Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Pavol Rusnák

Efektivní mechanismy XML komunikace

Effective XML-based communication mechanisms

Department of Software Engineering

Supervisor: Doc. Ing. Jan Janeček, CSc.

Study Program: Computer Science, Software Systems

2008

Hereby, I would like to thank my supervisor and the author of the idea of SOAP processing in embedded systems *Doc. Ing. Jan Janeček, CSc.* for his time, patience, advices and comments he gave me during our cooperation.

Next I would like to thank *Peter Libič* for his help with benchmarking of the proposed solution and with POSIX sockets, *Tomáš Dzetkulič* and *Jakub Galgonek* for showing me flaws in generation of grammar for permutations and steering me towards the better solution.

I would also like to thank my family, girlfriend and friends for their patience and good environment they created even though I had a very little time to spend with them during the last days.

I declare that I wrote the thesis by myself and listed all used sources. I agree with making the thesis publicly available.

Prague, April 18, 2008 Pavol Rusnák

Contents

List of Figures

List of Tables

Název práce: Efektivní mechanismy XML komunikace
Autor: Pavol Rusnák
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. Ing. Jan Janeček, CSc.
e-mail vedoucího: janecek@fel.cvut.cz

Abstrakt: XML komunikace je efektivní, protože je příjemná z programátorského hlediska a má obrovskou flexibilitu. Dodnes se rozšířila na rozmanité platformy kromě jedné oblasti, kterou jsou vestavěné systémy. Je to hlavně proto, že zpracovávaní textových správ je náročné na prostředky a má obvykle velkou režii. Cílem této práce je navrhnout, implementovat a otestovat novou metodu zpracovávaní SOAP zpráv pomocí formálních gramatik a automatů a tím vlastně umožnít rozšíření webových služeb také na tato malá zařízení, dokonce i na ta, která neobsahují vlastní operační systém. Naše řešení jsme pojmenovali EXCUSA, podle akronymu anglického názvu "Effective XML Communication Using SOAP and Automata".

Klíčová slova: vestavěný systém, gramatika, webová služba, WSDL, SOAP

Title: Effective XML-based communication mechanisms
Author: Pavol Rusnák
Department: Department of Software Engineering
Supervisor: Doc. Ing. Jan Janeček, CSc.
Supervisor's e-mail address: janecek@fel.cvut.cz

Abstract: XML-based communication is effective, because it is programmer-friendly and has great flexibility. It is widely spread on variety of platforms. However, penetration of this technology has not yet reached embedded devices, because parsing of textual messages has usually large overhead and is demanding on resources. The goal of this work is to devise, implement and test a new method of parsing SOAP messages involving formal grammars and automata, thus allowing the use of web services to expand also to small devices, even on the ones without OS support. Our solution is called EXCUSA – acronym for "Effective XML Communication Using SOAP and Automata".

Keywords: embedded system, formal grammar, web service, WSDL, SOAP

Chapter 1

Introduction

1.1 Motivation

Forty years ago the computers were predominantly massive and expensive systems, which could be afforded only by the large institutions like the universities and companies. Advent of the microprocessors made these systems smaller and cheaper, what originated the creation of workstations or so called personal computers. Later, when local area networks were spread across the world, the concept of distributed system was born – an idea to connect these independent machines and to aggregate their computing power, so they appear to user as a single coherent system. Technologies that interconnect individual software components, which can be run on distinct hardware configurations, are called *middleware*. Over the years several implementations were created, e.g. Sun/ONC RPC, DCE/RPC, MPI, Object Management Group's CORBA, Microsoft's DCOM or Java RMI. But today one of the most commonly used middleware architecture is so-called *service-oriented architecture* – a modern technology based on web services, XML and SOAP.

Communication based on XML messages is effective because of the programming simplicity and flexibility of the usage. That is the main reason why this method has penetrated into nearly all spheres of distributed architectures. However, there was one exception. If small (embedded) devices were part of the solution, some binary protocol was usually used instead. Binary protocol has lower requirements on bandwidth size and on computing power of device. Situation was getting better and better and now we have microcontrollers with 128–256 kilobytes of memory, clock speeds up to 32 MHz and transfer speeds up to 250 kbit/s. The biggest disadvantage of XML communication is a large overhead because of parsing text messages.

This drawback can be reduced by using appropriate techniques so that embedded web services can come closer to classic ones and we could use SOAP also in small devices – e.g. sensors in control systems and servomechanisms or other actuators in advanced appliances.

1.2 Goals

The aim of this work will be to describe usage possibilities of XML-based RPC communication on small (embedded) devices, design and create an environment, that will allow development of effective web services on these devices – even on the ones without operating system. This can be achieved by using formal description of the messages by grammars.

Focal point of the work will be the compiler, which was suggested in [1]. It should take the web service definition (in WSDL file) as input and create source code for client and server. Proposed technique should be language independent. However, we will put emphasis on C language because C is the most used language in microcontroller programming. Algorithms should be as simple as possible to minimize the usage of the libraries – only the standard C library will be used. At the same time we want these algorithms to be effective (no string copying, all operations done in static buffers and minimum dynamic allocations of memory). The responsiveness and simple integration of the generated solution should also be a priority.

An integral part of the work will be to test the proposed solution in a real environment, profile various parts of the process and to compare its performance with the performance of other existing solutions and approaches.

1.3 Overview of the chapters

The Chapter 2 presents various technologies and concepts that are used throughout the work. Technologies relate mostly to XML and the web service world, while concepts refer to languages, grammars, automatons, their description and formal definitions.

In the Chapter 3 we explain theoretical background of the work, how the XML and grammar areas can be used together to achieve better results in middleware communication, especially optimizing parsing of the XML messages. We show off some problems that we were facing and how to solve or avoid them.

In the Chapter 4 we describe our solution programmed in Java and provide detailed view on the internals of the generated code in C. We depict the generating process, especially the cooperation of the classes and the focus is also put on representation of data structures in the output code.

The Chapter 5 contains results obtained from various benchmarks and tests we realized. Performance of the proposed solution was tested and compared to other, up to date existing, alternatives.

The last chapter brings the conclusion, list of accomplishments, themes and ideas for the future work.

Chapter 2

Technologies and concepts

In this chapter we present a mechanism of Remote Procedure Call, which is one of the basic methods used in distributed computing. Later we describe web services and individual technologies they utilize. Chapter ends with selected concepts from automata and grammar theory[2].

2.1 Remote Procedure Call (RPC)

Remote Procedure Call is a method that allows one code to call another that is not located in the same address space as calling one. Called code does not have to be on the same system, it could be for example on another system connected to first one with network. This allows to create distributed applications based on client-server architecture. The basic principle of RPC is that programmer does not have to care about code interaction and network details, he uses remote functions as it would be local ones. If object oriented programming language is used, we are talking about *remote method invocation*.

If program uses remote functions, so called *stub* is compiled in code, which represents code of remote procedure. When running program encounters remote procedure call, this stub passes parameters to client runtime library, which creates request message and sends it to known remote server across the network. Server handles the message (i.e. parses it and calls remote function) and creates response message, which is sent back to client runtime library, where it is again processed and result is returned to stub, which passes it back to program (see the Figure 2.1). Program then continues in its flow. This applies only for *synchronous* calls, where program is blocked while waiting for response. Program continues immediately when using *asynchronous* calls and will pick up response later - with signal or handler.

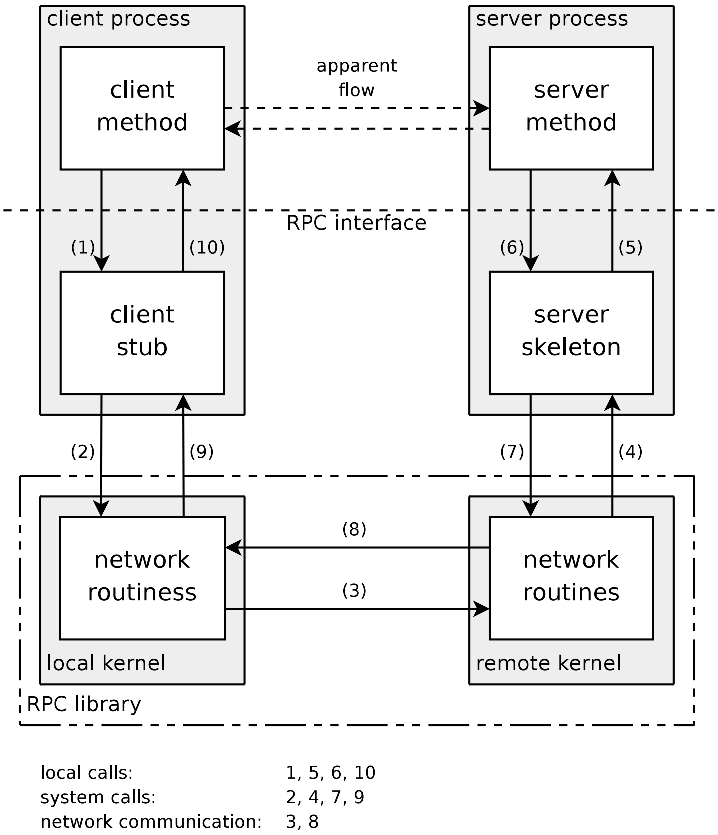

Figure 2.1: RPC model

Important difference between local and remote procedures is that remote calls can fail also because of problems caused by message transfer (most frequently network problems). This connects with another unpleasant problem: caller often does not know if the remote procedure was called or not and calling method more times with same parameters can cause trouble. (This does not happen with *idempotent* procedures, i.e. procedures that can be called safely multiple times.)

Messages used to represent remote calls have different formats and techniques use various protocols. This yields in often non-compatible implementations. In this work we will mention two RPC methods which use XML for transferring messages: XML-RPC (Section 2.5) and especially SOAP

(Section 2.6).

2.2 Web Services

Services, similar to components, are independent building blocks of application. Contrary to classic components services have some characteristic features, that allow them to become a part of service-oriented architecture. One of such features is a complete autonomy from other services. This means that every service is responsible for itself, what typically means, that it contains only limited range of logically related specific functions. This make it possible to create stand-alone units, which are loosely connected by a compliance to a standard communication framework. This characteristic enables the programmatic logic, which is encapsulated by services, to be platform and technology independent. The most widespread and successful type of web services are XML web services. This type utilizes lots of technologies, which will be described in following sections. Their relationship is pictured in the Figure 2.2.

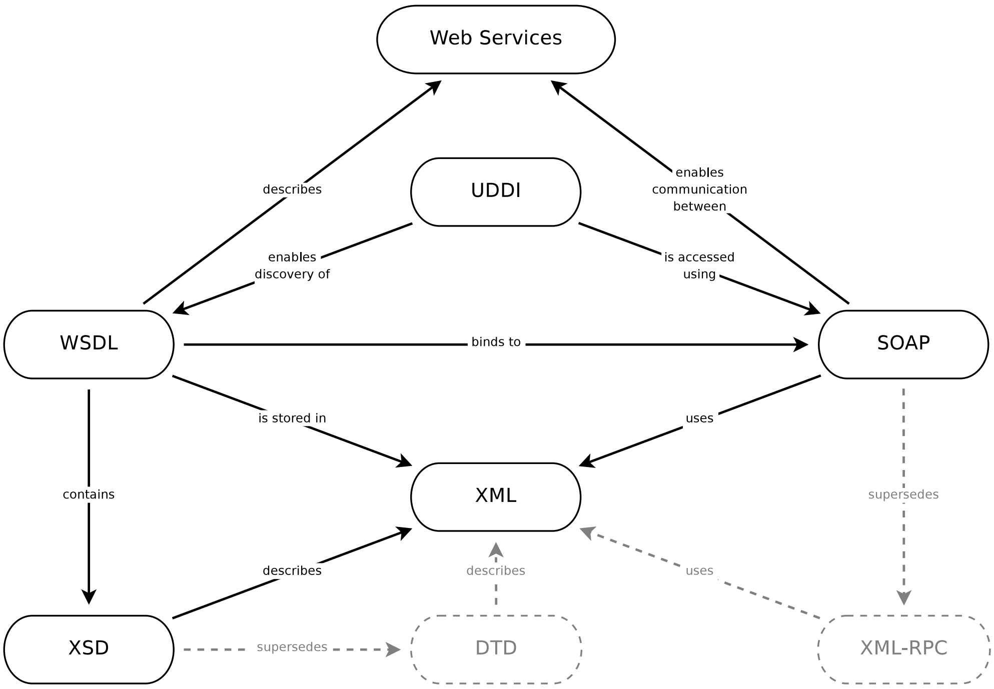

Figure 2.2: Relationship between the technologies used in XML Web Services

2.3 Extensible Markup Language (XML)

XML[3] is a markup language similar to HTML, but was designed to structure, store and transport data. The main difference between XML and HTML is that HTML was created with focus on how data looks, while XML was created with focus on what data is. XML data is stored in plain text form, so it is software and hardware independent and it can be easily exchanged between incompatible systems and platforms.

XML documents form a tree structure with each node represented as *element*. Element is a part of document enclosed in *tags*. It can contain other elements (called children), data or both children and data. Tags are identified by their name and are enclosed between `<` and `>` characters. Contrary to HTML tags are case sensitive and must be properly nested – they may not overlap.

Example of simple XML document, which stores information about countries of the world:

```
<?xml version="1.0" encoding="UTF-8"?>
<world>
  <country>
    <name>Czech Republic</name>
    <abbr>CZE</abbr>
    <population>10381130</population>
  </country>
  <country>
    <name>Slovak Republic</name>
    <abbr>SVK</abbr>
    <population>5447502</population>
  </country>
  <!-- TODO: add more countries later  -->
</world>
```

As we can see, XML format is pretty self descriptive, because tags are not predefined, but determined by creator of XML document. XML is also extensible, which means that structure of XML can be changed without breaking the applications. For example, when someone adds information about capital cities to our XML document (by adding `capital` element as a child of `country` element), applications can still retrieve `name`, `abbr` and `population` fields.

A *well-formed* XML document has correct XML syntax, meaning that document has one root element, each opening tag has a corresponding closing tag and elements are properly nested.

Valid XML document is a well-formed XML document, which also conforms to some semantic rules. We will tell more about the definition of these rules in next section.

2.4 XML Schema Definition (XSD)

As said in previous section, we can formally describe the content of XML documents by defining some semantic rules using a schema definition language. Situation is very similar to schemas used in database systems, where they create structural model for data. In XML world, schemas provide structure validation rules, type constraints and description of relationship between elements. The most used formal descriptions are Document Type Definition (DTD)[3] and XML Schema Definition[4]. We will focus on the second one as XSD is the one used in definition of web services.

In contrast to DTDs, XML schemas are themselves XML documents. They support wide variety of data types[5] and also *namespaces*[6]. These allow author of the schema to split definition into several logical domains to which some parts of a schema can be applied. This format is very flexible and extensible. Each schema definition can contain multiple schema definitions and each schema can be dynamically extended or have its parts overridden by another schema definition.

Example of XML Schema definition of the XML document shown in previous section:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ns="http://gk2.sk/">
  <xs:element name="world" type="ns:world"/>
  <xs:element name="country" type="ns:country"/>
  <xs:complexType name="ns:world">
    <xs:sequence>
      <xs:element name="country" type="ns:country"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ns:country">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="abbr" type="xs:string"/>
      <xs:element name="population" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

XML schema elements can be either of *simple* or *complex* type, according to whether they contain attributes or child elements or not. In our example **world** is a parent element, so it has a complex type. Situation is the same with **country** element. Schema element *sequence* groups children elements and tells us that they have to appear in the same order as they were specified. Similar to *sequence* is *all* element, but in this case children elements can appear in any order. The last grouping type is *choice*. Element can contain exactly one element from the listed ones as its child, when it is used. Elements **name**, **abbr** and **population** do not contain any attributes or child elements so they have simple types – built-in `xs:string` and `xs:integer` in particular.

2.5 XML-RPC

XML-RPC is a method which uses XML to encode the remote procedure calls. Hypertext Transfer Protocol is used as a transport mechanism. This approach is quite simple and defines only several data types with no inheritance. These types are listed in the following table:

`boolean`	boolean logical value (0 or 1)
`int`	whole number, integer
`double`	double precision floating point number
`string`	string of characters
`dateTime`	date and time in ISO 8601
`base64`	base64-encoded binary data
`array`	array of values
`struct`	associative array
`nil`	null value

Table 2.1: XML-RPC data types

The fact it could work only with simple data structures caused that this technology was not adopted as a W3C standard and is today considered as a legacy technology. We list it here because it is quite easy to understand and its successor SOAP makes use of the similar ideas.

Protocol recognizes two types of messages – requests and responses. They differ in their root element. Requests have `<methodCall>` as their root element, while responses have `<methodResponse>`. Every call contains method name and a list of method parameters. These are encapsulated in `<methodName>` and `<params>` elements respectively. Response does not have `<methodName>` element and contains only `<params>` element with return value(s).

Example of XML-RPC request and response:

```
<methodCall>
  <methodName>World.getCountryCode</methodName>
  <params>
    <param><value><string>Slovakia</string></value></param>
  </params>
</methodCall>

<methodResponse>
  <params>
    <param><value><string>SVK</string></value></param>
  </params>
</methodResponse>
```

However, XML-RPC defines one special type of method response – Fault. This message does not have `<params>` element. Rather it contains element `<fault>`, which holds information about an error which occurred. The body of such fault is typically composed of error code and error description.

Example of XML-RPC fault response:

```
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Country code not found.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

2.6 Simple Object Access Protocol (SOAP)

In spite of the fact that this protocol was created primarily as a bridge to connect repugnant RPC-based communication architectures, SOAP protocol has become the most widespread format for use with XML web services. That's why some interprets the abbreviation SOAP as Service-Oriented Architecture Protocol and not as its original meaning Simple Object Access Protocol.

SOAP allows to use both synchronous and asynchronous transfer of XML documents or remote procedure calls[7]. Frame of the SOAP message is called the *SOAP envelope*. This envelope can contain message header and has to contain message body[8]. Let's have a look at the skeleton of the SOAP message:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope">
  <env:Header>
    ...
  </env:Header>
  <env:Body>
    ...
  </env:Body>
</env:Envelope>
```

Header can include more blocks. Typical usage is to indicate implementation of predefined or application-specific SOAP extensions or to provide supplementary meta information about the SOAP message. Headers are massively used in second-generation specifications of web services and are not supported by EXCUSA.

Body serves as a container for transferred data, which are usually called *payload*. It can basically be arbitrary XML document, eventually a set of

XML documents. SOAP does not differentiate request and response messages at the level of XML documents like it is in XML-RPC, but it secerns fault messages. These messages contain `Fault` element in `Body` construction. This is a standard fault message used when communicating nodes have incompatible SOAP versions:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <env:Fault>
      <env:Code><env:Value>env:VersionMismatch</env:Value></env:Code>
      <env:Reason>
        <env:Text xml:lang="en">SOAP Version Mismatch</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

2.7 Web Services Description Language (WSDL)

In order that web services could communicate with each other we have to define them in a consistent manner. **Web Services Description Language (WSDL)** was created for this purpose. We will split the definitions in it to two separated parts – abstract interface and concrete implementation[9]. Both parts are stored in WSDL document's root element `definition`. Information about custom types also resides here.

2.7.1 Abstract definition – interface

Abstract definition can contain several service *interfaces*. Each interface is a group of logically related *operations*, which represent web service functions. We can liken the interface to interfaces known from object oriented languages and operations to their methods. These operations correlate one *input message* and one *output message* together. Operation can also have any number of custom *fault messages*. Every message consists of *parts*, while each part has an associated data type and corresponds to one input or output parameter. If operation has only one output parameter, we talk usually about a return value.

Example of simple interface – calculator with methods for adding and multiplying two numbers:

```
<wsdl:definitions>
  <wsdl:interface name="Calculator">
    <wsdl:operation name="Add">
      <wsdl:input message="TwoNumbers"/>
      <wsdl:output message="OneNumber"/>
      <wsdl:fault message="Error"/>
    </wsdl:operation>
    <wsdl:operation name="Multiply">
      <wsdl:input message="TwoNumbers"/>
      <wsdl:output message="OneNumber"/>
      <wsdl:fault message="Error"/>
    </wsdl:operation>
  </wsdl:interface>
  <wsdl:message name="TwoNumbers">
    <wsdl:part name="arg1" type="xs:double"/>
    <wsdl:part name="arg2" type="xs:double"/>
  </wsdl:message>
  <wsdl:message name="OneNumber">
    <wsdl:part name="ret" type="xs:double"/>
  </wsdl:message>
  <wsdl:message name="Error">
    <wsdl:part name="code" type="xs:int"/>
  </wsdl:message>
...
```

2.7.2 Concrete definition – implementation

Service in WSDL document represents one or more *endpoints* where web service can be accessed. These endpoints contain information about physical location and used protocol. *Binding* associates protocol and message format to operations, which correspond to operations described in interface. Thereby binding actually defines how each individual operation can be invoked.

Example of simple implementation – definition continues from interface shown above:

```
...
  <wsdl:service name="Calculator">
    <wsdl:endpoint name="CalculatorSoap" binding="CalculatorSoap">
```

```
      <soap:address location="http://localhost/Calculator" />
    </wsdl:endpoint>
  </wsdl:service>
  <wsdl:binding name="CalculatorSoap">
    <wsdl:operation name="Add">
      <soap:operation soapAction="http://localhost/Calculator/Add"/>
      <wsdl:input><soap:body use="literal"/></wsdl:input>
      <wsdl:output><soap:body use="literal"/></wsdl:output>
      <wsdl:fault><soap:body name="Error" use="literal"/></wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="Multiply">
      <soap:operation soapAction="http://localhost/Calculator/Multiply"/>
      <wsdl:input><soap:body use="literal"/></wsdl:input>
      <wsdl:output><soap:body use="literal"/></wsdl:output>
      <wsdl:fault><soap:body name="Error" use="literal"/></wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

2.7.3 Types

Web service definition contains typically also a definition of custom types and elements used in individual messages. Element `types` allows to insert XSD document into web service definition.

2.7.4 WSDL 1.1 vs WSDL 2.0

In previous two subsections we've used WSDL terminology and syntax in version 2.0[10]. Although the work on new version started already in the year 2003, it was not until June 2007, when it became W3C recommendation. Therefore this standard is still not massively used and vast majority of web services use WSDL in version 1.1[11]. Main and the most apparent differences are:

- `interface` element was called `portType` before
- `endpoint` element was called `port` before
- WSDL 2.0 allows modularization by including and importing of service descriptions

- inheritance of interfaces is supported by `extends` attribute in the interface element
- message constructs were obsoleted in favor of elements in types section (each message can be replaced by element of sequence complex type)

2.8 Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery and Integration (**UDDI**) is a platform independent directory for storing and retrieving information about web services and its interfaces described by WSDL. It is accessed using SOAP. Our work does not utilize this technology directly.

2.9 Generative grammar

Alphabet is any finite set Σ of members called *symbols*. String over the alphabet Σ is arbitrary sequence of the symbols of this alphabet. For example string $aabcccb$ is a string over the alphabet $\{a, b, c\}$. Grammar is an exact description of the language. In other words, it describes, which sequences of symbols – strings – are valid ones – words. We can generate all words using grammar and what is more important grammar can also be used to create *recognizer*, which decides whether the word is *grammatical* (belongs to language) or not. Formal descriptions of these recognizers are called *automata*. In our work we will focus on special type of generative grammars. They contain only one nonterminal on left side of each rewriting rule and are called *context-free* grammars.

Definition 1 *Generative context-free grammar is quad-tuple*

$$G = (\mathbb{N}, \Sigma, S, \mathbb{R})$$

where:

- $\mathbb{N}$ *is a finite nonempty set of nonterminal symbols or nonterminals*
- Σ *is a finite nonempty set of terminal symbols or terminals (disjoint from* $\mathbb{N}$*)*

- $S \in \mathbb{N}$ *is the start symbol also called grammar root*
- $\mathbb{R}$ *is a finite set of production or rewriting rules,*
  $\mathbb{R}$ *is a relation* $\mathbb{R}$*:* $\mathbb{N} \rightarrow (\mathbb{N} \cup \Sigma)^*$

Language of a formal grammar, denoted $L(G)$, is a set of all words over alphabet Σ, which can be generated from the start symbol S and applying the production rules from $\mathbb{R}$ until all nonterminal symbols are replaced with terminal symbols. Language generated by context-free grammar is context-free and is accepted by Pushdown automaton (see the Section 2.11).

2.10 Finite state automaton

A **finite state automaton** is an abstract model of behavior, which consists of finite count of states and transitions between them. It contains control unit, a reading head and a tape, on which the input string is written. At the beginning of computation the reading head is located at the beginning of the tape. In each step automaton reads one symbol from the tape, changes its state and moves the head one field forward. States changes according to the transition function, which means that the change depends on current state and last read symbol. Computation ends when the automaton is "blocked" or the whole input string is read. The string is *accepted* by automaton when it is read to the end **and** resulting state is one of the accepting states.

Definition 2 *Finite state automaton is a 5-tuple*

$$F = (\mathbb{Q}, \Sigma, \delta, q_0, \mathbb{A})$$

where:

- $\mathbb{Q}$ *is a finite set of states*
- Σ *is a finite set of input symbols (input alphabet)*
- δ *is a transition function,* $\delta : \mathbb{Q} \times \Sigma \rightarrow \mathbb{Q}$
- $q_0 \in \mathbb{Q}$ *is the start state*
- $\mathbb{A} \subseteq \mathbb{Q}$ *is a set of accepting states*

2.11 Deterministic pushdown automaton

Deterministic pushdown automaton is a state machine derived from deterministic finite automaton by adding auxiliary memory called *stack*. In each step automaton can read one symbol from top of the stack and can push down arbitrary count of symbols (even the one that was just read). Transition depends on the current state, read symbol from the input tape and value from the top of the stack. Automaton stops when the final state is reached (we talk about acceptance by *final state*) or there are no symbols on stack (we talk about acceptance by *empty stack*). These two acceptance criteria are not equivalent in deterministic pushdown automaton.

Definition 3 *Deterministic pushdown automaton is a 7-tuple*

$$M = (\mathbb{Q}, \Sigma, \Gamma, \delta, q_0, Z_0, \mathbb{A})$$

where:

- $\mathbb{Q}$ *is a finite set of states*
- Σ *is a finite set of input symbols (input alphabet)*
- Γ *is a finite set of stack symbols (stack alphabet)*
- δ *is a transition function,* $\delta : \mathbb{Q} \times (\Sigma \cup \{\lambda\}) \times \Gamma \to \mathbb{Q} \times \Gamma^*$
- $q_0 \in \mathbb{Q}$ *is the start state*
- $Z_0 \in \Gamma$ *is the start stack symbol*
- $\mathbb{A} \subseteq \mathbb{Q}$ *is a set of accepting states*

Chapter 3

Theory

In our work we tried to optimize XML communication using knowledge in formal language theory. To understand how the optimization process works, we introduce one particular type of context-free grammars and later we will show how we deal with concrete fragments of XML schemas.

3.1 XML grammar

All XML documents consist of tags and text located between these tags. Tags can be opening or closing and each opening tag has its corresponding closing tag and vice versa. Empty tags can be considered as opening tag followed immediately by closing tag, so we can assume that document contains no empty tags. Let's denote the set of opening tags as $\mathbb{T}$ and the set of corresponding closing tags as $\overline{\mathbb{T}}$. Since we are interested only in syntactic structure, we will omit attributes of the tags. Let's mark set of all valid values for type *int* as $\mathbb{V}_{int}$, for type *string* $\mathbb{V}_{string}$ and so on. Set $\mathbb{V}$ is union of all these sets. XML document is then word over the alphabet $\Sigma = \mathbb{T} \cup \overline{\mathbb{T}} \cup \mathbb{V}$ and we can formally define grammar describing XML documents:

Definition 4 *XML-grammar is quad-tuple*

$$G = (\mathbb{N}, \Sigma, S, \mathbb{R})$$

where:

- $\mathbb{N}$ *is a finite nonempty set of nonterminal symbols*

- $\Sigma = \mathbb{T} \cup \overline{\mathbb{T}} \cup \mathbb{V}$ *is a finite nonempty set of terminal symbols (disjoint from* $\mathbb{N}$*)*
- $S \in \mathbb{N}$ *is the start symbol*
- $\mathbb{R}$ *is a finite set of production or rewrite rules, each in one of these forms:*
  1. $n \rightarrow \mathbb{N}^*$
  2. $n \rightarrow tm\bar{t}$
  3. $n \rightarrow v$

  where $n, m \in \mathbb{N}$, $t \in \mathbb{T}$, $\bar{t} \in \overline{\mathbb{T}}$ *and* $v \in \mathbb{V}$

This definition of XML grammars is more simple than the one proposed by Berstel and Boasson in [12]. Their formalization is more general, but we wanted to achieve smaller complexity of the rewriting rules by having exactly one nonterminal on their left sides. These grammars are called *tree grammars* and are subset of *deterministic context-free grammars*, therefore *deterministic push-down automaton* recognizing them exists. An *XML language* is a language generated by some XML grammar.

3.2 Converting XML schema to XML grammar

XML documents are precisely described by XML schema. This suggests the idea to use XML schema to generate XML grammar, which will recognize given XML documents. We will show the technique in a couple of simple examples. Because contents of sets describing tags ($\mathbb{T}$ and $\overline{\mathbb{T}}$) are pretty straightforward ($\mathbb{T}$ contains all used t_{tag} symbols and $\overline{\mathbb{T}}$ contains all used $\overline{t_{tag}}$ symbols), we will list only rewrite rules (set $\mathbb{R}$) and mark nonterminals symbols as letters of Latin alphabet while A will be the start symbol. Symbol $v_{type} \in \mathbb{V}_{type}$ represents value of type *type*. Rules relevant to particular case are marked with full disc.

3.2.1 Simple type

Creating grammar for one element containing simple type is indeed simple. One rule is used to rewrite the start symbol to opening tag, another nonterminal symbol and corresponding closing tag. Another rule to change second nonterminal to particular value type.

XML example	XML grammar rewrite rules
`<size>3264</size>`	• $A \rightarrow t_{size}\ B\ \overline{t_{size}}$ • $B \rightarrow v_{int}$

3.2.2 Optional element

When element has attributes *minOccurs* and *maxOccurs* set to 0 and 1 respectively, we are talking about *optional element*. This element can but does not have to appear in document. We achieve this behavior by adding two rewriting rules into grammar. One is rewriting one nonterminal into another and the second one is changing the same nonterminal into empty symbol (ε).

XML example	XML grammar rewrite rules
`<error>`	○ $A \rightarrow t_{error}\ B\ \overline{t_{error}}$
`<code>3</code>`	• $B \rightarrow C$
`</error>`	• $B \rightarrow \varepsilon$
or	○ $C \rightarrow t_{code}\ B\ \overline{t_{code}}$
`<error></error>`	○ $D \rightarrow v_{int}$

3.2.3 Repeating element

Element can also repeat itself. Bounds are set with *minOccurs* and *maxOccurs* attributes of the element. When both these values are concrete numbers (let's mark them as m and n), we generate $n - m$ rules with the same nonterminal on their left side and their right side containing another nonterminal repeated $m, m+1, ..., n-1, n$ times. See the example where $m = 1$ and $n = 3$ (thus element `array` can contain one, two or three `val` elements):

XML example	XML grammar rewrite rules
	$\circ\ A \rightarrow t_{array}\ B\ \overline{t_{array}}$
`<array>`	$\bullet\ B \rightarrow C\ C\ C$
`  <val>1257</val>`	$\bullet\ B \rightarrow C\ C$
`  <val>358</val>`	$\bullet\ B \rightarrow C$
`</array>`	$\circ\ C \rightarrow t_{val}\ D\ \overline{t_{val}}$
	$\circ\ D \rightarrow v_{int}$

3.2.4 Infinitely repeating element

Element can repeat itself infinitely. This is the case when *maxOccurs* attribute is set to *unbounded* value. We solve this case by adding two rewrite rules with the same nonterminal on their left sides. One rewrites nonterminal to another nonterminal followed by the same nonterminal as on the left side (thus causing recursion), the second one rewrites nonterminal to empty symbol (ε) – stopping recursion.

XML example	XML grammar rewrite rules
`<array>`	$\circ\ A \rightarrow t_{array}\ B\ \overline{t_{array}}$
`  <val>974</val>`	$\bullet\ B \rightarrow C\ B$
`  <val>1465</val>`	$\bullet\ B \rightarrow \varepsilon$
`  ...`	$\circ\ C \rightarrow t_{val}\ D\ \overline{t_{val}}$
`</array>`	$\circ\ D \rightarrow v_{int}$

3.2.5 Complex content: Choice

Choice is done by adding one rewrite rule for each possibility. These rewrite rules contain the same nonterminal on their left side and exactly one nonterminal, equal to the start symbol of the element they refer to, on their right side.

XML example	XML grammar rewrite rules
`<value>`	○ $A \rightarrow t_{value}\ B\ \overline{t_{value}}$
`  <int>3</int>`	• $B \rightarrow C$
`</value>`	• $B \rightarrow E$
or	○ $C \rightarrow t_{int}\ D\ \overline{t_{int}}$
`<value>`	○ $D \rightarrow v_{int}$
`  <float>3.14159</float>`	○ $E \rightarrow t_{float}\ F\ \overline{t_{float}}$
`</value>`	○ $F \rightarrow v_{float}$

3.2.6 Complex content: Sequence

Contrary to choice, all listed elements have to appear in sequence. This means we manage this case by adding only one rewriting rule containing all the start symbols on its right side.

XML example	XML grammar rewrite rules
	○ $A \rightarrow t_{address}\ B\ \overline{t_{address}}$
`<address>`	• $B \rightarrow C\ E\ G$
`  <street>Evergreen Terrace</city>`	○ $C \rightarrow t_{street}\ D\ \overline{t_{street}}$
`  <number>742</number>`	○ $D \rightarrow v_{string}$
`  <city>Springfield</city>`	○ $E \rightarrow t_{number}\ F\ \overline{t_{number}}$
`</address>`	○ $F \rightarrow v_{int}$
	○ $G \rightarrow t_{city}\ H\ \overline{t_{city}}$
	○ $H \rightarrow v_{string}$

3.2.7 Complex content: All

All is similar to sequence, but listed elements can appear in any order. Unfortunately we have to calculate with all possibilities of ordering so we have to create all permutations to solve this problem. The most straightforward procedure is to generate rewriting rule for each permutation. This can be seen in following example. Only one permutation is shown in the XML document, but document can contain any other permutation of the inner tags.

XML example	XML grammar rewrite rules

`<address>`
`  <city>Springfield</city>`
`  <number>742</number>`
`  <street>Evergreen Terrace</city>`
`</address>`
	○ $A \longrightarrow t_{address}\ B\ \overline{t_{address}}$
	● $B \rightarrow C\ E\ G$
	● $B \rightarrow C\ G\ E$
	● $B \rightarrow E\ C\ G$
	● $B \rightarrow E\ G\ C$
	● $B \rightarrow G\ C\ E$
	● $B \rightarrow G\ E\ C$
	○ $C \longrightarrow t_{street}\ D\ \overline{t_{street}}$
	○ $D \longrightarrow v_{string}$
	○ $E \longrightarrow t_{number}\ F\ \overline{t_{number}}$
	○ $F \longrightarrow v_{int}$
	○ $G \longrightarrow t_{city}\ H\ \overline{t_{city}}$
	○ $H \longrightarrow v_{string}$

This technique brings obvious problem. When we have n inner elements, the number of rewriting rules increases by $n!$ (for 7 elements we have more than 5000 rules!). This count can be reduced by following method. Suppose we want to rewrite nonterminal S to every permutation of set of nonterminals $\{A, B, C, D\}$. We will introduce new nonterminals S_1 ... S_6. And add these rules:

$$\begin{array}{lll}
S \rightarrow S_1\ S_6 & S_1 \rightarrow A\ B & S_4 \rightarrow B\ C \\
S \rightarrow S_2\ S_5 & S_1 \rightarrow B\ A & S_4 \rightarrow C\ B \\
S \rightarrow S_3\ S_4 & S_2 \rightarrow A\ C & S_5 \rightarrow B\ D \\
S \rightarrow S_4\ S_3 & S_2 \rightarrow C\ A & S_5 \rightarrow D\ B \\
S \rightarrow S_5\ S_2 & S_3 \rightarrow A\ D & S_6 \rightarrow C\ D \\
S \rightarrow S_6\ S_1 & S_3 \rightarrow D\ A & S_6 \rightarrow D\ C
\end{array}$$

How did we construct these rules? We created rules $S \rightarrow S_i\ S_j$ for each possible break-up of sets into two halves. How many of these distributions we have? Since we are picking half of the elements from n the count is:

$$\binom{n}{\lceil n/2 \rceil} = \frac{n!}{\lceil n/2 \rceil! \lfloor n/2 \rfloor!}$$

If the set size is greater than 3, we repeat the procedure. If it is less than or equal to three we generate all permutations. For n nonterminals we can express the count of generated rules by this function:

$$f(n) = \binom{n}{\lceil n/2 \rceil} + \binom{n}{\lceil n/2 \rceil} * f(\lceil n/2 \rceil) = \binom{n}{\lceil n/2 \rceil} * (1 + f(\lceil n/2 \rceil))$$

where $f(1) = 1$, $f(2) = 2$ and $f(3) = 6$ (since we generate all permutations when the set has less than 4 members). This method gives us pretty better results than straightforward generating of permutations (see following table).

n	$n!$	$f(n)$
1	1	1
2	2	2
3	6	6
4	24	18
5	120	70
6	720	140
7	5040	665
8	40320	1330
9	362880	8946
10	3628800	17892

Despite this optimization the count of rewriting rules increases far too rapidly and we should avoid this type of complex content. This is usually not a problem, because in remote procedure calls we transfer messages with specified content, which correspond to structures defined in procedural languages and have thus fixed order of inner data. Therefore “sequence” complex type is a good replacement when designing the XML schema.

3.3 Automaton for XML grammar

How do we construct automaton which recognizes XML documents when we have already constructed XML grammar? We start with creating finite automata for each opening tag and each closing tag – automata F_{tag} and $\overline{F_{tag}}$ respectively. These automata will be very similar to each other because they differ only in tag name and whether they require a slash before tag or not. We will need also finite automaton for each supported value type – F_{type}. We now have to connect these automata so they can interact together. If grammar contained only rewrite rules with unique nonterminals on their left sides, another finite automaton would suffice. Rewrite rules would form some kind of static tree and the parsing procedure would be always the same independently of input. When we have more rules with the same nonterminal on their left side, we have to try more options at some point and often even revert last steps depending on input. Tracking back indicates that finite automata is not enough and we will need stack for storing previous steps. By adding stack to finite automaton we will get stronger apparatus - deterministic pushdown automaton (see the Section 2.11). If we limited the nesting level to some particular value we could have used finite automaton instead of stack automaton, but we would have to rebuild it when we wanted to alter schema and add for example one element. When we use stack automaton we have advantage that it suffices only to change the corresponding rewriting rule.

We will demonstrate the function of such automaton on a simple example from the Subsection 3.2.3. We have 5 finite automata. F_{array} accepts opening `<array>` tag, F_{val} accepts opening `<val>` tag, $\overline{F_{array}}$ and $\overline{F_{val}}$ accept corresponding closing tags and finally automaton F_{int} accepts all legal integer values. These automata are used by deterministic pushdown automaton, which calls them and if they accept the input word it continues and if they reject it it pops out the value of the stack and tries another rule.

Automaton takes the first rewrite rule and puts its left side nonterminal on the stack. Then it starts processing the right side. The first symbol is opening tag `array`, so finite automaton for it is called. It accepts the word, so we continue by pushing B on stack and processing this rule. Right side contains three Cs so we push first one onto stack and start processing it. We call automaton and it accepts tag `val`. Then D is pushed onto stack and processed by calling automaton for accepting integers. This will run smoothly and as we are at the end of D rule, we pop it out of stack and return to C. We call automaton for accepting closing `val` tag, pop C out of stack and return to B, which has 2 more Cs to test. Second one passes in the same way as the first one, but problem occurs with the third one. Automaton for opening tag `val` fails, so we have to pop all three Cs out of stack (B rule failed, so we have to clean it) and now we can test second B rule with only 2 Cs on its right side. This runs clearly to the end, so we can also pop out also B and after closing tag array we can also pop out A. At the end the stack is empty so we have successfully parsed our XML. If we run out of the rewriting rules during parsing (no more for match) or we

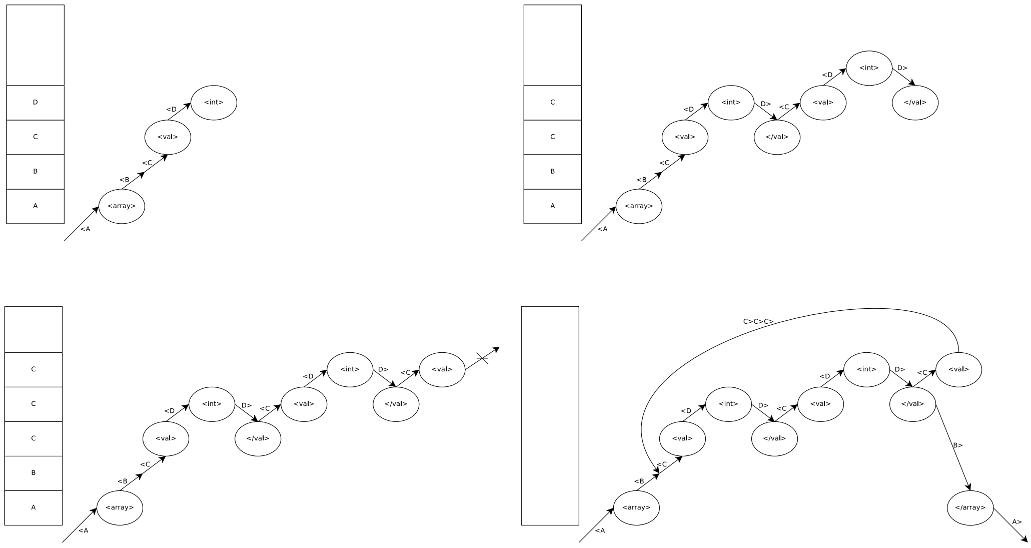

Figure 3.1: Automaton function

ended with non empty stack, XML is not processed.

Chapter 4

Implementation

The program is actually a command-line utility, which input is the only WSDL file with web service definition. This file could be hand crafted by user, if the user has sufficient knowledge, eventually retrieved from another running web server or some UDDI source. The most frequently used option is to generate WSDL file from sources in some higher object oriented programming language. There is *Java2WSDL* tool from Apache Axis project for Java and similar process also exists for C# language.

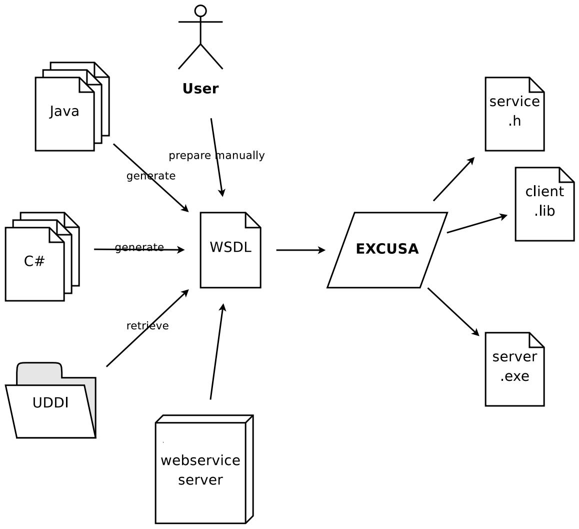

Figure 4.1: Usage schema

Program creates directory same as the name of the webservice and puts generated source files there. From these sources one can build independent server and library with header file for client. The only thing user has to do (apart from creating or retrieving WSDL file) is to move bodies of methods from client sources to server sources. Linking the library causes that these methods will be accessible in client and every call to them will lead to creation of request and communication with the server.

At the beginning we tried to implement the application in C++ language, but this later showed as a suboptimal solution, because there are not so many good libraries that simplify work with WSDL definitions and XML schemas and this inhibited the whole process. It is important that resulting generated code is not so demanding on resources, not the generator itself, so we switched development to Java. This made programming a lot easier, because there are many web service oriented frameworks for Java we could pick from. One of the best-known is the Apache Axis[13] project. We reused two packages from it – wsdl4j and XMLschema. This decision brought also another advantage: result code can be run on variety of operating systems supporting Java without any recompiling (for example Microsoft Windows, Apple Mac OS X, GNU/Linux, Solaris and others).

4.1 Program structure

The program could be divided in four logically separated components. These correspond to the packages used in Java programming language:

- `wsdl`
  objects for services, endpoints, operations, messages and other entities extracted from provided WSDL file
- `schema`
  objects for types and elements from XML schema definition found in WSDL file
- `grammar`
  objects for grammars, rewrite rules and grammar symbols (terminals and nonterminals)
- `code`
  objects for generating data structures and code

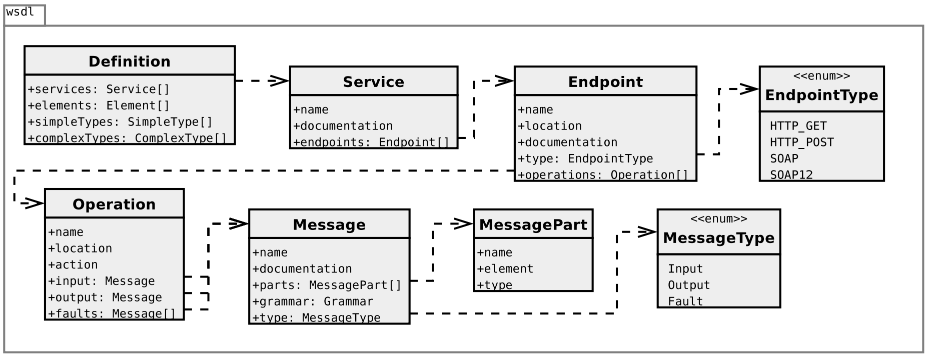

Figure 4.2: UML diagram of package wsdl

4.1.1 Package wsdl

Package **wsdl** contains classes which are related to web services and their definition. They provide methods to work with WSDL files. Wsdl4j package from Apache Axis project is used here.

The main class is **Definition**. It loads WSDL file, which can be either stored locally or retrieved from remote location. All entities such as services, operations and messages are converted from wsdl4j representation into our own. This is done because original representation was far too complex for our purposes. All services are stored in hashmaps, which are indexed with their name.

Class **Service** represents web service. Every service has its own unique name and could contain documentation, which is used as a comment in output code. These two attributes are present also in the next three classes. Service can define various access methods – these are called endpoints, which are stored in class **Endpoint**. Important attribute of endpoints is their type, which is enumerated in enum **EndpointType**. Valid endpoint types are: *HTTP_GET*, *HTTP_POST*, *SOAP* and *SOAP12*. Each service can have at most one endpoint of one type, what means that service can define at most four endpoints.

Every endpoint can have different set of operations, but in practice they are usually the same. Operations are stored in class **Operation**. HTTP operations (*HTTP_GET* and *HTTP_POST*) have attribute location and SOAP operations (*SOAP* and *SOAP12*) have attribute action. These have fundamentally the same meaning: they are sent to server in header, so server can identify to which method does the request belong to without parsing the message first. This also allows using one message for more methods, although this is not widely used to avoid confusion. Every operation has exactly one input message, exactly one output message and arbitrary number of fault messages.

All three types of messages are represented by class **Message** distinguishable by attribute type. These are enumerated in enum **MessageType** and valid values are: *Input*, *Output* and *Fault*. Every message contains at least one message part (class **MessagePart**) – containing reference to exactly one element or exactly one type (see next section).

4.1.2 Package schema

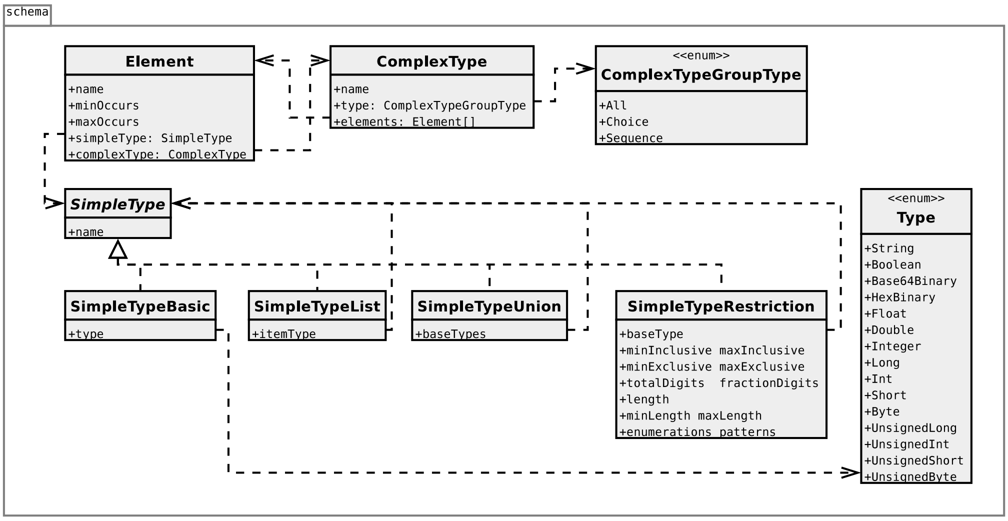

Figure 4.3: UML diagram of package schema

Objects in package **schema** hold information about types contained in XML schema defined in WSDL file. Class **Element** represents XML elements. Each element has its name, which is the same value as the identifier used in tags between opening and closing bracket. Element references exactly one simple type or exactly one complex type. If the simple type is used, particular value of this type can appear between the opening and closing tag and nothing else. In the case of complex type, element contains another elements in order specified by this complex type (see below). Very important attributes are *minOccurs* and *maxOccurs*. These denote whether the element is optional (*minOccurs* is zero) or can repeat in XML document (*maxOccurs* is larger than one).

Simple types are stored in classes based on abstract class **SimpleType**. They are: *SimpleTypeBasic*, *SimpleTypeList*, *SimpleTypeRestriction* and *SimpleTypeUnion*. Class **SimpleTypeBasic** represents some of the basic types defined in XML schema. These types are enumerated in enum **Type**, which is used also in other packages. Valid values are: *String*, *Boolean*, *Float*, *Double*, *Integer*, *Long*, *Int*, *Short*, *Byte*, *UnsignedLong*, *UnsignedInt*, *UnsignedShort*, *UnsignedByte*, *Base64Binary* and *HexBinary*. The last two are used to store arbitrary binary data, for example images. For information about mapping from XML to C types see the Table 4.2.4 in the Subsection 4.2.4.

Complex types are used for deriving more advanced types by grouping elements and are represented by class **ComplexType**. Each complex type contains list of elements and type of grouping enumerated in enum **ComplexTypeGroupType**, which lists three valid values: *Choice*, *Sequence* and *All*. *Choice* complex type contains exactly one element of listed ones, *Sequence* complex type contains all listed elements in the same order as they are listed and finally *All* complex type contains all listed elements in any random order.

4.1.3 Package grammar

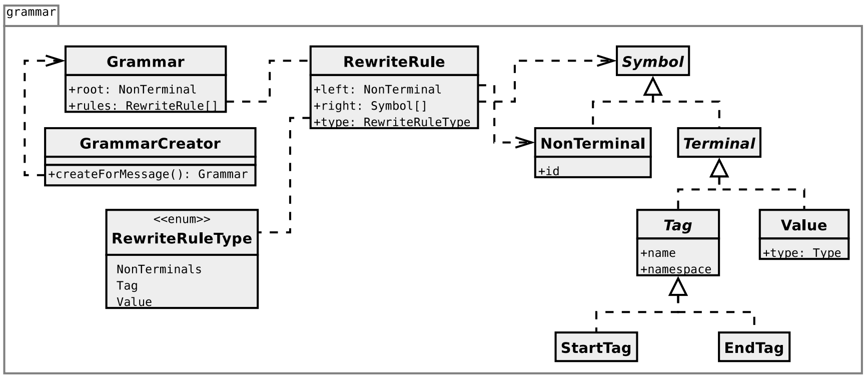

Figure 4.4: UML diagram of package grammar

Package **grammar** and its objects are used when generating grammars of the messages. Main class is the **Grammar** class. It contains one nonterminal (start symbol) and a list of rewrite rules, which are represented by object **RewriteRule**. This class contains one nonterminal (left side of the rule) and the list of symbols (right side of the rule). Each rewrite rule has its type, which are enumerated in enum **RewriteRuleType** and correspond to the types we introduced in the Section 3.1 – in exact order: *NonTerminals*, *Tag* and *Value*.

Abstract class **Symbol** is a base for all symbols used in rewrite rules. One implementation of it is class **NonTerminal** representing grammar nonterminals. These have only one attribute – unique identifier to distinguish them. Its logical counterpart is abstract class **Terminal**, which has two descendants: class **Value** and abstract class **Tag**. **Value** represents simple type values contained in elements. Classes **StartTag** and **EndTag** have class **Tag** as their common ancestor, and represent opening and closing tags of the element.

Special role in this package has class **GrammarCreator**. Messages are XML documents and their schema is available in web service definition, so procedure described in the Section 3.1 can be used to generate grammars for each of the messages.

4.1.4 Package code

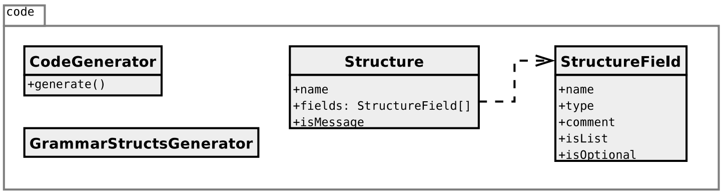

Figure 4.5: UML diagram of package code

Last package in EXCUSA project is package **code**. This contains classes for generating code and output files. Each message and element containing complex type is represented with class **Structure**. Its name correspond to message or complex type name. Class **Structure** contains list of **StructureField** classes. When the structure is message, fields stand for message parts. If the message part is simple type, it is used in structure directly, if not, pointer to structure representing the element is used. Similar situation is with structures of the complex types, with the difference that fields correspond to elements listed in complex type and these fields can be optional or list (message parts can not, they always appear in the count of one).

Field is optional when *minOccurs* attribute of the element is zero. In this case it can contain NULL value (and we have to check it for example when freeing structure). When the field is list (*maxOccurs* of the element is larger than one), it contains pointer to list of structures and one special field named `<variable>_count`. This stores the number of items previously described pointer is pointing to (this number is important for user when iterating through the elements and also when freeing the structure).

Structure field can also reference to simple type (when message part or member of complex type is of simple type) and in this case it holds type information (class **Type** from **schema** package). Each structure can generate code of four functions. Two for serializing to XML and deserializing from XML (functions `construct<Structure>` and `parse<Structure>`) and two for allocating and freeing memory needed by structure (functions `alloc<Structure>` and `free<Structure>`). Examples of structure contents and functions to manipulate them are listed in the Subsection 4.2.3 below.

Class **GrammarStructGenerator** generates representation of grammars in C language (described later in the Subsection 4.2.5) and finally class **CodeGenerator** glues all generators together and creates output files filled with the code provided by them.

4.2 Generated files

The application generates various source files. These are built using `make` utility, which uses provided Makefile and creates some intermediate files. Finally, after the build is done, we end up with client library and server binary. We can divide the files into these categories:

- input files (generated directly by EXCUSA tool)
  `Makefile`, `<service>.h`, `<service>.wsdl`
  `convert.h`, `grammar.h`, `server_methods.h`
  `client.c`, `common.c`, `convert.c`, `grammar.c`, `server.c`
- intermediate files (created by `make` during build)
  `wsdl.c`, object files (`*.o`)
- output (final redistributable) files
  client (`<service>.h`, `libclient.a`) and server (`server`)

In consequent subsections we will describe each of the file closely and show their interesting and nontrivial passages.

4.2.1 `Makefile`

Makefile contains build instructions how to build both client library and server binary. It can be used with GNU make or other compatible utility.

4.2.2 `<service>.wsdl`

WSDL definition file functionally identical to the WSDL file used to generate the service.

4.2.3 `<service>.h` and `common.c`

These two files are the core of the EXCUSA engine. Header file contains definition of structures which are used to represent all messages and elements containing complex types. They are followed by function prototypes for allocating, freeing, constructing and parsing these structures. File ends with declarations of remote methods from WSDL file, so they can be included by both client and server code.

File `common.c` contains mentioned functions to work with structures and automaton parser code. We will show by example how we generate structures and code from XML schema. Suppose we want to have one element with complex content. It contains number identifier, optional string description and unbounded array of points represented by another complex type. XML schema definition could be similar to this one:

```
<s:schema targetNamespace="http://localhost/">
  <s:complexType name="Point">
    <s:sequence>
      <s:element name="posx" type="s:double"/>
      <s:element name="posy" type="s:double"/>
    </s:sequence>
  </s:complexType>
  <s:element name="Test">
    <s:complexType>
      <s:sequence>
        <s:element name="id" type="s:int"/>
        <s:element name="desc" type="s:string" minOccurs="0"
                   maxOccurs="1"/>
        <s:element name="point" type="tns:Point" minOccurs="0"
                   maxOccurs="unbounded"/>
      </s:sequence>
    </s:complexType>
    </s:element>
</s:schema>
```

Generators mentioned in the Subsection 4.1.4 will generate following code for structures for previous schema. Types *Double*, *Int* and *String* are aliases for normal C types. They are created using `typedef` in `convert.h` header file – see following subsection.

```
struct Point {
    Double posx;
    Double posy;
};

struct Test {
    Int id;
    String desc; /* optional */
    struct Point **point; /* min: 0, max: unbounded */
    int pts_count;
};
```

Code for allocating memory for structures are created using macros because of their simplicity and effectivity. Each structure has one macro for allocating single instance and list of instances.

```
#define allocTest() \
        ((struct Test *)malloc(sizeof(struct Test)))
#define allocTestList(N) \
        ((struct Test **)malloc((N)*sizeof(struct Test*)))
#define allocPoint() \
        ((struct Point *)malloc(sizeof(struct Point)))
#define allocPointList(N) \
        ((struct Point **)malloc((N)*sizeof(struct Point*)))
```

Freeing structures is done with functions, because it is slightly more complex, but still pretty straightforward. For lists we have to iterate and free each member and then free the whole list. Other variables are freed in obvious manner.

```
void freePoint(struct Point *ptr)
{
    if (!ptr) return;
    free(ptr);
}
```

```
void freeTest(struct Test *ptr)
{
    int i;
    if (!ptr) return;
    if (ptr->desc) free(ptr->desc);
    for (i=0; i<(ptr->point_count); i++) {
        freePoint(ptr->point[i]);
    }
    free(ptr->pts);
    free(ptr);
}
```

Serializing into XML is done using function `snprintf` from standard C library. We use snprintf variant, because we do not want to overflow the buffer. Functions for serializing simple types (`construct<Type>`) are defined in header file `convert.h` – see next subsection. When we serialize optional elements, we have to check whether the pointer is not NULL. When serializing repeating elements, we can iterate through the list because we know the length of it.

```
int constructPoint(char *buf, int maxlen, struct Point *ptr)
{
    int pos = 0;
    pos += snprintf(buf+pos, maxlen-pos, ”<posx>”);
    pos += constructDouble(buf+pos, maxlen-pos, ptr->posx);
    pos += snprintf(buf+pos, maxlen-pos, ”</posx>”);
    pos += snprintf(buf+pos, maxlen-pos, ”<posy>”);
    pos += constructDouble(buf+pos, maxlen-pos, ptr->posy);
    pos += snprintf(buf+pos, maxlen-pos, ”</posy>”);
    return pos;
}

int constructTest(char *buf, int maxlen, struct Test *ptr)
{
    int i, pos = 0;
    pos += snprintf(buf, maxlen, ”<Test>”);
    pos += snprintf(buf, maxlen-pos, ”<id>”);
    pos += constructInt(buf+pos, maxlen-pos, ptr->id);
    pos += snprintf(buf, maxlen-pos, ”</id>”);
    if (ptr->desc) {
        pos += snprintf(buf, maxlen-pos, ”<desc>”);
        pos += constructString(buf+pos, maxlen-pos, ptr->desc);
        pos += snprintf(buf, maxlen-pos, ”</desc>”);
    }
    for (i=0; i<(ptr->point_count); i++) {
        pos += snprintf(buf, maxlen-pos, "<point>");
        pos += constructPoint(buf, maxlen-pos, ptr->point[i]);
        pos += snprintf(buf, maxlen-pos, "</point>");
    }
    pos += snprintf(buf, maxlen-pos, "</Test>");
    return pos;
}
```

Deserializing or parsing from XML into structures is the most complicated part of the engine. It uses the stack automaton, which function is described in the Section 3.3. The main part is written as one function called `eatRule`. It tries to consume rewriting rule identified by variable `desc`, which corresponds to the nonterminal on the left side of it. The parser iterates over all rewriting rules using macro `forEachRule`. Parser then determines the type of the rule and tries one of the following:

1. to consume all rules on the right side of the rule
2. to consume opening tag, rule between the tags and closing tags
3. to consume value of the type specified on the the right side of the rule

When some of the called functions returns zero, meaning that it encountered some error, calling function returns zero too. If nonzero value was returned, we advance to the position that was returned.

```
#define forEachRule(P,X) \
        for ((P)=((X)&GMASK); \
             (P)<=(((X)&GMASK)+((X)>>GSHIFT)); \
             (P)++)

int eatRule(char level, int pos, GUNIT desc)
{
    GUNIT i, j, r, *rule;
    forEachRule(i, desc) {
        rule = Parser_Grammar[i];
        if (rule[0]) Parser_Push(level, i, pos);
        r = pos;
        switch(rule[0]) {
            case 0:
                j = 1;
                while (rule[j] != -1) {
                    r = eatRule(level, r, rule[j]);
                    j++;
                    if (!r) break;
                }
                break;
            case 1:
                eatWhiteSpace(Parser_Buf, Parser_BufLen, r);
                r = eatStartTag(r, grammar_tag[rule[1]]);
                if (!r) break;
                r = eatRule(level+1, r, rule[2]);
                if (!r) break;
                r = eatEndTag(r, grammar_tag[rule[1]]);
                if (!r) break;
                eatWhiteSpace(Parser_Buf, Parser_BufLen, r);
                break;
            default:
                r = eatValue(r, rule[0]);
                break;
        }
        if (r) {
            break;
        } else {
            if (rule[0]) Parser_Pop();
        }
    }
    return r;
}
```

Described parser core is used in deserializing functions. They assign parser grammar, buffer to be parsed and call function `eatRule()`. If it returns the same length as the length of input buffer (meaning that parsing reached end of message) structures are created and filled with the data. This is done in two steps. Function `eatRule()` fills internal temporary stack with via `Parser_Push()` and `Parser_Pop()`. Nest level, rule identifier and position in XML is stored. Value is removed from stack only in case of bad decision, so the stack is not empty after the operation but contains information how the buffer was processed. Functions `findTag()`, `countTags()` and `getValue()` take an advantage of this. First one searches for given opening tag from some position. Second one counts opening tags of the same type on the given level, this is useful for determining how large structure list should we create. Last function returns pointer to buffer so we can parse individual values of simple types.

```
struct Point *parsePoint(int start) {
  int idx;
  struct Point *ret;
  ret = allocPoint();
  idx = findNextTag(start, tag_posx);
  ret->posx = parseDouble((idx>=0)?getValue(idx+1):NULL);
  idx = findNextTag(start, tag_posy);
  ret->posy = parseDouble((idx>=0)?getValue(idx+1):NULL);
  return ret;
}

struct Test *parseTest(char *buf, int len) {
  int idx;
  struct Test *ret;
  int i, cnt;

  Parser_Buf = buf;
  Parser_BufLen = len;
  Parser_Grammar = Test_grammar;
  Parser_StackCount = 0;
  if (eatRule(0, 0, 0) != len) return NULL;

  ret = allocTest();
  idx = findTagSame(0, tag_Test);
  if (idx >= 0) {
    idx = findNextTag(idx, tag_id);
    ret->id = parseInt((idx>=0)?getValue(idx+1):NULL);

    cnt = countTags(idx, tag_point);
    ret->pts_count = cnt;
    if (cnt > 0) {
      ret->pts = allocPointList(cnt);
      for (i=0; i<cnt; i++) {
        idx = findNextTagSame(idx, tag_point);
        ret->pts[i] = parsePoint(idx);
      }
    }
    idx = findNextTag(idx, tag_desc);
    ret->desc = parseString((idx>=0)?getValue(idx+1):NULL);
  } else {
    return NULL;
  }
}
```

4.2.4 `convert.h` and `convert.c`

Header file `convert.h` contains mapping from XML schema types to C types (using `typedef`, see the Table 4.2.4).

XML type	**C type**	**XML type**	**C type**
String	`char *`	*Boolean*	`char`
Base64Binary	`char *`	*HexBinary*	`char *`
Float	`float`	*Double*	`double`
Byte	`char`	*UnsignedByte*	`unsigned char`
Short	`short`	*UnsignedShort*	`unsigned short`
Int	`int`	*UnsignedInt*	`unsigned int`
Long	`long`	*UnsignedLong*	`unsigned long`
Integer	alias for Long		

Table 4.1: XML schema data types with mapping to C types used in EXCUSA

This header file also exposes functions from source file `convert.c`. They are used to parse basic simple types from XML and to serialize them back. Usually functions from the standard C library like `atoi` or `snprintf`) are used. All conversions are done inside static buffers given as parameters to these functions thus no unreasonable allocating or freeing of memory is done. We will list some examples to give idea how the parsing is done. Some of the functions are implemented as macros because of their simplicity.

```
Boolean parseBoolean(char *val) {
        if (!val) return 0;
        if (!strncmp(val, "true", 4)) return 1;
        if (val[0] == '1') return 1;
        return 0;
}

#define parseString(val) \
          ((val) ? (String)val : (String)NULL);

#define parseDouble(val) \
          ((val) ? (Double)atof(val) : (Double)0.0)

int constructBoolean(char *ptr, int maxlen, char val) {
        if (maxlen <= 0) return 0;
        *ptr = val ? '1' : '0';
        return 1;
}

#define constructString(ptr, maxlen, val) \
          snprintf((ptr), (maxlen), "%s", (val))

#define constructDouble(ptr, maxlen, val) \
          snprintf((ptr), (maxlen), "%f", (val))
```

4.2.5 `grammar.h` and `grammar.c`

These files contain grammar definitions for parser. Each tag and each type are enumerated, so we can index them with number. Each rewrite rule is represented as an array of numbers. Grammar is a list of these rewrite rules, thus it is stored as `int **`. We have only three types of rewriting rules when representing XML structure (see the Subsection 3.1). Each type is represented in different manner so the numbers have different meanings:

1. `grammar[i] =` $\{0, d_1, d_2, ...d_n, -1\}$`;`
   where d_k are rule descriptors (see below)
2. `grammar[i] =` $\{1, t, d\}$`;`
   where t represents tag (index of tag in tags enum), d represents rule descriptor (see below)
3. `grammar[i] =` $\{100 + t\}$`;`
   where t represents type of value (index of type in types enum)

Rule descriptor d is an integer where lower 24 bits are index i into grammar array. If higher 8 bits are set (let's mark this value c) parser knows it should try not only rewrite rule with index i but all rewrite rules with indexes between i and $i + c$. This is good when we have more rewrite rules with the same nonterminal on the left side. We could represent these values as two numbers, but with this little trick we can squeeze space requirements, which are pretty precious in microprogramming. Let's show representation by a simple example:

In our implementation we do not use actually integers to store grammar information. We use the following macros to be able to change `int` to some other type (when `int` it is too small or too big).

rewrite rule	**data structure**
$A \to$ `<root>` B `</root>`	`grammar[0] =` $\{1, tag\_root, 1\}$`;`
$B \to CD$	`grammar[1] =` $\{0, 2, 3 + 1 \ll 24, -1\}$`;`
$C \to$ `<int>` E `</int>`	`grammar[2] =` $\{1, tag\_int, 5\}$`;`
$D \to$ `<str>` F `</str>`	`grammar[3] =` $\{1, tag\_str, 6\}$`;`
$D \to$	`grammar[4] =` $\{0, -1\}$`;`
$E \to \{Int\}$	`grammar[5] =` $\{108\}$`;`
$F \to \{String\}$	`grammar[6] =` $\{100\}$`;`

Table 4.2: Representation of rewrite rules in EXCUSA

```
#define GUNIT int
#define GSHIFT (sizeof(GUNIT)*6)
#define GMASK ((1<<GSHIFT)-1)
```

4.2.6 `client.c`

Source file `client.c` contains client network communication core written using standard POSIX sockets. It holds also all method stubs which create and send requests to server, receive responses from it and parse them. One static buffer is used for request and one for response so we can avoid unnecessary copying.

4.2.7 `server.c`

Core of the web service server is also written using standard POSIX sockets. This file contains server skeleton which listens to requests and calls methods from `server_methods.h` to create responses. Server is meant to be lightweight, so no forking or threading is implemented. That is why we can also use static buffers for requests and responses. This behavior is similar to the one in client.

4.2.8 `server_methods.h`

At start this file contains only empty web service methods. It should be edited by user manually who adds bodies to them and thus provide logic functionality of the web service.

4.2.9 `wsdl.c`

This file consists of only one variable of type `char *` which contains WSDL definition from `<service>.wsdl`. Unneeded white characters are removed to decrease size of object and output files. This value is sent by server to client when it requests definition of the web service.

4.2.10 `libclient.a` and `server`

These are the final output files. Binary `server` is a monolithic web service server. Clients should be created by linking `libclient.a` with user code, which uses remote method prototypes defined in header file `<service>.h`.

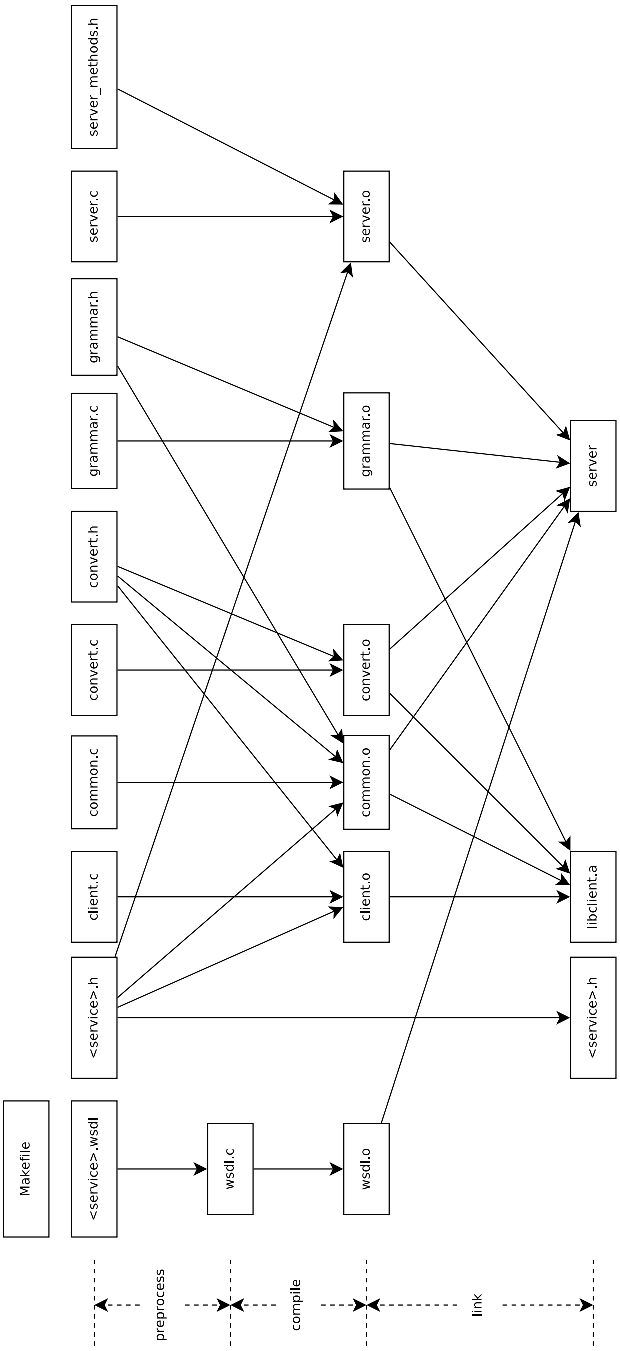

Figure 4.6: Visualized dependencies of the files

Chapter 5

Evaluation

To evaluate our solution we have chosen to benchmark it in real environment and to compare it with other existing implementations:

- **gSOAP**[14] is C or C++ based and momentarily regarded as a "standard" for programming thin solutions
- **bSOAP**[15] is a relatively new C++ based implementation that utilizes *differential serialization* and *deserialization* optimization techniques

5.1 Size

Size of the output is very important if we want to use the proposed solution to program microcontrollers. We used a simple WSDL file with one interface and four operations, generated the source codes for EXCUSA, gSOAP and bSOAP and compiled them on three architectures: **i386**, **avr** and **arm**. The first one was chosen for reference and the other two were picked, because they are used pretty often in microprogramming. We can expect similar sizes on the **m68k** platform like the ones we measured on the **arm**. Source size was calculated by adding sizes of all files that were being compiled – that means all C/C++ files and headers, without `Makefile`s and other helper scripts. Binary size was obtained by counting together the sizes of resulting executables and all non-standard shared libraries, that were created during the build time and are required for running the application. With rising complexity of the web service definition we can also expect the growth of the source and binary sizes. This increase will not be so dramatic in gSOAP and bSOAP case, because they have pretty large runtime. However, in EXCUSA we have to consider also this factor, because grammars representing XML schema together with stubs and skeletons of operations comprise significant fraction of the output. The measured sizes can be seen in the Table 5.1.

		i386		**avr**		**arm**	
	source	client	server	client	server	client	server
EXCUSA	47193	15032	23264	22704	36360	25127	38974
gSOAP	561753	161812	161772	-	-	253215	253158
bSOAP	1363528	326892	326780	-	-	-	-

Table 5.1: Output sizes (in bytes)

As we can see bSOAP and gSOAP have very similar size for client and server, because they link common runtime to both binaries. To compile on AVR architecture we had to remove all POSIX sockets code, so the shown value is the size without it. We hit the 128 KiB limit when compiling gSOAP or bSOAP for AVR and we did not succeed to create bSOAP binaries for ARM either, because of the various issues during linkage.

5.2 Network performance

We wanted to achieve not only good size but also to have fair performance when compared to other alternatives. We benchmarked on loopback of Intel Core Duo 1.83 GHz computer, network was tested when another Athlon XP 2600+ computer was connected to the mentioned one via 100Base-TX Ethernet and Cisco Catalyst 2950T 24 Switch. We tried both small and large packets to see how Ethernet frames fragmentation affects the performance. Request-response latency was measured with Wireshark, but the differences between implementation were not significant enough and the times were very similar to the ones which can be obtained by the `ping` command. The resulting measurements can be seen in the Table 5.2.

We see that bSOAP outperforms the other two competitors when sending large requests (that is because it sends only values that have changed). EXCUSA gets along pretty well especially when dealing with smaller requests.

	loopback		**network**	
packet size	**256**	**64k**	**256**	**64k**
EXCUSA	7692.9	233.0	257.4	28.3
gSOAP	2255.0	174.1	210.4	17.9
bSOAP	8956.4	816.8	922.2	135.7

Table 5.2: Performance (requests per second)

5.3 Profiling

We profiled EXCUSA to detect bottle-necks issues. Test were performed on the same computer clocked at 1.83 GHz using x86 `rdtsc` instruction[16]. This instruction returns a value that represents the count of ticks from processor reset. We can then calculate real duration in microseconds by dividing tick count with 1830. We again chose to test both small and large packets to see how the ratio between communication timeslot and parser timeslot changes. Measured values can be seen in the Table 5.3.

At the beginning client creates request (1). This involves traversing through structures and serializing them into static buffer. Socket is opened and buffer is sent to it (2a). On the other side of connection server opens the sockets and reads data until end of the message is reached (2b). Message is then processed by parser which finds position of variables in XML (3a). These values are deserialized into structures (3b) and the method is executed (4). Response is constructed in buffer in similar fashion like request (5). Server sends response to waiting client (6a) which returns from sleep and reads data (6b). Response is again parsed like request before (7a) and values deserialized into structures (7b).

		256		**64k**	
		ticks	**μsec**	**ticks**	**μsec**
client	1) create request	150 282	82	11 878 856	6491
	2a) send request	1 003 926	549	1 145 969	626
server	2b) receive request	180 795 109	98 795	180 849 878	98 825
	3) parse request	52 998	29	8 177 279	4468
	– 3a) stack automaton	20 427	11	4 594 216	2510
	– 3b) parsing values	11 913	7	3 541 692	1935
	4) execute method	1 100	1	101 783	56
	5) construct response	6 193	3	9 086	5
	6a) send response	35 002	19	49 214	27
client	6b) receive response	182 457 963	99 704	189 216 709	103 397
	7) parse response	65 582	36	77 275	42
	– 7a) stack automaton	16 753	9	23 771	13
	– 7b) parsing values	4 103	2	5 280	3

Table 5.3: Performance (ticks and microseconds)

Chapter 6

Conclusion

6.1 Summary

The primary objective of this work – to design and implement a tool that would allow automated code generation for web services using XML grammars – has been met. This objective can be divided into these partial accomplishments:

- we have devised the formal description of the grammars which represent the XML documents – Section 3.1
- we have invented a system for the automatic creation of these grammars from the XML schema definitions – Section 3.2
- we have found a way how to represent these grammars in procedural and object oriented languages (especially, but not exclusive, C) – Subsection 4.2.5
- we have created the effective parser, which forms pointers to the values in the XML document to avoid inefficient copying – Section 3.3 and Subsection 4.2.3
- we have put together a way how to compose these values into structures that represent messages (thus deserializing the data from the XML) and have also implemented the reverse process (serializing these structures back to the XML) – subsections 4.2.3 and 4.2.4
- we have built up a prototype application for creating web service's code from WSDL file by bringing mentioned parts together – Section 4.1

- we have tested our solution in the conditions similar to the real ones and have compared it to other existing alternatives – sections 5.1, 5.2 and 5.3

6.2 Discussion and future work

Since our application is in the prototype state there are still features that are not implemented or they are implemented partly. Most effort has been put to checking whether the grammar approach is competitive among other alternatives and optimizations. Therefore our compiler lacks complete WSDL support, in particular:

- only WSDL 1.1 is supported, WSDL 2.0 is rather new standard (see Subsection 2.7.4 for differences between 1.1 and 2.0) and even the large frameworks do not support it yet
- complex types in XML schema cannot have attributes (remote procedure calls have usually all information stored in elements, attributes are used primarily in XML databases, where we have to protect the constraints of the data)
- SOAP headers are not supported (we cannot create grammar for headers of unknown structure)
- server is lightweight so it listens for connections only on one port

These problems and optimization of critical parts of the code could be addressed in future, together with brand new ideas like:

- modifying the application to generate code for other languages than C (e.g. C++, Java, C#, Python, Ruby, Perl, PHP, etc.)
- creating module for Apache HTTP Server or any other server that supports extending the core functionality with modules
- trying to combine more optimization methods together, like using grammars together with differential (de)serialization
- investigating the possibilities of coexistence of the grammar approach with XOP[17] and MTOM[18]

Bibliography

[1] Janeček J.: *Efficient SOAP Processing in Embedded Systems*, ecbs, p. 128, 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04), 2004.

[2] Černá I., Křetínský M., Kučera A.: *Automaty a formální jazyky I.*, Elportál, Brno : Masarykova univerzita, 29 November 2006.
`http://is.muni.cz/elportal/?id=703389`

[3] Bray T., Paoli J., Sperberg-McQueen C. M., Maler E., Yergeau F.: *Extensible Markup Language (XML) 1.0*, W3C Recommendation, 16 August 2006.
`http://www.w3.org/TR/2006/REC-xml-20060816/`

[4] Fallside D. C., Walmsley P.: *XML Schema Part 0: Primer*, W3C Recommendation, 28 October 2004.
`http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`

[5] Biron P. V., Malhotra A.: *XML Schema Part 2: Datatypes*, W3C Recommendation, 28 October 2004.
`http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`

[6] Thompson H. S., Beech D., Maloney M., Mendelsohn N.: *XML Schema Part 1: Structures*, W3C Recommendation, 28 October 2004.
`http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`

[7] Mitra N., Lafon Y.: *SOAP Version 1.2 Part 0: Primer*, W3C Recommendation, 27 April 2007.
`http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`

[8] Gudgin M., Hadley M., Mendelsohn N., Moreau J.-J., Nielsen H. F., Karmarkar A., Lafon Y.: *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Recommendation, 27 April 2007.
`http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`

[9] Erl T.: *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*, Prentice Hall 2004.

[10] Chinnici R., Moreau J.-J., Ryman A., Weerawarana S.: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C Recommendation, 26 June 2007.
`http://www.w3.org/TR/2007/REC-wsdl20-20070626/`

[11] Christensen E., Curbera F., Meredith G., Weerawarana S.: *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001.
`http://www.w3.org/TR/2001/NOTE-wsdl-20010315`

[12] Berstel J., Boasson L.: *Formal properties of XML grammars and languages*, Acta Inf. 38(9): 649-671 (2002).

[13] Apache. Web Services Project – Apache AXIS2.
`http://ws.apache.org/axis2/`.

[14] van Engelen R. A., Gallivan K.: *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*, in the proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid, pages 128-135, May 21-24, 2002, Berlin, Germany.
`http://www.cs.fsu.edu/ engelen/soap.html`

[15] Abu-Ghazaleh N., Lewis M. J., Govindaraju M., ”Differential Serialization for Optimized SOAP Performance,” in proceedings of HPDC-13: IEEE International Symposium on High Performance Distributed Computing, Honolulu, Hawaii, pp. 55-64, June 2004.
`http://www.cs.binghamton.edu/ nayef/bsoap/`

[16] *Intel® 64 and IA-32 Architectures Software Developer's Manual* – Volume 3B: System Programming Guide, Part 2, Chapter 18
`http://download.intel.com/design/processor/manuals/253669.pdf`

[17] Gudgin M., Mendelsohn N., Nottingham M., Ruellan H.: *XML-binary Optimized Packaging*, W3C Recommendation, 25 January 2005.
`http://www.w3.org/TR/2005/REC-xop10-20050125/`

[18] Gudgin M., Mendelsohn N., Nottingham M., Ruellan H.: *SOAP Message Transmission Optimization Mechanism*, W3C Recommendation, 25 January 2005.
`http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/`

Appendix A

CD contents

`/readme.txt`	... readme file
`/excusa/`	... EXCUSA tool distribution
`lib/`	... Java archives needed to run the application
`src/`	... sources written in Java
`excusa`	... launcher script for UNIX systems
`excusa.bat`	... launcher script for Microsoft Windows
`/excusa.pdf`	... thesis in Portable Document Format
`/excusa.ps`	... thesis in Postscript format
`/javadoc/`	... EXCUSA source documentation generated in Javadoc
`/tex/`	... TeX sources of the thesis (+ Dia and Inkscape images)
`/wsdl/`	... example WSDL files