -noReplaceAnalysis</code>	Do not perform replacement/modification type checking
<code>-h, -help</code>	Print a synopsis of standard options

Table 5.1: The possible options for the command-line Doxpects compiler.

Values of type parameters are also lost during a compilation [16]. Without this information, the compiler does not know the component type of advice parameters with `vector` types. Therefore, the component types should be specified in the parameter-level `Component` annotation.

Finally, it is impossible to have dynamic types with the Reflection API compiler because its input doxpect class file cannot be created by the Java compiler in the first place due to the missing definition of the dynamic class.

5.3.2 Command-Line Interface

The command-line interface can be invoked with the following command

```
java doxpect.Compiler <options> <qualified class names>
```

where possible options are described in Table 5.1.

5.4 Connected XMLBeans and the `DoxpectBase` Class

When an XML element is parsed by an XMLBeans factory as part of a SOAP message, the created bean object is typed in the context of its parent: if the element is improperly placed or is not allowed by the schema of the parent element, it is represented by a generic `XmlObject` bean regardless of its XML name and type. However, if an element is parsed on its own and not as part of a larger element, it is always typed according to its XML name and type. There are many situations when we want to process XML elements which do not conform strictly to schemas. For example, the XML content encryption is rarely foreseen in schemas, which prevents encrypted messages from conformance. For example, in Figure 5.2, when the `foo` element in the instance document is parsed, the `bar` element is parsed into an `XmlObject` object because it is not a valid child of `foo` according to the `foo`'s definition. However, parsing the `bar` element on its own would result in a specialized `BarTypeXmlBean` object.

A `doxpect` wrapper parses improperly placed or disallowed elements separately to get them properly typed. However, such beans have no information about their parents, which is often useful information. The `DoxpectBase` class, which is a superclass of every `doxpect`, provides methods to retrieve beans parsed in the context of the whole message and, therefore, connected to their parents. To make things easier, both untyped and typed beans are accessible in this way. If one wants to make sure that a parameter passed to an advice is a bean (or an array of beans) connected to the rest of the message then a `getConnectioned()` method has to be called. The method takes an `XmlObject` (or an `XmlObject[]` for arrays) and returns a connected version of its parameter. The `DoxpectBase` class also provides access to the `doxpect` handler parameters and to the current message context.

XML Schema	XML Instance
<pre> <schema elementFormDefault="qualified" targetNamespace="http://domain.com" xmlns="http://www.w3.org/2001/XMLSchema" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://domain.com"> <element name="foo"> <complexType> <sequence> <element name="abc" type="ns1:abcType"> </sequence> </complexType> </element> <element name="bar" type="ns1:barType "> <!-- Other details are elided --> </schema> </pre>	<pre> <ns1:foo> <ns1:bar> </ns1:foo> </pre>

Figure 5.2: A sample XML instance not conforming to its XML Schema.

```

@Response
public void encrypt(@Pointcut("//fdx:Address | //fdx:CreditCard")
  XmlObject[] toEncrypt)
{
  // Make sure the beans are connected to the rest of the document
  XmlObject[] toEncrypt = getConnected(toEncrypt);
  // Encrypt
  try {
    DoxpectSecurity wss4j = new DoxpectSecurity(securityProps);
    wss4j.prepareSender(getMessageContext());
    wss4j.encrypt(toEncrypt);
  } catch (WSSecurityException se) { /* Details elided */ }
}

```

Figure 5.3: An encryption advice encrypts all `fdx:Address` and `fdx:CreditCard` body elements.

For example, the advice in Figure 5.3 encrypts all `fdx:Address` and `fdx:CreditCard` body elements. The encryption library cannot encrypt disconnected element and, therefore, we call the `getConnected()` method to get the connected version of the advice parameter `toEncrypt`.

Chapter 6

Related Work

The eXtensible Stylesheet Language Transformation (XSLT) is the predominant domain-specific language for XML transformation [6,22]. XSLT is a purely functional language based on pattern matching and template instantiation. XSLT supports modular transformations through the delegation of sub-tasks between templates. Similarly to Doxpects, it uses XPath for content matching. Unlike Doxpects, XSLT can transform XML documents into any content type (e.g. L^AT_EX, RTF, or plain text). However, this makes it impossible to express the interface of a template in terms of its output. Another important limitation of XSLT is the inability to transform the original document; rather, a new document is created based on the content of the original one.

XDuce is a statically typed functional XML processing language that takes XML documents as primitive values and represents them internally as sequences of nodes [13,14]. The document schema, represented by DTD, is described statically by regular expression types against which instance documents are matched. It supports a local form of type inference where types are specified explicitly for function arguments but inferred for pattern matching. In its current version, the XDuce type system does not model element attributes or unordered data.

JWIG is an extension of Java that, among other features, provides a mechanism for construction of XML documents using *XML templates* and *plug operations* [7,8]. It uses a technique – XACT – that provides an integration of XML values and operations for XML transformation into Java [26]. XACT guarantees type safety of the transformations: the approach is based on dataflow analysis rather than traditional type systems. The validity analysis uses the DSD2 schema language. The XML values are represented as XML templates, which are well-formed XML fragments containing named *gaps* where other templates or strings may be

plugged. The gaps may appear in place of element or attribute values. XPath is used for selecting fragments of XML values.

Both XDuce and Jwig perform a more sophisticated type safety analysis than does the Doxpects framework. However, unlike Doxpects, they do not have an explicit interface for transformations. Also, they make no contribution to the design, traceability, and maintenance of crosscutting document-oriented concerns. Tozawa and Hagiya compare algorithms for XML schema subtype checking and show how their algorithm outperforms XDuce [29].

XJ proposes mechanisms for integration of XML as a first-class construct into Java [12]. It uses the XML Schema and XPath standards, and supports in-place updates of XML data. The language syntax is based on Java, but has special syntactic constructs to represent XPath expressions and access XML data. At compile-time, an XPath expression is abstractly evaluated on a XML Schema to infer the type such that an evaluation of the XPath expression on any document conforming to the XML Schema results in an instance of this type. The XJ compiler emits the standard Java code that accesses XML data using DOM.

Chapter 7

Conclusion

We have described and demonstrated by examples an approach to modularizing crosscutting document-oriented concerns in the web service setting. The previous approach using handlers inadequately addressed important software engineering practices such as “programming to an interface”. This thesis contributes a further step towards providing a clean mapping from design to implementation of handlers with a useful static checking which takes advantage of a well specified interface.

The development of the Doxpects language has been primarily example driven. We have focused our attention on implementing those features of the language which addressed the programming difficulties encountered in those examples.

We demonstrated how a handler in the Doxpects framework can intercept, filter, and process messages. We explained message filtering, but could not provide a complete example where filtering by service endpoint URI would be used. We feel that this kind of message filtering would be more useful in Doxpects applied to more lightweight standards that use XML over HTTP without an additional messaging layer such as SOAP. We are looking into adapting the Doxpects framework to simple interfaces such as REST.

Due to the rare use of attributes in the commercial schemas that we used in our study, the current version of the language cannot bind XML attributes to advice parameters. In other words, pointcuts can only match XML elements. We may add this feature into the language later.

In our language, the insertion transformation is represented by a single advice parameter which is bound to the inserted content. Therefore, the matched content is not made directly available to the advice. It could be made available if the transformation was represented by a

pair of parameters as it is done in the replacement transformation. However, we did not have enough motivation to implement it this way because we could not find enough examples where this would be necessary.

We believe that our compile-time analysis of advice interactions will be quite useful for systems with large collections of handlers applied to messages. Even though the analysis reasons about the transformations on the schemas symbolically rather than manipulating specific document instances, we believe that it performs well. The current implementation can perform interaction checking in the context of a single doxpect and not yet across doxpects, although we feel that an extension will be straightforward.

We showed how a request advice can terminate a request flow and initiate a response. However, this feature could not be implemented without Axis-specific API and, therefore, it might be useful only for this platform.

We believe that our compile-time transformation sufficiency check will be useful for services which must be kept backwards-compatible with all previous versions of their XML Schemas. Our schema comparison algorithm has a very limited support of the structure `all` and facet `pattern` because they are hard to compare and are not very common in practice. Specifically, they were not used in our main sources of examples: FedEx and eBay XML Schemas. In the future, the algorithm could be improved to better support them.

We introduced dynamic types, which provide a simple way to access and mutate content common to matched elements. We believe that this feature will be very useful for handlers implementing crosscutting concerns and intercepting message elements of different types.

We described the two available compilers: one based on Java Mirror API and the Annotation Processing Tool and one based on the Java Reflection API. The first one takes doxpect source files, the second – doxpect class files. The second compiler does not support all features of the language and we expect it to be used only when the doxpect source files are unavailable.

Bibliography

- [1] Berners-Lee, T., R. Fielding, and L. Masinter, RFC 3986: Uniform Resource Identifiers (URI): Generic Syntax. <http://rfc.net/rfc3986.html>, 2005.
- [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] Boag, S., et al., XQuery 1.0: An XML Query Language. W3C Candidate Recommendation, <http://www.w3.org/TR/xquery/>, 2006.
- [4] Bray, T., et al., Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, <http://www.w3.org/TR/REC-xml/>, 2004.
- [5] Clark, J., XML Path Language (XPath) Version 1.0. W3C Recommendation, <http://www.w3.org/TR/xpath>, 1999.
- [6] Clark, J., XSL Transformations (XSLT) Specification. W3C Recommendation, <http://www.w3.org/TR/xslt/>, 1999.
- [7] Christensen A. S., C. Kirkegaard, and A. Møller, A Runtime System for XML Transformations in Java. In *Proceedings of the Second International XML Database Symposium*, 2004.
- [8] Christensen, A. S., A. Møller, and M. I. Schwartzbach, Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 6, pages. 814-875, 2003.
- [9] Common Object Request Broker Architecture: Core Specification, Version 3.0.3. <http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf>, 2004.
- [10] Fielding, R. T., Architectural Styles and the Design of Network-based Software Architectures. PhD Dissertation, University of California, Irvine, <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>, 2000.

-
- [11] Fontanella, J. and R. Saia, SOA in IT Benchmark Report. Aberdeen Group, 2005.
- [12] Harren, M., et al. XJ: Integration of XML processing into Java. Technical Report rc23007, IBM Research, 2003.
- [13] Hosoya H. and B. C. Pierce. XDuce: A Typed XML Processing Language. In *Proceedings of International Workshop on the Web and Databases*, 2000
- [14] Hosoya, H. and B. C. Pierce. XDuce: A Typed XML Processing Language. *Transactions on Internet Technology*, 3(2), 2003.
- [15] Imamura T., B. Dillaway, and E. Simon, XML Encryption Syntax and Processing. W3C Recommendation, <http://www.w3.org/TR/xmlenc-core/>, 2002.
- [16] Sun Microsystems, Generics in the Java Programming Language. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>, 2005.
- [17] Sun Microsystems, Java API for XML-Based RPC. <http://java.sun.com/webservices/jaxrpc/index.jsp>.
- [18] Sun Microsystems, Java Architecture for XML Binding (JAXB). <http://java.sun.com/xml/jaxb/>, 2002.
- [19] Sun Microsystems, Java Properties Files. <http://java.sun.com/docs/books/tutorial/118n/resbundle/propfile.html>
- [20] Sun Microsystems, Java Remote Method Invocation (Java RMI). <http://java.sun.com/products/jdk/rmi/>, 2003.
- [21] Kastner, P., Enterprise Service Bus: The Foundation of Successful Service-Oriented Architecture. Aberdeen Group, 2006.
- [22] Kay, M., XSL Transformations (XSLT) Version 2.0, W3C Candidate Recommendation, <http://www.w3.org/TR/xslt20/>, 2005.
- [23] Keen, M. et al., Patterns: Implementing an SOA using an Enterprise Service Bus. IBM Redbooks, 2004.
- [24] Kiczales, G., et al., An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327-353. Springer, 2001.

-
- [25] Kilpeläinen, P. and R. Tuhkanen. Regular Expressions with Numerical Occurrence Indicators – Preliminary Results. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools*, pages 163-173. University of Kuopio, Department of Computer Science, 2003.
- [26] Kirkegaard, C., Moller, A. and Schwartzback, M. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 2003.
- [27] Møller, A., Document Structure Description 2.0. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7, <http://www.brics.dk/DSD/>, 2002.
- [28] SOAP Version 1.2. W3C Recommendation, <http://www.w3.org/TR/soap/>, 2003.
- [29] Tozawa, A., and Hagiya, M. XML schema containment checking based on semi-implicit techniques. In *Proceedings of the Conference on Implementation and Application of Automata*, 2003.
- [30] Wohlstadter E. and K. De Volder, Doxpects: Aspects supporting XML transformation interfaces. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2006.
- [31] XML Schema. W3C Recommendation, <http://www.w3.org/XML/Schema>, 2004.
- [32] XMLBeans. Version 2.1. <http://xmlbeans.apache.org/>, 2005.

Appendix A

Filtering Messages by Service Endpoints

This class-level annotation² defines a list of URI regular expressions. Only messages targeted to or originating from the services matched by one of the regular expressions are processed by the doxpect to which the `Target` annotation belongs. To define a regular expression for services to which the doxpect does not apply, one should prepend the expression with a ‘!’ sign.

For example, the following expression allows a doxpect to be applied to messages targeted to or originating from `shipping.com` and `shipping.net`, but not from `shipping.org`.

```
@Target({"shipping\\.((com)|(net))", "!shipping\\.org"})
```

Note, that some common URI symbols, such as ‘.’ and ‘?’, have special meaning in regular expressions and must be escaped.

The syntax of a URI is `[scheme:][//authority][path][?query][#fragment]`. If a path, query, or fragment part is not specified in a pattern then any string is accepted in its place. Hence, for example, the above pattern also matches messages to and from `shipping.com/services/track`.

While the query and fragment parts are rarely used in SOAP-based web-services, they are extensively used in simpler HTML- or XML-based architectures like REST [10]. Therefore, we chose to support these URI parts in `Target` patterns.

A pattern of a query part may contain any number of parameter name-value pairs joined by ‘&’ signs. In a pattern, a parameter value is considered to be a regular expression, but a

² Alternatively, could be implemented at the method level

parameter name is always a pure string and must comply with the generic syntax defined in RFC 3986 [1]. For example, the following pattern matches a URI `shipping.com/getpage.html` which query path has the parameter `id` set to any string with prefix "user" and the parameter `action` set to either "shipRequest" or "shipReply".

```
"shipping\\.com/getpage\\.html\\?id=user.*&action=ship((Request)|(Reply))"
```

The order of parameters in a pattern does not matter. Therefore, the above pattern matches both `shipping.com/getpage.html?id=user325&action=shipRequest` and `shipping.com/getpage.html?action=shipRequest&id=user325`.

Appendix B

Graph Comparison Algorithm

What follows is a pseudo-code of the graph comparison algorithm outlined in Section 4.1.2.

```

// A stack item is a state of a comparison. A new state is created when
// either two element/wildcard nodes are matched or a structural node is
// stepped over in either graph.
Stack matchStack = new Stack();

// Stack of fork points. A point is identified by an index in the matchStack.
// The comparison state with the given index corresponds to the fork point.
Stack forkPoints = new Stack();

// True if a step was performed and no backtracking is necessary
boolean madeStep = false;

// True if the algorithm has just backtracked
boolean justBacktracked = false;

Node vNode = vGraph.getFirstNode(); // Current node in the validated graph
Node rNode = rGraph.getFirstNode(); // Current node in the reference graph

// When true forces the current node in the validated graph
// to be matched again
// (e.g. when the node was matched less times than its "minOccurs" value)
boolean mustMatchVNode = false;

// Same for the current node in the reference graph
boolean mustMatchRNode = false;

```

```

boolean compare()
{
while (true) {
    madeStep = false;

    if (!justBacktracked) {
        // Repeat for each graph:
        // If the current node in the graph is a all/sequence/choice head
        // and was already used, then reset occurrences of its
        // child nodes to their "maxOccurs."
        /* code elided */

        // Repeat for each graph:
        // If the current node in the graph is a sequence/choice member
        // and wasn't used enough times (less than its "minOccurs") then set
        // the corresponding flag (mustMatchVNode or mustMatchRNode).
        /* code elided */

        // Repeat for each graph:
        // If the current node in the graph is a choice/all end,
        // ensure that all children were used enough times.
        // If this condition is not satisfied, step back by discarding
        // the latest item on the matchStack.
        foreach(node: vNode, rNode) {
            if (node.isStructureEnd()) {
                Node head = node.getStructureHead();
                if ((head.isAllHead() || head.isChoiceHead())
                    && !ensureChildrenMinOccurs(head)) {
                    // The backtrackOneStepBack method returns false
                    // if the matchStack is empty
                    if (!backtrackOneStepBack(matchStack)) return false;
                    justBacktracked = true;
                }
            }
        }

        // Get iterator over nodes that follow the current validated node
        vIter = vNode.getIteratorOverNextNodes();
    }
}

```

```

// If vNode is a "choice" or "all" structure with more than one child,
// remember to return to it later to try other paths
if ((vNode.isChoice() || vNode.isAll ()) && vIter.size() > 1)
    forkPoints.push(new ForkPoint(matchStack.size()));

while (vIter.hasNext()) {
    vNext = cIter.next();

    // If mustMatchVNode is true, force reuse of vNode if necessary.
    // If vNode is the end of a structure, the whole structure has
    // to be reused and we want vNext to point to the head of the structure.
    // If vNode is not a structural node, then make vNext point to vNode.
    if (mustMatchCCur &&
        ((vNode.isStructureEnd() && vNext != vNode.getStructureHead()) ||
         (!vNode.isStructural() && vNext != vNode))) {
        vNext = /* code elided */
    }

    // If vNext wasn't used more times than its "maxOccurs"
    if (vNext.canUse()) {
        // If a structural node, skip it and push the current state
        // to matchStack
        if (vNext.isStructural() && !vNext.isEndOfGraph()) {

            // If vNext is a head node of a structure already used then the
            // occurrences of child nodes are restored to their original values.
            /* code elided */

            // However, to be able to restore the current state later (during
            // backtracking), the current occurrence values are saved as part of
            // the state pushed to the matchStack.
            matchStack.push(getCurrentState());
            justBacktracked = false;
            vNext.use(); // increase occurrence count by one
            vNode = vNext;
            vIter = vNode.getIteratorOverNextNodes();
            continue;
        }
    }
}

```

```

// If no backtracking occurred, construct a fresh copy of
// the iterator over the next nodes of rNode. Otherwise,
// we continue to use the popped iterator.
if (!justBacktracked)
    rIter = rNode.getIteratorOverNextNodes();

justBacktracked = false;
while (true) {
    // If no more elements in rIter, try to go forward
    // (skip optional nodes) or backtrack
    if (!rIter.hasNext()) {
        // If in sequence and the next node is optional, try to skip it
        if (rNode.isSequenceMember()) {
            // Upon a successful skip, rNode and rIter are changed
            skipOptionalNodeInSequence(rNode,rIter);
        }
        // If couldn't skip, try to backtrack to
        // the previous unpaired (structural) rNode
        if (!rIter.hasNext()) {
            // Peek from the matchStack. Recall that items of this stack are
            // either pairs of matched element/wildcard nodes or unpaired
            // structural nodes, which were skipped. If the picked item has
            // no vNode component then it's a skipped structural node from
            // the reference graph.
            // Continue to pop from the stack while popped items correspond
            // to skipped structural node from the reference graph
            // with exhausted iterators.
            ComparisonState peek = matchStack.peek();
            while (peek.vNode() == null && !peek.rIter().hasNext()) {
                if (!backtrackOneStepBack(matchStack)) return false;
                peek = matchStack.peek();
            }
            // If we found a stack item with a structural rNode and a
            // non-exhausted rIter, pop it
            if (peek.vNode() == null && peek.rIter().hasNext())
                if (!backtrackOneStepBack(matchStack)) return false;
        }
    }
}

```

```
// If rIter has no more items
if (!rIter.hasNext()) break;
rNext = rIter.next();

// If mustMatchRNode is true, force reuse of rNode if necessary
// (similar to forcing a reuse of vNode).
// If rNode is the end of a structure, the whole structure has
// to be reused and we want rNext to point to the head of the
// structure.
// If rNode is not a structural node, then we want rNext
// to point to rNode.
if (mustMatchRNode &&
    ((rNode.isStructureEnd() && rNext != rNode.getStructureHead()) ||
     (!rNode.isStructural() && rNext != rNode))) {
    rNext = /* code elided */
}

// If we've reached the ends of both graphs
if (vNext.isEndOfGraph() && rNext.isEndOfGraph()) {

    // See if there are fork points in the candidate graph that
    // we have to backtrack to
    if (forkPoints.empty()) {
        return true; // comparison succeeded
    } else {
        ForkPoint fork = backtrackPoints.peek();

        // Cut matchStack so that the state corresponding to
        // the fork point is at the top
        matchStack.setSize(fork.stackIndex + 1);

        // Now restore the state corresponding to the fork point
        if (!backtrackOneStepBack(matchStack)) return false;
        madeStep = true;
        break;
    }
}
```

```

// If rNext can be used (was used less than its maxOccurs)
if (rNext.canUse()) {
    // If rNext is a structural node, but not the end node of
    // the reference graph, push it to the stack and skip
    // (similar to the skipping of a structural vNext)
    if (rNext.isStructural() && !rNext.isEndOfGraph()) {

        // If rNext is a head node of a structure already used, then the
        // occurrences of child nodes are restored to their original
        // values.
        /* code elided */

        // However, to be able to restore the current state later (during
        // backtracking), the current occurrence values are saved as part
        // of the state pushed to the matchStack.
        matchStack.push(getCurrentState());
        justBacktracked = false;
        rNext.use(); // Increase occurrence count by one
        rNode = rNext;
        rIter = rNode.getIteratorOverNextNodes();
        continue;
    }

    // At this point we have non- structural vNext and rNext nodes that
    // we are now going to compare.
    // The method compareNodes compares names and types of element
    // nodes and analyses namespace containment of wildcard nodes.
    // The method checkPossibleMatchCounts is described below
    if (compareNodes(vNext, rNext) &&
        checkPossibleMatchCounts(vNext, rNext)) {

        // If nodes are equal we decrease their remaining occurrences
        // by the maximal possible common amount.
        int useCount = adjustOccurrences(vNext, rNext);

        // Push the matched pair along with the current state of the
        // comparison to the stack
        matchStack.push(getCurrentState());

```

```

        // Advance in both graphs
        vNode = vNext;
        rNode = rNext;
        madeStep = true;
        break; // Break from the inner infinite loop
    }
}
}
}
}
// If could match the current vNext to something in the reference graph
// then break from the outer infinite loop and prepare for the next step.
if (madeStep) break;
}
// If could match the current vNext to something in the reference graph,
// continue to the next step.
if (madeStep) continue;

// else try to backtrack one step
if (!backtrackOneStepBack(matchStack)) return false;
}

boolean checkPossibleMatchCounts(Node vNode, Node rNode)
{
    foreach(Node node: vNode, rNode) {
        // See if node has neighbour nodes which have the same name and type
        // and which elements/wildcards may appear directly before or after
        // the node's element/wildcard in an instance document.
        /* code elided */

        // Constructs an interval of possible cumulative occurrences that
        // the given name-type combination might have.
        /* code elided */
    }
    // Ensure that the interval of vNode lies within the rNode interval.
    /* code elided */
}

```