Interfaces for Web Service Intermediaries

Sara Forghanizadeh, Ivan Minevskiy, and Eric Wohlstadter
University of British Columbia
Software Practices Lab
Vancouver, British Columbia Canada
{forghani,ivan,wohlstad}@cs.ubc.ca

Abstract

The use of XML as a format for message exchange makes Web services well suited for composition of heterogeneous components. However, since the schema of these messages must be understood by all cooperating services, interoperability is still a significant problem. There is often some level of semantic overlap between schemas even when there is no syntactic match. We are interested in supporting interoperability through the use of partial interface adaptation. Using Web services, it is becoming common for clients to share adaptations provided by a Web service intermediary. Given a Web service and an intermediary, we generate the interface that can be used by a client taking into account what can happen at the intermediary. We provide examples using publicly available service schemas and a performance evaluation for the purpose of validating the usefulness of our approach.

1. Introduction

Web services are fluorishing on the web as an important part of the IT infrastructure, enabling exchange of information for a wide variety of domains. Nowadays, almost all major Internet sites provide Web service based solutions (e.g. Google, eBay, Paypal, Amazon, Fedex). The use of XML as a format for message exchange makes Web services well suited for composition of heterogeneous components (i.e. those programmed in potentially different languages and OS's). The structure (or schema) of these messages defines the service's interface and is often described by an interface definition language (e.g. WSDL). However, since the schema of these messages must be understood by all cooperating services, interoperability is still a significant problem. There is often *some level of semantic overlap* between schemas even when there is no syntactic match.

We are interested in supporting interoperability through the use of *partial interface adaptation*. This approach is motivated by the desire to keep client applications simple by only having to understand one particular schema format for those places where multiple schemas overlap. Unforunately, the space where schemas overlap does not always include all those elements which make up an entire Web service operation. In this case, partial interface adaptation is required to prevent an enterprise from becoming polylingual. We focus on client-side development as there are potentially thousands of clients using any one service and the onus is usually on the client to work with whatever interface is exported by a service.

Previous work on interface adaptation [14, 8, 17] has solved many problems related to the reuse of components in contexts that were not originally anticipated. Although there are a variety of approaches, the core problem is to take an interface provided by a component, an interface required by a component consumer, and to help produce an adaptor which adapts between the required and provided interface. This helps a programmer who takes two components off-the-shelf and needs a way to connect them. This approach is illustrated in Figure 1.a.

We seek to extend this line of work for use in a Web services setting. Using Web services, it is becoming common for clients inside of an intra-enterprise network to share adaptations provided by a Web service *intermediary*[6]. An *intermediary* is simply a network proxy which provides added non-functional features (e.g. encryption, logging, etc.) in the service message flow; in our work we are specially concerned with the feature of message transformation. Transformations follow an implicit-invocation [5] architecture, implemented using event-handlers which register for particular XML elements using a query language such as XPath.

Middleware vendors for Web services now specifically market intermediary products (under the rubric Enterprise Service Bus [2], e.g. IONA Artix, BEA AquaLogic, Sonic ESB). Transformations applied by an intermediary are intended to shield any client in the enterprise from the heterogeneity of the outside world allowing the enterprise to remain fixed on one common schema. Notice that this is

not so much a problem for servers inside an enterprise because they would most likely require any outside clients to conform to the enterprise schema.

Consider a company that uses Web services to schedule shipments with companies such as Fedex and UPS. Naturally, Fedex and UPS share many semantic elements in common and perhaps the company uses its own internal representation as well. In this case, the intermediary for the company would be programmed to transform to and from the company schema to Fedex and UPS. Now, when a programmer at the company starts to write client code to communicate with Fedex or UPS, they will need to know what the Fedex and UPS interfaces (WSDL) look like, after the semantically overlapping elements have been replaced with their internal schema. This way they can reason about the composition of their intermediary with either the Fedex or UPS service. We call this an *augmented schema*, and the act of retrieving an augmented schema is called *interface discovery*.

So we take a different approach from previous work: given a Web service, and an intermediary, we generate the interface that can be used by a client to talk to the service, taking into account what can happen at the intermediary. Essentially, we have chosen that the adaptations are known variables and the client interface is an unknown variable instead of vice versa. In previous interface adaptation work, it was assumed the client component was already written and that there needed to be a complete mapping from the old interface to the new one. Our research looks at this problem when neither of these assumptions holds. In many cases, the client code would not even have been written yet but the internal schemas used by the comapny would have. Our setup is then shown in Figure 1.b.

Our contribution is to show how client software can take advantage of static interfaces that take into account the composition between intermediaries and services. We describe our implementation of an intermediary interface discovery tool and middleware support which extends work on XML programming languages [9] and tree automata [3].

The rest of the paper is as follows: Section 2 presents a motivating scenario, Section 3 describes an overview of the approach, Section 4 describes useful background material, Section 5 presents technical details, Section 6 presents examples in more details, Section 7 is the evaluation, Section 8 presents related work and Section 9 concludes the paper.

2. Motivating Scenarios

Differences in schemas between services can create headaches for programmers tasked with the challenge of composing such services. We consider two causes for differences between schemas: schemas from different service vendors and schema version evolution. We provide examples for both using publicly available service schemas for the purpose of validating the usefulness of our approach.

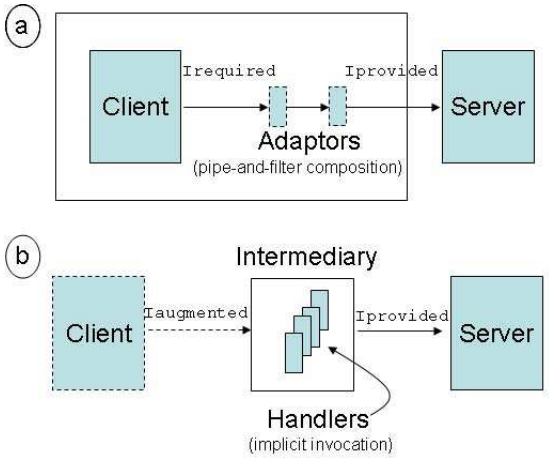

Figure 1. (a.) Component interface adaptation versus (b.) Web service intermediary interface discovery. Dotted lines represent the unknown variable. The large box in (a.) represents co-location (could also be at server).

2.1 Web service composition

Suppose a company works with the eBay Web service to market widgets to consumers. Consumers might choose between different options for shipping, such as Fedex and UPS.

Let's examine a typical process flow for the company. First, they send a request such as `GetSellerTransactions` to eBay. The response contains the transaction information, including a shipping address element and some other information such as a choice of shipping options (in this case Fedex).

Next, the company wants to verify the address of the buyer before scheduling with Fedex. Unfortunately, the Fedex Web service does not include a separate operation for validating address information, but UPS does. So, they go ahead and send an `AddressValidation` request to UPS. A part of the information in this message is specific to UPS, such as the UPS account number, and needs to be specified by the company. On the other hand the destination address can be extracted from the response message received from eBay and simply routed to UPS. So we can hope to adapt only part of the interface which includes the address.

So the seller must transform the eBay address to a UPS format before validating the address and then additionally, they must transform it to Fedex before sending the final request in the flow (e.g. `FDXShipRequest`).

Now, consider that modern enterprises are often built up of an internal network of systems connected by a business process workflow. Many pieces of the network may make use of external services and they are likely to benefit from many of the same transformations. Locating the transformations at an intermediary provides a single point to manage communications with the outside. In the case that some outside service changes its schema, we may only require an update at one location. Also, if the company adds a new business partner, a specialist in the schema semantics can add value to the entire system by adding new transformations to the intermediary. However, without additional tool and middleware support, this would break the sound engineering process of writing software against statically defined interfaces.

2.2 Schema evolution

The eBay Web service schema [1] has gone through numerous revisions since the service was started. At one point, a major change was introduced such that versions before the change are called the "legacy schema" and versions after are called the "unified schema" (because changes were required to integrate the schema with some partner Web services). eBay provides a complete set of mappings covering 50 operations between the legacy and unified schemas for each element type. Here we consider a scenerio where an intermediary has been deployed, which provides transformation between the legacy and unified schemas using these mappings.

Now, even when complete interoperability through transformation is achieved, vendors will inevitably roll out new operations to provide more features to clients. For example, `GetRecommendations` was added so that clients could receive feedback about common mistakes in the description of their auctioned `Items`. Since these new operations have no semantic counterparts in the legacy schema, it makes sense that some of the elements cannot be covered by any transformation from a legacy element. Although, for clients that wish to begin taking advantage of new operations quickly, they may wish to make use of any partial message transformations.

Consider this motivation for the use of partial message transformations. A core piece of the eBay schema is the `Item` element (and is used by `GetRecommendations`). This data structure has 75 elements (direct children of the root) and is 5 levels deep. A client system making use of the legacy schema over a period of time may come to depend on the structure of this element in many ways. For example, one part of the system written in Java may store particular `Items` in a local XML database, another part of the system written in XQuery might process the database to provide reports. So, upgrading the entire system to the new `Item` element is not a simple task, and it would be advantageous if evolving the application to make use of new eBay features is not held up while the rest of the system is upgraded. So, when evolving to new operations, it would help a programmer to know how much of the operation's schema is covered by existing transformations.

3. Overview of the Approach

Our approach is divided into two stages: interface discovery and the intermediary middleware runtime.

3.1 Interface Discovery

One technical challenge is to return a new schema to the client, the *augmented schema* (described in Section 5.2.2), which is an extended version of the server schema according to the transformations an intermediary can make. This requires that programmers assign a *transformation interface* to each handler on an intermediary to show the input and output types of that handler. Our tool adapts a *tree rewriting* algorithm described in Section 5.2.1. The inputs and outputs to the tool are shown in Figure 2. A digest is created which traces the execution of the tool for the purpose of runtime handler dispatching described in Section 5.3.

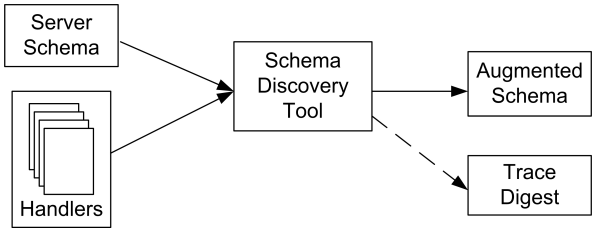

Figure 2. The schema discovery tool produces the augmented schema, given the set of handlers and the server schema. A digest is created ffor use at runtime.

3.2 Intermediary Message Processing

At runtime our middleware must select the set of handlers to be executed (and determine their order) when the intermediary receives a message from a client.

This stage is described in Section 5.3. When client messages are validated (i.e. determined to be of a particular type) at run-time, the middleware decides which handlers should be executed - and in what order - to transform the message type to the type expected by the server schema. The input and output of the run-time part of the algorithm are shown in figure 3.

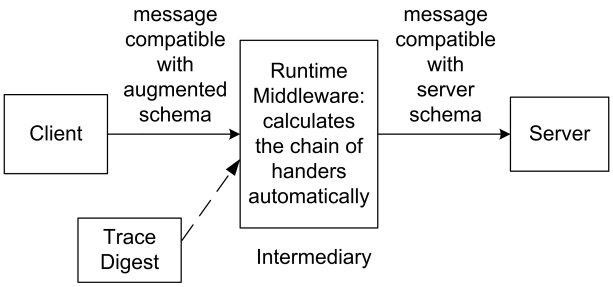

Figure 3. The intermediary calculates the handlers and their order to make the message compatible with the server schema using the digest dervied at development-time.

To provide a better understanding of the technical details, we provide a short background to some research from which we have derived our implementation.

4. Background

4.1 Handlers

In their simplest form, intermediaries are often a repository of message handlers (similar to interceptors [19] or connector wrappers [15]) which read and write *elements* of XML-based messages. When a message containing a particular type of element flows through the intermediary, the handler acts upon that element, using a form of implicit invocation. We call this *element-wise* transformation. For example, a handler might be responsible for just the translation of elements representing Addresses in two schemas with disparate formats.

So handlers are often programmed at the granularity of individual XML elements and not just at the course-grained message level. This practice helps make handlers generic and reusable because the transformations made on particular elements can be used no matter what message context the elements appear in. Separation of concerns is achieved in an aspect-oriented [16, 10] fashion because a handler is only concerned with certain element types and these element types crosscut (appear in the context of) a variety of message types.

4.2 Regular Expression Types

Recently, interest in the integration of programming languages and XML has led to renewed interest in *structural typing*. Unlike popular OO languages, under this discipline a sub-type relationship holds for two types if the value set of one subsumes the other. This is different from typing in standard OO languages where a sub-type relationship needs to be explicitly declared. OO typing makes sense in the context where classes are used to hide the representation details of implementations behind a well defined interface. However, in the context of document construction and transformation, it is overly restrictive and severely limiting as demonstrated in the XDuce [9] project.

In XDuce, the schemas of documents are described by combining XML elements using the standard regular expression operators on sequences of elements. These operators are as follows (which we have adopted from their research as well):

```
T ::= ()     empty sequence
      l[T]   label
      T,T    concatenation
      T|T    union
      T*     repetition
      T?     optional
```

So in XDuce given two types,

type $P1$ = Person[Name[*string*] Address[*AddressType*]?]
type $P2$ = Person[Name[*string*]]

$P2$ would be a subtype of $P1$ because the set of documents validated by the first type includes all of the those validated by the second type. This is written $P1 <: P2$. We use the sub-type algorithm described by XDuce during the process of creating an augmented schema.

5. Technical Details

5.1 Handlers

Automating the composition of handlers requires a formal representation of the input and output to handler components which we call a *transformation interface*. We provide support for programmers to describe transformation interfaces using the language of regular expression types. Programmers must decorate handler components to express the transformations that a handler can make. These interfaces are given to our tool as a configuration file which maps rules to Java classes implementing handlers.

There are two basic categories of transformations that can be expressed in a transformation interface.

First, sometimes elements between schemas differ only in their labeling (i.e. name). In this case, a simple transformation such as `PersonName` $\hookrightarrow$ `Name` suffices to express the conversion. In these cases the interface is exactly the implementation and no other code needs to be written by the programmer. Our tool uses the information in the interfaces for providing an augmented schema and also for generating an implementation.

Second, in many other cases handler code written in some other language such as Java will need to be used to implement more complicated transformations. This is required in case some other business logic or outside resources (such as a database) should be consulted to perform a conversion. This could be as simple as the logic required to convert between two postal code formats or as complicated as contacting an external Web service to perform an up-to-date currency conversion. In this case, our tool uses the rules as input to schema augmentations and associates an identifier with some Java method which will be used as a call-back handler.

In both cases transformation interfaces are used by our deployment-time tool to generate an appropriate augmented schema.

5.2 Interface Discovery Tool

A tool is used at deployment-time to discover the interface of a server with respect to an intermediary. Our intermediary exposes a Web service operation to accept schemas from clients which are compiled into an augmented schema and returned. Compilation is based on a tree-rewriting algorithm which needed to be adapted to meet the requirements for our domain. First we explain the original algorithm and then the changes which were required.

5.2.1 Tree Rewriting Algorithm

A *regular ground tree rewriting system* (RGTRS) is a set of rules of the form $T_i \hookrightarrow T_j$ where T_i and T_j are sets of trees (we use XDuce expressions for this purpose). Our transformation interfaces are compiled into a *tree automata* representation for use with this approach. The semantics of such rules are interpreted as: some tree accepted by T_i can properly be replaced by some tree accepted by T_j.

The *reachability problem* for RGTRS is informally: Given a RGTRS (i.e. set of rewrite rules) and a *target* (possibly infinite) set of trees, what is the set of trees which can be rewritten into a tree accepted by the target, using any composition (transitively) of the rewrite rules? Using the algorithm of [12] this output set is given as a tree automata, which we convert back into an XML Schema.

Since RGTRS work *bottom-up*, they help us model the effect that handlers can apply transformations element-wise. To illustrate these points consider the simple example in Figure 4. Intuitively we can see that `schema1:ContactElement` can be transformed into `schema3:PersonElement` by applying the first rule (element-wise) and then the second. Notice that if we consider these two rules as type signatures in a traditional programming model, it would not make sense to compose them in any fashion.

```
type schema1:ContactElement =
  Contact[Person[string], Number[int]]
type schema2:ContactElement =
  Contact[Person[string], Phone[int]]
type schema3:PersonElement =
  Person[Name[string], PhoneNumber[string]]

(1) schema1:Number[int] ↪ schema2:Phone[int]
(2) schema2:ContactElement ↪ schema3:PersonElement
```

Figure 4. Illustration of tree rewriting: If we apply transformation (1) to an instance of the first type (element-wise) and transformation (2) to an instance of the second, we are guarenteed an instance of the third type. Namespaces are qualified using the (:) notation.

Repository 1	Correct for both	1. `A ↪ (B | C)` 2. `(B | C) ↪ D`
Repository 2	Correct for both	1. `A ↪ (B | C)` 2. `B ↪ D` 3. `C ↪ D`
[6] Repository 3	Correct for original Incorrect for interfaces	1. `A ↪ (B | C)` 2. `B ↪ D`

Figure 5. Examples of differences between original and modified algorithm for inferring: `A ↪ D`

5.2.2 Modified Algorithm

Unfortunately, the semantics of the algorithm are not safe for our purposes. Since T_j is a type, any of the possible instances could be produced by the handler at run-time. If we are to model the execution of the handler as a black-box we are forced to consider all possible choices rather than being able to select one of them. To illustrate consider the simple example in Figure 5.

Here we show three possible repositories of handlers. In the first two cases, we can appropriately infer that an element of type `A` can be used where `D` is required by making certain transformations. However, in the third case we must assume that handler number (1.) could return an instance of type `C` for which there will be no appropriate transformation to `D` (assuming `C` is not a sub-type of `B`).

For our purposes, we must require that some tree accepted by T_i could be replaced by *any* tree accepted by T_j. We achieved this by replacing a test for non-empty intersection in the original algorithm with the sub-type algorithm described in [9]. So, while in the third example the output of handler (1.) intersects the input of handler (2.), the output

```
walk((T|U)) := reach(walk(T) | walk(U))
reach((T|U)) :=
1.   if(∃(T_i ↪ T_i') s.t. T_i' <: (T|U) and T_i ≮: (T|U))
2.     if(lookup(T_i))
3.       return var(T_i)
4.     else
5.       bind(T_i')
6.       return reach(walk(T_i) | (T|U))
7.   else
8.     return (T|U)
```

Figure 6. Simplified Algorithm for Illustration: showing the functions for processing choices (functions for other structures follow schematically). We assume functions on a mutable global environment: `lookup` for testing whether a type is bound to a reference, `var` for returning a reference binding, and `bind` for binding a type to a fresh reference. Psuedo-code uses pattern matching on type structures and infix notation for constructing new choice types.

of handler (1.) is not a sub-type of handler (2.) input.

An illustration of the modified algorithm is shown in Figure 6. The first function `walk` is simply a helper function to force processing types bottom-up (i.e. starting at the leaves). It walks the structure of types as laid out in Section 4.2 . Bottom-up processing is simpler because any changes to the lower levels become visible as we work our way up. Due to space, we are only showing the definitions for functions being applied to the `choice` structure. Definitions for the other structures are straightforward.

The main logic is in the function `reach`. The condition in line 1 checks for matches between the output of any interface and the current type $(T|U)$. A match occurs when the output is a sub-type. This means we can use the output of the interface wherever the current type is required. If there is no match, we simply return the current type unchanged. Note that any changes which occur higher up will not have any effect on this type.

The condition also checks that the left-hand side is not a sub-type of the current type. If it was, there would be no need to apply a transformation. This check prevents us from applying transformations redundantly.

In case there is a match, we first need to check (line 2) if we have already made this match, in order to prevent an infinite transformation loop. If we have, we return a reference (i.e. pointer) instead of the actual structure. This creates the appropriate recursion in the schema to represent the loop. If we have not matched the type already, then we first we create a new binding to record a new match (line 5).

Line 6 is the most interesting, we need to process the rule input itself in case any transformations apply to it – `walk` (T_i). This is why we need to create use references so we know when the output of the handler could be fed back to the input.

To illustrate why all of this machinery is necessary consider a simple transformation such as, $f[a] \hookrightarrow a$. First, the output of the transformation could later be used as input. Second, we cannot represent the set of transformations made by this rule without the use of recursion.

Once we get the results of processing the rule input (call to `walk` in line 6), we create a choice between the current type and the results. This is where the augmentation takes place. We have effectively widened the servers interface to allow types to be accepted which we have determined will be transformed appropriately. We call these added types `substitutions`.

Finally, we repeat the process on the choice. There are two reasons for this, first there may be other rules which apply to the current type. Second, there may be other rules which can only be applied to the new choice. Note that the situation in Repository 2 (Figure 5) is handled appropriately, because substitutions propogate backwards (from right-hand side to left-hand side). So, since B and C lead to a common type (directly or transitively) they will appear as choices wherever D could appear and the test $(B|C) <: (B|C|D)$ will succeed.

In our actual implementation, we rely on directly modifying the algorithm presented in [12]. That algorithm has superior performance but is more complex so we provided this discussion for illustrating the changes we made and for defining the meaning of augmented schemas concretely.

5.3 Middleware Runtime

We have seen how the reachability algorithm is used to construct an augmented schema at development-time. Still, at run-time when we receive a particular XML message, we need to know: for this message exactly which handlers should be used, on which message pieces, and in what order? This control over handler execution is important to control the order in which transformations are applied. Consider cases such as when two handlers apply to the same element, a completely dynamic approach could not tell which handler might lead to a useful conclusion.

During tool execution, we need to remember which handlers were responsible for adding new choices to the augmented schema. To do this, we keep a trace of the tool execution in the form of schema annotations. Intuitively, we annotate each choice with a trace of identifiers for the handlers responsible in creating that choice. In our implementation we just keep a separate digest that is the output of the compile-time process. This digest is then loaded when we start our middleware server.

1. G[D]
2. G[$(C^3 \mid D)$]
3. G[$(B^2 \mid C^3 \mid D)$]
4. G[$(A^{1,(2\mid3)} \mid B^2 \mid C^3 \mid D)$]

Figure 7. Compiling the schema G[D] with repository number 2 in Figure 5. Numbers refer to handlers in that repository. Trace annotations are shown as superscripts. In the case that a client sends a message of type G[A], for example, our middleware will dispatch to handler 1, then either 2 or 3 depending on the output of 1.

tr ::=	i	handler identifier
	$(tr \mid tr)$	choice
	tr, tr	sequence
	$()$	empty sequence

As above, a trace is a sequence of choices of traces. The choice is necessary in the case that a sub-type test can only succeed by taking into consideration a pair of handlers and not just a single handler. This is exactly the case described in repository (2) of Figure 5. Notice that it is not possible to know whether handler 2 or 3 should be exectuted until run-time. We implement tracing during the step on line 6 in Figure 6, as below.

T_{tmp} := walk (T_i)

$T_{tmp}.trace$:= (append(i, T.trace) | append(i, U.trace))

Before calling `reach`, the new substitution inherits traces (shown as a property of a type) from the type that it substitutes. This allows the trace to follow a chain of substitutions. These inherited traces are appened to the identifier for the handler which added the substitution. Figure 7 provides an illustration by showing how a small schema is augmented and annotated over four iterations of the while loop from the algorithm.

When a message is received by our intermediary, we validate that message bottom-up. When some sub-tree validates to a type with a trace, we dispatch that sub-tree to the sequence of handlers specified in the trace for transformation. Choices in the trace require a second disambiguation step to determine the type of message output by a handler. Since two types can usually be disambiguated based on the name of the top level element, we do not believe this will have an important impact on performance.

We hope the use of run-time dispatching of handlers informed by development-time analysis will help to relieve developers from worrying about manual composition of handlers or their possible interference.

```
type GetRecommendationsRequestElement =
  GetRecommendationsRequest[
    ListingFlow?,
    Item[legacy:ItemType | unified:ItemType],
    (ProductIdentifier[ProductIDType]?
     | ProductID[ProductIDType]?),
    ModifiedFields[ModifiedFieldsType]?,
    RecommendationEngine*,
    Query[string]?
  ]
```

(1) `ProductIdentifier` ↪ `ProductID`
(2) *`legacy:ItemType`* ↪ *`unified:ItemType`*

Figure 8. Augmented eBay unified GetRecommendation request message type and two transformations that applied.

6. Examples

6.1 Schema evolution

Here we discuss the advantages of the approach in concrete terms of the example.

Recall that a developer wants to take advantage of the new `GetRecommendations` operation. In the bottom of Figure 8 we see two transformations that may already be available at an intermediary since these transformation apply to the original legacy schema. However, the client-side programmer using `GetRecommendations` needs some way to know which transformations can be performed. Our tool provides an automated way to derive this information and the augmented schema provide a convenient way to deliver this information.

Looking over the augmented schema in Figure 8, a programmer can determine that there were 4 elements as part of the message for which no transformation was available. It turns out that none of these elements is actually required so the minimal functionality of the operation can be used without developing the business logic that creates these elements. Additionally, `ModifiedFields` is deprecated (as determined by the documentation) and `Query` is unchanged from the legacy schema. Now, the programmer can go ahead and begin using this operation using legacy `Item`'s that may be collected in their database, until such time as the database schema and XQuery reporting programs are updated.

6.2 Compatibility with Web Services

Returning to our second scenerio consider the second step in the Web service flow: the seller wants to validate

Legacy eBay to Fedex	*legacy:AddressType* ↪ *FDX:DestinationType*
Fedex to UPS Validator Address	*FDX:DestinationType* ↪ *UPS:AVAddressType*

Figure 9. eBay, Fedex, UPS transformations

```
type UPS:UPSAddressValidationRequest =
  AddressValidationRequest[
    RequestElement,
    AVAddressElement
  ]

type Int:UPSAddressValidationRequest =
  AddressValidationRequest[
    UPS:RequestElement,
    address[UPS:AVAddressType |
            legacy:AddressType  |
            FDX:DestinationType
    ]
  ]
```

Figure 10. Original and augmented UPS Address Validation Message using transformations from Figure 9

the buyer's address using the UPS AddressValidation (AV) service. The format of this message is shown at the top of Figure 10. Now consider an intermediary that supports common transformations from eBay to Fedex elements and also Fedex to UPS elements. It may include those shown in Figure 9.

The augmented message type for the UPS address validation is shown at the bottom of Figure 10. Notice that it considers all the address types equivalent for the purpose of the address validation operation. Now the client can use any of these Address types for the purpose of Address Validation.

The transformations shown in Figure 9 are just one modular decomposition that could achieve the output of 10. Suppose that the Address element itself was transformed element-wise. This other choice in modularity is motivated by the fact that UPS uses two different Address formats for different services it offers as in Figure 11. We see that if a developer had already created a transformation for `TrackingAddressElement`, it would have already covered `AVAddressElement`. Also if a transformation was only created for AVAddressElement, then a client developer could see which elements were not yet covered.

This shows two advantages of our approach. First, a developer does not have to compose specific transformations when they are already covered, reducing the amount of error-prone formatting of data. Second, and more importantly, consider how transformations would be composed in a traditional call-and-return style architecture. If we had individual functions for child elements of TrackingAddress, we could write a new function to transform AVAddress by composing three of those functions. However, since transformations are deployed on a distributed intermediary it is not reasonable to assume that any client-side programmers should have knowledge of the implementation details of the transformation functions deployed. Even if they did, a client-side composition of those functions would require three remote interactions to compose the individual functions. These reasons help show why the discovery of intermediary interfaces is important to enable reuse in the loosely coupled Web service environment.

```
type UPS:AVAddressElement =
  Address[
    City[string],
    StateProvinceCode[string]
    PostalCode[string]
  ]

type UPS:TrackingAddressElement =
  Address[
    AddressLine1[string],
    AddressLine2[string],
    AddressLine3[string],
    City[string],
    StateProvinceCode[string],
    PostalCode[string],
    CountryCode[string]
```

Figure 11. UPS AVAddress element and TrackingAddress element

7. Evaluation

In this section, we present some performance results of our tool and middleware implementation. We have used the eBay legacy–to–unified mapping tables (described in Section 2.2) to create these tests. We wanted to verify that the changes we made to the original reachability algorithm would not restrict its scalability.

The complexity for the original reachability algorithm is $O(|R|^2 * (|R| * |A|))$ where R is the size of all the transformation interfaces (i.e. rules) and A is the size of the target, which in our case represents a server interface. This bodes well for scalability since the complexity is low-order polynomial. Our run-time is based on validation of messages which can be done in linear time [3].

We have modified the reachability algorithm to use the

sub-type test described in [9]. This algorithm is known to run in exponential time in the worst case. Independent studies have shown the algorithm to perform well in practice [9, 18]. For example, testing between two versions of the schema for HTML runs in under a second. Also, the sub-type test is only used in the off-line portion of our approach.

In our implementation, the sub-type test is only used once we have determined that two types intersect, to avoid many sub-type tests. Our hypothesis was that since most handler interfaces on an intermediary repository will not intersect with those in a given target message type, having large repositories of handlers would not adversely effect the usability of the tool.

Table 1 shows how the computational time for the tool processing the eBay `AddDispute` request message using a repository with enough handlers implemented from the mapping tables to cover the given number of complete messages[1]. We see that time follows roughly as the square of the number of handlers. This matches with our expectation that the time would follow closer to the complexity without the sub-type test.

No. of Tables	1	10	20	40	60
Time(s)	0.14	0.53	1.76	6.89	17.10

Table 1. Off-line computational time according to the number of mapping tables involved in building the augmented schema for one of the messages

Table 2 shows the efficiency of the run-time part of the algorithm using a client and server connected on a LAN with and without an intermediary. The first row shows the total time and the average time for each request/response message, when a thousand messages are sent to the server. The Web service implemented using the open-source Apache Axis and is performing a light task[2].

The second row shows the overhead of the presence of an intermediary on the message path. The intermediary is only forwarding the message to the server and is not processing it. The third row shows the timing and the overhead of the on-line part of middleware runtime, which is fairly low. The handler chain is determined for an `AddDispute` message but the handlers are not actually executed. The forth row shows the timing and the overhead of the handler execution in addition to their dispatching. We feel these results meet our expectation that dispatching handlers through a modified document validation algorithm could reasonably be integrated into existing server implementations.

	1000 msg	avg/msg	overhead
No Intm	38.77 s	0.03	-
Intm Forwarding	54.08 s	0.05	0.42%
Intm + Alg	58.16 s	0.05	0.07%
Intm + Alg + Hdlrs	59.91 s	0.05 s	0.03%

Table 2. Efficiency table for on-line part of the algorithm. Intm = Intermediary, Alg = Algorithm, Hdlrs = Handlers, msg = message, avg = average

8. Related Work

8.1 Web services

The standard language for XML transformation is the eXtensible Stylesheet Language Transformation (XSLT). XSLT uses a *top-down* processing model. In general, processing begins working at the root of some document and works its way down to the leaves. When an element is encountered for which there is a matching *template*, that template is executed. We share this form of implicit-invocation with XSLT however XSLT does not support any kind of inference for reasoning about the composition of templates. Several tools [11] have been developed to type check an XSLT program given an input and output type, but this does not address the motivation described in this paper where all possible input types should be inferred given a particular output type.

In previous work [20] we proposed an AOP approach to programming transformation using the familiar advice-pointcut style. However that work did not address the generation of an augmented schema or automatic middleware dispatching and shares very little in terms of techical details to the work proposed here.

Web service composition is an important area including software engineering approaches such as formal modeling and specification checking [7]. We believe our work is novel in that the composition of service interfaces with intermediaries is largely unexplored.

Ponnekanti [13] presented a taxonomy of Web service interface mismatches that can occur when interfaces are allowed to evolve independently as well as a static and dynamic analysis tool to discover mismatches in WSDL. Their taxonomy provides a good overview to the kinds of problems where an intermediary can be useful.

8.2 Adapters

Nimble [14] shows how external adaptation can be valuable to reduce the non-functional constraints affecting application code. In some cases, the generation of adapters

[1] We ran the experiments for a several request messages other than `AddDispute` as well, but the change in computational time was not noticeable.

[2] Three computers with Pentium 4 CPU's have been used for running these tests. Each test was repeated ten times and the number shown in the table for each experiment is the average of the results

can be further automated [17] but neither of these works is based in an event-based adaptation paradigm.

In [8] Gschwind provides a solution for the problem of composing software component interfaces similar to ours. They work in the paradigm of OO typing where sub-type relationship are explicitly declared. In their approach reachability on types can be implemented using a shortest path algorithm on graphs. However partial interface adaptation is not supported.

Other works [4, 21] concentrate on mismatches between components at the behavioral level (i.e., the protocol between components). We have not addressed this issue in our research.

9. Conclusion

Based on our examination of several Web services, we found several opportunities where partial adaptation could be of benefit. We motivated some realistic scenerios as an illustration. We believe the scenerios make sense given the use of newer middleware architectures such as the Enterprise Service Bus.

We do require annotations for each handler deployed at an intermediary. However, the annotations correspond to elements already described in a service schema, so additional tool support could easily automate the bulk of this task.

The transformation of an entire schema can be broken down into handlers responsible for only specific pieces, motivating an event-drive style of programming. Unfortunately, as is the case with many component architectures, there is a tradeoff between creating too few, hard to reuse, easy to compose handlers, or too many, easy to reuse, hard to compose handlers. So developers use a more dynamic approach to avoid manual composition (call-and-return style) and rely on flexible composition based on an event-driven style. This comes at a price to service clients who are then unable to reason about the composition of an intermediary and a service. Our approach is tailored to provide flexibility to transformation programmers without sacrificing an explicit interface contract for clients.

References

[1] eBay developers web site. http://developer.ebay.com.

[2] D. A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[3] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree automata techniques and applications*. Draft book, 2005.

[4] R. DeLine. Avoiding packaging mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[5] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proc. of Foundations of Software Engineering*, pages 209–221, 1998.

[6] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Techbology*, 2(2):115–150, 2002.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *International Conference on Automated Software Engineering*, 2003.

[8] T. Gschwind. Automated adaptation of component interfaces with type based adaptation. Technical Report 1841 -2003-14, Technische Universitat Wien, 2003.

[9] H. Hosoya and B. Pierce. Xduce: A statically typed xml processing language, 2002.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP*. 1997.

[11] C. Kirkegaard, A. Moller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Trans. on Software Engineering*, 2004.

[12] C. Loding. *Infinite Graphs Generated By Tree Rewriting*. PhD thesis, RWTH Aachen, 2003.

[13] S. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proc. of the International Conference on Middleware*, 2004.

[14] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *Software Practice and Experience*, 21(6), 1991.

[15] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proc. of the International Conference on Software Engineering*, pages 374–384, 2003.

[16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the International Conference on Software Engineering*, pages 107–119, 1999.

[17] S. R. Thatte. Automated synthesis of interface adapters for reusable classes. In *Proc. of Symposium on Principles of programming languages*, 1994.

[18] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *Implementation and Application of Automata*, 2003.

[19] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware. In *Proc. of the COOTS*, pages 677–694, 2000.

[20] E. Wohlstadter and K. D. Volder. Doxpects: aspects supporting xml transformation interfaces. In *Aspect-Oriented Software Development*, 2006.

[21] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.