Interface Composition for Web Service Intermediaries

Sara Forghanizadeh and Eric Wohlstadter

University of British Columbia

Abstract. The use of XML as a format for message exchange makes Web services well suited for composition of heterogeneous components. However, since clients must manage differences in message schemas between services, interoperability is still a significant problem. Interoperability currently can be supported through the use of transformations provided by a Web service intermediary. However, intermediary technologies do not provide a way for clients to reason about the composition of services and intermediaries. We propose an approach to provide clients with an interface composed of schema information from a Web service and an intermediary. Composition is performed by applying rewriting rules, defined by the intermediary, to the server interface schema. This new interface takes into account what transformations are available at an intermediary. The advantage of the approach is that clients can continue to benefit from code-generation and static type-checking offered by interface definition languages such as WSDL; while still making use of the flexibility offered by intermediary transformations. We provide the algorithmic details of composition, including a proof of correctness and an upper bound on complexity. We demonstrate the approach in the context of a Web service composition of three publicly available Web services.

1 Introduction

Web services are flourishing on the Web as an important part of the information technology infrastructure. They provide building blocks for clients who can compose new applications or services out existing reusable services. Here, a client could also be a service itself, made up of a "mash-up" of existing services. The use of XML as a message exchange format makes Web services well suited for composition of heterogeneous components. The schemas [1] of these messages define the service's interface and are often described by an interface definition language such as the Web Services Definition Language (WSDL) [2]. However, since clients must manage differences in schemas between services, interoperability is still a significant problem. There is often some level of semantic overlap between schemas even when there is no syntactic match.

Since distinct services will naturally have certain distinct semantics, we cannot realistically hope to completely shield clients from differences in schemas. So, we are investigating support for interoperability through the use of *element-wise type-based adaptation*. This partial approach to adaptation is motivated by the desire to keep client applications simple. Complexity is reduced because clients only have to understand one particular schema for those XML elements where the types of multiple schemas semantically overlap (i.e. intersect).

Previous work on type-based adaptation [3, 4] has solved problems related to the reuse of components in contexts that were not originally anticipated. The core problem is to take a typed interface provided by a component, a typed interface required by a

client, and to help produce or locate a type adaptor which adapts between the required and provided interface. This helps a developer who takes components "off-the-shelf" and needs a way to connect them; as illustrated in Figure 1.a.

We seek to extend this line of work to an element-wise approach, for use in a Web services setting. Using Web service middleware, clients can make use of adaptive message transformations provided by a Web service *intermediary* [5]. Transformations can be implemented using *handlers* (described in Section 3) which can be applied to individual elements of XML messages. In our setting, an intermediary could either be a network proxy (e.g. Enterprise Service Bus [6]) or a local middleware layer. The important property of an intermediary is that it can be managed separately from the client business logic.

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. In this case, the intermediary for the company could be programmed to transform to and from the company schema to FedEx and UPS. Here is how our approach would apply: when a developer at the company starts to write client code to communicate with FedEx or UPS, they can see 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 a *composite schema*.

So we extend previous work in the following way: given a target Web service schema, $\mathcal{TS}$, and an intermediary, we generate a composite schema, $\mathcal{CS}$. Once this composite interface is provided to the client, they can then use it to generate stub code and perform type-checking. In our research the adaptations are known variables and the interface the client can use is an unknown variable instead of vice versa. Our setup is then shown in Figure 1.b.

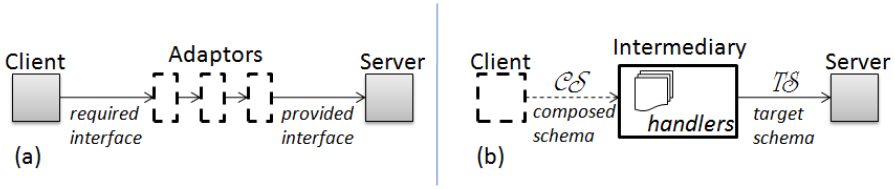

Fig. 1. (a.) Interface adaptation, and (b.) Intermediary interface composition. Dotted lines represent the unknown variable.

Our contribution is to demonstrate an architecture and algorithm for intermediary interface composition. We provide the algorithmic details including a proof of correctness and an upper bound on complexity. We demonstrate the approach in the context of a mash-up of three publicly available Web services.

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

2 Motivating Scenario

We consider a typical mash-up scenario where a Web service is built by composing services from eBay, FedEx, and UPS. Let's examine a typical business process flow for the mash-up.

First, a request is made to the `GetSellerTransactions` operation for eBay. This provides information regarding auctions managed by a particular user. The response contains transaction information, including a shipping address and a choice of shipping options for auction winners. Here we assume FedEx was chosen by some auction winner.

Second, the mash-up company wants to verify the address of the buyer before scheduling a shipment with FedEx. Unfortunately, the FedEx service does not include a separate operation for validating address information, but UPS does. So, the mash-up sends an `AVRequest` (address validation) to UPS. A part of the information in this message is specific to UPS, such as the UPS account number. On the other hand the address to be validated can be extracted from the response message received from eBay and simply routed to UPS.

Finally, after address validation, the mash-up must additionally transform address information to FedEx before sending a final `FDXShipRequest` request. This motivates transformation of the address information between eBay, UPS, and FedEx formats, but complicates things for the programmer without additional support.

Motivated by this scenario, a separate transformation layer deployed on a intermediary could be implemented which imports and exports address information in a common format. This has the benefit of factoring the entire transformation concern out of the core mash-up business logic. This could simplify the mash-up implementation since the address format will be a common element in many operations for all three services. However, without additional support, this would break the sound engineering process of writing software against statically defined interfaces. The WSDL of the three services does not take into account the intermediary transformations so their presence is implicit and not explicitly accounted for in a typed interface. The client could not generate stub-code from the WSDL that accounts for the transformations or type-check against these stubs.

3 Background

3.1 Handlers

Intermediaries are often a repository of message handlers (i.e. interceptors [7]) which read and write elements of XML-based messages. For example, a handler might be responsible for just the translation of elements representing shipping addresses in two schemas with disparate formats. Traditional handlers could be modeled as a function with a type `Document`→`Document`, since they accept any XML document type and do not place any guarantees on the output document type.

This type does not provide useful information for reasoning about the composition of document handlers. On the other hand, if handlers specified specific types at the granularity of entire documents, for example `PlaceEBayOrder`→`PlaceAmazonOrder`, the types could be too specific. This is because document handlers might apply specific element-wise transformations.

For example, they might transform just the address information in an eBay or Amazon order. These problems help motivate a different approach based on element-wise type information.

3.2 XML Schema

WSDL [2] is based on the standard W3C XML Schema [1] grammar for defining types of XML documents. Web service developers use this grammar to specify sets of XML documents that are accepted or provided by service operations. A document instance that is a member of the set, defined by a type, is said to be *valid* for the type. Figure 2 provides the grammar and notation of a mini-schema language that we use throughout the paper to model salient properties of XML Schema.

Here a schema consists of a root `ElementType` which defines an XML element tag name and the `ContentModel`. The types of elements can also be declared through reference to existing definitions by use of an `ElementRef`. The content model defines the valid structure of nested children. The content model [1] is either a `Primitive` type or a regular expression of element types. The use of recursion to other `ElementTypes` in an expression is what gives XML Schema its power to express tree structures; as opposed to flat strings which are normally described by regular expressions.

Web services middleware can perform *validation* to ensure messages match the types defined in their WSDL. Our approach combines the mechanism of message validation with a message transformation mechanism. Here we describe some technical details of standard message validation mechanisms so we can build on this background to describe details of our approach.

ElementType ::=	*name*[ContentModel]	//Named element with child content
	ElementRef	//Reference to a named element
ContentModel ::=	*Primitive*	//primitive type
	ContentModel, ContentModel	//sequence
	ContentModel || ContentModel	//choice
	ContentModel*	//unbounded occurences
	ContentModel?	//optional occurence
	ElementType	//recursive nesting of elements

Fig. 2. Mini XML schema language BNF. Note: This figure defines a grammar language, Mini-Schema, in terms of another grammar language, BNF. So the figure should be interpreted carefully. We avoid using any symbols from the BNF language except for the non-terminal assignment (::=). Non-terminal alternatives are separated by newlines. All other symbols are part of the defined Mini-Schema language. Terminals are shown in italics. Java-style comments are used.

Message validation can be performed by middleware at runtime using a parser. An important property of the W3C XML Schema specification is that many aspects of the specification are motivated by the requirement to keep this parsing process simple. In particular, the schema language is designed to prevent parsers from needing to *back-*

[1] For simplicity, we don't currently deal with some features of XML Schema, such as numeric occurence indicators or the `all` content model. These features could be added using the approach in [8]

track at runtime [2]. This is achieved by restricting the regular expressions describing a `ContentModel` to those that are unambiguous.

This property allows XML Schema parsers to be implemented using only small extensions to standard deterministic finite automaton (DFAs). As usual, a DFA is a mechanism for parsing sequences of strings which consists of: a set of states and a set of transitions between states. One state is distinguished to be the initial state and some subset of states are final (accepting) states. Each transition is associated with a symbol from some alphabet. Parsing begins at the initial state and is driven by reading symbols of an input string from left to right. As each symbol is read, the current state is changed by following a transition labelled with that symbol. If the sequence is consumed and the automaton is in a final state then the string is accepted.

But XML documents consist of trees instead of strings! Still, Thompson shows how XML Schema is parsed using a set of cooperating DFAs [9]. Intuitively, each level of sibling elements in the hierarchy of the XML tree is considered as a separate string. When an element is consumed, the parser simply parses the content of the element using a different DFA, before allowing the transition to occur. Parsing of XML trees is thus handled through this nesting of operations on separate DFAs. This process is illustrated in Figure 3.

Our approach to composition involves integrating this parsing process with the process of message transformation. This is achieved by building automaton that are aware of the transformations that handlers can make.

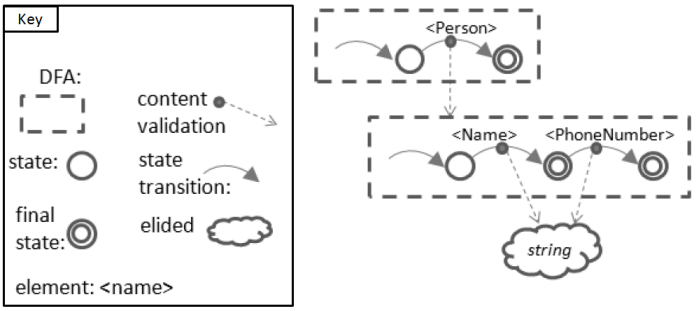

Fig. 3. A parser for the ElementType: Person[Name[$string$], PhoneNumber[$string$]?]. Parsing begins at the root (top-most) DFA and its final state is the final state for validating a complete document. Final states of internal DFAs are used to determine if transitions in a parent DFA are successful.

3.3 Regular Expression Types

We use the paradigm of regular expression types [10] for added flexibility during the process of creating a composite schema. Unlike popular OO languages, under this discipline a sub-type relationship holds for two types if the set of instances described by one type completely contains (i.e. subsumes) the set of instances described by another type. This is different from typing in standard OO languages where a sub-type relationship needs to be explicitly declared.

[2] This is called Unique Particle Attribution in W3C parlance.

So under regular expression typing, `person` (below) would be a sub-type of `personExtra` because the set of documents validated by `personExtra` includes all of the those validated by `person`. Here new types are declared using an `element` keyword in the obvious way. Intuitively, a `person` can be used wherever a `personExtra` is expected. This is commonly notated using the sub-type operator (`<:`), as in, `person <: personExtra`.

element person = Person[Name[$string$]]
element personExtra = Person[Name[$string$], PhoneNumber[$string$]?]

We will especially take advantage of these two axioms of regex typing:
Axiom 1. `(a <: b)` implies `(a <: (b || c))`, for all types `a`, `b`, and `c`.
Axiom 2. `(a || a)` = `a`, for all types `a`.

Axiom 1 is clear since `(b || c)` accepts a larger set of documents than `b` alone. Axiom 2 simply says that a choice between a type and itself is idempotent.

4 Overview of the Approach

Our approach is divided into two stages: interface composition and handler composition. We describe how these stages work from a high-level architectural viewpoint before going into low-level details in Section 5.

4.1 Interface Composition

One technical challenge is to return a new composed schema, $\mathcal{CS}$, to the client. This is an extended version of some target schema, $\mathcal{TS}$. This requires that programmers assign a typed interface to each handler on an intermediary (Section 5.1). This interface composition process (Section 5.2) occurs off-line at development-time; not during the actual execution of a deployed system. Our middleware uses our algorithm described in Section 5.3 to compose handler type information with a Web service's type information.

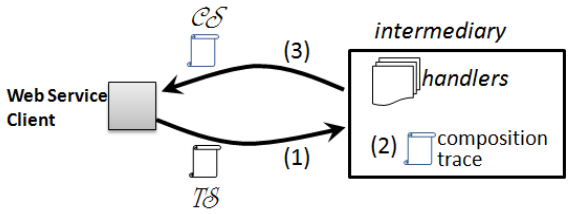

Fig. 4. The composition service produces a composite schema, given the set of handlers and some server interface chosen by the client. A digest of this process is saved by the intermediary for use at run-time. We place certain terms in italics; these terms will be referred to as Java objects in the algorithm of Figure 7.

The data-flow of this process is shown in Figure 4. First (1), a client chooses some $\mathcal{TS}$ that it wants to compose with an intermediary. It submits $\mathcal{TS}$ to any intermediary that makes use of our prototype middleware. Second (2), the intermediary runs an algorithm using $\mathcal{TS}$ and *handlers* deployed on the intermediary, to compose their type

information. Certain information about the algorithm execution is stored in a *composition trace*. This information will serve as handler dispatching instructions at run-time (Section 5.4). Finally (3), the client receives the composed interface, $\mathcal{CS}$ from the intermediary. The client can now use $\mathcal{CS}$ to generate stub code and type-check against.

4.2 Handler Composition

At run-time, our middleware dispatches the handlers to execute transformations. This occurs when the intermediary receives a message from a client (or server reply). The data-flow of this process is illustrated in Figure 5. In this paper, we focus only on the algorithmic details and complexity of our approach. In an online technical report [11], we have provided some initial benchmarks of our prototype performance.

First (1), a client sends a message valid for $\mathcal{CS}$, which is intercepted by the intermediary. Second (2), client messages are validated and transformed by the intermediary. Our prototype middleware decides which handlers should be executed, on which message elements, and in what order. These decisions are guided by the composition trace derived off-line, so no complex processing is required at this time. Finally (3), a message valid for the $\mathcal{TS}$ is forwarded by the intermediary. This process is described in more detail in Section 5.4. The reverse process for server replies is similar so we do not show this data-flow in the figure or discuss the details specifically.

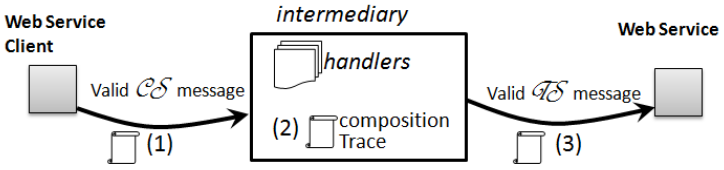

Fig. 5. The intermediary dispatches the handlers at run-time to make the message compatible with the Web service, using the digest derived at development-time.

5 Technical Details

5.1 Handler Interfaces

Our composition mechanism requires a formal representation of the input and output types of handler components. We provide support for programmers to describe *element-wise interfaces* using XML Schema.

Definition 1: *Element-Wise Interface*. An element-wise interface is a pair of `ElementTypes`, (*input*, *output*). The contract for a handler which implements the interface specifies that the handler can transform any element of type *input* in a document to some element of type *output*.

The implementation of the handler can be written as either Java or XSLT and can perform arbitrarily complicated computation to make a transformation. The implementation could be as simple as the logic required to convert between postal code formats or as complicated as contacting an external service to perform a currency conversion. Our tool uses the element-wise interfaces as input to schema composition

and associates an identifier with some implementation component which will be used as a call-back by our intermediary middleware. In our algorithm we assume each handler has a unique identifier, denoted as $handler.id$. The input and output element types are denoted $handler.input$ and $handler.output$ respectively.

To motivate how element-wise interfaces are used, consider the simple example of Figure 6. Here there are three type definitions for some user contact information in different formats i.e. contact1-3. Following there are two element-wise interfaces for some handlers listed: hander1-2. Notice that from the type information, we can infer that a message of type contact1 can be transformed into a message of type contact2; by applying handler1 to the `Number` element embedded within a contact1 message. From there we can now apply handler2 on the whole message to obtain a message of type contact3.

Composition in this case is not achievable by a traditional composition of function types. Here, the output type of handler1 does not match the input type of handler2. Still, composition is achievable if we consider element-wise composition. So we need a new approach for composition in this setting.

```
element contact1 = Contact[Person[string], Number[string]]
element contact2 = Contact[Person[string], Phone[string]]
element contact3 = Person[Name[string], PhoneNumber[string]?]

element numberType = Number[string]
element phoneType = Phone[string]
handler1: numberType ↪ phoneType

handler2: contact2 ↪ contact3
```

Fig. 6. If we apply handler1 to an instance of contact1 (element-wise) and then handler2 to that result, we are guaranteed an instance of contact3.

5.2 Interface Composition

At development-time an intermediary can be composed with the interface of a server. Our middleware prototype exposes a Web service operation to accept WSDL schemas from clients. The schema is processed into a composite schema and returned. Here we explain how our middleware can reason over the composition of element-wise transformations in an efficient way.

Using a naive approach, determining the correct composition of handlers could take exponential time in the number of handlers, because all permutations may need to be examined. Also, if handlers are allowed to execute multiple times (for example to perform element-wise operations) the problem is even more complicated. Here, we limit our approach to handlers providing *optional transformation*. This enables us to implement a tractable solution. This is exactly the case we have motivated: intermediaries provide transformations as an optional convenience for clients. These optional handlers are in contrast to handlers which enforce policies such as security. Our work does not address that kind of mandatory interposition of intermediaries.

Using the interfaces of handlers deployed on an intermediary, an inference algorithm is performed to determine the composed interface exposed to the client. This

process makes use of a *rewrite rule* which is a transformation to the server schema itself; as opposed to the transformations made by handlers at run-time on a message instance. We use the term "rewrite" here, rather than "transform", to keep it clear when changes are being applied to schema types versus message instances. For simplicity we assume there are no two handlers deployed on the same intermediary which have intersecting output types. This is sufficient to ensure the unambiguous nature of the schema is maintained during rewriting.

Definition 2: *Rewrite Rule*. Assume there exists a handler with the element-wise interface, (*input*, *output*), and also there exists an element type, `t` somewhere in the target server schema. If `output` is a sub-type of `t`, then any reference to `t` should be rewritten to the choice type: `t || input`.

Notice that we are matching *output*s and then adding *input*s to the schema. Intuitively, this process infers, in reverse, the transformations that handlers will be capable of making at run-time.

In our algorithm, we repeatedly apply the rewrite rule until it is no longer useful. Notice that since rewriting only causes the schema to grow, we do not need to worry about what order we apply the rewrite rules. This is because some application of the rewrite rule using one handler interface will never prevent the application of another rewrite rule, at any later point in time (by Axiom 1). In other words, the types of the schema only get wider.

However, we do need information to know at what point to terminate the algorithm. Termination occurs when only idempotent rewrites can be made (as in Axiom 2). For this purpose we introduce annotations on the composed schema that we call a composition trace. The trace also records the "backwards" type inference so that the trace can be followed in the forwards direction at run-time to actually transform message instances.

Definition 3: *Composition Trace*. Whenever a rewrite rule for the handler, h, with interface, (*input*, *output*), is applied at an element type t, we add an annotation on the reference to *input* that is created. The annotation is a sequence of handler identifiers, consisting of the identifier for h appended to the beginning of any existing sequence which annotates t. If t has no such annotation already, then the annotation is simply the single handler identifier.

As a convenience for clients, our intermediary supports an additional post-processing step to filter large composite schemas. Clients can provide preferences in the form of a partial-ordering on XML Schema namespaces. If a choice exists between two element types, `a` and `b`, and the namespace of `a` is preferred over `b`, then `b` will be removed as a choice. An example of this is demonstrated in Section 6.

With these definitions in place, we can define an algorithm that computes a composed schema. The question we need to answer is: given an intermediary and a server schema, what is the schema which describes all documents that can be transformed into

a document valid for the server schema, using any combination of the element-wise handlers?

5.3 Interface Composition Algorithm

The composite schema begins as a copy of the target schema, in Figure 7 line 1. The algorithm is structured as a comparison between all element types in the schema and all handler outputs. This is implemented as a double nested-loop, as in lines 3-4. Whenever the schema is changed by a rewrite, the entire process begins again, because references to new element types are added through the rewriting. This "restart" can be seen by the `break` statement on line 10, which breaks out to line 2. This is certainly not the most efficient implementation but it simplifies presentation of the algorithm and worst-case analysis.

Now, we can see that the algorithm will continue to execute until one of the `if` statements on line 5-6 fails for all iterations. The first `if` statement (line 5) implements part of the rewrite rule that checks whether the handler output type (as in Definition 1) is a subtype of some schema element type. The second `if` statement (line 6) checks the trace annotation (as in Definition 3) on the existing schema type to see ensure this rewrite would not be idempotent. If both of these checks succeed then a new reference to the handler input type is created (line 7). The trace annotation on the new reference is updated (line 8), according to Definition 3. Finally, the schema is rewritten (line 9), according to Definition 2.

```
1. CS = TS;
2. loop:
3.   forall(element in CS.elements)
4.         forall(handler in intermediary)
5.               if(handler.output.isSubTypeOf(element))
6.                     if(!element.trace.idempotent(handler.id))) {
7.                           ref = new ElementRef(handler.input);
8.                           ref.trace = concat(handler.id, element.trace);
9.                           CS.rewrite(element, new Choice(element, ref));
10.                          break loop;
11.                    }
```

Fig. 7. Composition Algorithm. Shown in Java pseudo-code. Local variable declarations are elided. Local variables shown in italics. We assume a method `Schema.rewrite(SchemaElement, SchemaElement)` which replaces the occurence of the first argument with the second argument by mutating the target schema. The constructors `Choice` and `ElementRef` simply model the construction of new schema productions as given in Figure 2.

Before proving the correctness and complexity of the algorithm. We walk through a specific example where a target schema is composed with handler information. The example makes use of the previous type definitions and handlers of Figure 6. In Figure 8, we see four schema files which consist of type definitions enclosed in curly braces. The first file, $\mathcal{TS}$, is the original schema chosen by the client. The next three files are versions of the composite schema as it is rewritten. We use a subscript to denote the number of rewriting steps.

$\mathcal{CS}_0$ starts out as a copy of $\mathcal{TS}$. Then, in $\mathcal{CS}_1$, handler2.output is matched, so handler2.input is added to the schema and added as a new choice. Next, in $\mathcal{CS}_2$, handler1.output is matched (element-wise), so handler1.input is added to the schema and added as a new choice. At that point no new matches can be made so the algorithm terminates. The superscripts on types are the composition trace and will be discussed in Section 5.4.

```
TS =  { element contact3 = Person[..]; }

CS0 = { element contact3 = Person[..]; }

CS1 = { element contact3 = Person[..] || contact2^(h2);
        element contact2 = Contact[Person[string], Phone[string]];
      }

CS2 = { element contact3 = Person[..] || contact2^(h2);
        element contact2 = Contact[PersonName[string],
                                   Phone[string] || numberType^(h1) ];
        element numberType = Number[string];
      }
```

Fig. 8. Composing a schema. The example makes use of the previous type definitions and handlers of Figure 6. We use a subscript on CS to denote the number of rewriting steps. The superscripts on types are the composition trace and will be discussed in Section 5.4.

Termination Proof: Termination is guaranteed because each rewrite does not actually copy the type information for some $handler.input$ type into a new location in the composite schema. Instead, a reference to that type is used as an alternative in the additional choice.

We know that the algorithm starts out with a fixed number of named types available as input at the beginning of execution. During execution, no new named types are created. Since the algorithm will not add any idempotent choices, it must eventually run out of rewrites that can be made and line 6 in the algorithm must fail for an entire iteration of both loops.

Correctness Proof: We need to show that a message, msg, is a $\mathcal{CS}$ message if and only if there is a sequence of element-wise transformations that can be performed on msg to create a $\mathcal{TS}$ message. This can be shown in two directions. First, we show that (1) if msg is a $\mathcal{CS}$ message, then it can be transformed into a $\mathcal{TS}$ message. Second, we show that (2) if msg can be transformed into a $\mathcal{TS}$ message, then it is a $\mathcal{CS}$ message.

1. This is shown by induction on the sequence of rewrites (line 9) performed during algorithm execution.

Base Case: When the algorithm begins, $\mathcal{CS}$ equals $\mathcal{TS}$. So, if msg is valid for $\mathcal{CS}$, it is already a valid $\mathcal{TS}$ message.

Induction: We assume that all valid $\mathcal{CS}_{k-1}$ messages can be transformed into $\mathcal{TS}$ messages. Now, we show, that $\mathcal{CS}_k$ messages can be transformed into $\mathcal{TS}$ messages. We know on the kth rewrite, one additional choice is added by some handler's

interface to $\mathcal{CS}_{k-1}$. So assume there is some message that is valid for $\mathcal{CS}_k$ but not for $\mathcal{CS}_{k-1}$. We know the message can only differ from a $\mathcal{CS}_{k-1}$ message by use of this additional alternative. However, we also know we can transform the alternative type back to one accepted by $\mathcal{CS}_{k-1}$ using the same handler. By the assumption, all $\mathcal{CS}_{k-1}$ messages are $\mathcal{CS}_k$ messages. Therefore, by induction, after the termination of the algorithm, all $\mathcal{CS}_i$ messages can be transformed to $\mathcal{TS}$ messages for any integer i.

2. This direction is straightforward. msg is valid for $\mathcal{TS}$, and $\mathcal{TS}$ is a sub-type of $\mathcal{CS}$ by Axiom 1 and Definition 2. Therefore, msg is valid for $\mathcal{CS}$.

Complexity: We show that our offline algorithm runs in polynomial-time with respect to the input size. This suffices to show that the algorithm is tractable. In other words, execution will not suffer from "exponential explosion". In the future we plan to give a tighter upper-bound.

As was described, the input to the algorithm consists of n handlers and $\mathcal{TS}$. Let the sum of the sizes of all handler inputs and outputs be denoted $|H|$. Let the size of $\mathcal{TS}$ be denoted $|TS|$. These sizes are in terms of the length of the respective type definitions.

In the pathological case, the $output$ of all handlers could match every element type in $\mathcal{TS}$, and also every element type in the $input$ of all handlers. Recall that handler.input types become referenced by the schema as the algorithm progresses, so they become fair game for matching. So the schema could have a maximum of $(|H|+|TS|)$ element types. Each element type could be a choice between all types, giving a size of $(|H|+|TS|)^2$. To simplify analysis we assume that each match might require $n*(|H|+|TS|)$ iterations of the double loop. In other words, we conservatively assume that matching fails all the way up to the last evaluation of the loop body. Also, for clarity we assume $(|H|+|TS|)$ is much larger than n. Therefore we have $\mathcal{O}(|H|+|TS|)^3$ total evaluations of the loop body.

The only step of the loop body which cannot be implemented in constant time is the check for sub-types (line 5). This check can be performed in polynomial-time [9] with respect to the size of the input types [3]. So, then the total complexity is this polynomial sub-typing cost times the number of loop body evaluations, which is clearly polynomial.

5.4 Handler Composition

When an intermediary receives a particular XML message, the intermediary needs to determine: for this message exactly which handlers should be used, on which message elements, and in what order? We use the trace information from the interface composition to construct a transformation-aware automata. This allows handlers to be dispatched as part of the standard message validation process.

Revisiting Figure 8 we now see how annotations were added during rewriting. These annotations are shown as superscripts on types: (h1) and (h2) for handler1-2. When a message is received by our intermediary, we validate that message according to the algorithm published by Thompson [9]. When some element validates to a type with a trace we know that type is not actually valid according to the target server schema. However, the trace annotation tells us the series of handlers which can be applied to

[3] As described in [9], this is because the regular expression types are restricted to be unambiguous.

create a valid message element. So, we dispatch the sub-tree rooted at that element to the sequence of handlers specified in the trace.

Finally, a transformation-aware automata is now illustrated by the example in Figure 9. Certain transitions introduced by rewriting are attached to handlers. When these transitions are followed, the handler is fed the XML message element for transformation.

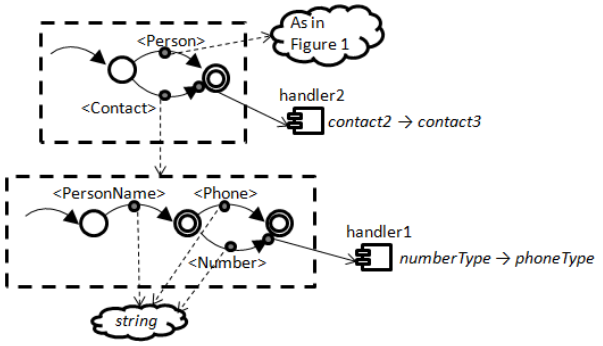

Fig. 9. Transformation-aware automata for the type constructed in Figure 8: `Person[..] || Contact[PersonName[..], Phone[..] || Number[..]]`.

6 Example Revisited

Recall the example client implements a mash-up of three services: eBay, FedEx, and UPS. This client prefers to use data formatted according to FedEx standards. So, they develop several transformation handlers, two of which are listed in Figure 10. Next, they compose these handlers with both the eBay and UPS schemas. Now we consider each of these in turn. We qualify elements with namespaces using a (:) to avoid confusion.

eBay to FedEx	`eBay:address[`*`Address`*`]` ↪ `FDX:destination[`*`Destination`*`]`
FedEx to UPS Validator Address	`FDX:destination[`*`Destination`*`]` ↪ `UPS:address[`*`AVContent`*`]`

Fig. 10. eBay, FedEx, UPS handlers

```
element SellTxResponseT =
  eBay:getSellerTxResponse[
    transaction[
      buyer[..,
        address[..]
      ],
  ..],
..]
```

```
element SellTxResponseT =
  eBay:getSellerTxResponse[
    transaction[
      buyer[..,
        address[..] ||
        UPS:address[..] ||
        FDX:destination[..]
      ],
  ..],
..]
```

Fig. 11. Original and composite eBay GetSellerTransactions response using transformations from Figure 10.

```
element AVType =
   UPS:AVRequest[
      RequestElement,
      address[AVContent]
   ]
```

```
element AVType =
   UPS:AVRequest[
      RequestElement,
      address[AVContent] ||
      eBay:address[Address] ||
      FDX:destination[Destination]
   ]
```

Fig. 12. Original and composite UPS Address Validation message using transformations from Figure 10. Elements in italics are references to schema elements not shown.

In Figure 11, on the left, we see part of the original type definition for the eBay `GetSellerTransactions` operation response. This is the first operation called by the client in our example workflow. On the right, is the composed type definition after being processed by the intermediary. Notice that the operation can now return addresses in either one of three formats. If the client needs to pass this returned information to another service as part of the mash-up, the client may no longer need to embed data formatting conversion as part of the mash-up business logic. This can be seen by considering the next operation in the example workflow.

Recall the second operation in the workflow was to validate the address returned from eBay. This is done through the UPS `AVRequest` operation, as shown on the left of Figure 12. Some of this information is specific to UPS, such as the `RequestElement` which carries UPS account information. We could not hope to provide any conversion from eBay account information to UPS. Still, we can see in the composite on the right of Figure 12, that at least the address information can be made uniform. Finally, in Figure 13 we see both the composite types after being filtered for

```
element SellTxResponseT =
  eBay:GetSellerTxResponse[
    transaction[
      buyer[..,
        FDX:destination[..]
      ],
  ..],
..]
```

```
element AVType =
   UPS:AVRequest[
      RequestElement,
      FDX:destination[Destination]
   ]
```

Fig. 13. Composite types for both eBay and UPS after being post-processed with a preference for FedEx.

a preference to FedEx, using the described post-processing step. Now, when the client receives an eBay response it will already include an address in FedEx format. That address can be taken and passed directly to the UPS address validation operation. So we can see how the implementation of the client might be simplified by making use of only the FedEx format whenever possible.

7 Related Work

7.1 Web services

The standard language for XML transformation is the eXtensible Stylesheet Language Transformation (XSLT). XSLT does not support any kind of inference for reasoning

about the composition of templates. We see our work, not as an alternative to such languages but as complementary. Our approach applies at the level of interface definition languages. So, it is agnostic to the implementation language that is used.

Several tools [12] 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 a composite type definition should be provided given a particular target output type.

In previous work [13] we proposed an AOP approach to programming transformation using the familiar advice-pointcut style. However that work did not address the generation of a composite schema or automatic middleware dispatching and shares very little in terms of technical details to the research here.

Work on the semantic web [14] including standards for semantic web services (WSDL-S) could help to ease the task of manually programming transformations for interoperability. However, since many enterprises still do not have semantic standards, we have focused on helping automate handler composition where the transformation handlers are programmed manually by developers using common languages such as Java and XSLT.

Ponnekanti [15] 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. We think their taxonomy provides a good overview to the kinds of problems where an intermediary can be useful.

7.2 Adapters for Components

Purtilo et al. [16] show how external adaptation can be valuable to reduce the non-functional constraints affecting application code. They provide a new language called Nimble that is used to generate adaptation code. In some cases, the generation of adapters can be automated [17] using type-based inference.

Gschwind et al. [3] provide a solution for the problem of composing software component interfaces and software built on different component models. That work was done under the paradigm of OO typing where sub-type relationships are explicitly declared. So an element-wise transformation approach was not applicable. To address element-wise transformation in a Web service environment, we have provided a new composition algorithm.

Other works [18] concentrate on mismatches between components at the behavioural level (i.e., the protocol between components). We have not addressed this issue in our research.

8 Conclusion

The transformation of an entire schema can be broken down into handlers responsible for only specific pieces. 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 to provide the flexibility of transformation without sacrificing an explicit interface contract for clients.

Currently we are planning an empirical study of the performance of our prototype implementation. We expect the performance overhead should not be prohibitive because the complex processing in our approach occurs mainly offline. We expect that at run-

time the actual message transformations will dominate the time for handler dispatching. Thus far this has been the case in our initial studies [11]. In addition to standard validation mechanisms, our approach only adds constant-time hashtable lookups to dispatch handlers at run-time. For high-performance applications, additional consideration will be needed to determine how the buffers which hold message streams should be allocated and copied. We have not yet considered these fine-grained performance characteristics.

This paper described an architecture and algorithm for composition of Web service interfaces with intermediaries. We provided a proof of algorithm correctness and showed that it executes in polynomial-time. Finally, we motivated and demonstrated a realistic mash-up scenario as an illustration of our approach.

References

1. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures. W3C Recommendation, 2004.
2. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web Services Description Language 2.0 Part 1: Core Language. W3C Recommendation, 2007.
3. Gschwind, T.: Type based adaptation: An adaptation approach for dynamic distributed systems. In: Proc. of the International Workshop on Software Engineering and Middleware. (2002)
4. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna component framework enabling composition across component models. In: International Conference on Software Engineering. (2003)
5. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Transactions on Internet Techbology **2**(2) (2002) 115–150
6. Chappell, D.A.: Enterprise Service Bus. O'Reilly (2004)
7. Wang, N., Parameswaran, K., Schmidt, D.: The design and performance of meta-programming mechanisms for object request broker middleware. In: Proc. of the COOTS. (2000) 677–694
8. Thompson, H.S.: Efficient implementation of content models with numerical occurrence constraints. In: Proc. of the European XML Conference (XTech). (2006)
9. Thompson, H.S.: Using finite state automata to implement W3C XML Schema content model validation and restriction checking. In: Proc. of the European XML Conference. (2003)
10. Hosoya, H., Pierce, B.: Xduce: A statically typed xml processing language (2002)
11. Forghanizadeh, S., Minevskiy, I., Wohlstadter, E.: Interfaces for web services intermediaries (Tech Report), Available online: `www.cs.ubc.ca/~wohlstad/extended.pdf`
12. Kirkegaard, C., Moller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Trans. on Software Engineering (2004)
13. Wohlstadter, E., Volder, K.D.: Doxpects: aspects supporting xml transformation interfaces. In: Aspect-Oriented Software Development. (2006)
14. Halpin, H., Thompson, H.: One document to bind them: Combining xml, web services, and the semantic web. In: World Wide Web Conference. (2003)
15. Ponnekanti, S., Fox, A.: Interoperability among independently evolving web services. In: Proc. of the International Conference on Middleware. (2004)
16. Purtilo, J.M., Atlee, J.M.: Module reuse by interface adaptation. Software Practice and Experience **21**(6) (1991)
17. Thatte, S.R.: Automated synthesis of interface adapters for reusable classes. In: Proc. of Symposium on Principles of programming languages. (1994)
18. DeLine, R.: Avoiding packaging mismatch with Flexible Packaging. In: Proceedings of the 21st International Conference on Software Engineering. (1999)