Compositionality of Handlers on Intermediate Servers in a Web Service Architecture

by

Sara Forghanzadeh

M.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
The Faculty of Graduate Studies
(Computer Science)

The University Of British Columbia
August, 2006
© Sara Forghanzadeh, 2006
Abstract

This research addresses the problem of inter-operability among different components in a Web service architecture on the Internet. "Message Handlers" are pieces of code that intercept the message between clients and services in order to create/modify different pieces of the message, or to perform a processing task. For several reasons, such as management, reusability, optimization, routing, providing value-added services and transformations, it might be desirable to have chains of handlers on intermediate servers on the path between the client and the server; however, automatic selection of the set of handlers to be executed and their order of execution is still a problem. We try to solve this problem based on the type of the messages that are sent by the clients and the expected server input type. We leverage two algorithms based on formal language theory in order to solve the handler composition problem. Our case studies show how our approach could help in solving the problem of schema migration and version incompatibility in a large system such as eBay.
# Table of Contents

Abstract ................................................................. ii

Table of Contents ....................................................... iii

List of Tables ............................................................ v

List of Figures ............................................................ vi

Acknowledgements ........................................................ vii

1 Introduction ............................................................ 1

2 Background Material .................................................... 4
  2.1 Service Oriented Architecture and Web services .......... 4
  2.2 XML (Extensible Markup Language) ....................... 4
  2.3 SOAP ............................................................... 6
  2.4 Message Handlers and Intermediaries ...................... 6

3 System Overview ......................................................... 10

4 Handler Composition Algorithms for Atomic and Sequential Types ......................................................... 12
  4.1 Atomic Types and Atomic Rules .............................. 12
  4.2 Sequential Types and Parametric Rules ..................... 13
    4.2.1 Context Free Grammar Solution ......................... 14
    4.2.2 Subtractive Parametric Rules ......................... 17
    4.2.3 Parsing Algorithm ........................................ 19
    4.2.4 Mandatory Handlers ...................................... 21
    4.2.5 Handler Precedence ...................................... 23
    4.2.6 Why Backtracking is not Needed ....................... 24
    4.2.7 Handlers that do not Change the Type ............... 24
    4.2.8 Disjunction ............................................... 24
  4.3 Case Studies ....................................................... 25
    4.3.1 Case Study 1: Purchase Order ......................... 25
    4.3.2 Case Study 2: Moblogging ............................. 27
  4.4 Summary of the Chapter .......................................... 30
# Table of Contents

5 **Handler Composition Algorithm for Tree Structured Types**  
5.1 The Tree Automata Solution  
5.1.1 Tree Automata Informal Definition  
5.1.2 Tree Automata Formal Definition  
5.1.3 Regular Tree Expression  
5.1.4 Regular Ground Tree Rewriting Systems  
5.1.5 The Membership Problem  
5.1.6 Regular Ground Tree Reachability and Handler Composition  
5.1.7 The Augmented Schema  
5.2 Case Study: eBay Schema Migration and Version Incompatibility  
5.2.1 Version Incompatibility  
5.2.2 Schema Migration  
5.2.3 Compatibility with other Web Services  
5.2.4 Localizing Data  

6 **System Implementation**  
6.1 Tools and Technologies  
6.2 Programming  
6.3 GTRS Solution Technical Details  

7 **Evaluation**  
7.1 Verification and Usefulness  
7.2 Timing and Efficiency  

8 **Related Work**  
8.1 XML Validation and Transformation  
8.2 Web Service Composition  
8.3 Dynamic Handler Chain  
8.4 Interface Adaptation  

9 **Conclusion**  
9.1 Contributions  
9.2 Future Work  

Bibliography  

A **First Appendix**  

B **Second Appendix**  

C **Third Appendix**
List of Tables

4.1 Possible list of handlers on the intermediary in case study 1 . . . 26
4.2 Possible list of handlers on the intermediary in case study 2 . . . 28
9.1 Table of comparison for different composition algorithms . . . . . 54
List of Figures

2.1 SOAP request and response message examples .................. 7
3.1 Request and response flow ....................................... 11
4.1 Intermediate server setup ........................................ 13
4.2 The LL(1) table-based parser architecture (taken from [1]) .... 19
4.3 Parsing table for the example grammar (taken from [1]) .......... 20
4.4 Application of intermediate server in Moblogging ............... 28
4.5 Summary of Chapter 4 ........................................... 30
5.1 SOAP Message Example – Tree Structure ........................ 32
5.2 The reduction and acceptance of f(a a) in Example 2 (taken from [2]) ......................................................... 34
5.3 (a) and (b) are the inputs to the reachability algorithm in examples 1 and 2 respectively ........................................ 37
5.4 Reachability algorithm inputs and outputs in our setup .......... 37
5.5 List of the calls affected by combination of SiteHostedPicture and VendorHostedPicture into PictureDetails ..................... 39
5.6 Mapping table for GetAccount request message migration .... 41
5.7 eBay ShippingAddress, UPS Address, and FedEx Destination type definitions ....................................................... 42
7.1 Efficiency test results .............................................. 49
Acknowledgements

Thanks to my supervisor, Eric Wohlstadter, for his great support, and the time and effort he put on this research with me. Thank you for spending time on answering my questions very quickly, even late at night or during the weekend.

Thanks to Gail Murphy for being my second reader and for your prompt and helpful comments on the thesis.

Thanks to my parents, and my brother, for being supportive, as you have always been in my life. You are the best.

Thanks to all my friends, especially Arash, Terry, Baharak and Shirin, for your support and your presence in my life. Thank you Arash for your patience, and for your help with Latex and Bibtex! Terry, thank you for helping me while moving.

Thank you Hermie for providing us with snacks in the lab whenever I was here, and also for all the administrative help you have given me.

Thanks to all the people in Software Practices Lab, and everyone I found the chance to know in Vancouver.
Chapter 1

Introduction

This research addresses the problem of inter-operability among different components in a Web service architecture on the Internet. Web services are pieces of software that can be accessed through a network; they provide many different services such as validating a credit card, providing information about the weather, stock quotes, or the existence of any ten digit phone number in Mexico. One of their main characteristics is flexibility, in the way that the clients do not have to use the same language or platform as the Web services in order to use them; this promises inter-operability between heterogeneous applications and technologies, as well as making the services highly reusable by different applications.

The most popular means of interaction with Web services is through XML based messages such as SOAP. Non-XML interactions with Web services are also possible, but for our purposes we will focus on the fact that services exchange XML messages and the service interface is based on a schema of these messages. An XML schema can be used to define the type of a Web service interface. Schemas are important in the software engineering of Web services and client applications. In addition to the XML schema, Web services provide other types of interface documents that show the other properties of their expected input message such as encryption or compression.

Before a request or a response message is sent or received by the client or the server, the message might be processed by several pieces of functionality that either create (or modify) different parts of it, or perform extra processing such as logging or authentication. These pieces of functionality represent different concerns, and are implemented using “message handlers” [3, 4, 5]. Handlers are pieces of code that intercept the message either on the client or the server. They might be named differently in other systems; for example, in SpringAOP [6] they are called “interceptors”. Handlers have the similar functionality as “before and after advice” in Aspect Oriented Programming [7] where the join point is the invocation or reception of a call to a remote method provided by a Web service.

For several reasons, it might be desirable to move a part of the functionality provided by message handlers to an intermediate server, which is a server located on the path between the client and the service provider. These reasons are discussed in Chapter 2, but in brief some of them could be management, reusability, optimization, routing, providing value-added services and transformations. An “intermediary” [8] intercepts the XML message between the client and the server. Message intermediaries are applications, or in other words, a set
of handlers located on intermediate servers that can process parts of a message as it travels from its origination point to its final destination point.

Composition of handler chains on an intermediate server is a challenge for the programmers and administrators. Currently, handlers are composed statically and manually through configuration files, or sometimes dynamically according to "precedence rules". The static method is sufficient if the number of handlers is limited and if there are no variations in the type of the messages that are received from different clients, but this is not always the case and this method is not scalable. Precedence rules are not sufficient either because sometimes according to the type of the message sent by the client, or the server's expected message type, the order of handler execution might vary. For example, one service might desire that the message should be signed first and then encrypted, while another service might expect the message to be signed after encryption. And besides, even though the handlers are composed dynamically, some programmer should write a code for every possible message input and output types to decide how the handlers should be composed in different situations.

In this research we try to solve the problem of dynamic and automatic handler composition on intermediate servers based on the type of the message. In other words, we try to solve the following problem: Given a client, a server and a set of handlers on an intermediate server, when the intermediary receives a message with a specific type from the client targeted to the server (or vise versa), what is the right set of handlers to be executed, and in what order should they be executed, so that the desired output type is generated?

One of the main applications of this approach is solving schema incompatibility problems. Differences in schemas between services can create headaches for programmers tasked with the challenge of composing such services. The differences can arise for variety of reasons, such as: evolving versions of services or semantically overlapping services from different vendors. For the first case consider the example of eBay. Since 2002 the schema has undergone thirty seven revisions for which five of the revisions break compatibility with previous versions. In the second case, consider the integration of eBay services with a package shipping service such as Fedex. Although both services deal with addresses and shipping locations, as is typical, there is no common understanding for the elements in distinct service interfaces. For each service composition, programmers are required to marshal data back and forth between various formats; but such marshaling should be confined to the connector code and not tangled with the business logic of applications. We discuss the details of these problems and the way our approach helps in solving them in the case study section in Chapter 5.

We introduce three different types for the messages, including atomic, sequential and tree-structured types. We provide solutions based on formal language theory to solve the composition problem for the last two type structures.

Chapter 2 provides background material on Service Oriented Architecture and Web services. Chapter 3 presents an overview of the system. Chapters 4 and 5 show how we have leveraged existing algorithms to implement handler composition, which is the main contribution in this research. The former chapter
Chapter 1. Introduction

presents algorithms for handler composition using atomic and sequential types, while the latter describes an algorithm considering tree structured types. The case studies at the end of each chapter show the application of these algorithms. Information about system implementation is provided in Chapter 6. Chapter 7 presents an evaluation of our approach. In Chapter 8 some related work is presented. Chapter 9 is the conclusion, which describes the contributions and the future work.
Chapter 2

Background Material

In this chapter, we present background material on Service Oriented Architecture and Web services, XML, SOAP, message handlers, and intermediaries.

2.1 Service Oriented Architecture and Web services

Service Oriented Architecture is the underlying structure supporting communications between services. Services are self-contained, reusable software modules with well-defined interfaces and are independent of the applications that are using them and the computing platforms on which they run [1].

SOA uses the find-bind-execute paradigm [9], where service providers register their service in a public registry. This registry is used by consumers to find services that match certain criteria. If the registry has such a service, it provides the consumer with a contract and an endpoint address for that service. Web services are the most popular representation of Service Oriented Architecture.

2.2 XML (Extensible Markup Language)

Part of our solution is based on the structure of XML messages; the information presented in this section will aid in the understanding of Chapter 5.

XML [10] is a simple, generic format for semi-structured data that has been standardized by the World Wide Web Consortium.

"Elements" [10] are the basic building blocks of XML markup, and may be thought of as containers: they may contain character data, other elements, and/or other markup (comments, entity references, etc.) Elements are delimited with a start-tag and an end-tag. If an element has no content, it is known as an empty element, and may be represented with either a start-tag/end-tag pair or using an abbreviation: the empty-element tag.

All three types of tags are shown in this example:

```xml
<image> <!— start-tag -->
<file src="logo.png" /> <!— empty-element tag -->
</image> <!— end-tag -->
```

Well-formed XML data is defined as being in the form of a simple hierarchical tree, with one, and only one, root node, called the "document root". The
document root will always contain a body. The body is comprised of a sub-tree of elements. The body sub-tree always has a single root node called the "root element". All other elements in an XML document are descendants ("children") of the root element, and each of them can be a root element for a new subtree.

Any XML data object is considered valid XML if it is well formed, and it meets certain further validity constraints and matches a grammar describing the document's content. XML can provide such a description of document structure in the form of an XML Schema [10].

In XML Schema, a "type" attribute may be assigned to an element. In programming languages types are the items that get reused. It has been suggested that XML Schemas are a "type-based system", therefore the idea behind schemas is to reuse types [10]. Types may be built-in (e.g., string, integer, etc.) or user-defined. User-defined types may be simple or complex. As an example, consider the complex type Address.

```
<xsd:complexType name = "Address">
    <xsd:sequence>
        <xsd:element name="Street1" />
        <xsd:element name="Street2" />
        <xsd:element name="City" />
        <xsd:element name="State" />
        <xsd:element name="Zip" />
    </xsd:sequence>
</xsd:complexType>
```

The attribute called "name" holds the name of the complex type we are creating. This is followed by a sequence element to indicate that the child elements should appear in the order in which they are declared. The element declarations themselves, representing the content model, are then nested inside.

The Address type may be used as follows:

```
<xsd:element name="mailingAddress" type="Address" />
<xsd:element name="billingAddress" type="Address" />
```

Each of these elements now contain the same child elements:

```
<mailingAddress>
    <Street1/>
    <Street2/>
    <City/>
    <State/>
    <Zip/>
</mailingAddress>

<billingAddress>
    <Street1/>
    <Street2/>
    <City/>
</billingAddress>
```
2.3 SOAP

The interactions in a Web service architecture are through a set of XML-based open standards. SOAP is currently the most popular XML/Web service technology; it is a framework for exchanging XML-based information in a network. A SOAP envelope includes a "header" section and a "body" section. A SOAP header typically contains authentication information, transaction ID, digital signature, routing information and so on.

Figure 2.1 shows the body sections for an example SOAP request message and its response. The message is sent to a service provided by a warehouse, requesting product information. `getProductDetails` is similar to a method that returns this information and its input parameter is the product ID. The SOAP envelope contains a header showing the client’s username and password. The output is returned in the response message.

2.4 Message Handlers and Intermediaries

"Message handlers" [3, 4] provide additional message-handling logic to Web services as extensions to the business logic implementations of those services. Handlers are pieces of code that intercept the message either on the client or the service side. Handlers might be named differently in other systems; for example, in SpringAOP [6] they are called "interceptors". They have the similar functionality as "before and after advice" in Aspect Oriented Programming [7] where the join point is the invocation or reception of a call to a remote method provided by a Web service. A "handler chain" represents an ordered group of handlers.

For several reasons we may desire to have handlers on intermediate servers as well as the client and the server. An "intermediary" [8] intercepts the XML message between the client and the server. Message intermediaries are applications, or in other words, chains of handlers located on intermediate servers that can process parts of a message as it travels from its origination point to its final destination point. The route taken by a message, including all intermediaries it passes through, is called the "message path". The following are some motivations for using intermediaries (which are not mutually exclusive):

- **Management**: To provide auditing, logging, metric collection, service lookup, monitoring, performance mediation, fault management, tracing, and so on. You may find examples of this kind in our two case studies in Chapter 4. Also in [11], intermediaries have been used for this purpose.

- **Optimization**: To offer improved performance, availability and scalability of services. Caching on behalf of the client or the server is one of
Chapter 2. Background Material

Request Message:

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsse='http://schemas.xmlsoap.org/ws/2002/04/secext'>
  <soap:Header>
    <wsse:Security>
      <wsse:UsernameToken Id='http://xmethods.net/xspace'>
        <wsse:Username>sara</wsse:Username>
        <wsse:Password>sara</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>99345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Response Message:

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsse='http://schemas.xmlsoap.org/ws/2002/04/secext'>
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Sharp 20" LCD EDTV</productName>
        <productID>99345</productID>
        <description></description>
        <realPprice>699.99</realPprice>
        <salePrice>599.99</salePrice>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2.1: SOAP request and response message examples
the examples. Servers can offload their processing by including processing instructions to intermediaries in their messages. Also, if there are clients that have CPU, power or memory constraints (such as mobile devices) an intermediary can perform a part of their tasks on their behalf. To learn more about optimization through intermediaries in a Web service architecture please refer to [12].

- **Reusability**: The handlers on an intermediate server could be shared by many different clients and many different servers. Besides, if a task on the clients has the potential to change very frequently, it is easier to – if possible – move the task to an intermediate server and update the task only on the intermediary rather than all of the clients. For more information please refer to [12].

- **Routing**: The most basic function of an intermediary is routing, where the intermediary acts as an access point for purposes of access or traffic control, logging or relay. Routing intermediaries are usually deployed at administrative boundaries in the network, in order to both control the traffic flowing across them, and simplify administrative tasks such as client configuration.

- **Providing value-added services**: As an example, message processing intermediaries may be able to add local information unavailable at the server, to provide localized services. Other examples are encryption and adding digital signature to a messages for security. Examples of this type are presented in our two case studies in Chapter 4.

- **Transformations**: Currency conversion, temperature conversion, world time conversion, and transformation between the old and new versions of an API are a few examples. Our case study in Chapter 5 shows how handlers can be used for this purpose.

The client may or may not be aware of the existence of the intermediaries on the path. A “transparent” intermediary [8] is one the client knows nothing about; the client believes it is sending messages to the actual service endpoint. An “explicit intermediary” involves specific knowledge on the part of the client; the client knows the message will travel through an intermediary before getting to the final server.

For example, a third-party intermediary that provides authentication for an organization could be transparent; the organization providing the service would publish the address of the intermediary as the service endpoint. A caching or monitoring intermediary that is owned by the clients in an organization (that are sending messages to a service that is not owned by that organization) is an example of explicit intermediate servers. In our setup both situations are possible.

In our system we deal with message handlers on intermediaries and the messages are XML based. We provided the information in this section to give
Chapter 2. Background Material

the reader a better understanding of the message handlers and the intermediaries as well as their application in real systems.
Chapter 3

System Overview

Our system – which runs on intermediate servers – provides a mechanism for dynamic and automatic composition of handlers based on the message type. There are two different ways to define a type for an XML based message: it might have a content based type and a non-content based type. There are a few variations in the definition of content based and non-content based types in the literature. In this research we define the content based type according to the structure and the elements of the message body. We define non-content based type according to the non-content data including message headers and other envelope features, such as encryption, signature, compression, etc.

Handlers can modify the content based and non-content based types of a message. We provide two different algorithms for handler composition; one of them is based mostly on non-content type compatibility, determining the set of handlers to be executed and their order of execution according to the non-content type of the message sent by the client, and the server’s expected input type. It also considers other factors such as handler precedence and mandatory handlers. The other algorithm selects the set of handlers based on the content based type of the input message as well as the server schema. In both algorithms, the chain of handlers might be composed to affect either the request or the response message. So, in brief, the input to our system is an XML message which has a type, and the output is the modified XML message whose type has been adapted to the server’s expected input type.

We assume that the type of the message accompanies the message (especially for the non-content based types), or as for the content based type, the type can also be inferred from the message content.

The server’s expected input type can be obtained from the metadata it provides, in other words, the schema documents including the XML schema.

Since the client does not have its own explicitly published interface, we assume that the client’s expected response type is accompanied with the request message implicitly or explicitly. For example, if the client uses a specific version of the schema, and the intermediate server provides a transformation between the versions, the version number is usually a message header in the request message. The intermediate server can then change the response message according to the version number in the request message. Also, the clients have the option of registering a response schema with the intermediate server. This is useful especially when the client is expecting “mixed messages”, for example, a mixture of old and new fields from different versions of the schema server.

Figure 3.1 shows the scenario of a request and a response message going
through an intermediary, and the chain of handlers' composition. We use the notation $\rightarrow$ to show a type conversion. The expected server input type is $B$, and the client's expected response type is $D$.

If the intermediary is explicit (means the client knows about it and should specify its path in the message), then the clients also have the option of asking the intermediate server about the possible conversions. For example, in figure 3.1, the client might want to make sure the conversion between $A \rightarrow B$ is possible using this intermediary as part of a deployment-time check, rather than discovering the error at run-time. For the non-content based typing, this question could be in terms of a composition question (like $A \rightarrow B$?) sent to the intermediary. The client may receive a negative or positive answer to the question.

For content based message type, we are using an algorithm that given a repository of handlers and a target schema, it will compute an augmented schema. The augmented schema is the transitive closure of the transformation relation induced by the repository on the target schema. Programmers need not specify any specific ordering of handlers, as all type correct orderings are taken into consideration automatically. So basically the clients can see what are the possible types for the message content according to the server schema and the transformations that intermediary can make.

Figure 3.1: Request and response flow
Chapter 4

Handler Composition
Algorithms for Atomic and Sequential Types

In this chapter we provide algorithms for handler composition based on atomic and sequential message types. We define simple types for the messages at the beginning, then make it more elaborate considering different parameters. The two case studies provided in this chapter give a better understanding of the algorithms.

4.1 Atomic Types and Atomic Rules

Each handler can change the type of an XML message. In this section we assume that every message has an atomic (single) type such as $A$. The handlers can make conversions of type $A \rightarrow B$ where $A$ and $B$ are both atomic types. By atomic we mean that the algorithm cannot reason about the internal structure of the type.

As an example, imagine an organization which has many daily transactions with a warehouse to order various items. There are many clients in the organization, but the whole organization uses a unique shipping address, as well as a unique account; also the organization provides the financial information for purchasing the items. The client sends a SOAP message containing the information about the required item to an intermediate server owned by the organization. Let's call the type of this message “WrhItem”. The intermediate server adds the shipping address, account information and financial information to the message and forwards it to the Web service provided by the warehouse. Let's call the new type “WrhTx”. Figure 4.1 shows this process.

On the intermediate server, there are handlers that make the following conversions (as well as a few other handlers):

- $WrhItem \rightarrow WrhAddressed$ (Adds the shipping address to the message)
- $WrhAddressed \rightarrow WrhAccount$ (Adds account information to the message)
- $WrhAccount \rightarrow WrhTx$ (Adds the financial information to the message)

The question is, given the above set of handlers, is the $WrhItem \rightarrow WrhTx$ conversion possible? More generally, the problem may be stated as follows:
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

Problem: Given an atomic input type $A$, an atomic output type $B$, and $n$ functions which convert between various types (in the format of $P \rightarrow Q$ where $P$ and $Q$ are atomic types) is there a composition of the functions which together convert from type $A$ to $B$?

In [13] Gschwind provides a solution for the problem of composing software component interfaces; this solution applies to our problem as well. According to this solution, we need to model each type as a node in a graph. The functions are directed edges between the function input type node and function output type node. To find a composition we just need to run a path finding algorithm such as Dijkstra's single source shortest path; the path between the source and destination nodes shows what handlers should be used in order to make the conversion. The time complexity of the algorithm is $O(|V|^2)$, where $|V|$ is the size of the handlers set.

4.2 Sequential Types and Parametric Rules

The type of an XML message cannot always be specified using only an atomic type. For example, the type of a message might also depend on its headers, or several operations that are performed on the envelope such as encryption or compression. In these cases, we allow the type of the message to be specified using a sequence of types, as described in this section. This sequence of types mostly specifies the non-content type of the message.

In the Web services domain, handlers may make *additive parametric conversions* (vs. atomic conversions). We specify these conversions using the following notation:

$$X \rightarrow X, Y$$

In this notation, $X$ is just any type (a parameter) and $Y$ is the new type element appended to the type sequence. In Java, this concept could be represented as $X \rightarrow Y<X>$. We use the above notation because in practice, new
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

type elements, like message headers, are simply appended to the current list of elements (headers and other features).

We define the sequential type as a sequence (i.e., string) of atomic types where ordering is important. In this chapter, we seek to model the SOAP header contents and other non-content features of the message envelope, so we represent the SOAP body using a single atomic type, which is the first element in the type sequence. In Chapter 5, we deal specifically with the SOAP body. The next elements in the type sequence are non-content properties of the message (i.e., headers and envelope features).

For example, suppose we have a handler that signs a SOAP message on an intermediate server before forwarding the message to the service. This handler is to be used by different applications, so it will sign the message no matter what the type of the message is \((X \rightarrow X, Signed)\). If the type of the message is \(A\), the output type will be \(A, Signed\).

Now we have two different categories of conversions: atomic conversions which change the atomic type of the message body, and additive parametric conversions which append new (non-content) type elements to the type sequence. We can rewrite the problem in the previous section as follows:

**Problem:** Given a sequential input type \(A\), a sequential output type \(B\), and \(n\) functions which convert between various types (either atomic type conversion in the format of \(A \rightarrow B\) or additive parametric conversions in the format of \(X \rightarrow X, Y\) where \(X\) could be any type, and \(Y\) is a single type element), is there a composition of the functions which together convert from type \(A\) to \(B\)?

The simple graph solution cannot handle this problem because of the parametric rules, and because it cannot answer the question whether \(A \rightarrow B\) is possible if \(A\) and \(B\) are sequential types. We provide a new solution to this problem in this section. Notice that time complexity is an important factor in finding a solution and we desire to minimize the time complexity as much as possible.

4.2.1 Context Free Grammar Solution

We reduce the above problem to the problem of parsing context free grammars in computational linguistics.

Parsing is the process of analyzing an input sequence in order to determine its grammatical structure with respect to a given formal grammar [1]. It also determines whether the input sequence belongs to the language described by the given formal grammar or not.

A grammar \(G\) is defined as a quadruple \(G = (V, S, T, P)\), where

- \(V\) is a finite set of objects called *variables*,
- \(T\) is a finite set of objects called *terminal symbols*,
- \(S \in V\) is a special symbol called the *starting symbol*,
- \(P\) is a finite set of *productions*. 
The sets $V$ and $T$ are non-empty and disjoint.

A grammar $G = (V, S, T, P)$ is said to be context-free [14] if all productions in $P$ have the form $A \rightarrow w$, where $A \in V$ and $w \in (V \cup T)^*$.

To solve our problem, we create a context-free grammar according to the handlers' type conversions. The question is whether $Source \rightarrow Destination$ conversion is possible, in other words, if $Destination$ is reachable from $Source$ using this grammar. To solve this reachability problem, we will have the starting symbol in the context free grammar produce $Source$, and then find out whether $Destination$ is accepted (can be produced) by this grammar.

**Claim:** What the following context-free grammar does is equivalent to how a set of handlers with atomic and additive parametric rules can change the type of a message:

$$
G = \{ \Sigma, V, R, S \}
$$

$\Sigma$ : All the existing types in intermediary’s dictionary and on the left side of composition question (in small letters)

$V$ : All existing types in atomic rules and left side of the question (in capital letters)

$R$ : \( \{ S \rightarrow SourceX \} \)

$U \{ \text{Atomic Rules} \}

$U \{ \forall a \in \Sigma : A \rightarrow a \}

$U \{ \forall Y \text{ type element in a parametric rule} : X \rightarrow Y' \}

$U \{ \forall Y \text{ type element in a parametric rule} : Y' \rightarrow yX \}

$U \{ X \rightarrow \lambda \}

Where $A$ is the empty string. We have a table that corresponds atomic rules and rules of type $X \rightarrow Y'$ to the respective handlers. As an example, suppose we have a set of handlers that make the following conversions possible:

- $A \rightarrow B$
- $B \rightarrow C$
- $X \rightarrow X, Signed$
- $X \rightarrow X, Encrypted$

$(X$ is a parameter, it could be any type/type sequence)

Suppose the composition question is if “$A \rightarrow C, Signed, Encrypted$” is possible. We build the following context free grammar according to the conversion rules and the question:
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

\[ G = \Sigma, V, R, S \]
\[ \Sigma = a, b, c, signed, encrypted \]
\[ V = A, B, C, Signed', Encrypted', X \]
\[ R = \{ S \rightarrow SourceX \} \]
\[ S \rightarrow AX \]
\[ \bigcup \{ \text{Atomic Rules} \} \]
\[ A \rightarrow B \]
\[ B \rightarrow C \]
\[ \bigcup \{ \forall a \in \Sigma : A \rightarrow a \} \]
\[ A \rightarrow a \]
\[ B \rightarrow b \]
\[ C \rightarrow c \]
\[ Signed \rightarrow signed \]
\[ Encrypted \rightarrow encrypted \]
\[ \bigcup \{ \forall Y \text{ added by an additive parametric rule} : Y' \rightarrow yX \} \]
\[ Signed' \rightarrow signedX \]
\[ Encrypted' \rightarrow encryptedX \]
\[ \bigcup \{ \forall Y \text{ added by an additive parametric rule} : X \rightarrow Y' \} \]
\[ X \rightarrow Signed' \]
\[ X \rightarrow Encrypted' \]
\[ \bigcup \{ X \rightarrow \lambda \} \]
\[ X \rightarrow \lambda \]

And, the question will change to whether \( c \text{ signed encrypted} \) is accepted by this grammar.

**Informal Justification:** For handlers with atomic conversions (like \( A \rightarrow B \), which correspond to the change in the type of the message body) we have the same rules in the grammar; so the grammar just reflects what the handlers of this type do. Handlers that perform additive parametric conversion (\( X \rightarrow XY' \)) add a type element to the end of the type sequence. The grammar is created in a way that it enforces type elements to be added to the end of the type sequence; this behavior results from the rule \( S \rightarrow SourceX \) and rules in the format of \( X \rightarrow Y' \), \( Y' \rightarrow yX \) and \( Y' \rightarrow y \). Notice that we use \( Y' \) instead of \( Y \) in the grammar so that if a \( Y \) already exists in the Source, it cannot be changed to \( YX \), and elements are only appended to the "end" of the sequence. Therefore the grammar exactly represents the handlers' behavior.
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

4.2.2 Subtractive Parametric Rules

There might be handlers which undo the effects of other handlers; in other words, they might remove a type element added by another handler. For example, suppose $Y$ is encryption. We want to encrypt a message on the intermediate server, and decrypt it on the server. So a handler on the server side should make a conversion of type $X, Y, X' \rightarrow X, X'$ (where $X$ and $X'$ are parameters, in other words, they can be any type or even null). In general these handlers can remove a type element from anywhere in the type sequence. We call this category of type changing "subtractive parametric conversion" (vs. additive parametric conversion). Let's see how the solution in the previous section could be extended to support this new category of conversions.

To support subtractive parametric conversions, the production $Y \rightarrow A$ should be added to the grammar for each subtractive parametric rule that can remove the type $Y$ from the type sequence. Everything else is just the same as before. However, we should make sure that we choose a parser that accepts context free grammars which include $A$ (empty) productions, as not all context free parsers accept grammars of this kind.

Prefix Type Elements

In Web services domain, sometimes when we add a type element to the message by additive rules, we cannot change the previous type elements in the type sequence before that type is removed. Specific headers that change the type of a message are not like that, but for example, compression and encryption are of this kind. If the type of a message is $A, Enc$ and $Enc$ is encryption, the type cannot be changed to $B, Enc$ even if we have the atomic rule $A \rightarrow B$; we have to remove $Enc$ first (decrypt the message), perform $A \rightarrow B$, and add $Enc$ again (encrypt again). In order to distinguish these especial type elements, we specify them using the $[ ]$ notation, and we call them prefix type elements. This means that nothing to the left of the $[ ]$ element can be modified until the $[ ]$ element is removed. For example, if part of a message is encrypted, it cannot be modified until it is unencrypted.

The CFG algorithm is not powerful enough to enforce this restriction, so we pre-process the composition question to take care of the prefix type elements. The following algorithm shows how to do this. Basically, we start processing the left hand side of the composition question (Source in $Source \rightarrow Destination$) from right to left. Whenever we reach a $[ ]$ notation, we just find out if the conversion can be done without removing it – if not, we look for a removing rule. If there is such removing rule, we remove it and continue; otherwise the conversion is not possible.

1The reason for naming these type elements this way is that they always apply to the prefix of the type element; for example, this approach does not deal with the case where the first half of a message is encrypted by one algorithm and the second half is encrypted by another algorithm, e.g. $A, [Enc1], B, [Enc2]$ means $A, [Enc1], B$ is encrypted using $[Enc2]$; we cannot show the case where $A$ is encrypted using $[Enc1]$ and $B$ is encrypted using $[Enc2]$.   

17
1. Start processing from the right side of Source

2. Continue moving to the left on Source until the sequence is finished or until you get to a symbol notified by [ ]. If the sequence is finished then goto 5

3. Name the symbol, [Y]. If the prefix of Source ending with [Y] is the same as the prefix of the Destination
   TempSource = Source without that prefix
   TempDest = Destination without that prefix
   if TempSource → TempDest is possible
     then Possible = true, goto 5
   else
     Look for a handler which removes Y
     if Y → λ is found
       Remove Y from Source, Goto 2
     else
       Possible = false, goto 5

4. if Source → Destination is possible
   then Possible = true
   else
     Possible = false

5. Done

Claim. Given a composition question including [Y] on the right hand side (Destination), we may ignore the [ ] notation and find out if the conversion is possible. The solution would be no different with or without [ ] on the right hand side.

Informal Justification. Either [Y] exists on the left hand side of the question (the Source) as well, or not.

If it exists, it is taken care of while running the previous algorithm – [Y] is removed from the source before anything on its left side can be changed; and if we have a rule to add it again, it will be added as a part of the destination type.

If it does not exist, it should be added by a parametric rule. As you will see later, we choose our context-free parser in the way that it produces the sentence from left to right (leftmost derivation); so the elements on the left side of [Y] are changed or added before [Y] is added. Therefore we can just ignore the "[ ]" notation and execute our normal parsing algorithm to build the destination type.
4.2.3 Parsing Algorithm

In this section we describe the parsing algorithm we use in order to parse the context free grammar. It is important to know how this parser works especially because we take advantage of its architecture in next sections, when we add more features to our system.

We use an LL parser to parse the introduced grammar. An LL parser is a top-down parser for a subset of the context-free grammars. It parses the input from left to right, and constructs a Leftmost derivation of the sentence (Hence LL, compare with LR parser). The class of grammars which are parseable in this way is known as the LL grammars [1]. Our grammar is also an LL grammar.

![LL(1) Table-Based Parser Architecture](image)

Figure 4.2: The LL(1) table-based parser architecture (taken from [1])

Figure 4.2 shows the structure of a table-based or LL parser. This parser has a buffer for the input string, a stack on which it keeps symbols from the grammar (it could contain non-terminals and terminals, as well as the $ symbol which shows the end of a sentence), and a parsing table. The parsing table tells what grammar rule to use given the symbols on top of its stack and its input tape. The following example, taken from [1], shows how this parser works.

Suppose we have the following grammar:

1. $ \rightarrow F$
2. $ \rightarrow (S + F)$
3. $F \rightarrow 1$

For this grammar, we will have the following parsing table:

The parser looks at the top-most symbol on the stack and the current symbol in the input string, and decides which rule should be applied according to the parsing table (If we have 'S' at the top of the stack and '1' is the next symbol in the input stream, rule 1 will be applied and 'S' is replaced by 'F'). The content
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>1</th>
<th>+</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4.3: Parsing table for the example grammar (taken from [1])

of the stack is [S, $] at the beginning. The parser may do one of the following according to the type of the symbol at the top of the stack:

1. If the top is a nonterminal: It looks up in the parsing table on the basis of this nonterminal and the symbol on the input stream which rule of the grammar it should use to replace it with on the stack. The number of the rule is written to the output stream. If the parsing table indicates that there is no such rule then it reports an error and stops.

2. If the top is a terminal: It compares it to the symbol on the input stream and if they are equal they are both removed. If they are not equal the parser reports an error and stops.

3. If the top is $ : If the latest symbol on the input stream there is also a $ then the parser reports that it has successfully parsed the input, otherwise it reports an error. In both cases the parser will stop.

This algorithm will result in either an error, or the left-most derivation of the input.

The algorithms for constructing the parsing table can be found in Appendix A. If the table contains at most one rule in every one of its cells, then the parser will always know which rule it has to use and can therefore parse strings without backtracking. It is in precisely this case that the grammar is called an LL(1) grammar. LL(k) parsing uses the next k input tokens to make parsing decisions. LL(1) is LL(k) for the case where k=1. So in a LL(k) parsing table, instead of assigning a single terminal to each column, some columns might be specified by more than one terminal (such as “aa” or “ab”).

Nondeterminism

A context-free grammar G is said to be ambiguous if there exists some $w\in L(G)$ which has at least two derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations [14]. If a grammar is ambiguous, then it is neither LL(1) nor LL(k). Many grammars can be made LL(1) by first removing ambiguity, then removing left-recursion, and finally left-factoring the grammar.
What happens when the same lookahead set or sequence predicts more than a single alternative – in other words, we have more than one rule in at least one entry of the parsing table? We have a nondeterminism as we cannot figure out what to do. Either this arises from a true ambiguity in the grammar or from a weakness in the parsing strategy; in general you cannot determine which one it is. We may call this situation \textit{nondeterminism}.

Theoretically our grammar is an LL(1) grammar and it is not ambiguous (because of its special structure). In this grammar, the rules that represent additive/subtractive parametric handlers cannot cause ambiguity (because they just add an element or remove one to/from the type sequence, and there is only one way of adding or removing such elements – using only the set of rules that represent those parametric conversions). The atomic rules are in the $A \rightarrow B$ format and cannot cause ambiguity; using these rules, either a atomic type is reachable by another one or not.

Although our grammar is not theoretically ambiguous, it has some type of nondeterminism that is specific to our domain. The first reason for nondeterminism is that there might be more than one handler making the same conversion (for example, both may do $A \rightarrow B$ or $X \rightarrow XY$). Consequently, we will have more than one instance of each rule. In theory these rules are not different, but in our domain, since these rules correspond to different handlers which are different entities in real world, they are different. So we may have more than one rule in some of the table entries.

Secondly, it might be possible to reach one atomic type from another using different sets of atomic rules. For example if the following three rules are a subset of our grammar:

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \rightarrow B$
4. $C \rightarrow b$

Then there will be more than one way to reach c from A: You may use rule number 1 or 2 for that purpose. In theory if you have $A \rightarrow B$ and $B \rightarrow C$, it means that naturally by transitivity you have $A \rightarrow C$ as well, and these are not considered as different derivations. But in our domain, again, since these rules correspond to different handlers in practice, they could be considered different derivations.

In the next sections of this chapter you will see how we deal with this nondeterminism and how we will choose a rule to execute when there is more than one rule in a table entry.

\section*{4.2.4 Mandatory Handlers}

As mentioned before, each intermediate server has a list of the target services for which it behaves as an intermediate server. This list may also contain some in-
formation about each service. This information may include a list of mandatory
handlers registered for each service; this means that those handlers should be a
part of the composition in any case. For example, caching might be a manda-
tory handler for the stock quote service (means every message that is pointed
to this service should be cached on the intermediate server). Logging could be
another mandatory handler for many service owners who wish to monitor the
requests through an intermediate server. The question is how should we make
sure that these handlers are certainly a part of the solution in handler composi-
tion. Let us call the grammar rules that correspond to mandatory handlers the
mandatory rules.

Let's consider the parametric handler conversions first. If there is a manda-
tory handler that makes an additive parametric conversion, it means that the
type element that this handler adds to the sequence should exist in the output
type. If it does not exist, then the composition is not possible. If it exists,
it means the handler will be used; the only reason why it may not be used is
that there is more than one handler that makes the same type conversion. This
means we will have more than one rule in a parsing table cell; so we will choose
the mandatory rule to execute.

In the above case, if there is more than one mandatory rule in a cell (e.g. a
message needs to be compressed with two different algorithms twice, but both
of them add the \texttt{[Compressed]} element to the type sequence), we choose one of
them to execute by random; then we will remove the selected rule from that cell
and from every other cell in the table (notice that if two mandatory rules exist
in one cell, it means that they perform the same task – so if one of them exists
in any other table entry, the other will be there too). This way, if a conversion
needs to be done more than once, the second one will be used, and we make
sure that the first mandatory rule does not prevent the second one from having
a chance to be executed in the same cell or elsewhere.

If one of the mandatory handler for a service is not being executed in the
end, this means there is an inconsistency between the server expected input type
and the assignment of mandatory handlers to the service, or inconsistency with
the client output type. In any case it means the composition is not possible if
we want to use all mandatory handlers.

For atomic rules – which change the type of the body – the same case is
possible (more than one handler performing the same conversion, and we will
choose the mandatory one). Also as explained in previous section, there might
be more than one way to make a conversion, \(A \rightarrow c\) when we have \(A \rightarrow B,\)
\(A \rightarrow C, B \rightarrow C, C \rightarrow c\). Again if there is a mandatory rule, it will be chosen.
If among \(A \rightarrow B\) and \(B \rightarrow C,\) only \(B \rightarrow C\) is mandatory, then for getting from
\(A\) to \(C, A \rightarrow B\) becomes mandatory as well, but we call it \texttt{virtual mandatory}.
The virtual mandatory rules in each cell can be found while we are finding the
first and follow sets for non-terminals\(^2\).

\(^2\)To be more specific, if we have three rules like \(A \rightarrow B, B \rightarrow C,\) and \(C \rightarrow c,\) where \(B \rightarrow C\)
is mandatory, \(c\) will be added to the first set of \(B\) because of \(B \rightarrow C;\) so in the first set of \(B,\)
we distinguish \(c\) from other elements. When \(c\) wants to be added (or re-added) to the first
set of \(A\) because of the rule \(A \rightarrow B,\) as \(c\) is a distinguished element, we specify \(A \rightarrow B\) as
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

For atomic body type elements, which are transformed by these atomic rules, we make the assumption that it is not possible to have more than one mandatory rule (virtual or non-virtual) in a table cell (conceptually it does not make sense either; if we want to reach the type $C$ from $A$, we can reach it only through one way – why should there be two mandatory ways for reaching it?). If the handler rules happen to be in the way that more than one mandatory atomic rule occurs in a table cell, we will return a warning saying that the algorithm might not be able to work properly with this setup.

4.2.5 Handler Precedence

Conceptually, there are handlers that should be executed before/after some other handlers. We define the following precedence attributes for handler $H$:

\[
\text{precedes}(H_1, H_2, ..., H_m) \\
\text{succeeds}(H_1, H_2, ..., H_m)
\]

Where $H_1$ to $H_m$ are handler names. \textit{Precedes} means if both handler $H$ and handler $H_i$ from the list are executed, $H$ should be executed before $H_i$. Similarly, \textit{succeeds} means if both handler $H$ and handler $H_i$ from the list are executed, $H$ should be executed after $H_i$.

A note here is that we define precedence only for handlers that do not change the type (as you will see in Section 4.2.7), or for the handlers that perform the same task (i.e. technically they will appear in the same table cell). The reason is, if they are not performing the same task, the handler sequence is composed according to the expected server input type, and that input type, as well as the prefix type element constraints, enforce the order of the handlers to be executed; in other words, if they change the type, the precedence will naturally be taken care of according to the server input type.

For the handlers that perform the same task, first of all, if there is more than one handler in a table entry, we can consider precedence to select a rule. Secondly, we assign a precedence list to each rule, which determines a list of the rules that should be executed before this rule. Obviously, this list includes the corresponding handler's \textit{succeeds} list, plus any other rules corresponding to handlers which this handler is in their \textit{precedes} list. Whenever this rule is executed, those rules cannot be used anymore, since if they were to be executed, they had to be executed before this rule. So we omit them from the grammar rules list and if it was not already removed from that list, we also remove them from the whole table.

\^

\footnote{Also for the atomic body rules that are in the same table cell but make different type changes we define precedence; e.g., $A \rightarrow B$ and $A \rightarrow C$ in the previous section. Notice that if $A \rightarrow C$ precedes $A \rightarrow B$ for reaching $C$ from $A$, the $A \rightarrow B$ rule will also be omitted from the cell where we reach $B$ from $A$; however, we assume that it is unlikely that we want to get back again to type $A$ by executing other handlers and need to change it to $B$!}

\footnote{a virtual mandatory rule for reaching $c$. These are true only for atomic types/rules that are assigned to the body, and not for non-content additive types added/removed by parametric rules}
Notice that removing the rules from the parsing table does not increase the time complexity greatly—though it might seem to do so. According to the aggregate analysis [15], the amortized cost of removing the grammar rules cannot be worse than $|G||T|$ times the parsing time of the grammar, where $|G|$ is the size of the grammar and $|T|$ is the size of the parsing table. The reason is, each grammar rule cannot be removed from the grammar rules list—and from the table—more than once.

### 4.2.6 Why Backtracking is not Needed

As discussed before, we may encounter nondeterminism in the parsing table because two or more handlers may make the same conversion, or there might be more than one way to make a conversion. We also saw that sometimes mandatory handlers and handler precedence can help in selecting a rule in a table entry. In general, in each table cell, we will just put the mandatory rules first, and put the other rules according to precedence and then in random order. If a non-mandatory handler precedes a mandatory handler, still the mandatory rule has priority, because the non-mandatory rule does not have to be executed in general. When we put it after the mandatory rule, it will be omitted from the table after the execution of the mandatory rule anyway; but no problems will occur because if we need to repeat that conversion, the mandatory rule is still in the cell and can perform the task once more. The process of finding the right output will not be affected (or prevented) by removing a non-mandatory handler that precedes the mandatory handler but performs the same task.

Every time we get to a table entry (which is ordered according to the mandatory, precedence, and then random rules), we just execute the first rule. We do not need to backtrack because if at some point the parser stops, it means that the composition is not possible.

### 4.2.7 Handlers that do not Change the Type

Remember that not all handlers change the type of a message, for example caching and logging handlers probably do not change the type. We say these handlers make $X \rightarrow X$ type conversion (where $X$ is any type).

Obviously, these handlers will show up in the composition only if they are mandatory. If they do not have any ordering (precedence) constraints, after the composition sequence is calculated, we will just insert them at the beginning or the end of the sequence. If they have ordering constraints, we will insert them in the composition sequence before or after the right handlers.

### 4.2.8 Disjunction

Sometimes a handler might accept more than one input type in order to produce the output. If a handler accepts types $A$ or $B$ and produces type $C$, we show the conversion in this format:

$$A|B \rightarrow C.$$
To handle this case we just separate these rules and add the two rules $A \rightarrow C$ and $B \rightarrow C$ to the grammar. These rules correspond to the same handler.

The right hand side of the composition question may also be a disjunction of types. That is because the destination server might accept more than one input type. If the question is $P \rightarrow Q|R$, then we should calculate the composition twice: once for $P \rightarrow Q$, and once for $P \rightarrow R$. If one composition is possible, we need not try the next one, as it is enough to make one of them possible.

4.3 Case Studies

We present two case studies which provide a better understanding of the usefulness of the CFG solution for handler composition.

4.3.1 Case Study 1: Purchase Order

At the beginning of this chapter, we introduced an example about an organization having many daily interactions with an online store. In this section, we will see an extended version of this example where sequential types are used.

Suppose the following is a part of the request message that should be sent to the service to place a purchase order. It contains information about the customer account, the item to be purchased, delivery and payment:

```xml
<PurchaseOrderRequest>
    <AccountInfo>
        <Username>
        <Password>
    </AccountInfo>

    <TransactionItem>
        <ItemID>
        <Quantity>
    </TransactionItem>

    <Shipment>
        <ShippingAddress>
        <DeliveryMethod>
        <Condition>
    </Shipment>

    <Payment>
        <CreditCardNumber>
        <Expiration>
    </Payment>
</PurchaseOrderRequest>
```

The whole organization has a unique account and payment method, and the shipping address is the same for all clients (the organization's address). The
clients can send a message containing the information about the items they need to an intermediate server. The handlers on the intermediate server will add the information about the customer account, payment and shipment to the message and forward it to the destination server.

The message includes credit card information, so it should be encrypted. Also, the server expects the message to be digitally signed, to make sure it is really sent by the client with the indicated account. Because of bandwidth limitations, the server also expects the message to be compressed.

A subset of the handlers on the intermediate server – along with the type of the conversions they make – is shown in Table 4.1. There might be other handlers on this intermediary which are not shown here; they might be used for other purposes and requests. In the constraints column, \( m() \) means mandatory, \( p() \) means precedes and \( s() \) means succeeds. \( m() \) without any parameters means the handler is mandatory for every service. It could also have the name of the services as parameters, meaning that the handler is mandatory for those services.

<table>
<thead>
<tr>
<th>Handler Name</th>
<th>Conversion Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logging</td>
<td>( X \rightarrow X )</td>
<td>( m() + p(\text{Encryption}) )</td>
</tr>
<tr>
<td>Caching</td>
<td>( X \rightarrow X )</td>
<td></td>
</tr>
<tr>
<td>Compression1</td>
<td>( X \rightarrow X ) [Compressed]</td>
<td></td>
</tr>
<tr>
<td>Compression2</td>
<td>( X \rightarrow X ) Compressed</td>
<td>( p(\text{Compression1}) )</td>
</tr>
<tr>
<td>Encryption</td>
<td>( X \rightarrow X ) [Encrypted]</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>( X \rightarrow X ) Signed</td>
<td></td>
</tr>
<tr>
<td>Decompression</td>
<td>( X ) [Compressed] ( X' \rightarrow X X' )</td>
<td></td>
</tr>
<tr>
<td>Decryption</td>
<td>( X ) [Encrypted] ( X' \rightarrow X X' )</td>
<td></td>
</tr>
<tr>
<td>Address Provider</td>
<td>Item \rightarrow AddressAdded</td>
<td></td>
</tr>
<tr>
<td>Account Info</td>
<td>AddressAdded \rightarrow AccAdded</td>
<td></td>
</tr>
<tr>
<td>Financial Provider</td>
<td>AccAdded \rightarrow PurchaseOrderRequest</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Possible list of handlers on the intermediary in case study 1

If the client sends a message of type Item to the intermediate server, the composition question would be as follows:

\[ \text{Item} \rightarrow \text{PurchaseOrderRequest}, [\text{Encrypted}], \text{Signed}, [\text{Compressed}] \]

According to the algorithm introduced for sequential types, the result is the execution of the following sequence of handlers:

\[ \text{Address Provider} \rightarrow \text{Account Info} \rightarrow \text{Financial Provider} \rightarrow \text{Logging} \rightarrow \text{Encryption} \rightarrow \text{Signature} \rightarrow \text{Compression2} \]

Compression1 and Compression2 represent two different compression algo-
rithms. Notice that Compression2 has priority over Compression1 (conceptually because it is a preferred method for compression if available), so it will be selected. Logging does not change the type, but it is a mandatory handler for this service. The algorithm will insert this handler in the composition sequence as late as possible, and that is before the Encryption handler, since Logging precedes Encryption.

The client might add the shipping address by itself (for some reason it may not want the item to be shipped to the company). In this case, the type of the message will be AddressAdded instead of Item, so the composition question will change to:

AddressAdded → PurchaseOrderRequest, [Encrypted], Signed, [Compressed]

And the composition sequence will be:

Account Info → Financial Provider → Logging → Encryption → Signature → Compression2

The above case study was designed to show the application of assigning sequential types to messages, as well as the application of the composition algorithm for messages with this type. It shows how mandatory handlers could be assigned to a service, and also how handlers might have precedence rules over each other.

### 4.3.2 Case Study 2: Moblogging

Weblogging is a technique that makes it easy for people to publish their thoughts on the World Wide Web, and is very popular nowadays. Moblogging – or mobile blogging – is the content posted to the Internet from a mobile or portable device, such as a cellular phone or PDA. You can send SMS text, photos or video taken by your cellular phone to your weblog instantly.

Very probably, it is not desirable to have the MIDlet – the application on the mobile device – talk to the blog server directly. The reason is, first of all the mobile device has its own constraints in terms of memory and battery power. The reason is, even if you have a parser generator or even just the prepared messages that their required fields should be filled up on the mobile device, there are many different moblogs and each has a different APIs. The APIs are big enough that might not be a good idea to support more than two or three of them on the mobile device. The second reason is, the application for sending photos to a Web site might be useful for other purposes, such as sending photos to storages and albums on the net. So in general it is a good idea to have an intermediate server between the clients and the moblog service providers that takes care of API conversion.

So basically, the client just sends a simple SOAP message including the photo or video as an attachment, and the name of the weblog or the photo sharing/storage service. It will also contain client's username and password for the Web site, so it should be encrypted by the client. The intermediate server
has the API’s for these services and will just convert this message to the right SOAP message format, as shown in figure 4.4.

![Simple SOAP + Photo](image)

**Figure 4.4: Application of intermediate server in Moblogging**

<table>
<thead>
<tr>
<th>Handler Name</th>
<th>Conversion Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>X → X</td>
<td>m()</td>
</tr>
<tr>
<td>Caching</td>
<td>X → X</td>
<td>m(PhotoZou)</td>
</tr>
<tr>
<td>Size Fixer</td>
<td>X → X</td>
<td>s(Decompression, Decryption)</td>
</tr>
<tr>
<td>Charging Registrar</td>
<td>X → X</td>
<td>m + p(Encryption)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ s(Decompression)</td>
</tr>
<tr>
<td>Compression1</td>
<td>X → X [Compressed]</td>
<td>m()</td>
</tr>
<tr>
<td>Compression2</td>
<td>X → X [Compressed]</td>
<td></td>
</tr>
<tr>
<td>Encryption</td>
<td>X → X [Encrypted]</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>X → X Signed</td>
<td></td>
</tr>
<tr>
<td>Date Adder</td>
<td>X → X [Dated]</td>
<td>m(PhotoZou, Pixagogo)</td>
</tr>
<tr>
<td>Decompression</td>
<td>X [Compressed] → X X'</td>
<td></td>
</tr>
<tr>
<td>Decryption</td>
<td>X [Encrypted] → X X'</td>
<td></td>
</tr>
<tr>
<td>TextAmerica API</td>
<td>ClientMessage → TextAmerica</td>
<td></td>
</tr>
<tr>
<td>MetaWeblog API</td>
<td>ClientMessage → Simsi</td>
<td></td>
</tr>
<tr>
<td>Atom API</td>
<td>ClientMessage → TextAmerica</td>
<td></td>
</tr>
<tr>
<td>Blogger API</td>
<td>ClientMessage → NokiaMB</td>
<td></td>
</tr>
<tr>
<td>MovableType API</td>
<td>ClientMessage → Phlogger</td>
<td></td>
</tr>
<tr>
<td>Pixagogo API</td>
<td>ClientMessage → Phlog</td>
<td></td>
</tr>
<tr>
<td>Bzznet API</td>
<td>ClientMessage → Buzznet</td>
<td></td>
</tr>
<tr>
<td>Amazon S3 API</td>
<td>ClientMessage → ebayStorage</td>
<td></td>
</tr>
<tr>
<td>Flickr API</td>
<td>ClientMessage → Flicker</td>
<td></td>
</tr>
<tr>
<td>PhotoZou API</td>
<td>ClientMessage → MyPhotoAlbum</td>
<td></td>
</tr>
<tr>
<td>MyBlogSite API</td>
<td>ClientMessage → SutterFly</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Possible list of handlers on the intermediary in case study 2
Because of bandwidth limitations, the mobile device compresses the message at the beginning. Some services might expect a specific message size, so the intermediate server might have handlers for taking care of that. The message will probably need to be encrypted so that it is not stolen on the way, and is signed so that the service will know it is not spam. There may also be handlers for adding date and time to the message. On this intermediary, Compression1 is mandatory for all messages, meaning it should be compressed and should use Compression1 as its compression algorithm. Charging Registrar precedes Encryption and succeeds Decompression handlers; this is because if the input message is compressed, the registrar can extract the information from the message after decompression. If the message is to get encrypted, the registrar should register the charge before the message is encrypted, while the handler can still extract the data. A possible set of handlers on the intermediate server is shown in Table 4.2.

A possible composition question is:

ClientMessage Signed,[Encrypted],[Compressed] → PhotoZou Signed,Dated,[Encrypted],[Compressed]

For this question, the composition sequence with the above set of handlers would be:

Decompression → Decryption → PhotoZou API → Date Adder → Charging Registrar → Encryption → Compression1 → Monitor

Notice that the message will be decompressed and decrypted before any other changes can be made to it. Compression1 is mandatory so it will be selected between the two compression methods. Charging Registrar handler is responsible for entering the data for billing the customers. It succeeds Decryption and precedes Encryption, so it is inserted before Encryption. There are no constraints for the mandatory handler Monitor, so it is inserted at the end of the sequence.

In addition to the features shown by the first case study, this case study shows how our algorithm supports the special types that should be removed (and probably be added again) such as encryption in this case study. It also demonstrates the application of the handler composition algorithm for a different domain which shows the approach is useful in a variety of domains.
Chapter 4. Handler Composition Algorithms for Atomic and Sequential Types

4.4 Summary of the Chapter

Figure 4.5 is the summary of the problems and the solutions we provided in this chapter.

<table>
<thead>
<tr>
<th>Type Structure</th>
<th>Assumption</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic</td>
<td>Atomic rules</td>
<td>Simple Graph Solution</td>
</tr>
<tr>
<td>Sequential, non-content based</td>
<td>Additive parametric rules + atomic rules</td>
<td>Context-free grammar solution (CFG)</td>
</tr>
<tr>
<td></td>
<td>Subtractive parametric rules</td>
<td>Add null productions to CFG</td>
</tr>
<tr>
<td></td>
<td>Prefix type elements</td>
<td>Defining special types + Pre-processing the CFG parser input</td>
</tr>
<tr>
<td></td>
<td>Mandatory handlers exist</td>
<td>Adjust the CFG solution</td>
</tr>
<tr>
<td></td>
<td>Handlers have precedence rules</td>
<td>Adjust the CFG solution</td>
</tr>
<tr>
<td></td>
<td>Intermediaries or the final server accept more than one type</td>
<td>Adjust the way handlers are mapped to grammar rules</td>
</tr>
<tr>
<td></td>
<td>There are handlers that do not change the type</td>
<td>Post-processing the CFG parser output</td>
</tr>
</tbody>
</table>

Figure 4.5: Summary of Chapter 4
Chapter 5

Handler Composition Algorithm for Tree Structured Types

The context-free grammar handles the composition problem with sequential types well, especially for non-content type elements. But this solution is not sufficient for covering the content based type assigned to a message, as it depends on the structure of the message body, and the message body is typically tree-structured. For example, consider figure 5.1 that shows an example of a SOAP message taken from [16]. As you can see in this figure, data in the SOAP message – or any XML based message – are ordered, labeled tree structures. To describe the body structure of this message, we define the type of the message as

\[ \text{ReservationCharge}(\text{Code CreditCard}(\text{Name Number Expiration})) \]

where the parenthesis show the children of a node.

Now consider another example: Suppose the message contains the information about an address book item, and there is a handler that changes this item to a telephone book item, as follows:

\[
<\text{addrbookitem}>
\begin{align*}
&<\text{name}> \text{Sara Forghani} </\text{name}> \\
&<\text{email}> \text{forghani@cs.ubc.ca} </\text{email}> \\
&<\text{tel}> 604 999 9999 </\text{tel}>
\end{align*}
\]<\text{addrbookitem}>
\[<\text{tellbookitem}>
\begin{align*}
&<\text{firstname}> \text{Sara} </\text{firstname}> \\
&<\text{lastname}> \text{Forghani} </\text{lastname}> \\
&<\text{number}> 604 999 9999 </\text{number}>
\end{align*}
\]<\text{addrbookitem}>

In other words, the handler changes from \textit{AddressBookItem(Name Email Tel)} to \textit{TellBookItem(FirstName LastName Number)}.

It is not easy – and sometimes impossible – to have a corresponding rule for this handler in the context-free grammar (either the grammar will not be context-free anymore, or the set of rules will make this change possible but will
also accept other conversions that could never happen). As we will see in this chapter, this problem can be solved using the tree automata, which is a specific type of term rewriting systems. This solution covers content based message types.

5.1 The Tree Automata Solution

5.1.1 Tree Automata Informal Definition

Tree automata are a specific type of term rewriting systems [17]. Rewriting, in its most basic form, consists of a set of terms, plus relations on how to transform these terms. A "term rewriting system" is a formal model that is quite often used to describe the semantics of programming languages. Each term consists of a composition of function symbols and constants. A rewrite rule $\alpha \rightarrow \beta$ implies
that it is proper to replace any occurrence of the left hand side in a term with
the right hand side.

Finite automata over finite words are a formalism to represent certain kinds
of infinite sets of words, called regular sets or regular languages. Tree automata
[2, 18] are similar to finite automata except that instead of sets of words, they
deal with trees. Tree automata are devices with finite memory that read input
trees and accept or reject them.

The tree automata have two important properties, they are non-deterministic
and they are bottom-up. This helps us to mitigate two difficulties. First, be­
because they are non-deterministic we demonstrate that it is not necessary to
compose handlers manually. Second, because they are bottom-up they support
reasoning over message pieces (sub-trees).

5.1.2 Tree Automata Formal Definition

The brief formal introduction to tree automata in this subsection is provided for
the interested reader but we do not think it is strictly necessary to understand
the contribution of this thesis. The figure and examples have been taken directly
from [17] and [2].

Formal Definition: A nondeterministic finite tree automaton (NFTA) is
a tuple $A = (Q, A, \Delta, F)$, where $Q$ is a set of finite states, $A = (A_i)_{i \in [k]}$ is a
ranked alphabet, $F \subseteq Q$ is a set of final states, and $\Delta \subseteq (\bigcup_{i=0}^{k} Q^i \times A^i) \times Q$. A
set of trees that can be accepted by an NFTA is called regular.

A "move relation" shown by $\rightarrow_A$ is applying a transition rule to the tree
to rewrite a node. $\rightarrow_A^*$ is the reflexive and transitive closure of $\rightarrow_A$. Notice that
there is no initial state in a NFTA, but when the symbol is a constant symbol
$a$, a transition rule is of the form $a \rightarrow q(a)$. Therefore, the transition rules for
the constant symbols, i.e. leaves, can be considered as the "initial rules". If the
direct subterms (its children in the tree) $u_1, \ldots, u_n$ of $t = f(u_1, \ldots, u_n)$ are
labeled with states $q_1, \ldots, q_n$, then the term $t$ will be labeled by some state
$q$ with $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(q(f(x_1, \ldots, x_n)) \in \Delta$ (informally, it means
that when all the children of a node are "states", the node can be reduced if
there is a rule that reduces it – Example 2 makes this point clearer).

Example 1. The following bottom up tree automaton recognizes the regular
tree language $f(g^+(a), f(a, a))$ [17] and [2]:

\[
\begin{align*}
    a & \rightarrow q_a \\
    g(q_a) & \rightarrow q_g \\
    f(q_a, q_a) & \rightarrow q_f \\
    f(q_g, q_f) & \rightarrow q_{accept}
\end{align*}
\]

The symbols denoted by $q$'s are the states. They never appear in input
expressions but they are used internally by the automata to provide the correct
semantics. As you can see, $g(q_g) \rightarrow q_g$ produces the $g^+(a)$. "$a"$ is the only
constant and for that we have $a \rightarrow q_a$. You can see that whenever all the
children of a symbol are states, it will be assigned a state (like $f(q_a, q_a) \rightarrow q_f$).
Example 2. Suppose we have the following transition rules in a tree automaton [2]:

\[
\begin{align*}
    a & \rightarrow q(a) \\
    g(q(x)) & \rightarrow q(g(x)) \\
    f(q(x), q(y)) & \rightarrow q(f(x, y))
\end{align*}
\]

Figure 5.2 shows how the tree \(f(a, a)\) is reduced but not accepted by this automaton; while \(f(g(a), g(a))\) is accepted as it is reduced to \(q_f\). A ground term \(t\) is accepted by a finite tree automaton \(A = (Q, A, A, F)\) if \(t \rightarrow_f^* q_f\). As you can see in the figure, \(f(a, a)\) is accepted because \(q_f\) is derived in the end.

![Figure 5.2: The reduction and acceptance of \(f(a, a)\) in Example 2 (taken from [2])](image)

In "ranked trees" (vs. unranked trees) each symbol in the alphabet has a number associated with it called the "arity" of the symbol. In functions, the arity is just like the number of arguments of each function, e.g. binary. In a tree, each symbol is associated with a node, so you can think of the arity as the number of children a node with a particular symbol has.

Giving a more in-depth description of the theory of tree automata is beyond the scope of this thesis; but you can learn more about it in [2] or [18]. We will just explain the algorithm we use and how we use it.

### 5.1.3 Regular Tree Expression

Just like every finite-state automata corresponds to a regular expression, every tree automaton corresponds to a "regular tree expression". For example, the regular tree expression for Example 1 is as follows:

\[ f(+[g](a) \ f(a \ a)) \]
In our implementation, a number of operators can be used in writing the tree expression. CHOICE (or \texttt{?} | \texttt{||}) accepts either of the two sequences given as operands. STAR (or \texttt{*} [ ] ) means there can be zero or more sequences inside the brackets, while PLUS (or \texttt{+} [ ] ) means one or more sequences. And, the operator OPTION (–[ ] ) means the sequence inside the brackets is optional.

### 5.1.4 Regular Ground Tree Rewriting Systems

A regular ground tree rewriting system (RGTRS) is a set of rules of the form $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are regular tree expressions representing tree automata, rather than tree instances. The semantics of such rules are interpreted in a similar way as term rewriting systems. Here we would say that any tree accepted by $\alpha$ can properly be replaced by any tree accepted by $\beta$.

### 5.1.5 The Membership Problem

The tree automata have been used in literature for XML type checking, for example in [19]. Similarly we use the membership algorithm in tree automata introduced in [18] for type checking. The input of this algorithm is a regular tree expression which describes a type, and a tree instance which we want to determine if it belongs to the language described by the expression, or in other words, it is of the right type. For example, the regular tree expression for Example 1 and a tree that is accepted by this expression given for our implementation are as follows:

\[
\text{EXPR}
\]

\[
\text{type } T = f(+[g](a) \ f(a \ a))
\]

\[
\text{TREE}(T)
\]

\[
f(g(g(a))) \ f(a \ a)
\]

### 5.1.6 Regular Ground Tree Reachability and Handler Composition

To solve the compositionality problem, we use the reachability algorithm introduced in [18]. To the best of our knowledge this algorithm has never been used for solving the handler (or any component) compositionality problem. The right hand side of the composition question (Destination in Source $\rightarrow$ Destination) is a regular tree expression that corresponds to the server schema; but the left hand side (Source) is an instance of a tree, as you will see later in the examples. For tree structured types, the handlers correspond to regular tree rewriting rules, in other words, they correspond to rules which transform one regular tree expressions to another one.

You can find the details of the reachability algorithm in [18], but informally, here is the definition of the reachability problem for a RGTRS:
Chapter 5. Handler Composition Algorithm for Tree Structured Types

Given a RGTRS (i.e. set of rewrite rules) and a target automaton (i.e. set of tree instances), what is the set of tree instances which can be rewritten into a tree accepted by the target using any composition (transitively) of the rewrite rules?

Our approach for solving the composition problem of Web service handlers is largely based on an implementation of this reachability problem. The following examples will help more in understanding of the input and output of the algorithm implementation.

Example 3. Suppose reachability – or composition – question is whether we can reach a tree like \( f(g(c) g(c)) \) from the tree \( f(g(a) g(a)) \) if we have handlers which make conversions of type \( A \rightarrow B \) and \( B \rightarrow C \). The algorithm will tell us which handlers should be used – and in what order – to make this reachability possible. The algorithm’s input is shown in figure 5.3.a. EXPR defines the tree regular language. RULES show the transition rules and each rule corresponds to a handler which can make that change. REACH is the tree expression which we want to reach from TREE. We have implemented the algorithm so that the output of the algorithm for this example is as follows:

\[
f( g(1\leftarrow 0\leftarrow a()) g(1\leftarrow 0\leftarrow a()) )
\]

which says that we start with the leaves (a’s) and apply rules 0 (a → b) and 1 (b → c) on them in order, and we will reach the expected tree.

Example 4. For the input shown in figure 5.3.b, the output will be as follows:

\[
f(g(2\leftarrow 0\leftarrow) g(2\leftarrow 0\leftarrow) ) \leftarrow 1\leftarrow 0\leftarrow a()
\]

which says that we start with a, then use rule 0, rule 1, then we get a tree where we apply rules 0 then 2, to the children of both g’s (which are children of f).

Figure 5.4 shows the inputs and outputs of this algorithm in our setup. For every request message, we build a tree automaton that accepts the required XML or SOAP message according to the server schema and transition rules which correspond to handler conversions. When a request message arrives, if it corresponds to the server schema, or if its elements can be converted by transition rules in the way that the required message can be generated, it is accepted by the tree automata corresponding to that request message.

5.1.7 The Augmented Schema

A tree automaton corresponds to an XML schema and vise versa. The new tree automaton that we have built using the server schema and the transition rules from the handlers, corresponds to a new XML schema which we call augmented schema. This is like an extended view to the server’s schema. For example, suppose all of client A’s messages are routed through the intermediate server.
Chapter 5. Handler Composition Algorithm for Tree Structured Types

Figure 5.3: (a) and (b) are the inputs to the reachability algorithm in examples 1 and 2 respectively.

Figure 5.4: Reachability algorithm inputs and outputs in our setup.

B. Now, client A finds out the URL of server C. Client A asks server C for the server’s schema. The intermediate server B intercepts and before returning the schema to the client, it runs the algorithm to get a new schema which it returns to the client.

So the client A has a different “view” of the server because it is looking at it
through the “lense” of the intermediate server. Notice that we will build a tree automata for each request or response message, so the unit of computation is a request or response message.

The case study in the next section of this chapter gives a better understanding of the GTRS solution application.

5.2 Case Study: eBay Schema Migration and Version Incompatibility

The following case study shows the application of the GTRS solution provided in this chapter.

Sometimes a service provider changes the service API, and the new API might be incompatible with the previous one; so the client applications that are using the service have to adopt to the new API. Migrating from the legacy API to the new one could be costly for the clients, because in general making changes to the applications is costly. eBay is one of the service providers which has changed its XML API to the new unified API. As of June 1st 2006, support for the legacy XML API was removed. If any application uses the legacy XML API, an error will be returned mentioning that the legacy XML API is not supported anymore. The original eBay XML API (legacy XML API) was rolled out in 2001 for developers making pure XML calls. In early 2004, eBay released the SOAP API which is now using the unified schema. The unified schema, part of eBay Web Services, supports all five interfaces to the eBay API including XML and SOAP [20].

Even in the unified API, there are currently thirty seven different versions and sometimes there are some incompatibilities between these different versions.

Intermediate servers can be helpful in handling the incompatibility and migration problems. Older systems will not have to migrate to the new API if an intermediate server changes the message from legacy to unified format. Also in the unified API, the intermediate server can take care of the incompatibilities between the different versions so that the client applications do not have to change when a new version is created.

Handlers can also help in solving the problem of incompatibility between the schemas for different Web services. There are types that exist in different schemas which are conceptually the same but have different structures in different schemas. As you will see in this section, handlers help converting between these pieces of messages.

In this section, first you will see examples of handlers which take care of the incompatibility problem between different versions of the unified API, and then we will see examples of handlers which change from the legacy API to the unified API. Then you will find an example that shows how handlers can help in solving the problem of incompatibility between different Web service schemas, and then you will also see how they can help in everyday conversions of data like measurement units in order to localize the data. You will find out how the
GTRS solution helps in solving these problems.

### 5.2.1 Version Incompatibility

Currently, there are thirty seven different versions for eBay unified schema (versions 399 to 473). Sometimes these versions are incompatible; for example, from version 439, two fields named SiteHostedPicture and VendorHostedPicture which are the children of the complex type ItemType are deprecated and replaced by a new child element named PictureDetails. As this type is being used in fourteen different calls, this change affects all these calls either in the request or the response message. Figure 5.5 shows a list of the affected calls. A handler on the intermediate server can take care of this incompatibility by combining SiteHostedPicture and VendorHostedPicture into PictureDetails. We show this conversion by:

\[
\text{ItemType(SiteHostedPicture VendorHostedPicture)} \rightarrow \text{ItemType(PictureDetails)}
\]

<table>
<thead>
<tr>
<th>Request Messages:</th>
<th>Response Messages:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddItem</td>
<td>GetBidderList</td>
</tr>
<tr>
<td>AddLiveAuctionItem</td>
<td>GetCategoryListings</td>
</tr>
<tr>
<td>GetItemRecommendations</td>
<td>GetItem</td>
</tr>
<tr>
<td>RelistItem</td>
<td>GetMyeBayBuying</td>
</tr>
<tr>
<td>ReviseItem</td>
<td>GetMyeBaySelling</td>
</tr>
<tr>
<td>ReviseLiveAuctionItem</td>
<td>GetSearchResults</td>
</tr>
<tr>
<td>VerifyAddItem</td>
<td>GetSellerList</td>
</tr>
</tbody>
</table>

Figure 5.5: List of the calls affected by combination of SiteHostedPicture and VendorHostedPicture into PictureDetails

As another example, there are incompatibilities between versions 415, 453, 459, 463 and their older versions. These incompatibilities are mostly caused by changes in element names. For example, one of the child elements of the complex type TransactionPlatformType is named Express instead of ebayExpress from version 459. This type is an element in GetMyeBayBuying and GetMyeBaySelling response messages.

An example of the possible conversions on a response message are shown here. The GetMyeBayBuying call returns items from the All Buying section of the user’s My eBay account, including items the user is watching, bidding on, has won, has not won, or has made best offers on.

If the client sends a GetMyeBayBuying request message to the server, the automata on the intermediate server may return the following handlers to convert the response message (notice that we are converting from new to old – for
example, the server version is 469 and the client version is 437):

1. TransactionPlatformType(express) → TransactionPlatformType(ebayExpress)
2. ItemType(..PictureDetails..) → ItemType(..SiteHostedPicture VendorHostedPicture..)
3. USD → CND

We have used the ellipsis here just to omit the details about other children of a root element. For example, the corresponding transition rule in the automata for the second rule is as follows:

\[
\text{item(-[biddingDetails (BiddingDetails)] -[eBayNotes(string)] itemID(string)}
\]
\[
\text{-[listingDetails (ListingDetails)] pictureDetails(PictureDetails))}
\]
\[
\rightarrow
\text{item(-[biddingDetails(BiddingDetails)] -[eBayNotes(string)] itemlD(string) -}
\]
\[
\text{[listingDetails(ListingDetails)] siteHostedPicture(SiteHostedPicture) vendorHostedPicture(VendorHostedPicture))}
\]

The third rule is for currency conversion and is described in Section 5.2.4.

### 5.2.2 Schema Migration

In the eBay developers Web site [20], you can find the mapping tables for migrating from legacy to unified XML API. Each row of the mapping table shows the new (unified) and old (legacy) versions of a message element. Mapping tables are available for fifty different calls, each for the request and response messages. As an example, the mapping table for GetAccount request message includes the first two columns of the table shown in figure 5.6. The third column represents the handlers which make these conversions possible. There could be a large number of handlers in the system where each of them takes care of a specific change; these handlers should be combined to transform each message, and many of them are shared between different messages. For example, changing from PageNumber to Pagination.PageNumber (which means PageNumber will be a part of the Pagination element instead of the message itself) should be done in several messages, and all these messages will use this handler to make this change.

The schema for GetAccount request message is shown in Appendix B. The input to the reachability algorithm for converting this request message is shown in Appendix C. Using EXPR, RULES and REACH – which are extracted from the server schema and the handlers – we build an automata for this request message. Each time a new message arrives we will test and convert it as the TREE input.

Notice that the two rules that combine two children of the request message and make pagination and invoiceDate should be merged while we are writing the rules that are the input to our algorithm (in other words, one single handler should be responsible for making the two changes); the reason is, they should be converted at the same time, otherwise we would not know which one will
Chapter 5. Handler Composition Algorithm for Tree Structured Types

<table>
<thead>
<tr>
<th>Legacy</th>
<th>New</th>
<th>Handler Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountPageType</td>
<td>AccountHistorySelection</td>
<td>AccountPageType $\rightarrow$ AccountHistorySelection</td>
</tr>
<tr>
<td>BeginDate</td>
<td>BeginDate</td>
<td></td>
</tr>
<tr>
<td>Currency</td>
<td>Currency</td>
<td></td>
</tr>
<tr>
<td>EndDate</td>
<td>EndDate</td>
<td></td>
</tr>
<tr>
<td>EntriesPerPage</td>
<td>Pagination.EntriesPerPage</td>
<td>ReqMsg</td>
</tr>
<tr>
<td>PageNumber</td>
<td>Pagination.PageNumber</td>
<td>ReqMsg</td>
</tr>
<tr>
<td>ExcludeBalance</td>
<td>ExcludeBalance</td>
<td></td>
</tr>
<tr>
<td>InvoiceMonth</td>
<td>InvoiceMonth</td>
<td>ReqMsg</td>
</tr>
<tr>
<td>InvoiceYear</td>
<td>InvoiceYear</td>
<td>ReqMsg</td>
</tr>
<tr>
<td>ErrorLanguage</td>
<td>ErrorLanguage</td>
<td></td>
</tr>
<tr>
<td>Period</td>
<td>AccountHistorySelection</td>
<td>Period $\rightarrow$ AccountHistorySelectionCodeType</td>
</tr>
<tr>
<td>AccountPageType</td>
<td>AccountHistorySelectionCodeType</td>
<td></td>
</tr>
<tr>
<td>AccountEntrySortType</td>
<td>AccountEntrySortType</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td>ExcludeSummary</td>
<td>Summary $\rightarrow$ ExcludeSummary</td>
</tr>
</tbody>
</table>

Figure 5.6: Mapping table for GetAccount request message migration

be converted first and in that case we would need more rules to take care of different combinations of old/new children.

For the following input tree:

TREE

getAccountRequest(
    sort(accountEntryCreatedTimeAscending)
    accountPageType(accountEntryFeeTypeDescending)
    beginDate(2006-5-10T12:00:00-05:00 )
    currency(10d)
    endDate(2006-10-10T12:00:00-05:00 )
    excludeBalance(false) summary(true)
    invoiceMonth(8) invoiceYear(2006)
    entriesPerPage(5) pageNumber(4))

The output of our algorithm is rules (handlers) 4, 3, 0 and 1 in the order (from the rules list in appendix C).

In this part of the case study we provided a complete example and the complete input and output of our algorithm. We saw that some transformations should be merged in one single handler in order to prevent inconsistency. We also found out how the GTRS solution could be helpful for composing handlers to solve the problem of schema migration.
5.2.3 Compatibility with other Web Services

Our algorithm not only determines which handlers should be executed to make the API conversion possible, but also returns the right order of execution that leads to the type-correct output. To understand how the order of execution could be important, consider the following example:

Suppose a seller who has recently sold an item in eBay wants to send the item to the buyer through FedEx. She sends a request such as GetSellerTransactions to eBay. The response contains the transaction information, including the shipping address and some other information in respect to the buyer.

Now the seller needs to send a shipping request to FedEx – the FDX-ShipRequest. A part of the information in this message is specific to FedEx, such as the FedEx account number, and should be specified by the seller. On the other hand, some parts of the information in this request message can be extracted from the response message received from eBay, such as the destination address; however, not all Web services use the same structure for the Address type in their schema. You can see the different address type definitions for eBay, FedEx and UPS in figure 5.7.

```
<eBayShippingAddress>
  <ShippingAddress>
    <Addressed> string </Addressed>
    <CityName> string </CityName>
    <Country> string </Country>
    <CountryName> string </CountryName>
    <ExternalAddressID> string </ExternalAddressID>
    <Name> string </Name>
    <Phone> string </Phone>
    <PostalCode> string </PostalCode>
    <Street1> string </Street1>
    <Street2> string </Street2>
  </ShippingAddress>

<UPS Address>
  <City> string </City>
  <StateProvinceCode> string </StateProvinceCode>
  <PostalCode> string </PostalCode>
</UPS Address>

<FedExDestination>
  <Destination>
    <Contact>
      <PersonName> string </PersonName>
      <PhoneNumber> string </PhoneNumber>
      <FaxNumber> string </FaxNumber>
      <E-MailAddress> string </E-MailAddress>
    </Contact>
    <Address>
      <Line1> string </Line1>
      <Line2> string </Line2>
      <City> string </City>
      <StateOrProvinceCode> string </StateOrProvinceCode>
      <PostalCode> string </PostalCode>
      <CountryCode> string </CountryCode>
    </Address>
  </Destination>
```

Figure 5.7: eBay ShippingAddress, UPS Address, and FedEx Destination type definitions

So the seller has to make a conversion between the address type in eBay schema and the address type in FedEx schema. Suppose she also wants to validate the address, but the FedEx Web service does not provide address validation; however, UPS provides this service. The seller can validate the address by sending a AddressValidationRequest message to UPS Web service, so she has to make a conversion from the address type in eBay schema to the address type in UPS schema; however, if we have handlers on the intermediaries which make these conversions between pieces of messages and different type structures.
in Web service schemas possible, the clients will not have to write the code for
making the conversions between the pieces of messages that are conceptually
the same but have different structures.

Now, suppose the client uses the eBay legacy schema; the address structure in
eBay legacy schema is different from the unified schema. We also have a handler
that converts between eBay ShippingAddress type to FedEx Destination type.
We also have another handler that converts between FedEx Destination type
to UPS Address type. Notice that if the client wants to put the address from
eBay response in its request message and send it to UPS validation service, the
following three groups of rules should be executed in order. All the rules in
group number 1 transform between eBay legacy and unified schemas:

1: (Legacy to Unified eBay:)
City -> CityName
Country -> CountryName
CountryCode -> Country
Zip -> PostalCode

(The following three rules are equivalent to the following transition
in migration mapping table:

g getSellerTransactionsResult(GetSellerTransactionResultType)
-> GetSellerTransactionResultType
Transactions -> TransactionsArray
User -> BuyerInfo

2: (eBay to FedEx)
shippingAddress(ShippingAddressType) -> destination(DestinationType)

3: (FedEx to UPS)
destination(DestinationType) -> address(AddressType)

In this part of the case study, we found out how ordering could be important
in execution of transformation handlers. Our algorithm takes care of finding
the right execution order for handlers automatically so that the output will be type-
correct.

5.2.4 Localizing Data

Another application of handlers on intermediaries for services like eBay is mea-
measurement unit conversion so that they make the data in request/response mes-
messages localized. Temperature, currency and world time conversions are of this
kind. For example suppose the client is in Canada and the server is in the United States. In GetMyeBayBuying response message, there are fields which have the price in U.S. dollar, such as

```xml
<TotalWinningCost currencyID="USD"> 316.0 </TotalWinningCost>
```

in `<BuyingSummary>`, or

```xml
<StartPrice currencyID="USD"> 10.0 </StartPrice>
```

in `<Item>`. The client wants to see the prices in Canadian dollar; so there is a handler which makes USD → CAN conversion, and also changes the value. This handler might update its currency rating information everyday by sending a remote call to a Web service which provides this information.
Chapter 6

System Implementation

6.1 Tools and Technologies

Apache Axis

Apache Axis is essentially a SOAP engine – a framework for constructing SOAP processors such as clients, servers, gateways, etc. The current version of Axis is written in Java, but a C++ implementation of the client side of Axis is being developed. Axis is the third generation of Apache SOAP (which began at IBM as "SOAP4J") [3]. It also makes it possible to have chains of request and response handlers on the client and server side. Apache Axis is an open source product.

ANTLR Parser Generator

ANTLR (ANother Tool for Language Recognition) [21] is a language tool that provides a framework for constructing recognizers, compilers, and translators.

6.2 Programming

Our system is written in Java and is built on top of Apache Axis 1.3. We have modified this tool so that it supports intermediaries. We have built intermediate servers that are very light versions of axis server. Basically we use the transport layer (message http listener and sender) from the server, and we have chains of request and response handlers on intermediate server. We have also made dynamic registration of handlers on the intermediate server possible.

Currently, we have implemented the GTRS solution for content based message types. We have started implementing the CFG solution as well, but because of time limitations the implementation is not complete yet. Some details about the implementation of the GTRS algorithm are presented in the next section. We use ANTLR generated lexer and parser to define and parse the input to the algorithm. The codebase for the GTRS solution includes 45 java files, with 2963 lines of code\textsuperscript{15}.

\textsuperscript{15}Without considering the parser and the lexer generated by ANTLR.
6.3 GTRS Solution Technical Details

To solve the compositionality problem, we use the reachability algorithm introduced in [18]. To the best of our knowledge, this algorithm has never been used for solving any kind of software component composition problem or in the context of XML document research. We faced two challenges in our research which required changes to the standard algorithm given in the literature. Before describing how we deal with these problems, we provide some informal details about the algorithm’s execution.

Algorithm Overview

Similar to a finite state automata for strings, tree automata consist of sets of states and transitions. Likewise, a union and product automata (intersection) can be created by taking the union and product of states and transitions. First, the reachability algorithm takes the union of all automata that are the left-hand sides of rules and also the target (i.e. the type denoted by REACH). We call these the effective targets.

Next, it takes the union of the right-hand sides of rules. We call these the effective source. Now, it takes the intersection of the effective source and targets. Starting at the leaf product states it runs a reachability algorithm on product states. When a product state is reached, which has as one of its components the final state of a right-hand side, a free transition (i.e. a transition that consumes no input symbols) is created from the final state of the left-hand side of that rule to the other component of the product state.

This captures the intuitive notion that tree instances accepted by the left-hand side of a rule can be used anywhere that one of the tree instances accepted by the right-hand side is required. This is true regardless of whether the final state of a right-hand side automata appears as a non-final state of a left-hand side automata or the initial target, this is why message pieces (sub-trees) are handled properly.

Once the algorithm is finished the effective source and all the newly added transitions are equivalent to our augmented schema. We required two changes to this initial algorithm.

Algorithm Soundness

Unfortunately, the semantics of the algorithm as given are not sound with respect to an intuitive notion of safe handler composition. The algorithm will create a transition from an effective source to an effective target (or sub-tree) when there exists a tree instance in the source that is accepted by the target. We need to require that a transition is created when all tree instances in the source are accepted by the target (or sub-tree). Fortunately, this can be rectified.

\[\text{Algorithm Soundness} \]

Unfortunately, the semantics of the algorithm as given are not sound with respect to an intuitive notion of safe handler composition. The algorithm will create a transition from an effective source to an effective target (or sub-tree) when there exists a tree instance in the source that is accepted by the target. We need to require that a transition is created when all tree instances in the source are accepted by the target (or sub-tree). Fortunately, this can be rectified.

\[^{16}\text{It may seem awkward to call the left-hand sides targets, we use this terminology because the algorithm in effect works it's way backwards}\]

\[^{17}\text{Intuitively this is the same as reachability on finite state automata for model checking}\]
using the sub-type check on tree automata [19]. Before creating a free transition between two states, we simply check that the source is indeed a sub-type of the target (or sub-tree). This makes the algorithm sound and since there are no useful handler compositions that would require this transition it does not affect the completeness. In the future we hope to provide a more formal justification for this claim.

Algorithm Tracing

We have seen how the reachability algorithm is used to construct a sound augmented schema. 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?

Initially we thought of using this approach: at run-time, when we validate a message to the augmented schema, if any augmented transitions are used during validation, we record which ones are needed for what pieces of the message. This would tell us that the handler which generated that transition was required on that piece.

This initial approach is incomplete. During reachability, some states may only become reachable once a transition is added. So, we need to take this cause and effect relationship into consideration. We have instrumented our implementation to create a trace during reachability which records these relationships. When a state is reached because of a transition addition, the trace for that state is initialized to be the trace for the current state appended with the identifier for the handler responsible. Now, as reachability continues, each transition followed from a state inherits the trace of that state. When a state is reached from a transition it likewise inherits the transitions trace. We keep a digest of all the traces for the augmented transitions which is used at run-time to give a complete record of what handlers to use, in what order, and on what part of a message.
Chapter 7

Evaluation

7.1 Verification and Usefulness

We have run over forty test cases on our implementation for the GTRS solution to verify its correctness. These test cases were designed based on the practical examples we have found on schema migration and version incompatibility, as well as the differences between various Web service schemas.

Our case studies show the usefulness of our approach, especially because they try to solve real-world examples. If the programmer is supposed to write manual code for composing the chain of handlers, this task will become complicated as the number of handlers and the number of server schema versions, as well as variety of input message calls, increase. Besides, every new change that is made on the server schemas will make the programmer change the code and decide what the right place for the new handler is; while with automatic composition of handlers, it is enough to add the handler conversion rule to the rule list, and the system will automatically find out where the handler should be executed.

7.2 Timing and Efficiency

A part of our GTRS algorithm that builds the tree automata given the handlers and the server schema could have exponential time complexity according to the complexity of the schema, but that part is being executed off-line: the tree automata for every message is created only every time a new handler registers with the intermediary, or when the server schema is modified; these changes are not made very frequently. This exponential time is caused by the sub-type test for which we use the algorithm described in XDuce [19], which they have shown to be practical on real examples such as the complete HTML schema.

In order to estimate the efficiency of our method and its general overhead on the system, we have designed a few efficiency tests. The results are presented in Figure 7.1

\footnote{These tests were performed using three computers with Pentium 4 CPU's. Each test was repeated five times and the numbers shown in the table for each experiment are the average of the results.}
Chapter 7. Evaluation

either. A comparison between the result of this test and the first test shows the networking overhead caused by the existence of the intermediary. On the third test, we parse the message on the intermediary and also run the composition algorithm to choose a handler chain that makes the message compatible with the server schema; the handlers are not being executed though. The goal is to estimate the overhead of our handler composition algorithm. In the last test, the composed handler chain is also being executed.

<table>
<thead>
<tr>
<th></th>
<th>Total Time for 1000 Req/Res</th>
<th>Average Time for Each Req/Res</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Intermediary</td>
<td>36.735 s</td>
<td>0.036 s</td>
<td>0%</td>
</tr>
<tr>
<td>Intermediary, only Forwarding</td>
<td>39.687 s</td>
<td>0.039 s</td>
<td>8.33%</td>
</tr>
<tr>
<td>Intermediary + Composition Algorithm</td>
<td>42.852 s</td>
<td>0.042 s</td>
<td>16.66%</td>
</tr>
<tr>
<td>Intermediary + Composition Algorithm + Handlers Execution</td>
<td>47.047 s</td>
<td>0.047 s</td>
<td>30.55%</td>
</tr>
</tbody>
</table>

Figure 7.1: Efficiency test results

As you can see in the table, the existence of intermediaries on the message path has some networking overhead; however, nowadays the usage of intermediaries is becoming more and more common; Web services use them at least for routing the messages. As our intermediaries are lighter than the final destination SOAP servers, the total time caused by the existence of intermediaries in the second test is not multiplied by two, because a part of it is caused by the overhead of the SOAP server. Plus the Web service is performing XML parsing and a light task and this has some overhead too.

As you can see in the results, the overhead of having an intermediary and executing our algorithm is quite little compared to the overall system performance. An important point here is that our Web service was performing a light task and was not really doing what a Web service does. In reality the time the service itself takes to execute could be the major overhead in the whole process. Invoking a SOAP call produces costs similar to running an SQL statement in a relational database; maybe even more. Indeed, a database likely gets accessed whenever a SOAP call is made. If our Web service was making a call to a

\[19\] The average number of handler chain for each message was 6
database, then the impact of the intermediary and the composition algorithm would have been even less than the shown results.
Chapter 8

Related Work

8.1 XML Validation and Transformation

Recently there has been a lot of research on XML type checking and transformation. There are also a number of languages that try to treat XML documents as first-class values. The closest research to our approach in terms of theory is the XDuce project [19]. XDuce is a typed programming language that takes XML documents as primitive values. It provides pattern matching for these values and uses regular expression types for describing their structure statically. The XML types correspond to the tree automata and this makes this language powerful, as it provides an intuitive notion of sub-typing. One of our algorithms corresponds each XML schema to a tree automata as well – but the goal in the XDuce paper is generally different. To the best of our knowledge, the theory of tree automata has not been used for solving the problem of handler composition in existing research.

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language used for the transformation of XML documents. The XSLT language is declarative; an XSLT stylesheet consists of a template rule collection, each of which specifies what to add to the result tree when the XSLT processor, scanning the source tree (in a top-down manner) finds a node that meets conditions. Instructions within template rules are processed as if they were sequential instructions; but, in fact, they comprise functional expressions, representing their evaluated results – ultimately, nodes to be added to the result tree [1]. Their approach is comparable to our algorithm for content based (tree structured) type conversion (but not to the non-content method). One of the main differences between the XSLT approach and ours (using the tree automata) is that our method returns an augmented schema to the client, while it is not possible to do that using XSLT.

8.2 Web Service Composition

Web service composition is the ability of one business to provide value-added services through composition of basic Web services, possibly offered by different companies [22]. There are currently many proposals and standards for Web service composition, but only a few of them are actually supported by real products. The most famous standard for this purpose is IBM's BPELAWS (Business Process Execution Language for Web Services) [23] which is a language
to specify business processes and business interaction protocols. It lets you define workflows for Web services statically, though you can include conditions and loops in your definition.

There are similarities and differences between Web service composition and intermediate handler chain composition. The similarity is that in both cases we would like to compose pieces of functionality interacting via SOAP and other XML based messages. The difference is, first of all, all of the handlers intercept the same request or response message, and process, modify or build pieces of that message message, whereas in Web service composition, this is not usually the case. Web services have different input formats, so different messages should be exchanged between them. Second of all, the communication between Web services is more complicated than handlers. Most of the times when a message is sent to a Web service, more information or some form of confirmation is needed from the user before the message can travel to the next Web service.

One of the most related works on Web service composition is the Ninja.Paths project [24]. One of the components in their systems is named “automatic path creator” which builds a path for Web services given a source, destination, and a set of operators (services) that have to be included in the path. The input and output types of the services on the path should match. They never mention how they actually build the path though and it seems that the input and output types are atomic. It does not seem that this system has been implemented.

8.3 Dynamic Handler Chain

Apache Axis2 [25] provides dynamic handler chains on the client and the server (their tool does not support handler chains on intermediate servers, though it should not be difficult to extend the tool for that purpose). “Dynamic” chain means handlers can be added to the chain dynamically when a method that adds those handlers to the chain is called. It can be specified where in the chain the handlers should be added. Although the set of handlers to be executed – and their order – can be specified based on dynamic conditions, the composition is not done “automatically”; in other words, a programmer should write the code for every possible dynamic condition, e.g. every possible message input type where the type could be complex or have many different versions, so that the right handlers are added to the chain at the right time. If any new input or output versions are added in future, the programmer will need to convert the code to take care of the new situation. We call this situation “manual” dynamic handler composition as opposed to our “automatic” composition strategy, where an algorithm decides which handlers should be composed and what is their order of execution.
8.4 Interface Adaptation

There has been some work in the area of type and interface adaptation, such as the research in [26]. In this research Purtilo and Atlee offer a language named Nimble which allows the programmers to adapt the interfaces of existing software without having to operate on the source manually; i.e., it takes care of different order of data types and type adaptation in different interfaces. Their system is suitable for use either in conjunction with existing module interconnection languages, or stand-alone with C, Pascal and Ada source programs.
Chapter 9

Conclusion

9.1 Contributions

In this research we address the problem of inter-operability among different components in a Web service architecture. We provide algorithms for dynamic composition of handlers on intermediaries. We introduce atomic, sequential and tree-structured types for XML based request and response messages. The set of handlers to be executed and their execution order is determined mostly based on the type of the XML message received by the intermediate server and the destination server input type, as well as other factors such as handler precedence and mandatory handlers bound to each service. To solve the composition question for atomic message types, we use the simple graph theory. For sequential types we use context-free grammar parsing, and for tree-structured types, we use the theory of tree automata. Table 9.1 shows a summary of the three methods.

The context-free grammar solution supports sequential types which basically cover non-content based message types. The Ground Term Rewriting Systems solution supports tree structured types which correspond to content based message types. Type checking (validation) of the message is only provided by the GTRS solution. It is easier to extract the server input type from the schema for content based types rather than non-content based types, because in the former case we only need the XML schema, while the latter requires extraction of data from other schema documents as well.

<table>
<thead>
<tr>
<th></th>
<th>Simple Graph</th>
<th>CFG</th>
<th>GTRS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Structure</strong></td>
<td>Atomic</td>
<td>Sequential</td>
<td>Tree</td>
</tr>
<tr>
<td><strong>Type Category</strong></td>
<td>General</td>
<td>Non-content based</td>
<td>Content based</td>
</tr>
<tr>
<td><strong>Type Checking</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Server Input Type</strong></td>
<td>Not currently supported in practice</td>
<td>Several schema documents</td>
<td>Server XML schema only</td>
</tr>
<tr>
<td><strong>Determination</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: Table of comparison for different composition algorithms
9.2 Future Work

This research may be improved and extended in different ways; a few directions are as follows:

**Automatic Extraction of Server Input Type**

For the tree automata solution, given the server schema, the algorithm takes care of making the automata that accepts the right server input type. However, for the context-free grammar solution with sequential types, the type needs to be extracted from different documents such as WSDL, XML API and WS-Policy. Automatic extraction of the server input type from these documents could be a part of our future work.

**Un-Ordered Data**

In our algorithms (both in the CFG solution for sequential types and the GTRS solution for the tree-structured types), we have assumed that the data should be ordered. Support for un-ordered data raises the time complexity of the algorithms, however, it might be useful in practice. Supporting this feature is a part of the future work. We may be able to use the theory of sheaves automata [27] for this purpose.

**Web Service Composition**

Composition of Web services has received much interest to support business-to-business or enterprise application integration. As mentioned in the Introduction Chapter, there are differences between Web service and handler composition. However, we might be able to extend this research to solve the problem of Web service composition as well.

**Multiple Intermediate Servers**

Our approach only covers the existence of one intermediary on the path, but obviously more than one intermediary could exist, and our desired handler set could be distributed over these intermediaries. There are two main differences between the one and many intermediaries approach:

1. If there is more than one intermediary and we want to change the type of the client's output to be compatible with final server's input, then a central server should know about the handlers on all of the intermediaries so that it can compose them. This central server could be for example the first intermediate server which we know (or decide that) every client will route the message through it, and every handler on other intermediaries should be registered on this intermediate server.

2. Clearly the networking time should be optimized, so we prefer that if there are two handlers that should be executed on one intermediary, one of them
Chapter 9. Conclusion

is executed right after the other one, rather than having a third handler executed on another intermediary in between the two of them. Finding the algorithm that minimizes the networking time for the execution of these distributed handlers could be a part of the future work; however heuristics like assigning priority (not precedence, but optional precedence) to the handlers could optimize the networking time.
Bibliography


Chapter 9. Conclusion


58
Chapter 9. Conclusion


Appendix A

First Appendix

In order to construct the parsing table, we need to calculate the “first set” and the “follow set” for every non-terminal in the grammar. The first set of _w_ (Fi(_w_)) is the set of terminals that can be found at the start of any string in _w_, plus _e_ if the empty string also belongs to _w_. The follow set (Fo(_w_)) is the set of terminals that can follow a symbol. We have to compute that for non-terminals because a non-terminal might produce an empty string (λ) so we want to know what non-terminals may follow it.

If our grammar is made up of the rules _A_ → _w_1, ..., _A_ → _w_n, the following algorithms show how to calculate the first set and the follow set for the grammar [1]:

Algorithm for calculating the first set:

1. initialize every Fi(_w_i_) and Fi(_A_i_) with the empty set
2. add Fi(_w_i_) to Fi(_A_i_) for every rule _A_i_ → _w_i_, where Fi is defined as follows:
   - Fi(_aw'_) = _a_ for every terminal _a_
   - Fi(_Aw'_) = Fi(_A_) for every nonterminal _A_ with _λ_ not in Fi(_A_)
   - Fi(_Aw'_) = Fi(_A_) ∪ Fi(_w'_), for every nonterminal _A_ with _λ_ in Fi(_A_)
   - Fi(_A_) = _λ_
3. add Fi(_w_i_) to Fi(_A_i_) for every rule _A_i_ → _w_i_
4. do steps 2 and 3 until all Fi sets stay the same.

Algorithm for calculating the follow set:

1. initialize every Fo(_A_i_) with the empty set
2. if there is a rule of the form _A_j_ → _w_A_i_ _w'_, then
   - if the terminal _a_ is in Fi(_w'_), then add _a_ to Fo(_A_i_)
   - if _λ_ is in Fi(_w'_), then add Fo(_A_j_) to Fo(_A_i_)
3. repeat step 2 until all Fo sets stay the same

If _T_ is the parsing table, _T_[_A_, _a_] will contain the rule _A_ → _w_ if and only if _a_ is in Fi(_w_), or _λ_ is in Fi(_w_) and _a_ is in Fo(_A_).
Appendix B

Second Appendix

The following is the GetAccount Request Message Schema.

```xml
<?xml version="1.0" encoding="utf-8"?>

<GetMyeBayBuyingRequest xmlns="urn:ebay:apis:eBLBaseComponents"
<!-- Standard Input Fields -->
<DetailLevel> DetailLevelCodeType </DetailLevel>
<ErrorLanguage> string </ErrorLanguage>
<MessageID> string </MessageID>
<Version> string </Version>
<WarningLevel> WarningLevelCodeType </WarningLevel>
<!-- Call-specific Input Fields -->
<BestOfferList>
  <DurationInDays> int </DurationInDays>
  <Include> boolean </Include>
  <IncludeNotes> boolean </IncludeNotes>
  <ListingType> ListingTypeCodeType </ListingType>
  <Pagination>
    <EntriesPerPage> int </EntriesPerPage>
    <PageNumber> int </PageNumber>
  </Pagination>
  <Sort> ItemSortTypeCodeType </Sort>
</BestOfferList>

<BidList>
  <IncludeNotes> boolean </IncludeNotes>
  <Pagination>
    <EntriesPerPage> int </EntriesPerPage>
    <PageNumber> int </PageNumber>
  </Pagination>
  <Sort> ItemSortTypeCodeType </Sort>
</BidList>

<FavoriteSearches>
  <Include> boolean </Include>
  <MaxResults> int </MaxResults>
  <Sort> SortOrderCodeType </Sort>
</FavoriteSearches>
```
Appendix B. Second Appendix

<FavoriteSellers>
  <Include> boolean </Include>
  <MaxResults> int </MaxResults>
  <Sort> SortOrderCodeType </Sort>
</FavoriteSellers>

<LostList>
  <DurationInDays> int </DurationInDays>
  <IncludeNotes> boolean </IncludeNotes>
  <Pagination>
    <EntriesPerPage> int </EntriesPerPage>
    <PageNumber> int </PageNumber>
  </Pagination>
  <Sort> ItemSortTypeCodeType </Sort>
</LostList>

<SecondChanceOffer>
  <Include> boolean </Include>
  <MaxResults> int </MaxResults>
  <Sort> SortOrderCodeType </Sort>
</SecondChanceOffer>

<WatchList>
  <IncludeNotes> boolean </IncludeNotes>
  <Sort> ItemSortTypeCodeType </Sort>
</WatchList>

<WonList>
  <DurationInDays> int </DurationInDays>
  <IncludeNotes> boolean </IncludeNotes>
  <Pagination>
    <EntriesPerPage> int </EntriesPerPage>
    <PageNumber> int </PageNumber>
  </Pagination>
  <Sort> ItemSortTypeCodeType </Sort>
</WonList>

</GetMyeBayBuyingRequest>
Appendix C

Third Appendix

The following is the Input of the Reachability Algorithm in the Case Study in Chapter 5.

EXPR

type USD = usd
type CAN = can
type LBP = lbp

AccountHistorySelectionCodeType = ?[ ?[ ?[ ?[ ?[ ?[ ?[accountEntryCreatedTimeAscending | accountEntryCreatedTimeDescending] | accountEntryItemNumberAscending] | accountEntryItemNumberDescending] | accountEntryFeeTypeAscending] | accountEntryFeeTypeDescending] | customCode]

AccountEntrySortType = ?[ ?[ ?[ ?[ ?[ ?[ ?[accountEntryCreatedTimeAscending | accountEntryCreatedTimeDescending] | accountEntryItemNumberAscending] | accountEntryItemNumberDescending] | accountEntryFeeTypeAscending] | accountEntryFeeTypeDescending] | customCode]

AccountEntrySortTypeCodeType = ?[ ?[ ?[ ?[ ?[ ?[ ?[accountEntryCreatedTimeAscending | accountEntryCreatedTimeDescending] | accountEntryItemNumberAscending] | accountEntryItemNumberDescending] | accountEntryFeeTypeAscending] | accountEntryFeeTypeDescending] | customCode]

AccountHistorySelection = accountHistorySelection(AccountHistorySelectionCodeType)

BeginDate = beginDate(dateTime)

EndDate = endDate(dateTime)

ExcludeBalance = excludeBalance(boolean)

ExcludeSummary = excludeSummary(boolean)

InvoiceDate = invoiceDate(dateTime)

Pagination = pagination(entriesPerPage(int) pageNumber(int))

Sort = sort(AccountEntrySortTypeCodeType)

AccountPageType = accountPageType(Period)

Summary = summary(boolean)

Period = ?[ ?[ ?[ ?[ ?[ ?[ ?[accountEntryCreatedTimeAscending | accountEntryCreatedTimeDescending] | accountEntryItemNumberAscending] | accountEntryItemNumberDescending] | accountEntryFeeTypeAscending] | accountEntryFeeTypeDescending] | customCode]

GetAccountRequestOldPaginationInvoice =

getAccountRequest(accountEntrySortType(AccountEntrySortTypeCodeType))
accountHistorySelection(AccountHistorySelectionCodeType)
beginDate(dateTime) currency(CurrencyType) endDate(dateTime)
excludeBalance(boolean) excludeSummary(boolean)
invoiceMonth(int) invoiceYear(int)
entriesPerPage(int) pageNumber(int))
type GetAccountRequest =
getAccountRequest(accountEntrySortType(AccountEntrySortTypeCodeType)
accountHistorySelection(AccountHistorySelectionCodeType)
beginDate(dateTime) currency(CurrencyType) endDate(dateTime)
excludeBalance(boolean) excludeSummary(boolean)
invoiceDate(dateTime) pagination(Pagination))

RULES

Sort -> AccountEntrySortType
GetAccountRequestOldPaginationInvoice -> GetAccountRequest
Period -> AccountHistorySelectionCodeType
AccountPageType -> AccountHistorySelection
Summary -> ExcludeSummary

REACH(GetAccountRequest)

TREE

getAccountRequest(
  sort(accountEntryCreatedTimeAscending)
  accountPageType(accountEntryFeeTypeDescending)
  beginDate(2006-5-10T12:00:00-05:00 )
  currency(usd)
  endDate(2006-10-10T12:00:00-05:00 )
  excludeBalance(false) summary(true)
  invoiceMonth(8) invoiceYear(2006)
  entriesPerPage(5) pageNumber(4))