Georg-August-Universität Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2007-14

Masterarbeit

im Studiengang „Angewandte Informatik“

Web Service Test Framework with TTCN-3

Stefan Troschütz

am Institut für

Informatik

Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

15. Juni 2007

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 15. Juni 2007

Master's Thesis

Web Service Test Framework with TTCN-3

Stefan Troschütz

June 15, 2007

Software Engineering for Distributed Systems Group

Institute for Informatics

Georg-August-University Göttingen

Abstract

Web services are standards-based software systems designed to facilitate interoperable application-to-application integration over a network. The broadening adoption of Web services and especially their use for business purposes or critical applications introduce a growing need for efficient testing approaches that allow assuring the correctness and interoperability of Web services.

This thesis presents a framework for the testing of Web services with the standardized test specification and implementation language TTCN-3. Foremost, the mapping of a Web service description to a TTCN-3 abstract test suite, which facilitates basic testing of the Web service, is discussed in detail. A time-saving automation of the proposed mapping implemented as a Java console application is introduced afterwards. Finally, the enhancement of TTworkbench Basic, a TTCN-3 test development and execution environment, for Web service testing is presented. The implemented extension enables the execution of a TTCN-3 abstract test suite derived from a Web service description. In addition, it provides dialog-based wizards for using the automation of the mapping from within TTworkbench or defining new, more complex test cases.

Acknowledgements

First and foremost, I would like to thank Edith Werner for her kind support during all phases of creating this master's thesis. Without her knowledge, valuable suggestions, and other efforts, this thesis would never have been written.

Special thanks are dedicated to Robert and my brother Michael who spent a great deal of their precious time for proof-reading. I am really thankful for their comments and constructive criticism.

Finally, I want to thank my family and all of my friends for accompanying my journey through life. It is their supporting love and friendship that have made me who I am and have brought me this far.

Contents

List of Tables

List of Figures

Abbreviations and Acronyms

ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ATS	Abstract Test Suite
CD	Coding and Decoding
CDATA	Character Data
DTD	Document Type Definition
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
GFT	Graphical Presentation Format
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
J2SE	Java Platform, Standard Edition
JAR	Java Archive
MLF	Module Loader File
MSDN	Microsoft Developer Network
OSI	Open Systems Interconnection
PA	Platform Adapter
QName	Qualified Name
RPC	Remote Procedure Call
SA	SUT Adapter
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
SUT	System Under Test
TCI	TTCN-3 Control Interface
TE	TTCN-3 Executable
TFT	Tabular Presentation Format
TRI	TTCN-3 Runtime Interface

TTCN	Tree and Tabular Combined Notation
TTCN-2	Tree and Tabular Combined Notation version 2
TTCN-3	Testing and Test Control Notation version 3
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
UTF-8	8-bit Unicode Transformation Format
W3C	World Wide Web Consortium
WSDD	Web Service Deployment Descriptor
WSDL	Web Services Description Language
WSDL4J	Web Services Description Language for Java Toolkit
WS-I	Web Services Interoperability Organization
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSD	XML Schema Definition

1 Introduction

Web services are a standards-based, service-oriented approach to distributed computing that enables application-to-application integration over a network, Inter- or intranet. The maturity of the Web service core standards, the increasing need of automating cross-enterprise business processes over the Internet, and the mature tool support from major middleware as well as smaller, Web service-specific vendors are leading to a broad adoption of Web services. Whereas they are already widely used in corporate intranet environments or on a prototype scale, Web services are currently making their way into commercial use too and a growing number of companies use them with their customers, suppliers, and business partners. However, the broadening adoption of Web services and especially their usage for business purposes or critical applications introduce a growing need for efficient testing approaches that allow assuring correctness and interoperability of Web services.

In [23], [24], and [46], a specification-based, automated testing approach is developed that uses the standardized test specification and test implementation language TTCN-3 (Testing and Test Control Notation version 3). From a Web service description given by means of a WSDL (Web Services Description Language) document, an abstract test specification expressed with TTCN-3 is derived that is independent of the test platform and the concrete system to be tested. In conjunction with the standardized TTCN-3 notation, which eases understanding, documentation, communication, and discussion, this improves the transparency of the test process, increases the objectiveness of the tests, and makes test results comparable. Overall, the approach is considered superior to other, proprietary test solutions. Although there are case studies that prove the concept of the specification-based, automated Web service testing approach, it lacks a thorough specification of the mapping between WSDL and TTCN-3.

1.1 Scope of this Thesis

Based upon the existing research work, this thesis presents a framework for Web service testing with TTCN-3 that includes the following contributions:

- The detailed specification of a mapping between a Web service description given by means of a WSDL document and a TTCN-3 abstract test suite (ATS) that facilitates basic testing of the Web service.

- The implementation of the WSDL2TTCN utility, a command line application written in the Java programming language, which automates the specified mapping between WSDL and TTCN-3.
- The extension of TTworkbench Basic ([28]), a TTCN-3 test development and test execution environment, for Web service testing. This includes the implementation of:
  - A test adapter and codec that enable the execution of TTCN-3 abstract test suites derived from WSDL descriptions.
  - Two wizards providing dialog-based usage of the WSDL2TTCN utility. Whereas one simply maps a WSDL document to a TTCN-3 abstract test suite, the second wizard previously creates a TTCN-3 project that is configured for the testing of Web services (using the implemented test adapter and codec).
  - A third wizard that provides dialog-based definition of more complex test cases based on TTCN-3 abstract test suites created according to the specified mapping between WSDL and TTCN-3.

1.2 Structure of this Thesis

The content of this thesis is structured as follows: After this introduction, the basics of this thesis are given in Chapter 2. This includes a presentation of the Extensible Markup Language (XML), the concept of Web services and their key technologies, and the basic concepts and constructs of TTCN-3. Furthermore, an overview of the related research work is given.

Subsequently, Chapter 3 specifies the mapping between a Web service description given by means of a WSDL document and a TTCN-3 abstract test suite that facilitates basic testing of the Web service. After an introduction of the chosen TTCN-3 communication paradigm, the overall mapping rules, and a number of supportive TTCN-3 definitions, the mapping rules for the major WSDL elements are presented in detail.

An automated mapping of a Web service description to a TTCN-3 abstract test suite is provided by the WSDL2TTCN utility that is introduced in Chapter 4. In addition to the overall architecture of the Java console application, two implementation aspects are discussed in detail and the invocation and the available command line options of the WSDL2TTCN utility are presented. Furthermore, an option is introduced that facilitates changing the proposed mapping between WSDL and TTCN-3 in some points in order to create TTCN-3 output fully compatible to TTworkbench Basic.

In Chapter 5, the extension of the TTCN-3 test development and execution environment TTworkbench Basic for Web service testing is discussed. This includes an introduction of the test adapter and codec, which enable the execution of TTCN-3 abstract test suites derived from WSDL descriptions. In addition, three wizards are presented that provide dialog-based means to either use the WSDL2TTCN utility from within TTworkbench or define new, more complex test cases based upon TTCN-3 abstract test suites generated according to the mapping specified in Chapter 3.

The content of this thesis is concluded by Chapter 6, which gives an overall summary as well as an outlook to further work.

2 Foundations

This chapter provides the basics that are used in the subsequent chapters of this thesis. First, Section 2.1 introduces the Extensible Markup Language, which is at the heart of all Web service standards and techniques. Thereafter, Section 2.2 presents the concept of Web services as well as their core standards SOAP (originally Simple Object Access Protocol, since version 1.2 only SOAP) and WSDL. In Section 2.3, the basic concepts and constructs of the Testing and Test Control Notation version 3 are discussed. At last, Section 2.4 gives an overview of the related work.

2.1 XML Primer

The Extensible Markup Language is a meta-language that provides text-based means to structure and store information. XML defines a syntactic foundation that facilitates the definition of arbitrary, structural equivalent markup languages that can be used in many areas of application such as XHTML (Extensible Hypertext Markup Language) for the creation of web pages or WSDL for the description of Web services.

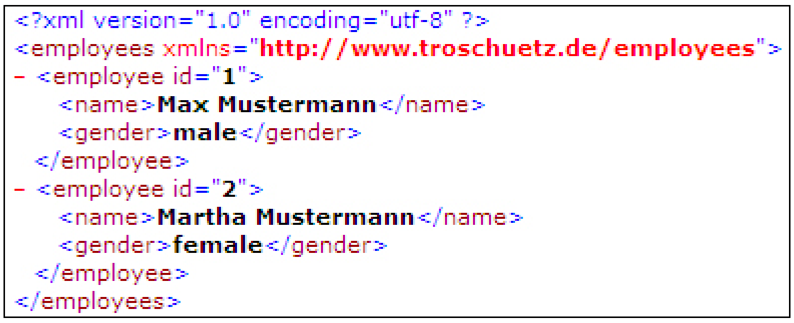
The XML standard is defined by the World Wide Web Consortium (W3C) and there are two current versions. The first, XML 1.0, became a W3C Recommendation in February 1998. It has undergone minor changes since then and is currently in its fourth edition ([30]), as published in August 2006. XML 1.0 is widely implemented and recommended for general use. The second version, XML 1.1, was first published in February 2004 and is currently in its second edition, as published in August 2006. It contains new features that are intended to ease the use of XML in certain scenarios. At the time of this writing, XML 1.1 is not very widely implemented and therefore its use is only recommended, if its features are necessarily needed.

An XML document is a hierarchical tree-structure that is composed by properly nesting the following types of nodes:

- Elements are the most important building block of an XML document. They are represented by a matching pair of start and end tags (`<element></element>`) or an empty tag (`<element />`). Elements may have attributes appearing either on the start tag or on the empty tag. In addition, they may have content enclosed by the matching start and end tags. The element content is an arbitrary sequence of text nodes and/or child elements.

- Attributes provide additional information on elements and they are the only carrier of information besides text nodes. Attributes are represented by a name-value pair in the start or the empty tag of elements (`<element attributeName="attributeValue" />`). The names of attributes are required to be unique per element.
- Text nodes are the most important carrier of information in an XML document. They may appear as normal text or in form of a CDATA (Character Data) section.
- Processing instructions and comments

On its top level, an XML document must have exactly one element, the so-called root or document element, which directly or indirectly contains all other elements, attributes, and/or text nodes. In front of the root element there may appear an optional line, the XML declaration, which states what version of XML is in use and contains information about the used character encoding. An XML document that conforms to all syntactical rules of the XML standard is called well-formed and assured to be at least readable by any XML-aware software.



```
<?xml version="1.0" encoding="utf-8" ?>
<employees xmlns="http://www.troschuetz.de/employees">
- <employee id="1">
    <name>Max Mustermann</name>
    <gender>male</gender>
  </employee>
- <employee id="2">
    <name>Martha Mustermann</name>
    <gender>female</gender>
  </employee>
</employees>
```

Figure 2.1: XML document

Figure 2.1 shows an exemplary XML document that is well-formed and provides basic information about two employees. The XML declaration states that the document uses version 1.0 of the XML standard and characters are encoded in UTF-8 (8-bit Unicode Transformation Format). The document element named *employees* contains two nested *employee* elements. Each of those contains further elements providing the employee's name and gender as well as an *id* attribute that holds the employee's personnel number.

2.1.1 XML Namespaces

An important amendment to the XML standard is the concept of XML namespaces. It enables the qualification of element and attribute names used in XML documents by associating them with namespaces identified by URI (Uniform Resource Identifier) references. XML namespaces are a W3C Recommendation that was first published in January 1999 and is currently in its second edition ([31]), as published in August 2006.

The qualification of element and attribute names gains in importance when an XML document should not only be read but also interpreted. A software system must be able to recognize the elements and attributes, which it is designed to process, even in the face of "collisions" i.e. the occurrence of markup using the same element or attribute names but intended for some other software. For example, an XML document could contain a *table* element being the root for an ordered arrangement of data in rows and columns and a second *table* element, which is the root for the description of a piece of furniture. Such a XML document is well-formed, but it would be very hard for a software system to interpret and process the XML document correctly, unless the two *table* elements are associated with differing namespaces.

Since URI references identifying XML namespaces are often inconveniently long and can contain characters not allowed in element and attribute names, they are not directly incorporated. Instead, the name of an element or attribute appears as a qualified name (QName) i.e. the name is either unprefixed or appended to a prefix and a delimiting colon (s. [31] ch. 4). An attribute-based declaration syntax is provided to bind prefixes to URIs identifying XML namespaces and/or to bind a default namespace that applies to unprefixed element names. The latter binding technique is used in the XML document shown in Figure 2.1 to associate all elements with a namespace identified by the URI "http://www.troschuetz.de/employees". The first technique is exemplified by the XML document displayed in Figure 2.2. The same namespace is now bound to the prefix *em* that is used to qualify all element names and thereby associate them with the namespace explicitly. Both types of namespace declaration are scoped by the element on which they appear, so different bindings may apply in different parts of an XML document.

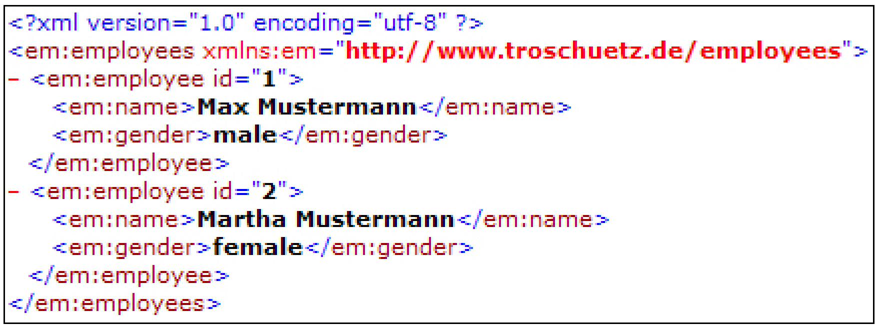

```
<?xml version="1.0" encoding="utf-8" ?>
<em:employees xmlns:em="http://www.troschuetz.de/employees">
- <em:employee id="1">
    <em:name>Max Mustermann</em:name>
    <em:gender>male</em:gender>
  </em:employee>
- <em:employee id="2">
    <em:name>Martha Mustermann</em:name>
    <em:gender>female</em:gender>
  </em:employee>
</em:employees>
```

Figure 2.2: XML document with explicit namespace qualification

2.1.2 XML Schema

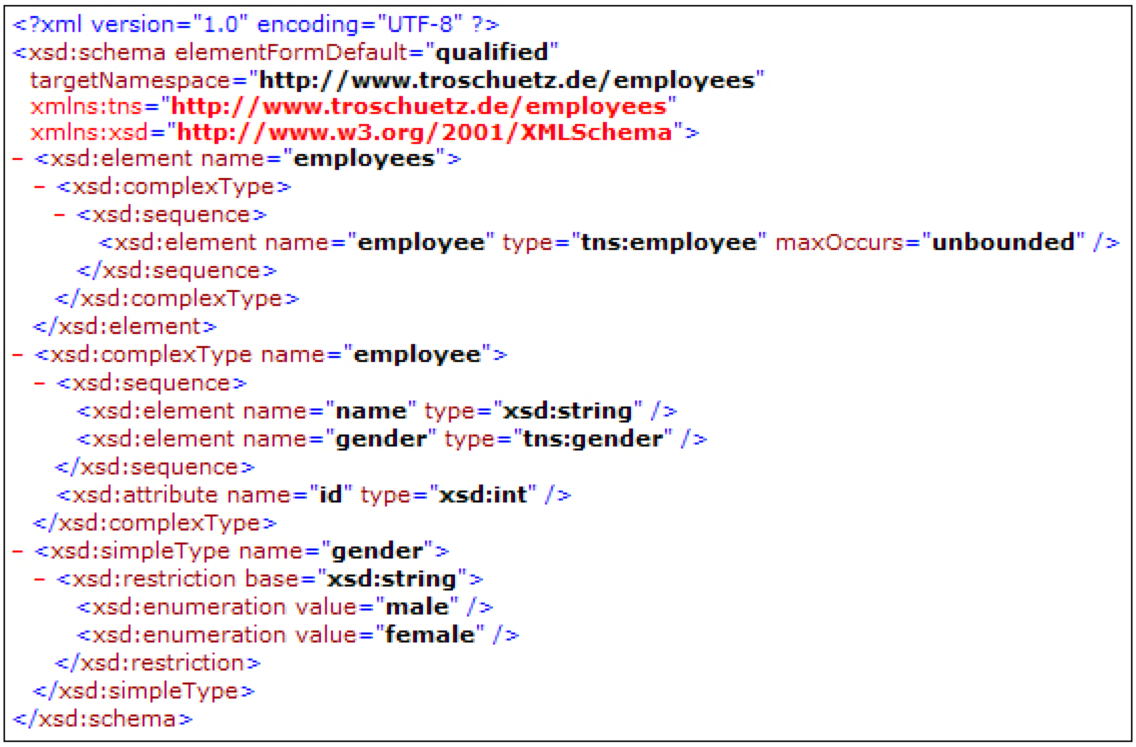
As initially mentioned, the XML standard provides a syntactic foundation that enables the definition of arbitrary, structural equivalent markup languages. In order to describe a specific markup language or type of XML document, the basic XML syntax rules must be supplemented with a schema that restricts the set of allowed element and attribute names as well as the valid document structures i.e. how the nodes are allowed to be nested. A schema is therein equivalent to the grammar for a language as it defines a "vocabulary" and the structure of valid "sentences". In case an XML document is well-formed and additionally conforms to every constraint specified by a given schema, the document is valid according to that schema.

XML Schema is one of the most widely used languages that facilitate defining schemas for XML documents. It is specified by the W3C in a multi-part standard that was first published in May 2001 and is currently in its second edition ([38], [39], and [40]), as published in October 2004. Henceforth, this thesis refers to the second edition without further notice or reference by using the term XML Schema Standard for the standard as a whole or XML Schema Standard Part 0, 1, or 2 for a specific part, respectively.

The XML Schema language is itself XML-based, which makes it legible for any XML-aware software and allows authoring a schema and the conforming documents with a single tool. Besides the description of the allowed vocabulary and structures for a type of document, XML Schema adds the notion of data types to XML, which by default treats all information as text. The provided type system is very rich, making the XML Schema language powerful and complex at the same time. Furthermore, XML Schema is aware of XML namespaces and each schema can be viewed as a collection of type definitions and element declarations whose names belong to a specific namespace, the so-called target namespace.

The XML Schema type system differentiates two fundamental sorts of types. On the one hand, simple types describe and restrict character-based content such as attribute values and text nodes. The XML Schema Standard provides a large set of built-in simple types such as *string*, *integer*, or *date*. Those can be used either directly or as base type for a custom simple type. In order to obtain a new simple type, its base is restricted along a number of constraining facets that for example allow restricting the number of digits in a numeric value or listing all valid values. The built-in simple types and the mechanism for defining custom types are standardized in Part 2 of the XML Schema Standard. On the other hand, complex types describe the valid document structures by defining the sequencing, multiplicity, and content type of (child) elements as well as the appearance and content type of attributes. There are numerous ways of defining complex types, which are specified in detail in the XML Schema Standard Part1.

To conclude this section, Figure 2.3 shows an exemplary XML Schema document also called XML Schema Definition (XSD). The displayed schema describes a type of XML document to which the documents given in Figure 2.1 and Figure 2.2 conform i.e. both documents are valid according to the schema. At the beginning, the schema document describes the *employees* element, which is of an anonymous complex type that allows the element to contain a sequence of an unlimited number of *employee* elements. Those child elements are of the complex type named *employee* that allows them to contain a sequence of one *name* and one *gender* element as well as one *id* attribute. Since both child elements as well as the attribute should contain character-based content, they are of simple types. The *name* element and the *id* attribute are of the XML Schema built-in data types *string* or *int*, respectively. The *gender* element on the other hand is of the custom simple type *gender* that restricts the built-in data type *string* to the values "`male`" and "`female`".



```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://www.troschuetz.de/employees"
  xmlns:tns="http://www.troschuetz.de/employees"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- <xsd:element name="employees">
  - <xsd:complexType>
    - <xsd:sequence>
        <xsd:element name="employee" type="tns:employee" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
- <xsd:complexType name="employee">
  - <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="gender" type="tns:gender" />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:int" />
  </xsd:complexType>
- <xsd:simpleType name="gender">
  - <xsd:restriction base="xsd:string">
      <xsd:enumeration value="male" />
      <xsd:enumeration value="female" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 2.3: XML Schema document

2.2 Web Services

Web services are a standards-based, service-oriented approach to distributed computing that enables application-to-application integration over a network, Inter- or intranet. The development of Web services was especially driven by the growing need of automating cross-enterprise business processes over the Internet for which conventional middleware is not suitable (s. [2] ch. 5.1.3). The term Web services was introduced in 2000, but at that time there was no universally adopted definition what it means and many software vendors had their own initiatives and products. In the following, the concept of Web services refined through the process of open standards development and in November 2002, the W3C published a first working draft describing the Web services architecture and defining a Web service as follows:

> "A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols." ([36] ch. 2)

The above definition captures the essence of Web services very well without specifying the standards that should be used for their description and discovery or the interaction with them. Whilst this ambiguity was intended and no standards should be presupposed, the further development of Web services proved WSDL, UDDI (Universal Description, Discovery and Integration), and SOAP to be the leading standards for the respective tasks.

SOAP is a lightweight protocol for exchanging XML-based messages over a network and the leading standard for the interaction with Web services. The SOAP protocol is discussed in detail in Section 2.2.2. WSDL is an XML-based language that provides means to describe the interface and location of a Web service as well as how to access it. The Web Service Description Language is presented in Section 2.2.3. UDDI is an XML-based registry for businesses and the services provided by them. It can be queried by SOAP messages and it provides access to the WSDL documents of the listed Web services. Because the publication and discovery of Web services are not relevant in the context of this thesis, the Universal Description, Discovery and Integration standard is not discussed in detail.

The basic Web services architecture and the application of the three core standards are illustrated in Figure 2.4 whereat emphasis is placed on the underlying service-oriented architecture. A service provider, the Web service, and a service requestor, any software system potentially another Web service, interact on the basis of the service description, which is published by the provider and discovered by the requester through a service broker. Because all involved entities communicate by means of well-defined standards and the requester and provider need no or only little shared understanding besides the service description, the architecture allows software systems to be loosely coupled and integrated independent of their platform or programming language.

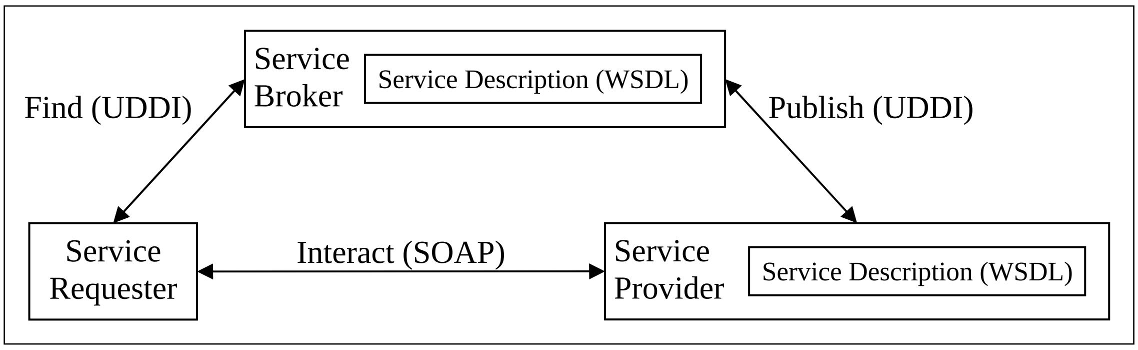


Figure 2.4: Web services architecture

Besides the three core technologies SOAP, WSDL, and UDDI, there are a number of supplementing standards that add notions of security, transactions, etc. to Web services. Furthermore, an open industry organization, the WS-I (Web Services Interoperability Organization), was founded to promote Web services interoperability across platforms, operating systems, and programming languages. For that purpose, the WS-I publishes profiles, which provide guidelines for how related Web services standards should be used together for best interoperability, as well as sample applications and test tools. The WS-I Basic Profile provides interoperability guidelines for the three core technologies WSDL, SOAP, and UDDI. It is currently available in version 1.0 ([44]), as published in April 2004, and 1.1 ([45]), as published in April 2006.

2.2.1 Exemplary "Movie Database" Web Service

This section introduces a simple Web service that simulates access to a movie database. It is used in the Sections 2.2.2 and 2.2.3 to exemplify the Web services core standards SOAP and WSDL and throughout Chapter 3 to illustrate how a WSDL description maps to TTCN-3. Furthermore, the exemplary "Movie Database" Web service was used to put the software developed in the context of this thesis to a first test. Its WSDL description was mapped to a TTCN-3 abstract test suite using the WSDL2TTCN utility presented in Chapter 4. Afterwards the generated test suite was executed against the exemplary Web service using TTworkbench Basic ([28]), which was enhanced for Web service testing with the custom test adapter and codec introduced in Section 5.1.

The "Movie Database" Web service is implemented with two distinct frameworks that are widely used for that purpose. On the one hand, it is written in the Java programming language and exposed as a Web service by using Apache Axis ([1]) version 1.4. On the other hand, the Web service is developed with the .NET Framework ([21]) 2.0 and the C# programming language. Figure 2.5 visualizes the structure of both implementations and their sources are listed in Appendix A. The .NET implementation is furthermore publicly available at the URI "http://www.troschuetz.de/MovieDatabase.asmx".

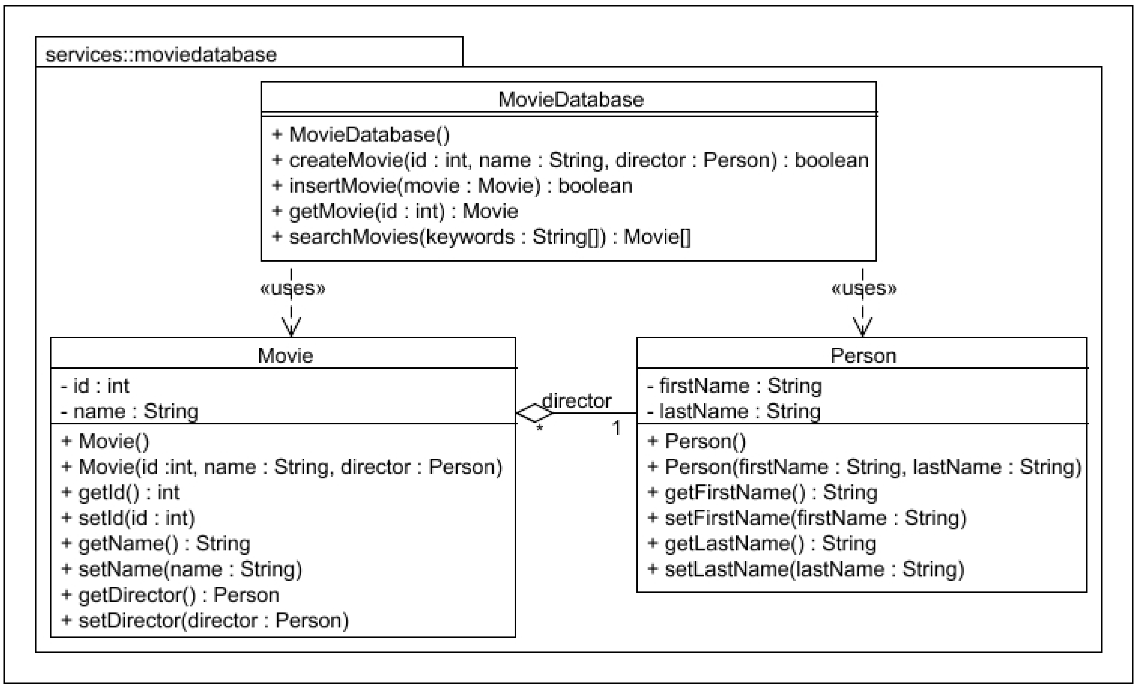

services::moviedatabase
MovieDatabase
+ MovieDatabase()
+ createMovie(id : int, name : String, director : Person) : boolean
+ insertMovie(movie : Movie) : boolean
+ getMovie(id : int) : Movie
+ searchMovies(keywords : String[]) : Movie[]
«uses»
«uses»
Movie
- id : int
- name : String
+ Movie()
+ Movie(id :int, name : String, director : Person)
+ getId() : int
+ setId(id : int)
+ getName() : String
+ setName(name : String)
+ getDirector() : Person
+ setDirector(director : Person)
director
*
1
Person
- firstName : String
- lastName : String
+ Person()
+ Person(firstName : String, lastName : String)
+ getFirstName() : String
+ setFirstName(firstName : String)
+ getLastName() : String
+ setLastName(lastName : String)

Figure 2.5: Class diagram of the "Movie Database" Web service

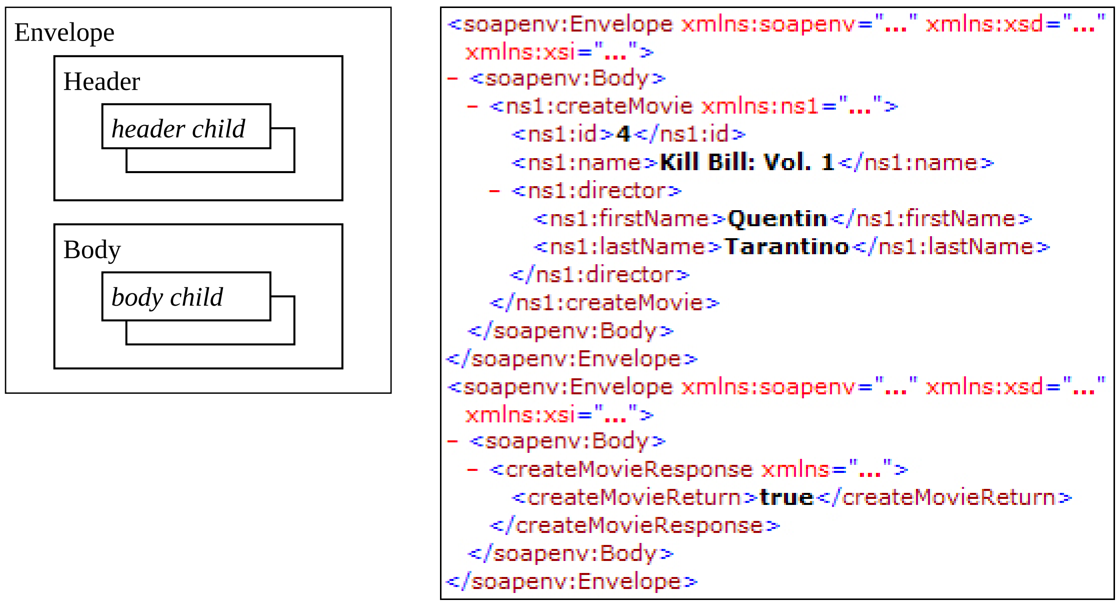
The interface of the exemplary Web service is implemented by the *MovieDatabase* class and it comprises operations for the creation, insertion, and retrieval of movies. Although the interface is kept quite simple, the exposed operations are designed to cover a wide range of possible signatures. First, the *createMovie* operation takes multiple parameters. Second, the *createMovie* and the *insertMovie* operations take complex arguments i.e. objects of the *Movie* and *Person* classes and the *getMovie* operation returns one. Finally, the *searchMovies* operation covers the use of arrays of both simple and complex types.

2.2.2 SOAP

SOAP is a lightweight protocol for exchanging XML-based messages in a distributed, decentralized environment and constitutes the leading standard for the interaction with Web services. It consists of four parts:

- A messaging framework that describes what is in a message, who should deal with it, and whether it is optional or mandatory.
- A set of encoding rules that specify a serialization mechanism for the expression of instances of application-defined data types.
- A convention that can be used to represent Remote Procedure Calls (RPC).
- A protocol binding framework that enables the exchange of SOAP messages using a variety of underlying protocols. A binding of SOAP to HTTP (Hypertext Transfer Protocol) is included.

The name SOAP was coined in early 1998 by Microsoft, DevelopMentor, and Userland that were working on XML-based distributed computing. After its further development was slowed down by Microsoft politics, SOAP was submitted to the World Wide Web Consortium for standardization. In May 2000, SOAP 1.1 ([32]) was published as a W3C Note and thenceforward it formed the foundation for the further standards work. In June 2003, the SOAP 1.2 protocol became a W3C Recommendation, which is published as a multi-part deliverable ([33], [34], and [35]).



```
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <ns1:createMovie xmlns:ns1="...">
      <ns1:id>4</ns1:id>
      <ns1:name>Kill Bill: Vol. 1</ns1:name>
    - <ns1:director>
        <ns1:firstName>Quentin</ns1:firstName>
        <ns1:lastName>Tarantino</ns1:lastName>
      </ns1:director>
    </ns1:createMovie>
  </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <createMovieResponse xmlns="...">
      <createMovieReturn>true</createMovieReturn>
    </createMovieResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.6: SOAP message format

Figure 2.6 shows an abstraction of the SOAP message format along with two exemplary messages that were exchanged during a call to the *createMovie* operation of the .NET implementation of the "Movie Database" Web service. To aid display, all namespace declarations occurring in the exemplary messages have been shortened. The document element of each SOAP message is named *Envelope* and it encloses an optional *Header* and a mandatory *Body* element. Both *Body* and *Header* are allowed to contain arbitrary XML, making the message format of the SOAP protocol both simple and flexible.

The SOAP protocol assumes that every message has a sender, an ultimate receiver, and an arbitrary number of so-called intermediaries that process the message and route it to the receiver. The *Body* element carries the core information of a SOAP message and its content must be processed by the ultimate receiver. The *Header* element is used to add functionality or additional information to a SOAP message without modifying the core of the message. For example, the *Header* could contain a session identifier or security credentials. SOAP provides a flexible role model that allows each child element of the *Header* to define for which of the intermediaries and/or ultimate receiver it is intended.

2.2.3 WSDL

The Web Services Description Language is the leading standard for the description of Web services and provides an XML-based syntax to specify the exposed interface and the location of a Web service as well as how to access it.

WSDL was initially created by Ariba, IBM, and Microsoft. The three vendors merged their proprietary description languages and submitted the result to the World Wide Web Consortium for standardization. In March 2001, WSDL 1.1 ([37]) was published as a W3C Note and since then forms the basis for the current standards work. At the time of this writing, several working drafts have been published of WSDL 2.0 that is expected to become a W3C Recommendation. Since WSDL 2.0 is currently not widely supported or used, this thesis considers only WSDL 1.1 and henceforth refers to it without further notice or reference by using the term WSDL 1.1 Standard.

The overall structure of WSDL documents is illustrated in Figure 2.7 and can be split up into two parts. The abstract part, which consists of the *types*, *message*, and *portType* elements, describes the interface of the Web service i.e. the exposed operations and used data types. The concrete part comprising the *service* and *binding* elements describes a concrete implementation of the service's interface at a network endpoint i.e. where the Web service is located and how its operations can be accessed. All WSDL elements depicted in Figure 2.7 are named using an attribute-based syntax and belong to a certain namespace, the target namespace declared on the root element of the WSDL document. This allows the WSDL elements to be referenced by their qualified name within their defining document or any other WSDL description that imports the defining document.

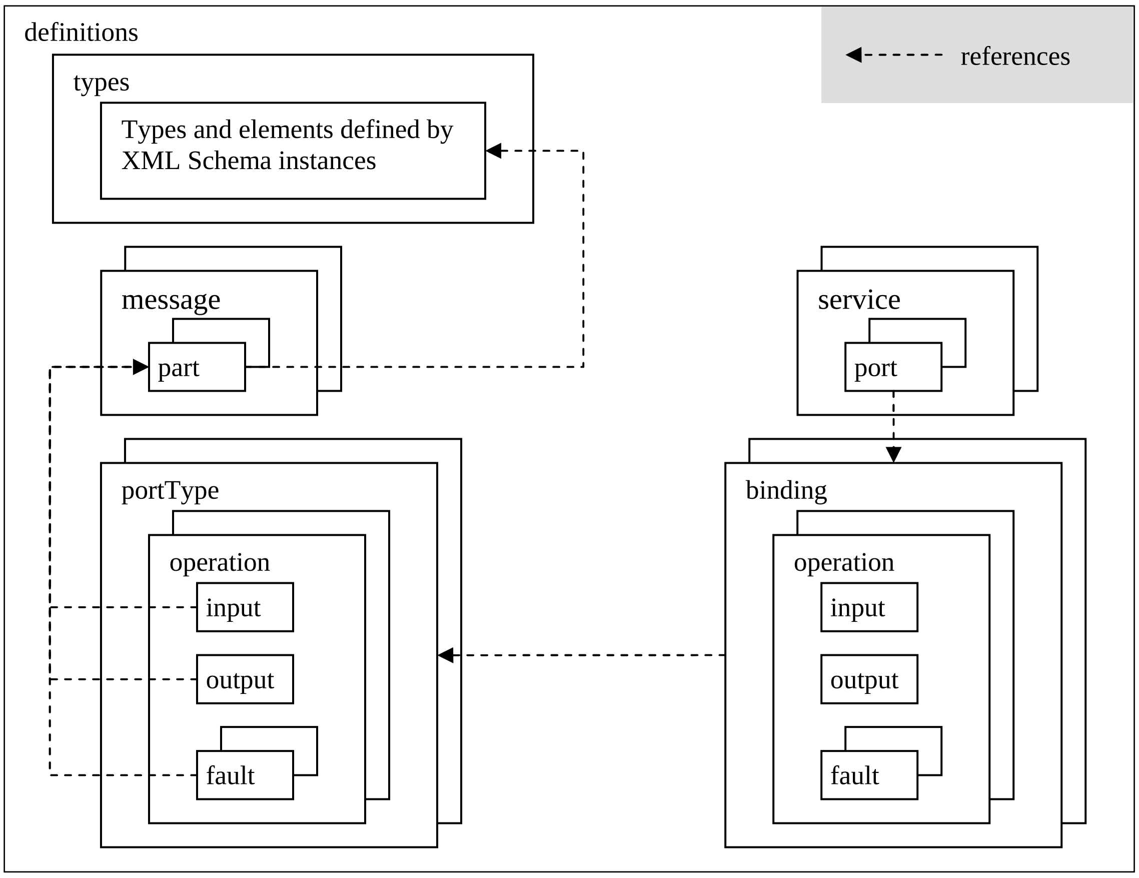


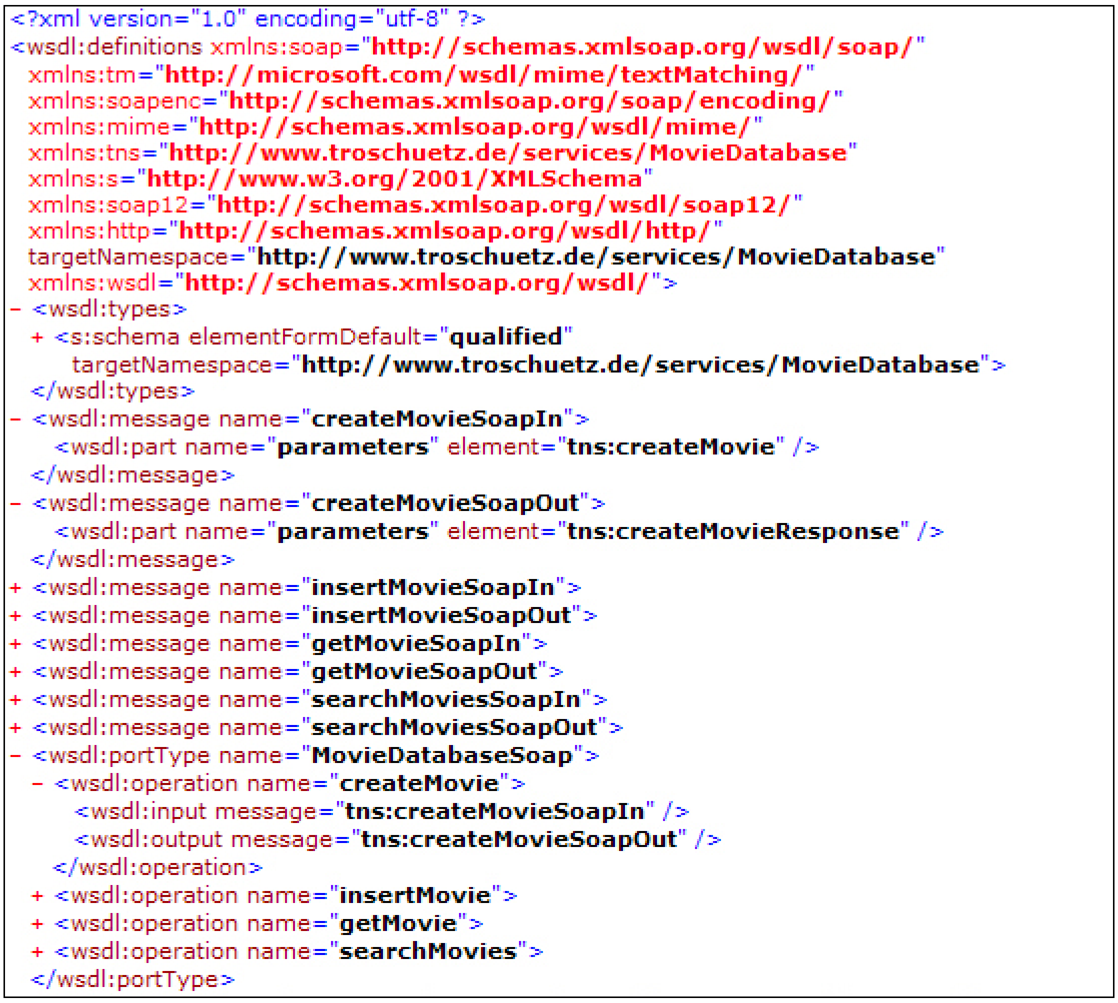
Figure 2.7: Abstraction of the WSDL document structure

Figure 2.8 shows the abstract part of the WSDL description that is generated by the .NET implementation of the "Movie Database" Web service. In order to shorten the document and aid display, some elements are collapsed, but their content is comparable to this of expanded elements of the same type.

The *types* element describes all data structures, which will be exchanged as parts of messages. The default type system in WSDL is XML Schema, but other type systems can be added via extensibility elements. The subsequent *message* elements describe the messages, which will be exchanged between applications, as sets of parts. Each *part* is characterized by a name and by a type that in turn is described in the used type system.

All messages of the .NET implementation of the "Movie Database" Web service are composed of a single part named *parameters* whose type is defined in the XML Schema language.

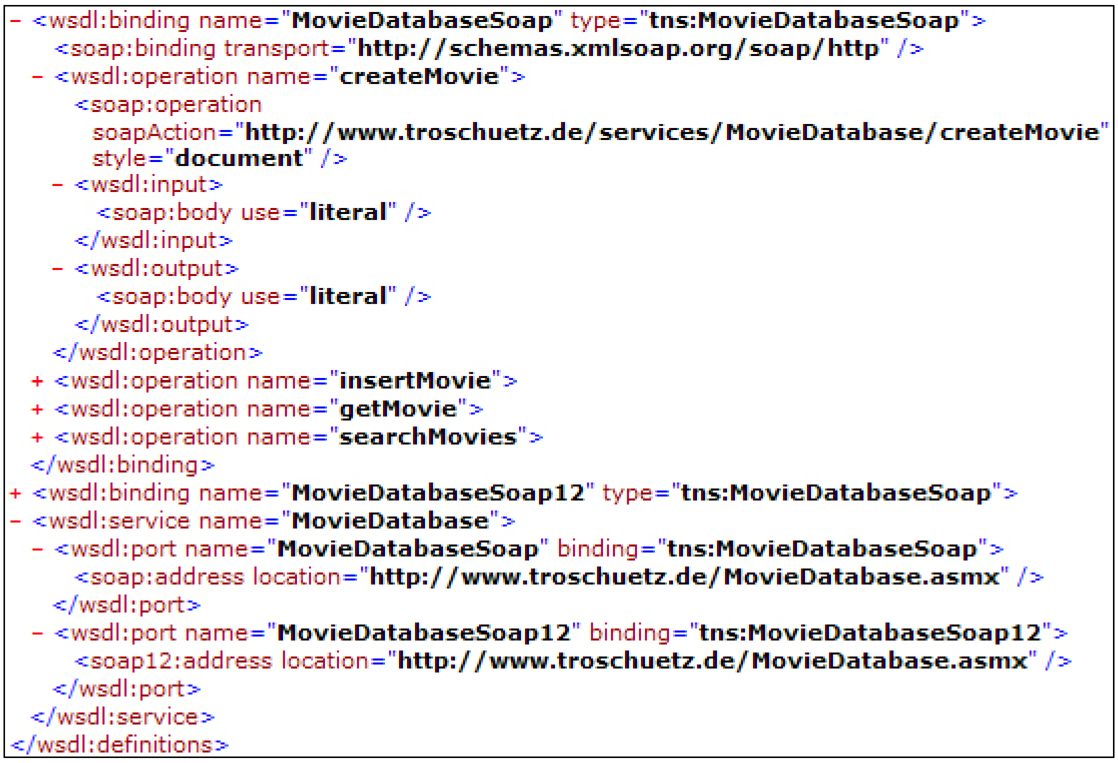
The *portType* element finally describes the Web service interface as a set of operations with every *operation* being a collection of previously defined input, output, and/or fault messages. The WSDL 1.1 Standard defines four operation types: request-response, one-way, solicit-response, and notification. The two latter operation types, which require the Web service to initiate communication, are not well defined in the WSDL 1.1 Standard and there is only little support for them. Because of this, the WS-I Basic Profile Version 1.0 and 1.1 forbid the usage of the solicit-response and the notification operation types (s. [44] ch. 5.4.2 and [45] ch. 4.5.2). The four operations of the .NET implementation of the "Movie Database" Web service make use of the request-response operation type.



```
<?xml version="1.0" encoding="utf-8" ?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://www.troschuetz.de/services/MovieDatabase"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  targetNamespace="http://www.troschuetz.de/services/MovieDatabase"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
- <wsdl:types>
  + <s:schema elementFormDefault="qualified"
      targetNamespace="http://www.troschuetz.de/services/MovieDatabase">
  </wsdl:types>
- <wsdl:message name="createMovieSoapIn">
    <wsdl:part name="parameters" element="tns:createMovie" />
  </wsdl:message>
- <wsdl:message name="createMovieSoapOut">
    <wsdl:part name="parameters" element="tns:createMovieResponse" />
  </wsdl:message>
+ <wsdl:message name="insertMovieSoapIn">
+ <wsdl:message name="insertMovieSoapOut">
+ <wsdl:message name="getMovieSoapIn">
+ <wsdl:message name="getMovieSoapOut">
+ <wsdl:message name="searchMoviesSoapIn">
+ <wsdl:message name="searchMoviesSoapOut">
- <wsdl:portType name="MovieDatabaseSoap">
  - <wsdl:operation name="createMovie">
      <wsdl:input message="tns:createMovieSoapIn" />
      <wsdl:output message="tns:createMovieSoapOut" />
    </wsdl:operation>
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:portType>
```

Figure 2.8: WSDL document of the "Movie Database" Web service – abstract part

Figure 2.9 shows the concrete part of the WSDL description that is generated by the .NET implementation of the exemplary "Movie Database" Web service. Again, some elements are collapsed to shorten the document and aid display.



```
- <wsdl:binding name="MovieDatabaseSoap" type="tns:MovieDatabaseSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  - <wsdl:operation name="createMovie">
      <soap:operation
        soapAction="http://www.troschuetz.de/services/MovieDatabase/createMovie"
        style="document" />
    - <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    - <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:binding>
+ <wsdl:binding name="MovieDatabaseSoap12" type="tns:MovieDatabaseSoap">
- <wsdl:service name="MovieDatabase">
  - <wsdl:port name="MovieDatabaseSoap" binding="tns:MovieDatabaseSoap">
      <soap:address location="http://www.troschuetz.de/MovieDatabase.asmx" />
    </wsdl:port>
  - <wsdl:port name="MovieDatabaseSoap12" binding="tns:MovieDatabaseSoap12">
      <soap12:address location="http://www.troschuetz.de/MovieDatabase.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure 2.9: WSDL document of the "Movie Database" Web service – concrete part

Each *binding* element concretizes a specific port type by defining the message format and the protocol bindings for all of its operations and messages. The Web Services Description Language defines a common binding mechanism that allows bindings to any number of messaging and transport protocols via extensibility elements. The WSDL 1.1 Standard defines a binding extension for SOAP 1.1 (s. [37] ch. 3), which is the most widely used binding and the only one supported by the WS-I Basic Profile Version 1.0 and 1.1 (s. [44] ch. 5.6.1 and [45] ch. 4.6.1). A binding extension for the latest version of the SOAP protocol is specified in the "WSDL 1.1 Binding Extension for SOAP 1.2" ([41]), a W3C Member Submission published in April 2006. WSDL can be bound to the SOAP protocol using one of five message styles that are presented in detail in Section 2.2.4. The .NET implementation of the "Movie Database" Web service contains two *binding* elements that both use a refinement of the document/literal style, the wrapped message style, and bind the service's interface to SOAP 1.1 on the one hand and SOAP 1.2 on the other hand.

The *service* element finally groups a set of related *port* elements. The latter describe single endpoints hosting a Web service interface by associating a specific binding with a network address. The bindings of the .NET implementation of the "Movie Database" Web service are associated with the same network address by two ports that are grouped by the *MovieDatabase* service.

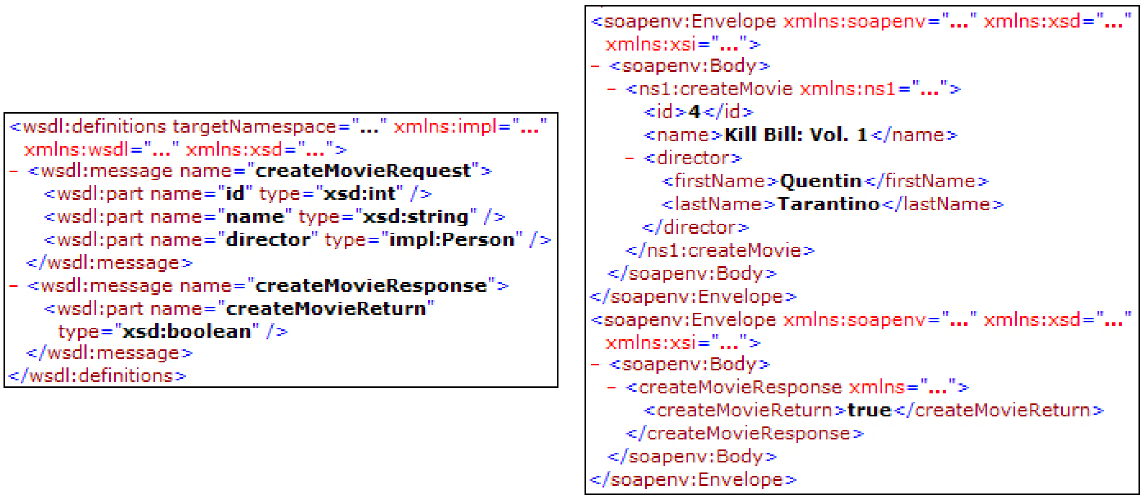
2.2.4 Message Styles

The abstract interface description of a WSDL document can be bound to SOAP using one of five message styles. Those differ in the construction of the *Body* and the *Header* elements of SOAP messages.

The SOAP binding extensions introduced in Section 2.2.3 provide two attributes for adjusting the message style. On the one hand, the *style* attribute indicates whether an operation is document-oriented i.e. the messages contain documents, or RPC-oriented i.e. the messages contain parameters and return values. The *use* attribute on the other hand indicates whether the message parts are encoded using the SOAP encoding rules or whether the parts define the concrete schema of the message. The combination of the attribute values gives four of the five message styles: rpc/encoded, document/encoded, rpc/literal, and document/literal. The fifth message style called wrapped represents a special application of the document/literal style.

Because "it was determined that there was no way to guarantee interoperability when using an RPC-encoded binding" ([14] p. 628), the WS-I Basic Profile Version 1.0 and 1.1 require a WSDL description to use either the rpc/literal or the document/literal message style (s. [44] ch. 5.6.3 and [45] ch. 4.7.3). Besides this general restriction, the WS-I Basic Profile imposes further, specific requirements for the supported message styles. Following the WS-I Basic Profile, only three message styles are supported in the context of this thesis: the ones mentioned above as well as the wrapped style being a special application of the document/literal style.

The rpc/literal message style uses the SOAP convention for the representation of remote procedure calls and responses i.e. the *Body* element of SOAP messages contains a single child element that encloses all message parts. Inside the request message, this wrapper element is named identically to the operation, and inside the response, its name is the operation name with "Response" being appended. The message parts are represented by elements whose names are identical to the part's names and whose types are defined by the WSDL type system, by default XML Schema. An example of the rpc/literal style is given in Figure 2.10. The left hand side partly shows the WSDL description of the Axis implementation of the "Movie Database" Web service using the rpc/literal message style. Concretely, it shows the description of the request and response messages of the *createMovie* operation. The right hand side of Figure 2.10 displays the corresponding SOAP messages exchanged during a call to the *createMovie* operation. To aid display, all occurring namespace declarations have been shortened.

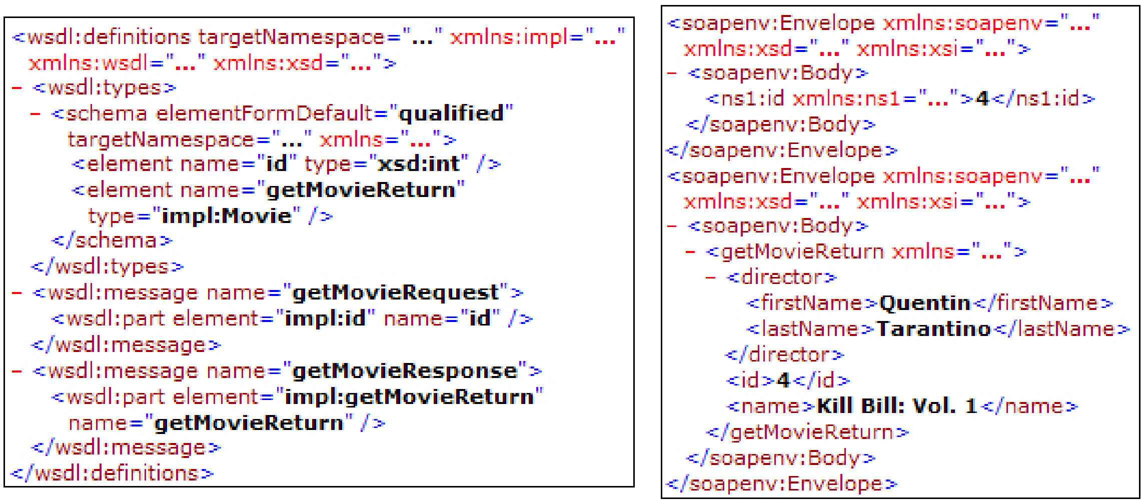


```
<wsdl:definitions targetNamespace="..." xmlns:impl="..."
  xmlns:wsdl="..." xmlns:xsd="...">
- <wsdl:message name="createMovieRequest">
    <wsdl:part name="id" type="xsd:int" />
    <wsdl:part name="name" type="xsd:string" />
    <wsdl:part name="director" type="impl:Person" />
  </wsdl:message>
- <wsdl:message name="createMovieResponse">
    <wsdl:part name="createMovieReturn"
      type="xsd:boolean" />
  </wsdl:message>
</wsdl:definitions>
```

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <ns1:createMovie xmlns:ns1="...">
      <id>4</id>
      <name>Kill Bill: Vol. 1</name>
    - <director>
        <firstName>Quentin</firstName>
        <lastName>Tarantino</lastName>
      </director>
    </ns1:createMovie>
  </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <createMovieResponse xmlns="...">
      <createMovieReturn>true</createMovieReturn>
    </createMovieResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.10: Rpc/literal message style

In case the document/literal message style is used, the SOAP *Body* element contains no additional wrapper elements. Instead, the message parts appear directly under the *Body* element, so that everything within the body is described by the WSDL type system. If the default type system XML Schema is used, a SOAP message can be validated using any XML validator. A drawback of the document/literal style is the non-inclusion of the operation name in the SOAP message, which could make it difficult or even impossible to associate a message with an operation. Furthermore, the WS-I Basic Profile Version 1.0 and 1.1 allow only one child i.e. only one message part in the SOAP body (s. [44] ch. 5.3.1 and [45] ch. 4.4.1). An example of the document/literal style is given in Figure 2.11. The left hand side partly shows the WSDL description of the Axis implementation of the "Movie Database" Web service using the document/literal style. Concretely, it shows the description of the request and response messages of the *getMovie* operation. This operation replaces the *createMovie* operation used by the example of the rpc/literal message style, since the latter requires more than one message part and therefore cannot be exposed by the Axis implementation using the document/literal style. The right hand side of Figure 2.11 displays the corresponding SOAP messages exchanged during a call to the *getMovie* operation. In the shown SOAP response, the identifier of the only child of the SOAP body was changed to "getMovieReturn", because it was wrongly named "idReturn" in the message actually returned by the Axis implementation. Similar to the previous example, all namespace declarations have been shortened to aid display.

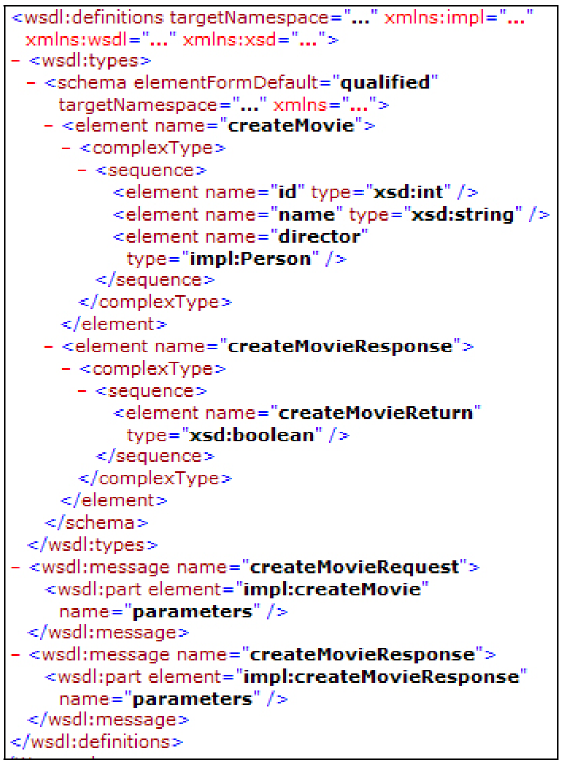


```
<wsdl:definitions targetNamespace="..." xmlns:impl="..."
 xmlns:wsdl="..." xmlns:xsd="...">
- <wsdl:types>
  - <schema elementFormDefault="qualified"
      targetNamespace="..." xmlns="...">
      <element name="id" type="xsd:int" />
      <element name="getMovieReturn"
        type="impl:Movie" />
    </schema>
  </wsdl:types>
- <wsdl:message name="getMovieRequest">
    <wsdl:part element="impl:id" name="id" />
  </wsdl:message>
- <wsdl:message name="getMovieResponse">
    <wsdl:part element="impl:getMovieReturn"
      name="getMovieReturn" />
  </wsdl:message>
</wsdl:definitions>
```

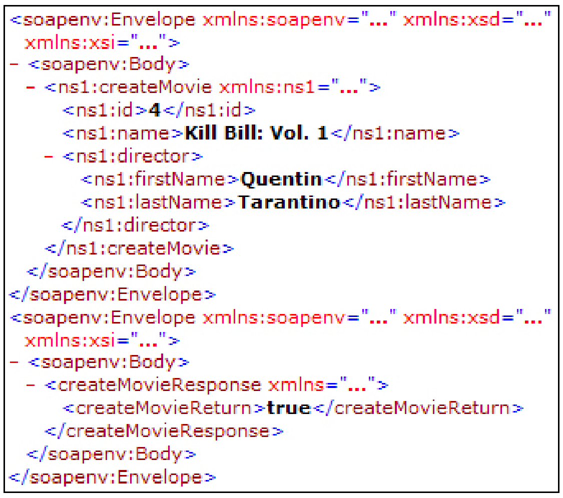
```
<soapenv:Envelope xmlns:soapenv="..."
 xmlns:xsd="..." xmlns:xsi="...">
- <soapenv:Body>
    <ns1:id xmlns:ns1="...">4</ns1:id>
  </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="..."
 xmlns:xsd="..." xmlns:xsi="...">
- <soapenv:Body>
  - <getMovieReturn xmlns="...">
    - <director>
        <firstName>Quentin</firstName>
        <lastName>Tarantino</lastName>
      </director>
      <id>4</id>
      <name>Kill Bill: Vol. 1</name>
    </getMovieReturn>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.11: Document/literal message style

The wrapped message style, which is a refinement of the document/literal style, resolves the drawbacks of its parent by using a wrapper element. Similar to the rpc/literal style the wrapper is named identically to the operation and it encloses all message parts. However, the wrapper now also reflects in the WSDL document, as it is fully described by the WSDL type system and becomes the only message part. On account of this, a SOAP message can still be validated using any XML validator, if the WSDL description uses the default type system XML Schema. Drawbacks of the wrapped message style are its proprietary origin from Microsoft and the non-existence of a specification for this style. The wrapped message style is nevertheless considered interoperable, as it is well understood and supported by many Web services implementations besides the Microsoft products. An example for the wrapped message style is given in Figure 2.12. The left hand side partly shows the WSDL document of the .NET implementation of the "Movie Database" Web service. Concretely, it shows the description of the request and response messages of the *createMovie* operation. The right hand side of Figure 2.12 displays the corresponding SOAP messages exchanged during a call to the *createMovie* operation. Again all occurring namespace declarations have been shortened.



```
<wsdl:definitions targetNamespace="..." xmlns:impl="..."
  xmlns:wsdl="..." xmlns:xsd="...">
- <wsdl:types>
  - <schema elementFormDefault="qualified"
      targetNamespace="..." xmlns="...">
    - <element name="createMovie">
      - <complexType>
        - <sequence>
            <element name="id" type="xsd:int" />
            <element name="name" type="xsd:string" />
            <element name="director"
              type="impl:Person" />
          </sequence>
        </complexType>
      </element>
    - <element name="createMovieResponse">
      - <complexType>
        - <sequence>
            <element name="createMovieReturn"
              type="xsd:boolean" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
- <wsdl:message name="createMovieRequest">
    <wsdl:part element="impl:createMovie"
      name="parameters" />
  </wsdl:message>
- <wsdl:message name="createMovieResponse">
    <wsdl:part element="impl:createMovieResponse"
      name="parameters" />
  </wsdl:message>
</wsdl:definitions>
```



```
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <ns1:createMovie xmlns:ns1="...">
      <ns1:id>4</ns1:id>
      <ns1:name>Kill Bill: Vol. 1</ns1:name>
    - <ns1:director>
        <ns1:firstName>Quentin</ns1:firstName>
        <ns1:lastName>Tarantino</ns1:lastName>
      </ns1:director>
    </ns1:createMovie>
  </soapenv:Body>
</soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="..." xmlns:xsd="..."
  xmlns:xsi="...">
- <soapenv:Body>
  - <createMovieResponse xmlns="...">
      <createMovieReturn>true</createMovieReturn>
    </createMovieResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.12: Wrapped message style

2.3 TTCN-3

The Testing and Test Control Notation version 3 is a standardized test specification and implementation language dedicated to black-box testing of a wide range of computer and telecommunication systems. Typical application areas are the testing of protocols, services, modules, and APIs (Application Programming interface).

TTCN-3 is a redesign of the Tree and Tabular Combined Notation (TTCN), which was developed by ISO (International Organization for Standardization) since 1984 as part of the overall methodology for conformance testing of the Open Systems Interconnection (OSI) protocol layers. TTCN was first standardized in 1992 and revised in 1998 with the publication of TTCN version 2 (TTCN-2). Though TTCN was successfully used for conformance testing of a wide range of communication protocols, it was not adequate for various kinds of testing such as interoperability, robustness, regression, system, and integration testing or for various emerging testing application areas such as mobile protocol, Internet protocol, service, module, and API testing. Therefore, a more flexible and powerful test specification language was called for and in 1998, ETSI (European Telecommunication Standards Institute) started the development of an improved version of TTCN, the Testing and Test Control Notation version 3.

After two years of development, ETSI published the first version of TTCN-3 in October 2000. Since then TTCN-3 is maintained through a well-defined change request process handled by ETSI and it is currently available in version 3.2.1, as published in February 2007. Because the latest version was published in the middle of this writing, this thesis refers to the prior version 3.1.1 that was released in June 2005. TTCN-3 is standardized in a multi-part deliverable that comprises ten parts. Whereas the parts 1 to 7 are already included in version 3.1.1 of TTCN-3, the remaining parts are still in the process of standardization and only available as drafts. Of interest to this thesis are the parts 1, 5, and 6 ([6], [7], and [8]) as well as part 9, which was published as finalized draft ([9]) in October 2006. Henceforth, this thesis refers to the parts included in TTCN-3 version 3.1.1 and part 9 without further notice or reference by using the term TTCN-3 Standard Part 1-7 or 9, respectively.

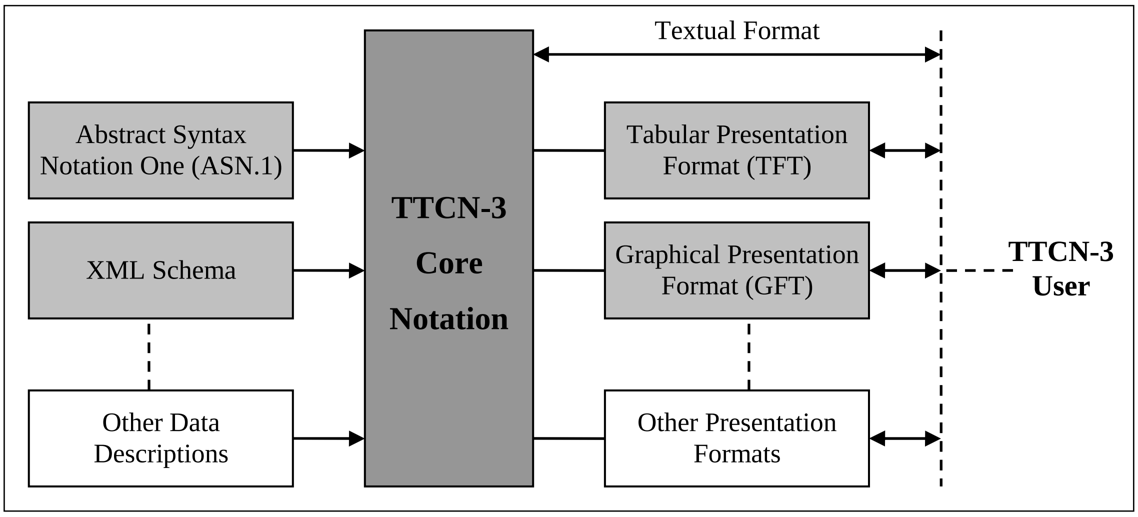

Textual Format
Abstract Syntax Notation One (ASN.1)
XML Schema
Other Data Descriptions
TTCN-3 Core Notation
Tabular Presentation Format (TFT)
Graphical Presentation Format (GFT)
Other Presentation Formats
TTCN-3 User

Figure 2.13: Overall picture of TTCN-3 (cp. [13] Figure 1)

At the heart of TTCN-3 is the textual core notation, also called core language, which is specified in the TTCN-3 Standard Part 1. The core notation is similar to conventional programming languages like Java or C++ but provides additional features dedicated to testing such as built-in data matching, test verdicts, timer handling, or the concurrent execution of test components. As illustrated in Figure 2.13, the TTCN-3 core notation can be represented using other formats than the textual one. The depicted presentation formats TFT and GFT are specified in the TTCN-3 Standard Part 2 and 3. Furthermore, the core notation provides interfaces to various data description languages to use the data described by these languages besides its own data model. The use of the depicted languages ASN.1 and XML Schema is defined in the TTCN-3 Standard Part 7 and 9.

In the following Section 2.3.1, the TTCN-3 core language and its key constructs are explained along the simple example of testing the login function of a fictitious service. In doing so, the necessary definitions of types, test data, and test behavior are presented. Subsequently, Section 2.3.2 introduces the TTCN-3 test system, a conceptual collection of entities that manage test execution and interpret or execute compiled TTCN-3 code.

2.3.1 TTCN-3 Core Language

TTCN-3 code written in the TTCN-3 core language is collected in modules. The module for the login example is shown in Figure 2.14 and Figure 2.15. Every TTCN-3 module can be split up into two parts. On the one hand, the module definitions part (Figure 2.14 and Figure 2.15 lines 2-52) comprises the definitions of data (types, templates, and constants), test behavior (test cases, functions, and altsteps), and import statements that reference the definitions of external modules. On the other hand, the optional module control part (Figure 2.15 lines 54-56) controls and sequences the execution of test cases. It is therein similar to the main function of conventional programming languages.

The module definitions part can be further structured using groups. In TTCN-3, groups do not form separate scopes i.e. the identifiers of definitions must be unique across the whole module, but the utilization of groups aids readability and adds logical structure. In addition, groups can be used for more selective importing of definitions. The module for the login example contains two groups that separate the type and data definitions (Figure 2.14 lines 2-26) from the definition of test behavior (Figure 2.15 lines 28-52).

The TTCN-3 core notation provides a rich data model that can be used to define types of test data. Similar to conventional programming languages, there are predefined basic types such as *boolean*, *integer*, *float*, and various string types. Moreover, TTCN-3 has some specific types. The *verdicttype*, an enumeration of the distinguished values "`pass`", "`fail`", "`inconc`", "`none`", and "`error`", represents the possible outcomes of a test case called test verdicts. The *anytype* on the other hand is defined as shorthand for the union of all types known inside a TTCN-3 module. On the basis of the basic types, a powerful subtyping mechanism can be used to define new types. Depending on the base type, the new subtype can restrict its parent to a specific length, value range, character pattern, or list of values. Line 8 of Figure 2.14 defines the type for response messages of the login function. The *LoginResponse* type is defined as a name alias of the built-in *boolean* type i.e. it is a subtype without restrictions.

From the built-in types and already defined subtypes, structured types such as records, sets, or unions can be constructed. A record type is an ordered sequence of named and typed fields. Lines 3-6 of Figure 2.14 define the record type for requests messages send to the login function. The *LoginRequest* type is composed of the *user* and the *password* field that are both of the built-in *charstring* type. A set type is similar to a record except that the sequencing of the fields is not significant. Whereas records and sets specify structures where all fields are always present, a union type defines a collection of named and typed fields, called variants, of which only one will ever be present in a value. The record-of and set-of types specify an ordered or unordered collection of instances of a single type.

```
01 module LoginExample {
02    group TypeAndDataDefinition {
03       type record LoginRequest {
04          charstring user,
05          charstring password
06       }
07
08       type boolean LoginResponse;
09
10       template LoginRequest a_LoginRequest := {
11          user := "Max Mustermann",
12          password := "1234"
13       }
14
15       template LoginResponse a_LoginResponse := true;
16
17       type port LoginPort message {
18          out LoginRequest;
19          in LoginResponse;
20       }
21
22       type component LoginTestComponent {
23          port LoginPort pt_login;
24          timer t_timeoutGuard;
25       }
26    }
27
```

Figure 2.14: TTCN-3 module for the login example – type and data definition

With the request and response message types being defined, Lines 10-15 of Figure 2.14 specify actual values of them, the so-called templates. During test execution, templates are going to be used to either transmit distinct values or test whether received values are contained in the set of expected messages. A template declaration is required to define a specific value or a matching expression for each field of its type i.e. it is fully specified. Furthermore, templates can be parameterized and provide a simple form of inheritance. The matching expressions allow specifying ranges or lists of values and thus provide a powerful mechanism to describe sets of expected messages. Both templates for the login example define messages with specific values.

Before the test behavior can be defined, there are two special types associated with test configurations that need to be specified preliminarily. On the one hand, the port type defines an endpoint at which communication takes place. Ports can be message-based for communication via message exchange and/or procedure-based for communication via remote procedure calls. A port type declaration contains one or more lists indicating the allowed (message) types and/or procedures in conjunction with the permitted communication direction. The port type for the login example (Figure 2.14 lines 17-20) allows sending messages of the *LoginRequest* type and receiving messages of the *LoginResponse* type. On the other hand, the component type is used to describe test components on which the test behavior will be executed. A component type declaration defines the ports, timers, constants, and variables owned by a test component of that type. The component type for the login example (Figure 2.14 lines 22-25) owns an instance of the *LoginPort* port type for communication with the service whose login function should be tested. Furthermore, the component type owns the *t_timeoutGuard* timer that will be used to secure test execution from inactivity of the service under test.

```
28    group BehaviorDefinition {
29       testcase tc_login() runs on LoginTestComponent {
30          pt_login.send(a_LoginRequest);
31
32          t_timeoutGuard.start(5.0);
33
34          alt {
35             [] pt_login.receive(a_LoginResponse) {
36                t_timeoutGuard.stop;
37                setverdict(pass);
38             }
39             [] alt_ReceiveAnyOrTimeout();
40          }
41       }
42
43       altstep alt_ReceiveAnyOrTimeout() runs on LoginTestComponent {
44          [] pt_login.receive {
45             t_timeoutGuard.stop;
46             setverdict(fail);
47          }
48          [] t_timeoutGuard.timeout {
49             setverdict(fail);
50          }
51       }
52    }
53
54    control {
55       execute(tc_login());
56    }
57 }
```

Figure 2.15: TTCN-3 module for the login example – behavior definition

With all necessary types and templates being described, finally the test behavior can be specified by test cases, functions, and altsteps. Functions and altsteps are used to specify and structure test behavior, to define default behavior, and to structure computation in a module. Test cases are a special kind of functions that always return a value of the type *verdicttype* and that can be executed in the module control part.

Lines 29-41 of Figure 2.15 define the test case for the login example named *tc_login*. The test case refers to the *LoginTestComponent* component type on which the described behavior will be executed. This reference is defined in the test case header (Figure 2.15 line 29) with the "`runs on`" clause. The header also declares a possibly empty list of parameters. The test case body (Figure 2.15 lines 30-40) specifies how to stimulate the system under test (SUT) and the expected reactions to the stimulation. The test behavior is described in sequential order and the TTCN-3 core language provides basic control structures known from conventional programming languages such as if-else-blocks or for-loops.

At first, the *tc_login* test case sends the request message template via the *pt_login* port owned by the test component (Figure 2.15 line 30). Subsequently, the test component's *t_timeoutGuard* timer is started (Figure 2.15 line 32) to safeguard test execution against inactivity of the service under test. The timer will run for 5 seconds. Finally, the test case contains an *alt* statement (Figure 2.15 lines 34-40) that defines an ordered set of alternatives. The first alternative (Figure 2.15 lines 35-38) is executed, if the defined response message template is received on the *pt_login* port i.e. the login function of the service under test answers as expected. The alternative stops the *t_timeoutGuard* timer and sets the test verdict to "`pass`". The second alternative defined in line 39 of Figure 2.15 simply refers to the *alt_ReceiveAnyOrTimeout* altstep.

Altsteps are used to structure the alternatives of *alt* statements. Similar to those, altsteps define an ordered collection of alternatives but in addition, they are given a name. This allows altsteps to be referenced by possibly multiple *alt* statements and thereby avoids code duplication. Furthermore, altsteps can be used to specify default behavior. Altsteps may have parameters and they may refer to a component type with the "`runs on`" clause. The *alt_ReceiveAnyOrTimeout* altstep (Figure 2.15 lines 43-51) refers to the component type *LoginTestComponent* and has two alternatives that react on faulty communication with the service under test. The first alternative (Figure 2.15 lines 44-47) is executed, if any (unexpected) message is received on the *pt_login* port. Lines 48-50 of Figure 2.15 define the second alternative that is executed, if the *t_timeoutGuard* timer expires. Since both alternatives react on faulty communication, they set the test verdict to "`fail`".

2.3.2 TTCN-3 Test System

Similar to any other programming language, TTCN-3 code is not executable by itself. Tests written in the TTCN-3 core language form a so-called abstract test suite, which lacks any platform and system-specific information like an actual timer implementation or how to communicate with the system under test. The ATS has to be interpreted or translated into an executable format, the executable test suite (ETS), which is then executed by a TTCN-3 test system. A standardized adaptation for TTCN-3 test system implementations such as TTworkbench Basic is provided by the TTCN-3 Standard Part 5 and 6. As illustrated in Figure 2.16, a TTCN-3 test system can be conceptually defined as a collection of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation.

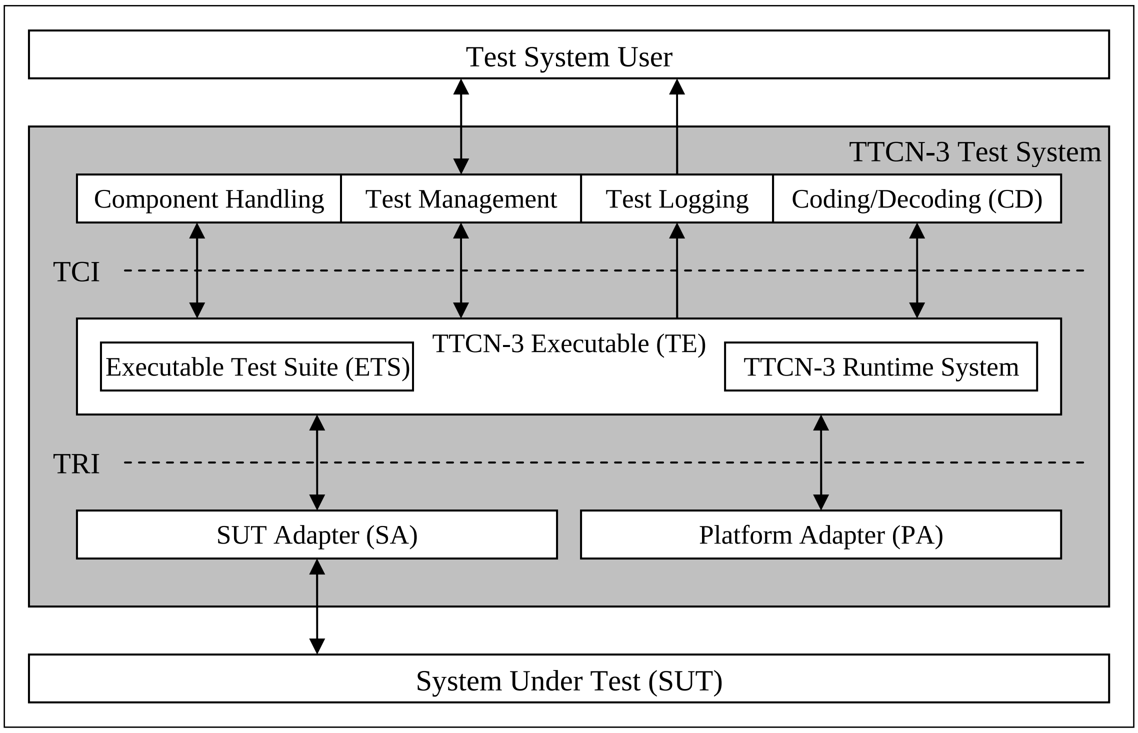


Figure 2.16: Structure of a TTCN-3 test system (cp. [7] Figure 1 and [8] Figure 1)

At the heart of a TTCN-3 test system is the TTCN-3 Executable (TE). The name shall thereby only indicate that the TE entity is responsible for the interpretation or execution of the ATS and does not necessarily imply that it is a separate executable. Conceptually, the TE consists of the executable test suite resulting from interpretation or compilation of the ATS as well as the TTCN-3 Runtime System. The latter manages the ETS and handles the interaction with the other test system entities via the standardized TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI).

The TRI defines the communication between the TE, SUT Adapter (SA), and Platform Adapter (PA) entities. Conceptually, it provides means for the TE to transmit test data to the SUT or manipulate timers, and similarly to notify the TE of received test data or timeouts. The TCI describes the interaction between the TE, Component Handling, Test Management, Test Logging, and Coding/Decoding (CD) entities. It provides means for the TE to distribute execution of test components among different test devices, manage test execution, log information about test execution, and encode/decode test data.

The standardized distribution of functionality into different entities makes TTCN-3 test system implementations very flexible. For example, a SUT Adapter can be reused in multiple test suites and possibly even in multiple test system implementations. TTCN-3 tools like TTworkbench Basic provide implementations for most of the entities, so that a user typically needs to implement only those entities that are test suite or SUT-specific, namely the SUT Adapter and the Coding/Decoding entity.

```
testcase tc_login() runs on LoginTestComponent {
   pt_login.send(a_LoginRequest);
   t_timeoutGuard.start(5.0);
   alt {
      [] pt_login.receive(a_LoginResponse) {
         t_timeoutGuard.stop;
         setverdict(pass);
      }
      [] alt_ReceiveAnyOrTimeout();
   }
}
```

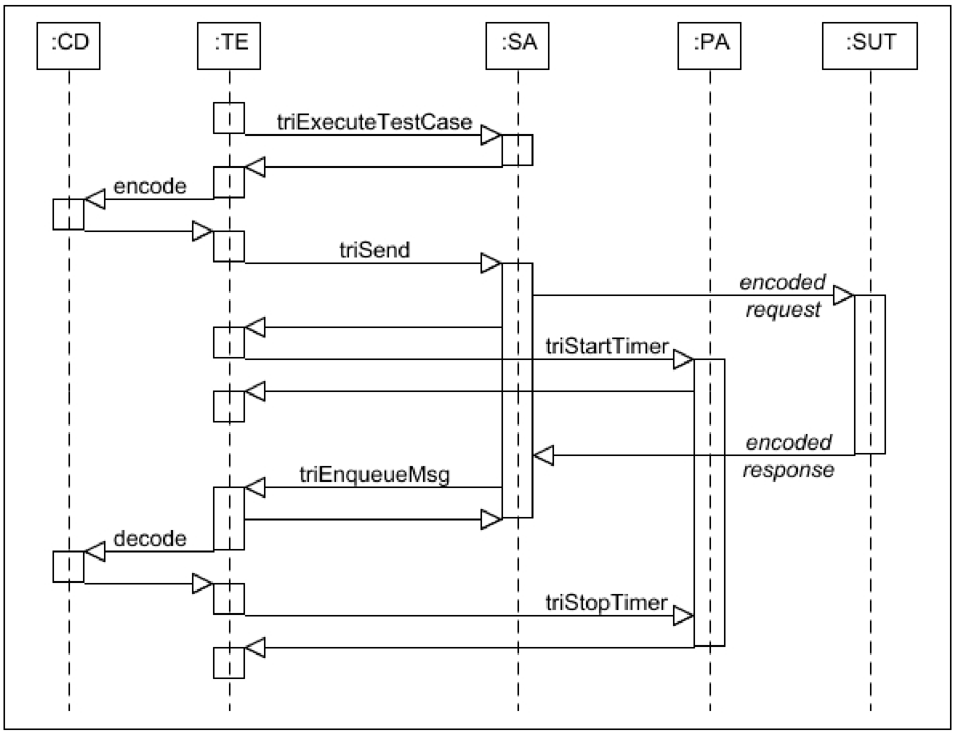


Figure 2.17: Interaction of TTCN-3 test system entities (cp. [43] Figure 12.2)

To conclude this section, Figure 2.17 illustrates the interaction of the test system entities via the TRI and TCI interfaces that takes place during execution of the login test case defined in Section 2.3.1. First, the TE invokes the *triExecuteTestcase* operation on the SA to inform it that the *tc_login* test case is about to be started and thus allows the SUT Adapter to prepare its communication facilities. Next, the *a_LoginRequest* message is send to the SUT. In a first step, the TE invokes the *encode* operation on the CD entity, which encodes the message from a structured TTCN-3 value into a form that will be accepted by the SUT. The encoded message is passed back to the TE as a binary string. In a second step, this binary string is passed on to the SA via the *triSend* operation, which is then responsible for transmitting it to the SUT. After the request message is sent, the *t_timeoutGuard* timer is started to guard the test behavior against inactivity of the SUT. To achieve this, the TE invokes the *triStartTimer* method on the PA.

In case the SUT accepts the encoded login request, it returns an encoded response. This message is received by the SUT Adapter, which forwards it to the TE by invoking the *triEnqueueMsg* operation. Inside the TE, the message receipt triggers an evaluation of the *alt* statement whose first alternative calls for a matching attempt of the received message against the *a_LoginResponse* template. For that purpose, the encoded message first has to be decoded into a structured TTCN-3 value and is therefore passed to the CD entity via the *decode* operation. If the operation succeeds, the decoded value is passed back to the TE and used in a template match attempt, which then causes the selection or rejection of the currently considered alternative. If the decode operation fails, the TE is informed and the currently considered alternative is rejected. Assuming the received message matches, the first alternative is selected and it stops the *t_timeoutGuard* timer. In order to achieve this, the TE finally invokes the *triStopTimer* method on the PA.

2.4 Related Work

There are both research and tools from industry to facilitate Web service testing. Many tools and approaches are language-specific and sometimes even bound to a specific test platform. In [12] and [26], the authors introduce testing approaches based on XML and Perl or Java, respectively. In [20], the author presents how to automate ASP.NET Web service testing using means provided by the .NET Framework. In [16], the authors present how to automate Web service testing using technologies such as JUnit, Apache Commons HttpClient, and Apache XMLUnit. However, these tools and approaches do not guarantee any systematic test and do not sufficiently cope with the language and platform-independency of Web services.

In [46], a formal approach to Web service testing based on TTCN-3 is discussed, which distributes the test activities to both server and client side. Specification of a TTCN-3 abstract test suite based on the WSDL description, models, and/or source code of a Web service is conducted at server side. At client side, the ATS is executed by a TTCN-3 test system implemented in a native programming language. Thus, the approach facilitates Web service testing while traditional software testing does not work well due to the language and platform-independency of Web services. However, the approach lacks generality especially at test implementation and execution conducted at client side and the binding between WSDL descriptions and TTCN-3 is not automated.

In [23], the authors discuss the automated testing of XML/SOAP based Web services by use of TTCN-3 and present a mapping between XML data descriptions and TTCN-3 data to enable the automated derivation of test data. The paper is the first proposal of an automated mapping between DTD (Document Type Definition) or XML Schema types and TTCN-3 type definitions. Nonetheless, the (automated) generation of TTCN-3 types associated with test configuration as well as generic test implementation and execution are not properly considered.

Additional work on importing XML Schema types into TTCN-3 is presented in [18] and [19]. It became the foundation of the TTCN-3 Standard Part 9, which was published as finalized draft ([9]) by ETSI in October 2006 and which is going to standardize the use of XML Schema with TTCN-3.

In [24], the authors discuss the import of WSDL descriptions into TTCN-3 targeting Web service testing and define mapping rules between WSDL and TTCN-3 definitions. The mapping enables the automated derivation of basic TTCN-3 test suites from WSDL descriptions where only specific test data must still be entered manually. Although the approach is evidenced by means of a simple case study, it lacks a thorough specification of the mapping between WSDL and TTCN-3. For example, it is not considered how the unambiguousness of TTCN-3 identifiers obtained from WSDL elements can be ensured. In addition, the derived TTCN-3 test suite uses procedure-based communication, which is considered less suitable in the context of Web service testing than message-based communication (s. ch. 3.1.1).

3 Mapping between WSDL and TTCN-3

This chapter presents the mapping of a Web service description given by means of a WSDL document to a TTCN-3 abstract test suite that allows basic testing of the Web service. The WSDL document is thereby expected to adhere to the WSDL 1.1 Standard and to be bound to the SOAP 1.1 or the SOAP 1.2 protocol using the document/literal, the rpc/literal, or the wrapped message style. Furthermore, the Web service interface(s) described by the WSDL document are expected to contain only request-response and/or one-way operations.

This chapter is structured as follows: First, Section 3.1 introduces the chosen TTCN-3 communication paradigm as well as the overall mapping rules. Afterwards, Section 3.2 presents a number of supportive TTCN-3 definitions. This includes naming conventions for the generated TTCN-3 abstract test suite as well as predefined TTCN-3 types needed by the mapping. Finally, the Sections 3.3 to 3.7 present detailed mapping rules for the major elements of a WSDL description.

3.1 Overview

3.1.1 Chosen TTCN-3 Communication Paradigm

An essential choice that influences the further mapping rules is whether the TTCN-3 abstract test suite should use message-based or procedure-based communication for the interaction with Web services. In the related research work, both communication ways are used almost equally often. Thereby, no reasons are given for the usage of either of the communication paradigms except in [24] where the authors motivate the usage of procedure-based communication with "the possibility of defining exceptions (from their *fault* counterparts)" ([24] ch. 4.5). The mapping presented in this thesis nevertheless generates a TTCN-3 abstract test suite that uses message-based communication for the interaction with Web services. This choice is based on the following reasons:

- Web services use message-based communication themselves, so the use of the same communication paradigm for testing suggests itself.
- The input, output, and fault messages described by the WSDL document can be mapped directly to a message type definition in TTCN-3.
- The *call* statement, a procedure-based communication operation, is not allowed to reference altsteps and during its evaluation, all active defaults are ignored. Therefore,

common behavior such as the reaction on inactivity of the system under test must be specified repeatedly. In contrast, these restrictions do not apply to the *alt* statement that is used in message-based communication to specify alternative *receive* and/or *timeout* operations.

- The syntax of the message-based communication operations *send* and *receive* is less complex than that of the procedure-based operations *call* and *getreply*, so the former provide a better readability of test cases.

3.1.2 Overall Mapping Rules

With the communication paradigm chosen, the rules for mapping a WSDL description to a TTCN-3 abstract test suite can be specified. Table 3.1 gives a first overview of the mapping rules for the major elements of a WSDL document that are discussed in detail in the Sections 3.3 to 3.7.

WSDL Element			**TTCN-3 Construct**
documentation			A comment before the TTCN-3 construct to which the enclosing WSDL element maps
types			Create a module for each target namespace, which is declared by directly enclosed XML Schema instances or the schemas that are imported, included, or redefined by them. The components defined by each XML Schema are mapped to TTCN-3 type definitions inside the module that has been created for the target namespace of the schema.
portType			A module with a TTCN-3 port type
	operation		A group
		input output fault	A record type (messages of this type are allowed to be send or received by the TTCN-3 port type that has been defined for the enclosing WSDL port type)
binding			Maps to the module that represents the WSDL port, which refers to this binding
	operation		A constant with SOAP binding information
port			A module with a constant containing SOAP binding information for each WSDL operation defined by the referenced binding

WSDL Element	TTCN-3 Construct
service	Create a module for each WSDL port type referenced by the service via port and binding, which contains basic test behavior and execution control for testing the WSDL port type. First, the module contains a component type definition owning a port, which is of the TTCN-3 port type that has been created for the WSDL port type. Second, the module contains test cases and necessary templates for testing all operations defined by the WSDL port type. Finally, the module control part executes the defined test cases for each WSDL port of the service, which references the WSDL port type. Thereby, the appropriate TTCN-3 constant with SOAP binding information is passed in.

Table 3.1: Overall rules for mapping WSDL to TTCN-3

The rules given in Table 3.1 show that the mapping makes wide use of modularization, which aids to increase the readability and maintainability of the resulting test suite. In addition, the modularization reduces the number of rules that aim at the elimination of name ambiguities. These rules are required, since the identifiers of TTCN-3 constructs are mostly obtained from the *name* attributes of the WSDL elements to which they map and those are only required to have unique names within the scope of their enclosing WSDL element. For example, the *input* and *output* elements must have unique names among the same elements within the enclosing WSDL port type, but a second port type can define *input* and *output* elements with equal names. The use of modularization is an essential improvement in comparison to the mapping presented in [24], which maps a WSDL description to a single module.

3.2 Supportive TTCN-3 Definitions

3.2.1 Naming Conventions for TTCN-3 Identifiers

Besides the modularization discussed in the Section 3.1.2, the utilization of prefixes is another approach to increase the readability and maintainability of a TTCN-3 test suite. Distinct prefixes allow distinguishing TTCN-3 identifiers of different kinds more easily and furthermore aid to prevent name ambiguities when TTCN-3 identifiers are obtained from the *name* attribute of WSDL elements. The prefixes employed by this mapping are listed in Table 3.2 and can be subdivided into two groups. The first prefixes including the one for timers are suggestions made by [43] and usable in many areas of application, whereas the remaining ones are closely bound to the domain of Web service testing.

TTCN-3 Construct	**Prefix**	**Comment**
type		In general, type identifiers are written without prefix, instead the first letter should be written in upper case
template	a_	
altstep	alt_	The same prefix is used for altsteps independent of whether they are used as defaults or not
constant	c_	
parameter	p_	Formal parameters of test cases, functions, altsteps, and templates
port	pt_	
test case	tc_	
timer	t_	
module	T_*x*	Modules that represent a target namespace declared by one or more XML Schema instances. In fact, this prefix is the final module identifier.
module	PT_*x*_	Modules representing a WSDL port type
module	P_*x*_	Modules representing a WSDL port and its associated binding
module	TPT_*x*_	Modules containing basic test behavior and execution control for testing a WSDL port type
group	g_	
record type	I_	Record types representing an input message of an operation defined by a WSDL port type
record type	O_	Record types representing an output message of an operation defined by a WSDL port type
record type	F_	Record types representing a fault message of an operation defined by a WSDL port type

TTCN-3 Construct	Prefix	Comment
field	mp_	Fields of the above record types representing the part of a WSDL input, output, or fault message
port type	TP_	Port types representing a WSDL port type
component type	TC_	Component types defining test components on which behavior for testing a WSDL port type is executed
constant	b_	Constants that encapsulate SOAP binding information for an operation of a WSDL port and its associated binding
Note: The "*x*" inside the module prefixes represents a target namespace qualifier as specified beneath.		

Table 3.2: Prefixes of TTCN-3 identifiers (cp. [43] Table 13.1)

Every module prefix specified in Table 3.2 incorporates a target namespace qualifier whose purpose is the uniqueness of module identifiers. In case of the modules, which represent a target namespace declared by XML Schemas contained in the WSDL *types* element, the prefix is the final module identifier. Thus, the incorporation of a qualifier for the target namespace is the only possibility to create unique identifiers. The module prefixes specified for the other WSDL elements are prepended to the value of the *name* attribute on the respective element. Although those names are required to be unique amongst the same elements within the enclosing WSDL description, elements with equivalent names can be imported from another WSDL document. Because of that, each module prefix incorporates a qualifier for the target namespace of the WSDL document, which encloses the WSDL element that maps to the module. The mapping presented in this thesis does not specify a concrete format for the target namespace qualifiers, but it suggests the usage of either user-defined qualifiers or hashes of the target namespace URIs, because both of them can guarantee unique identifiers even for the subsequent mapping of WSDL documents. All examples that are included in this thesis make use of user-defined qualifiers.

As aforementioned, many TTCN-3 identifiers are obtained from the *name* attribute of WSDL elements. It is thereby imperative to ensure that the resulting identifiers do not contain illegal characters. The syntax and set of valid characters for TTCN-3 identifiers are defined as follows:

"TTCN-3 identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore (_) symbol is also allowed. An identifier shall begin with a letter (i.e. not a number and not an underscore)." ([6] ch. A.1.3)

In contrast to this clear definition for TTCN-3 identifiers, the set of valid characters for the *name* attribute of WSDL elements is only inconsistently specified. Following the normative text of the WSDL 1.1 Standard, the *name* attributes of WSDL elements are of the XML Schema data type *NMTOKEN* i.e. a valid value is an arbitrary concatenation of letters, digits, points, hyphens, underscores, and colons. Against this, the schema for WSDL documents given in Section A.4.1 of the WSDL 1.1 Standard defines that some *name* attributes are of the XML Schema data type *NCName* i.e. a valid attribute value must now start with a letter or underscore and it is not allowed to contain colons. This redefinition is crucial, since otherwise the names of the affected WSDL elements e.g. the *port* and *portType* element could not be used to construct a QName by which the elements can be referenced. Finally, the WS-I Basic Profile Version 1.0 and 1.1 address this inconsistency of the WSDL 1.1 Standard and each of them references a new schema for WSDL documents, which both define that the *name* attribute of all WSDL elements is of the XML Schema data type *NCName* (s. [44] ch. 5.1.1 and [45] ch. 4.2.1).

In order to support WSDL documents conforming to any of the mentioned schemas, the translation of *name* attributes to valid TTCN-3 identifiers considers values of the XML Schema data types *NCName* and *NMTOKEN*. Because the *name* attribute values are appended to prefixes specified in Table 3.2, the resulting TTCN-3 identifiers are already guaranteed to start with a letter. In order to ensure that the identifiers contain only valid characters, the occurrences of point, hyphen, and colon characters are replaced by the escape sequences defined in Table 3.3. Due to the fact that all escape sequences start with an underscore, it is mandatory to escape the actual underscore character too.

Illegal character	**Replacing escape sequence**
`"_"`	`"_U"`
`"."`	`"_P"`
`"-"`	`"_H"`
`":"`	`"_C"`

Table 3.3: Replacement of illegal characters in WSDL identifiers

3.2.2 Predefined TTCN-3 Types

This section discusses a number of TTCN-3 types that are predefined for the mapping. On the one hand, this includes TTCN-3 representations of the XML Schema built-in data types (s. [40] ch. 3) and the schema-related attributes defined for the direct use in any XML document being validated (s. [39] ch. 2.6). They are needed for the mapping of the data type definitions contained in the WSDL *types* element, because the default WSDL type system is XML Schema. The representations of the XML Schema built-in data types and the schema-related attributes are defined in a distinct TTCN-3 module named *XSDAUX*, which will be imported as needed by the modules that are generated during the mapping. On the other hand, TTCN-3 types are predefined, which allow the encapsulation of SOAP binding information using a fixed format. Those TTCN-3 types are defined in another distinct module named *WebServices*, which will also be imported as needed.

The TTCN-3 representations of the XML Schema built-in data types base upon those presented in Part 9 of the TTCN-3 Standard (s. [9] Annex A), whereat the following changes apply:

- All subtypes of "useful TTCN-3 types", which are specified inside Annex E of the TTCN-3 Standard Part 1, are directly subtyped from built-in TTCN-3 types with the same restrictions as the former base type to reduce the number of predefined types. The concerned type definitions are *long_*, *unsignedLong*, *int*, *unsignedInt*, *short_*, *unsignedShort*, *byte_*, *unsignedByte*, *float_*, and *double*. The type identifiers having a trailing underscore character to prevent ambiguities with the "useful TTCN-3 types" remain the same to keep identifiers consistent with the TTCN-3 Standard Part 9.
- The pattern restriction of the TTCN-3 representations of XML Schema string types are revised, so that they follow the rules for character patterns (s. [6] ch. B.1.5) and conveniently mimic the restrictions for the XML Schema string types (s. [40] ch. 3):
  - If a set expression contains an unescaped hyphen character, which is preceded and followed by another character but should not denote a value range, the hyphen is escaped, so it loses its special meaning and no longer denotes a value range. The concerned type definitions are *ENTITY*, *Name*, *NCName*, *ID*, and *IDREF*.
  - The pattern of the *token* type contains a trailing "|" character that has no use and is therefore removed.

  - The pattern of the *base64Binary* type is changed to "`[0-9a-zA-Z+/= ]#(0,)`".
  - Occurrences of the metacharacter "`\d`" in a set expression together with the "`\w`" metacharacter are removed, because the latter also matches numerical digits. The concerned types are *NMTOKEN*, *ENTITY*, *Name*, *NCName*, *ID*, and *IDREF*.
  - The first character of the XML Schema data types *ENTITY*, *Name*, *NCName*, *ID*, and *IDREF* is not allowed to be a digit. Thus, the set expressions in the patterns of the respective TTCN-3 representations, which describe the first character, have to contain the range "`a-zA-Z`" instead of the "`\w`" metacharacter.

- The *extension* attributes, which are used by the TTCN-3 Standard Part 9 to allow codecs to keep track of the original XSD nature of a given TTCN-3 type by storing a value of the fixed format "XSD:<localName>", are removed. Instead the XSD nature is stored in the *encode* attribute, which is more appropriate for this information (cp. [6] ch. 28), whereat the string representation "{<namespaceURI>}<localName>" is used to store the qualified names of the XML Schema built-in data types.

The TTCN-3 representations of the four schema-related attributes, which are defined for the direct use in any XML document being validated, follow the rules of mapping an attribute declaration, as defined by the TTCN-3 Standard Part 9 and this thesis. Though those attributes are defined in a different namespace than the XML Schema built-in data types, their TTCN-3 representations are defined in the same module, so that the number of modules with supportive definitions is kept as low as possible.

Figure 3.1 shows parts of the *XSDAUX* module containing the TTCN-3 type definitions for the XML Schema built-in data types and the schema-related attributes. The complete module is listed in Figure B.1.

```
/*
 * This module defines TTCN-3 representations of XML Schema types basing on
 * the types presented in the ETSI standard ES 201 873-9 V1.1.1 Annex A
 */
module XSDAUX {
   // String types
   type charstring token (pattern "([^ \t\r\n]#(1,)( [^ \t\r\n]#(1,))#(0,))") with {
      encode "{http://www.w3.org/2001/XMLSchema}token";
   }

   type charstring string with {
      encode "{http://www.w3.org/2001/XMLSchema}string";
   }
   ...

   // Integer types
   type integer integer_ with {
      variant "{http://www.w3.org/2001/XMLSchema}integer";
   }

```

```
    type integer positiveInteger (1 .. infinity) with {
       variant "{http://www.w3.org/2001/XMLSchema}positiveInteger";
    }
    ...

    // Boolean types
    type boolean boolean_ with {
       encode "{http://www.w3.org/2001/XMLSchema}boolean";
    }

    // XSI attribute declarations
    type XSDAUX.QName Attribute_type with {
       encode "{http://www.w3.org/2001/XMLSchema-instance}type";
    }
    ...
 }
```

Figure 3.1: Extract of the *XSDAUX* module

In order to be able to define the TTCN-3 types, which allow encapsulation of SOAP binding information using a fixed format, the SOAP bindings supported by the mapping have to be examined first. The SOAP 1.1 binding is defined in Chapter 3 of the WSDL 1.1 Standard and it extends a WSDL document as shown in Figure 3.2. The informal syntax, which is used by the figure to describe the XML grammar, is defined in Section 1.2 of the WSDL 1.1 Standard, but it should be accessible without further reading.

```
<definitions .... >

  <binding .... > *
    <soap:binding style="rpc|document" ? transport="uri" />
    <operation .... > *
      <soap:operation soapAction="uri" ? style="rpc|document" ? /> ?
      <input> ?
        <soap:body parts="nmtokens" ? use="literal|encoded"
                   encodingStyle="uri-list" ? namespace="uri" ? />
        <soap:header message="qname" part="nmtoken" use="literal|encoded"
                     encodingStyle="uri-list" ? namespace="uri" ? > *
          <soap:headerfault message="qname" part="nmtoken" use="literal|encoded"
                            encodingStyle="uri-list" ? namespace="uri" ? /> *
        <soap:header>
      </input>
      <output> ?
        <soap:body parts="nmtokens" ? use="literal|encoded"
                   encodingStyle="uri-list" ? namespace="uri" ? />
        <soap:header message="qname" part="nmtoken" use="literal|encoded"
                     encodingStyle="uri-list" ? namespace="uri" ? > *
          <soap:headerfault message="qname" part="nmtoken" use="literal|encoded"
                            encodingStyle="uri-list" ? namespace="uri" ? /> *
        <soap:header>
      </output>
      <fault> *
        <soap:fault name="nmtoken" use="literal|encoded"
                    encodingStyle="uri-list" ? namespace="uri" ? />
      </fault>
    </operation>
  </binding>

  <service .... > *
    <port .... > *
      <soap:address location="uri" />
    </port>
  </service .... >

</definitions>
```

Figure 3.2: WSDL extended by the SOAP 1.1 binding (cp. [37] ch. 3.2)

Since the SOAP binding versions do not differ much and all distinctions can be adapted with little effort, first TTCN-3 types are defined that allow encapsulation of SOAP 1.1 binding information. Subsequently, the differences between the SOAP binding versions are discussed and the defined TTCN-3 types are extended to support SOAP 1.2 binding information too. The completed types are shown in Figure 3.4.

At first, subtypes of the TTCN-3 built-in type *charstring* are specified for the *use* and *style* attributes and both subtypes are restricted to the values allowed for the respective attribute. Furthermore, two set-of types are specified for the *parts* and *encodingStyle* attributes, whereat the TTCN-3 representations of the XML Schema built-in data types *NMTOKEN* and *anyURI* are used as element type.

As can be clearly seen in Figure 3.2, almost any information provided by the SOAP 1.1 binding is operation specific. Therefore, the binding information is encapsulated on the operation level as specified by the overall mapping rules, whereat common details given by the *soap:address* and the *soap:binding* elements are also included. For that purpose, a record type named *SoapBinding* is defined, which contains a field for every piece of information given by attributes of the *soap:address*, *soap:binding*, and *soap:operation* elements. The fields are named equivalent to their corresponding attribute and are of a type that is appropriate for the attribute values i.e. either a TTCN-3 representation of XML Schema built-in data types or a type already defined for the SOAP binding. The *style* attribute optionally present on *soap:binding* and *soap:operation* elements maps to a single field, since the value specified on a *soap:binding* element is only the default for the *style* attributes of the *soap:operation* elements enclosed by the same WSDL binding. Because the information defined by the other SOAP binding elements is too complex to be stored in simple fields of the *SoapBinding* record type, additional TTCN-3 types are defined. Those types encapsulate the information defined by the complex SOAP binding elements and the *SoapBinding* type contains fields of them.

For the *soap:body*, *soap:header*, *soap:headerfault*, and *soap:fault* elements, record types are defined, which follow the rules that are specified above i.e. attributes become simple fields and nested elements become fields of additional types. Because multiple *soap:header* and *soap:fault* elements are allowed for each operation, an additional set-of type needs to be defined for each of them. The same applies to the *soap:headerfault* element that is allowed to occur multiple times inside a *soap:header* element.

Finally, all fields, which correspond to nodes marked to be optional with a succeeding "?" or "*" character, are also defined optional in TTCN-3. An exception is the *style* field of the *SoapBinding* type, which is mandatory although the corresponding *style* attribute is declared optional on the *soap:binding* and *soap:operation* elements. This bases upon the fact that if "the *soap:binding* element does not specify a style, it is assumed to be "document"" ([37] ch. 3.4).

The binding for the SOAP 1.2 protocol, which is specified in Chapter 3 of the "WSDL 1.1 Binding Extension for SOAP 1.2" ([41]), extends a WSDL document as shown in Figure 3.3. The XML grammar is described with an informal syntax, which is specified in Section 2.2 of [41], but should be accessible without further reading.

```
<wsdl:definitions ...>

  ...

  <wsdl:binding ...>
    <wsoap12:binding style="rpc|document" ?
                     transport="xs:anyURI" />
    <wsdl:operation ...>
      <wsoap12:operation soapAction="xs:anyURI" ?
                         soapActionRequired="xs:boolean" ?
                         style="rpc|document" ? /> ?
      <wsdl:input>
        <wsoap12:body parts="list of xs:NMTOKEN" ?
                      use="literal|encoded" ?
                      encodingStyle="xs:anyURI" ?
                      namespace="xs:anyURI" ? />
        <wsoap12:header message="xs:QName"
                        part="xs:NMTOKEN"
                        use="literal|encoded"
                        encodingStyle="anyURI" ?
                        namespace="anyURI" ? > *
          <wsoap12:headerfault message="xs:QName"
                               part="xs:NMTOKEN"
                               use="literal|encoded"
                               encodingStyle="anyURI" ?
                               namespace="anyURI" ? /> *
          <wsoap12:header> *
      </wsdl:input> ?
      <wsdl:output>
        <!-- Same as wsdl:input -->
      </wsdl:output> ?
      <wsdl:fault>
        <wsoap12:fault name="xs:NMTOKEN"
                       use="literal|encoded"
                       encodingStyle="xs:anyURI" ?
                       namespace="xs:anyURI" ? />
      </wsdl:fault> *
    </wsdl:operation> *
  </wsdl:binding> *

  <wsdl:service ...>
    <wsdl:port ...>
      <wsoap12:address location="xs:anyURI" />
    </wsdl:port> *
  </wsdl:service>

</wsdl:definitions>
```

Figure 3.3: WSDL extended by the SOAP 1.2 binding (cp. [41] ch. 3)

As aforementioned, the SOAP 1.2 binding does not differ much from the SOAP 1.1 binding. The four key differences are:

- Introduction of a new namespace ("http://schemas.xmlsoap.org/wsdl/soap12/").
- The *style* attribute of the *binding* element has become mandatory.
- If present, the *encodingStyle* attribute is a single URI, instead of a list of URIs.
- The *operation* element has a new attribute called *soapActionRequired*, which is used to indicate that the server needs the *soapAction* value.

Except for the last difference, none of them imposes the need to change the TTCN-3 types. The new namespace is only relevant for the parsing of WSDL descriptions. The *style* attribute of the *binding* element, which is now mandatory, has a default value in the SOAP 1.1 binding, so that an attribute value is always given independent of the SOAP binding version. Finally, the single URI possibly specified by an *encodingStyle* attribute can be treated as a list of URIs with a single element.

For the new *soapActionRequired* attribute, an optional field is added to the *SoapBinding* record type. In case of the SOAP 1.1 binding, the value of the *soapActionRequired* field is always "`omit`". In case of the SOAP 1.2 binding, the values "`true`" and "`false`" are possible as well. Besides the *soapActionRequired* field, the *soapVersion* field is added to the *SoapBinding* record. The mandatory field stores the SOAP binding version and is of the additional *SoapVersion* type, which is restricted to the values "`V_11`" and "`V_12`".

The completed TTCN-3 types representing SOAP binding information are displayed in Figure 3.4.

```
module WebServices {
   import from XSDAUX all;

   type charstring SoapVersion ("V_11", "V_12");

   type charstring SoapStyle ("rpc", "document");

   type charstring SoapUse ("literal", "encoded");

   type set length (1 .. infinity) of XSDAUX.NMTOKEN SoapParts;

   type set length (1 .. infinity) of XSDAUX.anyURI SoapEncodingStyles;

   type record SoapBody {
      WebServices.SoapUse use,
      WebServices.SoapParts parts optional,
      WebServices.SoapEncodingStyles encodingStyles optional,
      XSDAUX.anyURI namespace optional
   }

   type record SoapHeader {
      WebServices.SoapUse use,
      XSDAUX.QName messageName,
      XSDAUX.NMTOKEN part,
      WebServices.SoapEncodingStyles encodingStyles optional,
      XSDAUX.anyURI namespace optional,
      WebServices.SoapHeaderFaults headerFaults optional
   }

   type set length (1 .. infinity) of WebServices.SoapHeader SoapHeaders;

   type record SoapHeaderFault {
      WebServices.SoapUse use,
      XSDAUX.QName messageName,
      XSDAUX.NMTOKEN part,
      WebServices.SoapEncodingStyles encodingStyles optional,
      XSDAUX.anyURI namespace optional
   }

   type set length (1 .. infinity) of WebServices.SoapHeaderFault SoapHeaderFaults;

   type record SoapFault {
      WebServices.SoapUse use,
      XSDAUX.NMTOKEN name,
      SoapEncodingStyles encodingStyles optional,
      XSDAUX.anyURI namespace optional
   }

   type set length (1 .. infinity) of WebServices.SoapFault SoapFaults;

   type record SoapBinding {
      XSDAUX.QName operationName,
      WebServices.SoapVersion soapVersion,
      XSDAUX.anyURI location,
      XSDAUX.anyURI transport,
      WebServices.SoapStyle style,
      XSDAUX.anyURI soapAction optional,
      XSDAUX.boolean_ soapActionRequired optional,
      WebServices.SoapBody inputBody,
      WebServices.SoapHeaders inputHeaders optional,
      WebServices.SoapBody outputBody optional,
      WebServices.SoapHeaders outputHeaders optional,
      WebServices.SoapFaults faults optional
   }

   type record length (1 .. infinity) of WebServices.SoapBinding SoapBindings;
}
```

Figure 3.4: TTCN-3 types representing SOAP binding information

3.3 Mapping of the WSDL *types* Element

The WSDL 1.1 Standard introduces the *types* element as follows:

> "The *types* element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system.
>
> […]
>
> However, since it is unreasonable to expect a single type system grammar can be used to describe all abstract types present and future, WSDL allows type systems to be added via extensibility elements." ([37] ch. 2.2)

The mapping described in this thesis considers only the default type system in WSDL namely XML Schema, because other type systems are regarded as proprietary and rare.

A mapping of XML Schema data type definitions to TTCN-3 is defined in the TTCN-3 Standard Part 9. The mapping presented in this thesis widely adopts the rules specified by the standard but also applies some changes and additions, which are discussed in the Sections 3.3.1 to 3.3.8.

3.3.1 Mapping of XML Schemas as a Whole

The mapping of XML Schema data type definitions has to take notice of two essential points. On the one hand, it has to support the simultaneous mapping of multiple XML Schemas, because the WSDL *types* element is allowed to contain an arbitrary number of *schema* elements, which in each case represent an XML Schema and can include or import further schemas. On the other hand, the mapping has to support the concept of target namespaces. The TTCN-3 Standard Part 9 specifies the following rules for the mapping of XML Schemas and the handling of namespaces:

> "A single XSD Schema will be translated to a single TTCN-3 module. Any XSD *include* / *import* statements are mapped to equivalent TTCN-3 *import* statements.
>
> XML namespaces are supported by means of incorporating any namespace qualifiers into the translated TTCN-3 identifier, as TTCN-3 does not offer a namespace concept. To do this, a translator has to first bring all XSD identifiers to a unique qualified form ([…]) and than translate these unique qualified names to TTCN-3 identifiers […]." ([9] ch. 5.1)

The mapping described in this thesis chooses another approach that mainly aims at a better support of namespaces. A TTCN-3 module is created for each declared target namespace instead of each XML Schema i.e. the data type definitions of schemas declaring the same target namespace map to a single TTCN-3 module. This poses no threat to the unambiguousness of TTCN-3 identifiers, because the names of each kind of definition and declaration component are required to be unique within their distinct symbol space within the target namespace. In this context, the term symbol space "is similar to the non-normative concept of namespace partition introduced in [XML-Namespaces]" ([39] ch. 2.5) and "denote[s] a collection of names, each of which is unique with respect to the others" ([39] ch. 2.5). The translation mechanism that makes the names unique across the distinct symbol spaces, so that they can be used as TTCN-3 identifiers, is discussed in Section 3.3.2. The identifier of a created module becomes the prefix specified in Section 3.2.1, which incorporates a qualifier for the target namespace to which the TTCN-3 module maps. The main advantages of the chosen approach to the mapping of XML Schemas are:

- No need to incorporate namespace qualifiers into all translated TTCN-3 identifiers.
- The module containing the TTCN-3 representation of an XML Schema component can be resolved unambiguously from its qualified name.

3.3.2 Translation of XML Schema Identifiers

As mentioned in Section 3.3.1, the names of each kind of XML Schema components are only unique within their distinct symbol space. In order to use them as identifiers for the resulting TTCN-3 type definitions, they need to be made unique across the distinctive symbol spaces first. Furthermore, it has to be ensured that the identifiers resulting from names of schema components are unequal to TTCN-3 keywords. The TTCN-3 Standard Part 9 does not standardize those translations but demands the following:

"A name conversion algorithm has to guarantee that the translated identifier name…

a) is unique within the scope it is to be used

b) contains only valid characters

c) is not a TTCN-3 keyword

d) is not a reserved word ("base" or "content")" ([9] ch. 5.2)

Besides those requirements, the TTCN-3 Standard Part 9 specifies an algorithm that is used for all examples within the standard (s. [9] ch. 5.2). The mapping presented in this thesis utilizes a similar algorithm that introduces some amendments and improvements.

The TTCN-3 Standard Part 9 does not specify how to deal with characters that are not allowed in TTCN-3 identifiers. Since the names of schema components are of the XML Schema data type *NCName*, they can contain invalid point and/or hyphen characters. Those characters are replaced by escape sequences, as defined in Table 3.3.

The name of an XML Schema component is made unique across the symbol spaces by appending it to a prefix that reflects the component type e.g. *Attribute_foo*. Compared to the TTCN-3 Standard Part 9, which defines that the names of element declarations remain unchanged and all other names are prepended to two underscores and the type of the component e.g. *foo__attribute*, the chosen approach has two advantages. First, the usage of prefixes guarantees that the resulting TTCN-3 identifiers start with a letter as demanded by the TTCN-3 Standard Part 1 (s. [6] ch. A.1.3). This cannot be ensured by using suffixes, because the names of XML Schema components are allowed to start with a letter or an underscore. Second, the prefixing of all component names removes the need to check whether an element name possibly matches a TTCN-3 keyword or the concatenation of another component name and its respective suffix. For example, an element schema component could be named *foo__attribute*, so following the translation algorithm of the TTCN-3 Standard Part 9 the resulting TTCN-3 identifier would be equal to that of an attribute component named *foo*. In contrast, the prefixing approach creates the distinct identifiers *Element_foo__attribute* and *Attribute_foo*. The prefixes that can occur during the mapping to TTCN-3 are: *Element_*, *Attribute_*, *SimpleType_*, *ComplexType_*, *Group_*, *AttributeGroup_*, *All_*, *Choice_*, *Sequence_*, *Extension_*, and *Any_*.

The mapping of XML Schema components to TTCN-3 sometimes requires the creation of multiple TTCN-3 type definitions. The name of the schema component is thereby used for the main type, whereas the names for the additional, anonymous types need to be generated. The TTCN-3 Standard Part 9 specifies that those generated identifiers are the concatenation of the word *ANONYM* and a sequential number that must be unique per module. The translation algorithm of the mapping presented in this thesis uses only a sequential number for the generated names, which again must be unique per module. Those generated names cannot match the name of an XML Schema component, because the latter are not allowed to start with a digit. At the same time, the resulting TTCN-3 identifiers are still valid, because the sequential number is appended to the prefix that corresponds to the type of the structure that is mapped to the anonymous TTCN-3 type. For example, the TTCN-3 identifier for a local simple type definition, which needs to be mapped to an anonymous TTCN-3 type, could be *SimpleType_5*.

The above rules guarantee that the identifiers of TTCN-3 type definitions resulting from the mapping meet the requirements specified by the TTCN-3 Standard Part 9. In order to guarantee the same for the identifiers of fields or variants of those type definitions, additional rules need to be specified. Thereby, the translation algorithm must only make sure that the names of local element or attribute declarations, which map to fields or variants, do not match any TTCN-3 keyword or a reserved word. Uniqueness is already ensured, because the names of the local declaration components must be unambiguous within their local symbol spaces (s. [39] ch. 2.5) and the local element and attribute declarations map to different TTCN-3 constructs.

The TTCN-3 Standard Part 9 defines that the names matching a keyword or a reserved word should be converted to uppercase, but this approach has the following drawback. If the symbol space contains another name that already denotes the keyword or reserved word in uppercase letters, a name ambiguity occurs that has to be resolved with another translation rule. In order to circumvent this problem, the translation algorithm that is used by the mapping presented in this thesis simply prefixes the names of locally declared elements and attributes. This prefixing approach furthermore has the advantage that the resulting identifiers of fields and variants are guaranteed to start with a letter, as demanded by the TTCN-3 Standard Part 1 (s. [6] ch. A.1.3). On the one hand, the used prefix depends on the type of the local declaration i.e. element or attribute. On the other hand, it depends upon the information whether the appearance of the local element or attribute declaration in an XML document must be qualified by a namespace. The latter can be specified by a schema author as follows:

> "Qualification of local elements and attributes can be globally specified by a pair of attributes, *elementFormDefault* and *attributeFormDefault*, on the schema element, or can be specified separately for each local declaration using the *form* attribute. All such attributes' values may each be set to *unqualified* or *qualified*, to indicate whether or not locally declared elements and attributes must be unqualified." ([38] ch. 3.1)

If the appearances of a local element declaration must be qualified, the identifier of the field or variant to which the declaration maps becomes the declaration name appended to the prefix *eq_*. If otherwise the appearances have to be unqualified, the prefix *e_* is used. In an analogous manner, the name of a local attribute declaration is appended to the prefix *aq_* or *a_*, respectively.

Finally, the TTCN-3 *encode* attribute is used to store the non-translated names of global and local XML Schema components. Therewith, codecs are able to keep track of the original name of a specific TTCN-3 type, field, or variant. The string representation "{<namespaceURI>}<localName>" is utilized to store the qualified name of an XML Schema component.

```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="naming" targetNamespace="naming"
  attributeFormDefault="unqualified"
  elementFormDefault="unqualified">
- <complexType name="person">
  - <sequence>
      <element name="age" type="xsd:string" />
      <element name="age" type="xsd:int"
        form="qualified" />
    </sequence>
    <attribute name="age" type="xsd:string" />
    <attribute name="age" type="xsd:int"
      form="qualified" />
  </complexType>
  <element name="person" type="tns:person" />
</schema>
```

```
module T_naming {
  import from XSDAUX all;

  type set ComplexType_person {
      T_naming.Sequence_1 base,
      XSDAUX.string a_age optional,
      XSDAUX.int aq_age optional
  } with {
      encode "{naming}person";
       extension "Attribute: a_age,
         aq_age";
      encode (a_age) "age";
      encode (aq_age) "{naming}age";
  }

  type record Sequence_1 {
      XSDAUX.string e_age,
      XSDAUX.int eq_age
  } with {
      encode "{naming}1";
      encode (e_age) "age";
      encode (eq_age) "{naming}age";
  }

  type T_naming.ComplexType_person
         Element_person with {
      encode "{naming}person";
  }
}
```

Figure 3.5: Translation of XML Schema identifiers

To conclude this section, Figure 3.5 illustrates the presented name translation rules with the mapping of two exemplary XML Schema components. At first, the mapped complex type definition and element declaration are both named *person* to exemplify how names are made unique across the symbol spaces of different kinds of components. Second, the complex type definition requires the generation of an anonymous record type for which the identifier *Sequence_1* is created. Finally, the complex type definition contains four local element and attribute declarations that are all named *age* but are either qualified or unqualified and map to fields in different TTCN-3 type definitions.

3.3.3 Mapping of XML Schema Constraining Facets

The mapping of XML Schema constraining facets is covered by the TTCN-3 Standard Part 9 in Section 6.1. The following explanations supplement this definition by giving a more detailed overview on the mapping support for the individual constraining facets as well as some corrections and additions for the mapping rules of the *pattern* facet.

The XML Schema Standard allows constraining simple type definitions with a given set of facets. If the simple type definition is derived by union or list, the set of valid facets is fixed. If the simple type definition is otherwise derived by restriction, the used base type definition restricts the set of facets. A detailed listing of the constraining facets that are valid for each kind of simple type definition is given in Section 4.1.5 of the XML Schema Standard Part 2. Table 3.4 merges this specification with information about the mapping support for the individual constraining facets as specified in Section 6.1 of the TTCN-3 Standard Part 9.

Base Type Definition	**Constraining Facet**											
	length	**minLength**	**maxLength**	**pattern**	**enumeration**	**whitespace**	**minInclusive**	**maxInclusive**	**minExclusive**	**maxExclusive**	**totalDigits**	**fractionDigits**
Simple Type definition derived by union												
all data types				*	x							
Simple Type definition derived by list												
all data types	x	x	x	*	*	x						
Simple Type definition derived by restriction												
NMTOKENS	x	x	x	*	*	x						
IDREFS	x	x	x	*	*	x						
ENTITIES	x	x	x	*	*	x						
string	x	x	x	x	x	x						
normalizedString	x	x	x	x	x	x						
token	x	x	x	x	x	x						
language	x	x	x	x	x	x						
NMTOKEN	x	x	x	x	x	x						
Name	x	x	x	x	x	x						
NCName	x	x	x	x	x	x						
ID	x	x	x	x	x	x						
IDREF	x	x	x	x	x	x						
ENTITY	x	x	x	x	x	x						
boolean				*		x						
float				*	x	x	x	x	x	x		
double				*	x	x	x	x	x	x		
decimal				*	x	x	x	x	x	x	*	*
integer				*	x	x	x	x	x	x	x	*

Base Type Definition	**Constraining Facet**											
	length	**minLength**	**maxLength**	**pattern**	**enumeration**	**whitespace**	**minInclusive**	**maxInclusive**	**minExclusive**	**maxExclusive**	**totalDigits**	**fractionDigits**
negativeInteger				*	x	x	x	x	x	x	x	*
nonPositiveInteger				*	x	x	x	x	x	x	x	*
nonNegativeInteger				*	x	x	x	x	x	x	x	*
positiveInteger				*	x	x	x	x	x	x	x	*
long				*	x	x	x	x	x	x	x	*
int				*	x	x	x	x	x	x	x	*
short				*	x	x	x	x	x	x	x	*
byte				*	x	x	x	x	x	x	x	*
unsignedLong				*	x	x	x	x	x	x	x	*
unsignedInt				*	x	x	x	x	x	x	x	*
unsignedShort				*	x	x	x	x	x	x	x	*
unsignedByte				*	x	x	x	x	x	x	x	*
duration				x	x	x	*	*	*	*		
dateTime				x	x	x	*	*	*	*		
time				x	x	x	*	*	*	*		
date				x	x	x	*	*	*	*		
gYearMonth				x	x	x	*	*	*	*		
gYear				x	x	x	*	*	*	*		
gMonthDay				x	x	x	*	*	*	*		
gDay				x	x	x	*	*	*	*		
gMonth				x	x	x	*	*	*	*		
hexBinary	x	x	x	*	x	x						
base64Binary	x	x	x	x	x	x						
anyURI	x	x	x	x	x	x						
QName	x	x	x	x	x	x						
NOTATION	*	*	*	*	*	*						

x Allowed by the XML Schema Standard and a mapping to TTCN-3 is supported.

* Allowed by the XML Schema Standard but a mapping to TTCN-3 is not supported.

Table 3.4: Mapping support of XML Schema constraining facets (cp. [40] ch. 4.1.5)

The XML Schema *pattern* facet is expected to contain a regular expression that follows the specification given in Chapter F of the XML Schema Standard Part 2. The mapping of these regular expressions to a TTCN-3 pattern is discussed in Section 6.1.4 of the TTCN-3 Standard Part 9, but some of the given rules must be corrected as follows:

- The character class "`\s`" should become the term "`[^ \t\n\t]`" that is presumably erroneous due to a typing error. According to the regular expression specification the correct mapping result is the expression "`[^ \t\n\r]`".
- The character class "`\i`" should be mapped to "`[\w\d:]`", but the specification of regular expressions defines this character class to be the set of initial name characters i.e. those matched by "`_`", "`:`", or a character from the character class *Letter* defined by the XML standard (s. [30] ch. B). The "`_`" character has therefore to be included and digits to be excluded, so the mapping result becomes the term "`[a-zA-Z_:]`". The same correction applies to the translation of the character class "`\I`".
- The mapping result for the character classes "`\c`" and "`\C`" should contain the term "`\w\d`". Thereby, the metacharacter "`\d`" can be removed, because the metacharacter "`\w`" already matches numeric digits.
- The TTCN-3 Standard Part 9 defines that an escaped plus character "`\+`" should be mapped directly to the plus character. This mapping rule needs to be excluded, since the plus character is a metacharacter in TTCN-3 patterns too (s. [6] Table B.1).
- The mapping of Unicode characters is defined to be a direct translation that simply changes the syntax from "`&#xgprc;`" to "`\q{g,p,r,c}`" with "`g,p,r,c`" representing single hexadecimal characters. First, this mapping rule has to be supplemented by three more, which translate Unicode characters with only one, two, or three values given. The missing values become zero, so that for example the expression "`&#xrc;`" maps to "`\q{0,0,r,c}`". Second, it is not clearly defined whether TTCN-3 patterns allow hexadecimal characters to occur in representations of Unicode characters. The TTCN-3 Standard Part 1 specifies in Table B.1 that a Unicode character is matched by the quadruple "`\q{group, plane, row, cell}`", but no details are given about the value format. Since the quadruples occurring in code examples of the standard use only decimal characters (s. [6] ch. 6.2.4 and B.1.5.2), it is assumed that hexadecimal characters are not supported in the Unicode character representations of TTCN-3 patterns and thus have to be replaced by their corresponding decimal character.

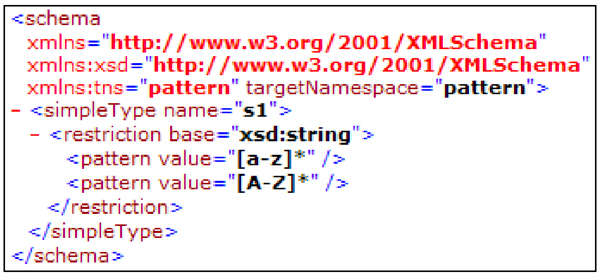
Table 3.5 summarizes the rules for mapping the regular expression given by an XML Schema *pattern* facet to a TTCN-3 pattern. For those mapping rules that are corrected versions of mapping rules specified by the TTCN-3 Standard Part 9, both versions are contained in the table to recapitulate the made corrections.

XSD regular expression	TTCN-3 pattern	
	Actual value	Value in [9]
`.`	`?`	
`\s`	`[ \t\n\r]`	
`\S`	`[^ \t\n\r]`	`[^ \t\n\t]`
`\D`	`[^\d]`	
`\W`	`[^\w]`	
`\i`	`[a-zA-Z_:]`	`[\w\d:]`
`\I`	`[^a-zA-Z_:]`	`[^\w\d:]`
`\c`	`[\w.\-_:]`	`[\w\d.\-_:]`
`\C`	`[^\w.\-_:]`	`[^\w\d.\-_:]`
`\.`	`.`	
`"`	`\"`	

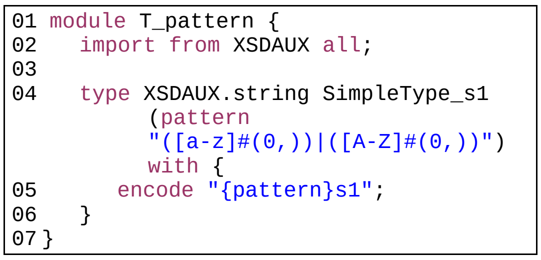
XSD regular expression	TTCN-3 pattern
`?`	`#(0,1)`
`+`	`#(1,)`
`*`	`#(0,)`
`{n}`	`#(n)`
`{n,}`	`#(n,)`
`{n,m}`	`#(n,m)`
`&#xgprc;`	`\q{g,p,r,c}`
`&#xprc;`	`\q{0,p,r,c}`
`&#xrc;`	`\q{0,0,r,c}`
`&#xc;`	`\q{0,0,0,c}`
Note: `g,p,r,c` in `[0-9a-fA-F]`	

Table 3.5: Mapping rules for XML Schema regular expressions (cp. [9] ch. 6.1.4)

Finally, an additional mapping rule must be specified for the simultaneous occurrence of multiple *pattern* facets, since this is not covered by the TTCN-3 Standard Part 9. The XML Schema Standard Part 2 states: "If multiple <pattern> element information items appear as [children] of a <simpleType>, the [value]s should be combined as if they appeared in a single ·regular expression· as separate ·branch·es" ([40] ch. 4.3.4.3). Based on this definition, the values of multiple *pattern* facets are mapped to a single TTCN-3 pattern in two steps. First, every *pattern* value is mapped to a TTCN-3 pattern according to the rules given in Table 3.5. In the second step, those translations become alternatives in the finally used TTCN-3 pattern as illustrated in Figure 3.6.



```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="pattern" targetNamespace="pattern">
- <simpleType name="s1">
  - <restriction base="xsd:string">
      <pattern value="[a-z]*" />
      <pattern value="[A-Z]*" />
    </restriction>
  </simpleType>
</schema>
```



```
module T_pattern {
   import from XSDAUX all;

   type XSDAUX.string SimpleType_s1
         (pattern
         "([a-z]#(0,))|([A-Z]#(0,))")
         with {
      encode "{pattern}s1";
   }
}
```

Figure 3.6: Mapping of multiple *pattern* facets

3.3.4 Mapping of the XML Schema *id* and *ref* Attributes

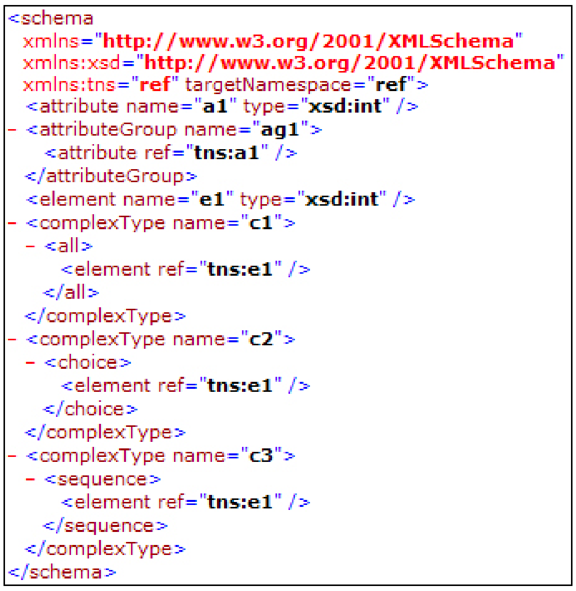
The TTCN-3 Standard Part 9 specifies the following mapping rules for the *id* and the *ref* attributes:

> "The attribute *id* enables a unique identification of an XSD component. They are mapped to TTCN-3 as simple type references, e.g. any component mapping to a type with name `typeName` and an attribute `id="ID"` should result in an additional TTCN-3 type declaration:
>
> `type typeName ID;`" ([9] ch. 7.1.1)
>
> "The *ref* attribute may reference an id or any global type (see clause 7.2).
>
> If the attribute is referring to an *id* it's directly mapped as a simple type, e.g. a component with an attribute `ref="REF"` is translated to:
>
> `type REF typeName;`
>
> In the case that `REF` references a global type the name of the global type has to be substituted, e.g.
>
> `type globalType typeName;`" ([9] ch. 7.1.2)

The mapping presented in this thesis redefines the rule for the *ref* attribute and omits the rule for the *id* attribute based on the following reasons:

The XML Schema Standard only specifies how to resolve a global schema component based upon a QName given by a *ref* attribute (s. [39] ch. 3.15.3). It does not describe how an id should be resolved based on a QName and defines no semantics for the *id* attribute. Hence, the *id* attribute has no significance for the mapping of XML Schema type definitions to TTCN-3 and it can be ignored.

The *ref* attribute is only permitted to appear on *attribute*, *element*, *attributeGroup*, and *group* elements that are no immediate children of the *schema* element (s. [39] ch. 3.2.2, 3.3.2, 3.6.2, and 3.7.2). Hence, a schema component with a *ref* attribute is always part of another structure e.g. a complex type definition and the mapping to a subtype is not suitable. The TTCN-3 Standard Part 9 discusses the appearance of nested *group* and *attributeGroup* elements with a *ref* attribute (s. [9] ch. 7.6.5.2, 7.6.6.2, and 7.6.7), but it disregards the occurrence of nested *element* or *attribute* elements with a *ref* attribute. Therefore, additional rules are defined for the latter mappings, which are similar to the rules for the nested *group* and *attributeGroup* elements. The nested *element* or *attribute* components with a *ref* attribute are mapped to a field or variant in the TTCN-3 type definition to which their enclosing schema component maps. The type of this field or variant is the TTCN-3 type that represents the global element or attribute declaration to which the value of the *ref* attribute refers. The name of the field, to which an *attribute* component with a *ref* attribute maps, is the concatenation of *attribute_* and a sequential number that must be unique amongst all fields of the enclosing TTCN-3 type. The name of the field or variant, to which a nested *element* component with a *ref* attribute maps, is the concatenation of a reserved string and a sequential number that needs to be unique amongst all fields or variants of the enclosing TTCN-3 type. The reserved string is *all_*, *choice_*, or *sequence_*, depending on the schema component that encloses the nested *element* component with the *ref* attribute. The preceding explanations are illustrated in Figure 3.7, which shows the mapping of all varieties of nested *element* and *attribute* components with a *ref* attribute.



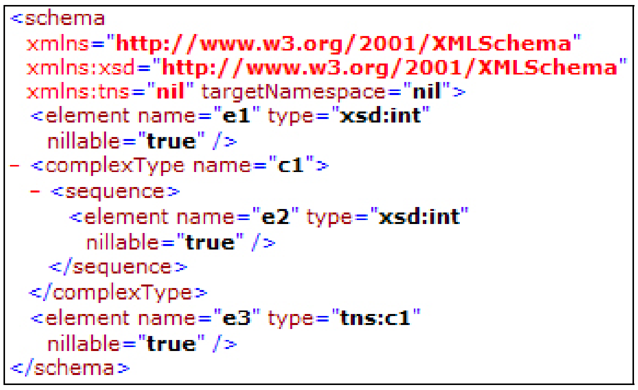
```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="ref" targetNamespace="ref">
   <attribute name="a1" type="xsd:int" />
- <attributeGroup name="ag1">
     <attribute ref="tns:a1" />
   </attributeGroup>
   <element name="e1" type="xsd:int" />
- <complexType name="c1">
  - <all>
       <element ref="tns:e1" />
     </all>
   </complexType>
- <complexType name="c2">
  - <choice>
       <element ref="tns:e1" />
     </choice>
   </complexType>
- <complexType name="c3">
  - <sequence>
       <element ref="tns:e1" />
     </sequence>
   </complexType>
</schema>
```

```
module T_ref {
   import from XSDAUX all;

   type XSDAUX.int Attribute_a1 with {
      encode "{ref}a1";
      extension "Attribute";
   }

   type set AttributeGroup_ag1 {
       T_ref.Attribute_a1 attribute_1 optional
   } with {
      encode "{ref}ag1";
      extension "AttributeGroup";
   }

   type XSDAUX.int Element_e1 with {
      encode "{ref}e1";
   }

   type set ComplexType_c1 {
      T_ref.Element_e1 all_1 optional
   } with {
      encode "{ref}c1";
   }

   type union ComplexType_c2 {
      T_ref.Element_e1 choice_1
   } with {
      encode "{ref}c2";
   }

   type record ComplexType_c3 {
      T_ref.Element_e1 sequence_1
   } with {
      encode "{ref}c3";
   }
}
```

Figure 3.7: Mapping of the *ref* attribute

3.3.5 Mapping of the XML Schema *nillable* Attribute

The *nillable* attribute, which is allowed to occur on element declaration components, is of great importance for the validation of XML documents. If the *nillable* attribute is present on an element declaration and set to *true*, "then an element may also be ·valid· if it carries the namespace qualified attribute with [local name] *nil* from namespace *http://www.w3.org/2001/XMLSchema-instance* and value *true* (see xsi:nil (§2.6.2)) even if it has no text or element content despite a {content type} which would otherwise require content" ([39] ch. 3.3.1). Because the TTCN-3 Standard Part 9 does not specify a mapping for the *nillable* attribute, an additional rule is introduced by the mapping presented in this thesis.

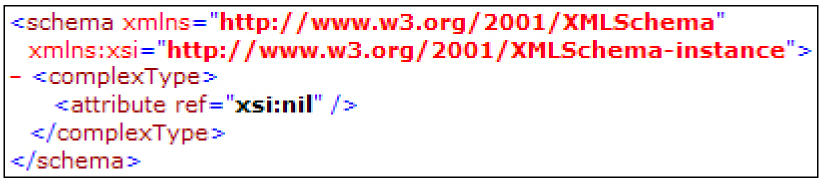


```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="nil" targetNamespace="nil">
  <element name="e1" type="xsd:int"
    nillable="true" />
- <complexType name="c1">
  - <sequence>
      <element name="e2" type="xsd:int"
        nillable="true" />
    </sequence>
  </complexType>
  <element name="e3" type="tns:c1"
    nillable="true" />
</schema>
```

```
module T_nil {
   import from XSDAUX all;

   type union Element_e1 {
      XSDAUX.int notNil,
      T_nil.ComplexType_1 nil
   } with {
      encode "{nil}e1";
   }

   type set ComplexType_1 {
      XSDAUX.Attribute_nil
         attribute_1,
   } with {
      encode "{nil}1";
      extension "Attribute:
         attribute_1";
   }

   type record ComplexType_c1 {
      T_nil.Element_2 e_e2
   } with {
      encode "{nil}c1";
      encode (e_e2) "e2";
   }

   type union Element_2 {
      XSDAUX.int notNil,
      T_nil.ComplexType_1 nil
   } with {
      encode "{nil}2";
   }

   type union Element_e3 {
      T_nil.ComplexType_c1 notNil,
      T_nil.ComplexType_1 nil
   } with {
      encode "{nil}e3";
   }
}
```

Figure 3.8: Mapping of the *nillable* Attribute

An element declaration with a present *nillable* attribute having the value *true* always maps to a TTCN-3 union with exactly two variants, as illustrated by the three exemplary mappings displayed in Figure 3.8. The first variant named *notNil* is of the TTCN-3 type that represents the type of the element declaration component. Thus, it allows specifying normal text or element content according to the content type of the element declaration. The second variant is named *nil* and it allows specifying that the element only carries a namespace qualified *nil* attribute, as defined by the XML Schema Standard Part 1. For that purpose, the second variant is of an anonymous TTCN-3 set type that represents the following hypothetical complex type definition.



```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <complexType>
    <attribute ref="xsi:nil" />
  </complexType>
</schema>
```

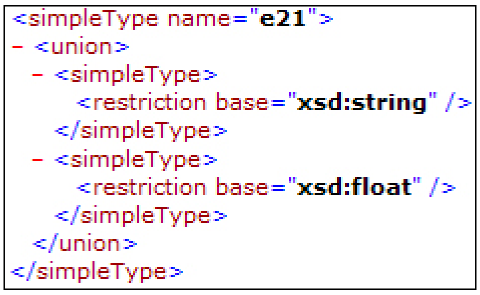
Figure 3.9: Hypothetical complex type definition with an *xsi:nil* attribute

In the exemplary mapping shown in Figure 3.8, this anonymous TTCN-3 set type is named *ComplexType_1* and commonly used in all TTCN-3 union types representing an element declaration component with a *nillable* attribute. In order to keep the number of additional types as low as possible, it is recommended but not mandatory to define only one TTCN-3 set type of that kind per module.

3.3.6 Mapping of the Simple Type Definition Schema Component

The mapping of simple type definitions to TTCN-3 is thoroughly defined in Section 7.5 of the TTCN-3 Standard Part 9. Yet, the specified rules contain an inconsistency that is corrected by the mapping presented in this thesis.

The TTCN-3 Standard Part 9 illustrates the specified rules with various examples of mapping simple type definitions derived by restriction, list, or union. All identifiers of the resulting TTCN-3 definitions except one end with the suffix *__simpleType*. The exception is a simple type definition derived by union that maps to a TTCN-3 definition whose identifier ends with the suffix *__union* (s. [9] ch. 7.5.3). Since this behavior is inconsistent and not motivated by the TTCN-3 Standard Part 9, the mapping in question is corrected. Following the naming rules of the mapping presented in this thesis, all identifiers of the resulting TTCN-3 definitions start with the prefix *SimpleType_*. Figure 3.10 shows the simple type definition in question as well as the inconsistent and the corrected TTCN-3 mapping results.



```
<simpleType name="e21">
- <union>
  - <simpleType>
      <restriction base="xsd:string" />
    </simpleType>
  - <simpleType>
      <restriction base="xsd:float" />
    </simpleType>
  </union>
</simpleType>
```

```
type union e21__union {
   XSDAUX.string union_1,
   XSDAUX.float_ union_2
}
```

```
module T_union {
   import from XSDAUX all;

   type union SimpleType_e21 {
      XSDAUX.string union_1,
      XSDAUX.float_ union_2
   } with {
      encode "{union}e21";
   }
}
```

Figure 3.10: Mapping a simple type definition derived by union (cp. [9] ch. 7.5.3)

3.3.7 Mapping of the Complex Type Definition Schema Component

The mapping of complex type definitions to TTCN-3 is described in detail in Section 7.6 of the TTCN-3 Standard Part 9, but again some of the specified rules are changed by the mapping presented in this thesis.

First, the mapping of a complex type definition with complex content, which is derived by extension, is unified. A component of this kind "is translated to TTCN-3 by creating a record containing a reference to the base type using the reserved word "base" and the extension components" ([9] ch. 7.6.2.1). Two alternatives are defined for the mapping of the extension content. If the base content and the extension content are of the same structure e.g. a sequence, the extension content should map directly to the resulting TTCN-3 record type. If both are of different structures, an anonymous type is generated and referenced by a second field with the reserved name "content". The mapping of the extension content is unified by disabling the first alternative for the following reasons: First, the unification simplifies the mapping, which is considered more useful than the slight decrease in complexity of the TTCN-3 output. Second, the precondition of the first alternative is not always suitable. Consider an example where the base content and the extension content define a choice, so that both are of the same structure and the extension content should map directly to the resulting TTCN-3 record type. This would be an unsuitable and inconsistent mapping, since it is specified that choice content maps to a TTCN-3 union type (s. [9] ch. 7.6.5). To illustrate the preceding explanations, Figure 3.11 shows the complex type definition in question together with the mapping result given by the standard and the changed mapping result.

```
type record e25__complexType {
   XSDAUX.string title,
   XSDAUX.string forename,
   XSDAUX.string surname
}

type record e26__complexType {
   e25_complexType base,
   XSDAUX.integer_ age
}
```

```
<complexType name="e25">
- <sequence>
    <element name="title" type="xsd:string" />
    <element name="forename" type="xsd:string" />
    <element name="surname" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="e26">
- <complexContent>
  - <extension base="tns:e25">
    - <sequence>
        <element name="age" type="xsd:integer" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
module T_extension {
   import from XSDAUX all;

   type record ComplexType_e25 {
      XSDAUX.string e_title,
      XSDAUX.string e_forename,
      XSDAUX.string e_surname
   } with {
      encode "{extension}e25";
      encode (e_title) "title";
      encode (e_forename) "forename";
      encode (e_surname) "surname";
   }

   type record ComplexType_e26 {
      T_extension.ComplexType_e25 base,
      T_extension.Sequence_1 content
   } with {
      encode "{extension}e26";
   }

   type record Sequence_1 {
      XSDAUX.integer_ e_age
   } with {
      encode "{extension}1";
      encode (e_age) "age";
   }
}
```

Figure 3.11: Mapping complex content derived by extension (cp. [9] ch. 7.6.2.1)

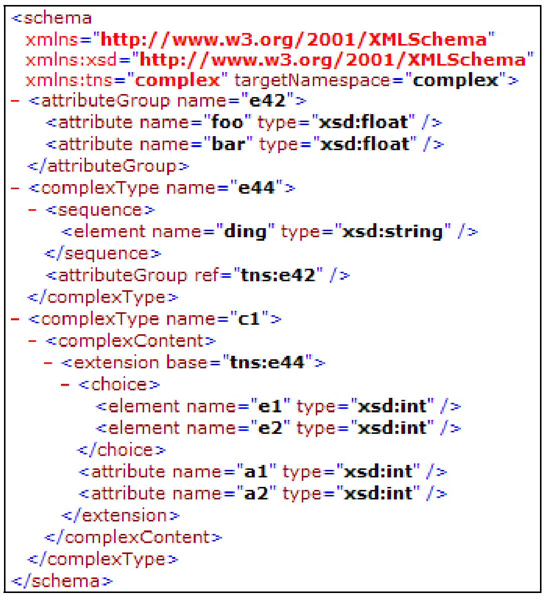
Besides the discussed unification, the mapping of an attribute group that is referenced from within a complex type definition is corrected. The TTCN-3 Standard Part 9 states that "if *attributeGroup* components are referenced from a *complexType*, *restriction*, or *extension*, a reference to the *attributeGroup* is generated and inserted in the mapped construct" ([9] ch. 7.6.7). To exemplify this mapping rule, the TTCN-3 Standard Part 9 includes the example shown in Figure 3.12.

```
<attributeGroup name="e42">
  <attribute name="foo" type="xsd:float" />
  <attribute name="bar" type="xsd:float" />
</attributeGroup>
<complexType name="e44">
- <sequence>
    <element name="ding" type="xsd:string" />
  </sequence>
  <attributeGroup ref="tns:e42" />
</complexType>
```

```
type set e42__attributeGroup {
   XSDAUX.float_ foo optional,
   XSDAUX.float_ bar optional
} with {
   extension "AttributeGroup"
};

type set e44__complexType {
   XSDAUX.string ding,
   e42__attributeGroup attributeGroup_1
}
```

Figure 3.12: Mapping of a referenced attribute group (s. [9] ch. 7.6.7)

The resulting TTCN-3 definition *e44__complexType* is thereby inconsistent with regard to the mapping rules for sequence content of a complex type definition, which specify that such content maps to a TTCN-3 record type (s. [9] ch. 7.6.6). Because mapping the referenced attribute group to a field inside a set type is nevertheless reasonable, this is retained. The sequence content is however mapped to an anonymous record type, which is referenced by a field of the set type that has the reserved name "base". The corrected mapping result for the complex type definition in question is shown in Figure 3.13.



```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="complex" targetNamespace="complex">
- <attributeGroup name="e42">
    <attribute name="foo" type="xsd:float" />
    <attribute name="bar" type="xsd:float" />
  </attributeGroup>
- <complexType name="e44">
  - <sequence>
      <element name="ding" type="xsd:string" />
    </sequence>
    <attributeGroup ref="tns:e42" />
  </complexType>
- <complexType name="c1">
  - <complexContent>
    - <extension base="tns:e44">
      - <choice>
          <element name="e1" type="xsd:int" />
          <element name="e2" type="xsd:int" />
        </choice>
        <attribute name="a1" type="xsd:int" />
        <attribute name="a2" type="xsd:int" />
      </extension>
    </complexContent>
  </complexType>
</schema>
```

```
module T_complex {
   import from XSDAUX all;

   type set AttributeGroup_e42 {
      XSDAUX.float_ a_foo optional,
      XSDAUX.float_ a_bar optional
   } with {
      encode "{complex}e42";
      extension "AttributeGroup";
      encode (a_foo) "foo";
      encode (a_bar) "bar";
   }

   type set ComplexType_e44 {
      T_complex.Sequence_1 base,
      T_complex.AttributeGroup_e42
         attributeGroup_1
   } with {
      encode "{complex}e44";
   }

   type record Sequence_1 {
      XSDAUX.string e_ding
   } with {
      encode "{complex}1";
      encode (e_ding) "ding";
   }

   type set ComplexType_c1 {
      T_complex.Extension_2 base,
      XSDAUX.int a_a1 optional,
      XSDAUX.int a_a2 optional
   } with {
      encode "{complex}c1";
      extension "Attribute: a_a1, a_a2";
      encode (a_a1) "a1";
      encode (a_a2) "a2";
   }

   type record Extension_2 {
      T_complex.ComplexType_e44 base,
      T_complex.Choice_3 content
   } with {
      encode "{complex}2";
   }

   type union Choice_3 {
      XSDAUX.int e_e1,
      XSDAUX.int e_e2
   } with {
      encode "{complex}3";
      encode (e_e1) "e1";
      encode (e_e2) "e2";
   }
}
```

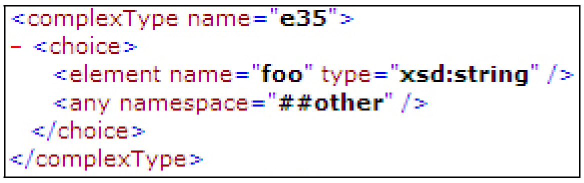
Figure 3.13: Mapping of nested *attribute* and *attributeGroup* elements

The corrected mapping rule furthermore applies to all cases where an *attributeGroup*, *attribute*, or *anyAttribute* element is contained directly in a complex type definition or in nested complex content derived by restriction or extension. In any case, a TTCN-3 set type is created that contains a field for each *attributeGroup*, *attribute*, or *anyAttribute* element as well as a field of the anonymous type to which the complex type definition or the complex content maps. The preceding explanations are illustrated in Figure 3.13, which shows the exemplary mapping of the complex type definition *c1*.

Finally, the mapping of *any* elements, which are contained in a complex type definition with choice content, is changed. The *any* element is a so-called wildcard component that facilitates element content according to namespaces i.e. only the namespace to which an element belongs is restricted. For example, the *any* element could permit the appearance of arbitrary XHTML elements. In the TTCN-3 Standard Part 9, the following mapping is specified:

> "As the TTCN-3 *anytype* is defined to be a union of all types of the present module and every other imported module, a *choice* containing XSD *any* types will translate to *anyType* in TTCN-3." ([9] ch. 7.6.5.5)

That is, a plain subtype of the TTCN-3 *anytype* is created regardless of other definitions contained in the choice content. Figure 3.14 displays the exemplary mapping of the complex type *e35* as given in Section 7.6.5.5 of the TTCN-3 Standard Part 9.



```
<complexType name="e35">
- <choice>
    <element name="foo" type="xsd:string" />
    <any namespace="##other" />
  </choice>
</complexType>
```

```
type anytype e35__complexType;
```

```
module T_any {
   import from XSDAUX all;

   type union ComplexType_e35 {
      XSDAUX.string e_foo,
      anytype choice_1
   } with {
      encode "{any}e35";
      encode (e_foo) "foo";
   }
}
```

Figure 3.14: Mapping an *any* element (cp. [9] ch. 7.6.5.5)

The mapping to a subtype of the TTCN-3 *anytype* seems reasonable at first glance, but it is not suitable. None of the local definitions, which are also contained in the choice content e.g. the element *foo* in the above example, are part of the special union type represented by the TTCN-3 *anytype*, because they do not map to a type in the present or any imported module. Thus, the mapping of the *any* element is adapted to that of other definitions contained in the choice content of a complex type definition i.e. it is mapped to an additional variant that is of the TTCN-3 *anytype*. The name of the variant becomes the concatenation of *choice_* and a sequential number that must be unique amongst all variants of the enclosing union type. The changed mapping is shown in Figure 3.14.

3.3.8 Mapping of the Annotation Schema Component

The mapping of the annotation schema component is only partially specified in the TTCN-3 Standard Part 9:

> "Annotations may appear in every component and will be mapped to a corresponding comment in TTCN-3. The comment should appear in the TTCN-3 code just before the mapped structure that it belongs to. This standard does not describe a format in which the comment is to be put into the TTCN-3 code." ([9] ch. 7.7)

In the following, the specification given above is refined by defining concrete rules for the mapping of the annotation schema component to a TTCN-3 comment.

An annotation is not only allowed to appear in other schema components but can also appear anywhere at the top level of schemas, as specified in Figure 3.15. In case the annotation schema component appears before *include*, *import*, or *redefine* elements, it supposedly provides information on the schema. Therefore, it is mapped to a TTCN-3 comment before the TTCN-3 module that represents the schema. In case the annotation occurs elsewhere on the top level, it will be mapped to a TTCN-3 comment in front of the TTCN-3 representation of the first subsequent schema component (a *simpleType*, *complexType*, *group*, *attributeGroup*, *element*, *attribute*, or *notation* element). In case multiple annotation schema components are mapped to a comment in front of the same TTCN-3 construct e.g. a module, the created comments should preferably be merged to a single block comment wherein each of them starts in a new line.

```
<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  blockDefault = (#all | List of (extension | restriction | substitution))  : ''
  elementFormDefault = (qualified | unqualified) : unqualified
  finalDefault = (#all | List of (extension | restriction | list | union))  : ''
  id = ID
  targetNamespace = anyURI
  version = token
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)*, (((simpleType | complexType |
    group | attributeGroup) | element | attribute | notation), annotation*)*)
</schema>
```

Figure 3.15: XML representation of the *schema* element (s. [39] ch. 3.15.2)

The annotation schema component is described in Section 3.13 of the XML Schema Standard Part 1 and its XML representation is shown in Figure 3.16. The *annotation* schema component is allowed to contain an arbitrary number of *documentation* and/or *appinfo* child elements that are furthermore permitted to occur in any order. The nested elements are very similar and therefore translated to analog string representations. Each of those resulting string representations starts in a new line inside the TTCN-3 comment to which the enclosing annotation schema component maps.

```
<annotation
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</annotation>
```

```
<appinfo
  source = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</appinfo>
```

```
<documentation
  source = anyURI
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</documentation>
```

Figure 3.16: XML representation of the *annotation* element (s. [39] ch. 3.13.2)

The string representation of the *appinfo* and *documentation* elements is similar to their XML representation. The unchanged start and end tags are copied into subsequent lines. The value of each nested text node is put into a new line between the matching element tags, whereupon the value is indented by two space characters (*#x20*). If the text node value contains any line breaks, the following white space characters are replaced with two space characters, so the text node value is consistently indented in the resulting string representation. The child elements of the *appinfo* and the *documentation* elements are processed in the same manner as the elements themselves and the resulting string representations are also inserted into a new line between the matching element tags and indented by two space characters.

To conclude this section, Figure 3.17 illustrates the preceding explanations with some examples. The first annotation schema component is mapped to a module comment, since it appears in front of an *include* element. The contained text node value stretches across multiple lines, which gets lost in the schema representation shown on the left, but is preserved in the resulting TTCN-3 comment. The second annotation is more complex as it has attributes as well as a nested element. It is mapped to a comment in front of the TTCN-3 representation for the subsequent complex type definition named *c1*. Inside the final complex type definition named *c2*, multiple annotations appear. Each of them is mapped to a comment in front of the TTCN-3 definition that represents the structure to which it belongs. Hence, the two first annotations map to a block comment in front of the TTCN-3 record type *ComplexType_c2*, whereas the innermost annotation schema component becomes a simple line comment in front of the *Sequence_1* record type.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:t="annotation" targetNamespace="annotation">
- <annotation>
    <appinfo>global, before include, multiline</appinfo>
  </annotation>
  <include schemaLocation="EmptySchema.xsd" />
- <annotation>
  - <documentation source="anyURI" xml:lang="en">
      global, after include,
      <nestedElement>nested element</nestedElement>
      source and xml:lang attribute
    </documentation>
  </annotation>
- <complexType name="c1">
  - <sequence>
      <element name="e1" type="xsd:int" />
    </sequence>
  </complexType>
- <complexType name="c2">
  - <annotation>
      <appinfo t:local="complexType" />
    </annotation>
  - <complexContent>
    - <extension base="t:c1">
      - <annotation>
          <appinfo t:local="extension" />
        </annotation>
      - <sequence>
        - <annotation>
            <appinfo t:local="sequence" />
          </annotation>
          <element name="e2" type="xsd:int" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

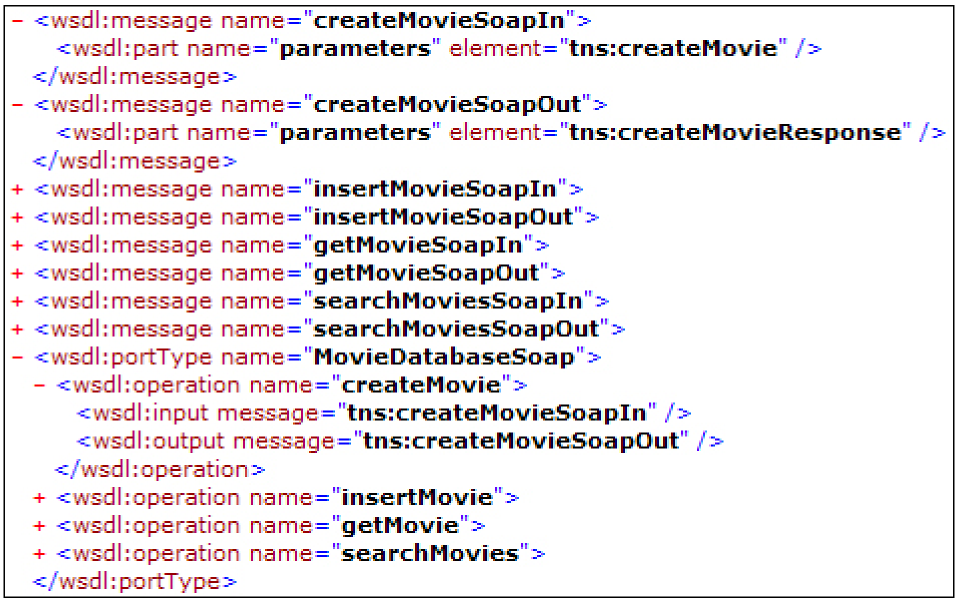
```
/*
 * <appinfo>
 *   global, before include
 *   multiline
 * </appinfo>
 */
module T_annotation {
   import from XSDAUX all;

   /*
    * <documentation source="anyURI"
         xml:lang="en">
    *   global, after include
    *   <nestedElement>
    *     nested element
    *   </nestedElement>
    *   source and xml:lang attribute
    * </documentation>
    */
   type record ComplexType_c1 {
      XSDAUX.int e_e1
   } with {
      encode "{annotation}c1";
      encode (e_e1) "e1";
   }

   /*
    * <appinfo t:local="complexType" />
    * <appinfo t:local="extension" />
    */
   type record ComplexType_c2 {
      T_annotation.ComplexType_c1 base,
      T_annotation.Sequence_1 content
   } with {
      encode "{annotation}c2";
   }

   // <appinfo t:local="sequence" />
   type record Sequence_1 {
      XSDAUX.int e_e2
   } with {
      encode "{annotation}1";
      encode (e_e2) "e2";
   }
}
```

Figure 3.17: Mapping of the *annotation* element

3.4 Mapping of the WSDL *portType* Element

The overall mapping rules specify that the WSDL *portType* element maps to a module, which contains a record type definitions for each input, output, and fault message of the operations defined by the WSDL port type as well as a TTCN-3 port type definition that allows sending or receiving messages of the above record types. Figure 3.18 shows an extract from the WSDL document of the .NET implementation of the "Movie Database"

Web service. It contains all elements that are relevant for the current mapping step and serves as an example along the discussion of the detailed mapping rules. Some elements are collapsed in order to shorten the document, but their content is comparable to that of expanded elements of the same type.



```
- <wsdl:message name="createMovieSoapIn">
    <wsdl:part name="parameters" element="tns:createMovie" />
  </wsdl:message>
- <wsdl:message name="createMovieSoapOut">
    <wsdl:part name="parameters" element="tns:createMovieResponse" />
  </wsdl:message>
+ <wsdl:message name="insertMovieSoapIn">
+ <wsdl:message name="insertMovieSoapOut">
+ <wsdl:message name="getMovieSoapIn">
+ <wsdl:message name="getMovieSoapOut">
+ <wsdl:message name="searchMoviesSoapIn">
+ <wsdl:message name="searchMoviesSoapOut">
- <wsdl:portType name="MovieDatabaseSoap">
  - <wsdl:operation name="createMovie">
      <wsdl:input message="tns:createMovieSoapIn" />
      <wsdl:output message="tns:createMovieSoapOut" />
    </wsdl:operation>
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:portType>
```

Figure 3.18: WSDL extract showing *portType* and *message* elements

For every WSDL port type, a TTCN-3 module is created whose identifier becomes the concatenation of the prefix specified in Table 3.2 and the value of the *name* attribute on the port type. This identifier is ensured to be unique, because of the prefix and the fact that the "port type *name* attribute provides a unique name among all port types defined within in the enclosing WSDL document" ([37] ch. 2.4). In the current example, a single module named *PT_mdb_MovieDatabaseSoap* is created, whereat *mdb* denotes a user-defined target namespace qualifier.

In the modules, a TTCN-3 group is defined for each *operation* element of the respective WSDL port type. In general, the group identifier becomes the prefix *g_* with the value of the *name* attribute on the *operation* element being appended. Because the WSDL 1.1 Standard allows multiple WSDL operations with equivalent names, this naming rule is overridden, if and only if there are equally named operations. In this case, the group identifier is composed of the prefix, the operation name, an underscore, and the name of the enclosed *input* element, which has to be unique among all *input* elements within the enclosing WSDL port type. This special case creates a rather long identifier, but it is considered to occur rarely. Furthermore, it is completely forbidden by the WS-I Basic

Profile Version 1.0 and 1.1, which both require a WSDL port type to have operations with distinct values for their *name* attributes (s. [44] ch. 5.4.3 and [45] ch. 4.5.3). In the current example, four groups named *g_createMovie*, *g_insertMovie*, *g_getMovie*, and *g_searchMovies* are defined. A hypothetical example for a composed group identifier is *g_createMovie_createMovieRequest*.

Inside of each group, a TTCN-3 record type is defined for every *input*, *output*, and *fault* element of the respective WSDL operation. The identifiers of these record types are the concatenation of the proper prefix *I_*, *O_*, or *F_* and a base part, which differs for *input* and *output* elements on the one hand and *fault* elements on the other hand. In the first case, the identifier base part can be obtained from the *name* attribute value, because it "provides a unique name among all input and output elements within the enclosing port type" ([37] ch. 2.4.5). In case the *name* attribute is not specified on the *input* or *output* element as in the current example, the WSDL 1.1 Standard specifies that its value is by default the name of the enclosing operation with either "Request" or "Response" being appended. In contrast, the *name* attribute of a *fault* element is required to be present but only unique within the set of faults that are defined for the enclosing operation. Thus, the identifier base part of the corresponding TTCN-3 record type will be the name of the enclosing operation with the name of the fault being appended. As aforementioned, the operation name is not required to be unique, so in case of multiple *operation* elements having equal *name* attributes, the operation name is first made unique by appending an underscore and the name of the enclosed *input* element. In the current example, the *input* and *output* elements of the WSDL *createMovie* operation map to the TTCN-3 record types *I_createMovieRequest* and *O_createMovieResponse*.

The first field of each TTCN-3 record type representing an *input* element is named *soapbinding* and of the TTCN-3 *SoapBinding* type predefined inside the *WebServices* module. The purpose of this field is to accompany each input message with the proper SOAP binding information that is necessary to make a call to the Web service upon test execution.

The other fields of the TTCN-3 record types are derived from the WSDL message that is referenced by the *input*, *output*, or *fault* element to which they correspond. Each *part* element of a WSDL message is mapped to a field whose name is the concatenation of the previously specified prefix *mp_* and the value of the *name* attribute, which "provides a unique name among all the parts of the enclosing message" ([37] ch. 2.3). The type of the field is the TTCN-3 representation of the XML Schema element declaration or type definition, which is specified by the *element* or *type* attribute on the *part* element. If an XML Schema built-in data type is referenced, the TTCN-3 representation is defined in the *XSDAUX* module. Otherwise, it has been generated according to the rules discussed in Section 3.3. In the current example, both record types contain a single field called *mp_parameters* that is of the generated *createMovie* or *createMovieResponse* type.

Finally, each TTCN-3 module representing a WSDL port type contains a TTCN-3 port type, which uses message-based communication and allows sending or receiving the defined record types. The identifier of the TTCN-3 port type is the concatenation of the prefix *TP_* and the value of the *name* attribute of the WSDL port type, so in the current example it is named *TP_MovieDatabaseSoap*.

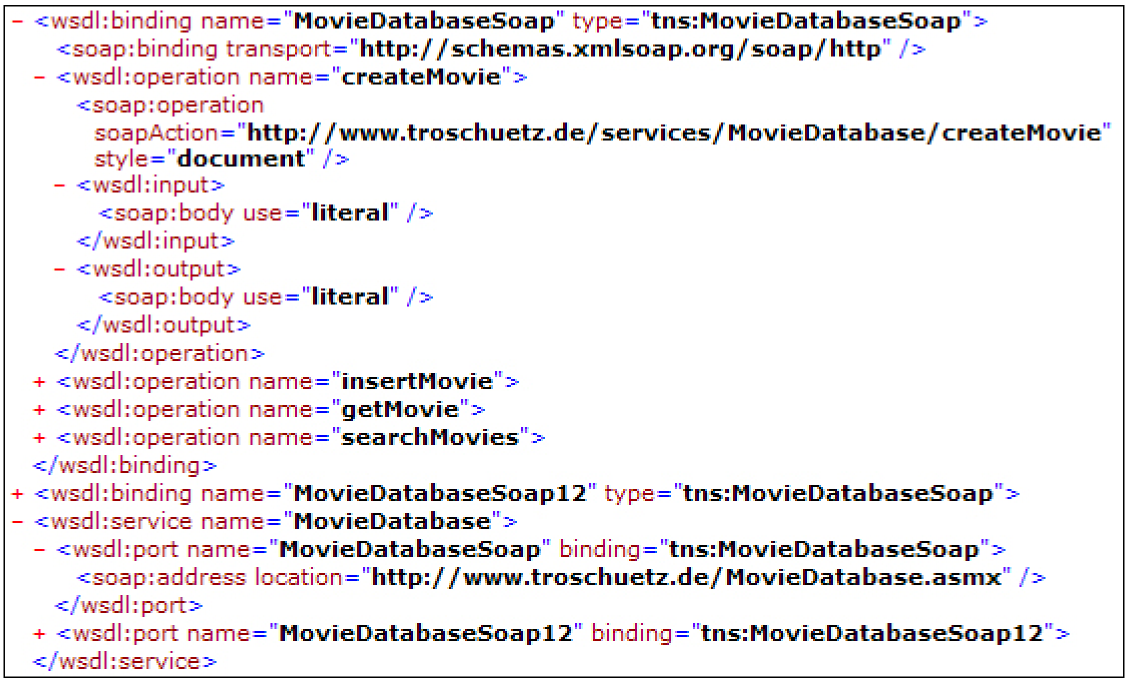
To conclude this section, Figure 3.19 lists the mapping result for the exemplary WSDL port type. In order to shorten the figure, the TTCN-3 groups corresponding to WSDL operations, which are collapsed in the WSDL description (s. Figure 3.18), are likewise collapsed in Figure 3.19.

```
module PT_mdb_MovieDatabaseSoap {
   import from WebServices all;
   import from T_mdb all;

   type port TP_MovieDatabaseSoap message {
      out PT_mdb_MovieDatabaseSoap.I_createMovieRequest;
      in PT_mdb_MovieDatabaseSoap.O_createMovieResponse;
      out PT_mdb_MovieDatabaseSoap.I_insertMovieRequest;
      in PT_mdb_MovieDatabaseSoap.O_insertMovieResponse;
      out PT_mdb_MovieDatabaseSoap.I_getMovieRequest;
      in PT_mdb_MovieDatabaseSoap.O_getMovieResponse;
      out PT_mdb_MovieDatabaseSoap.I_searchMoviesRequest;
      in PT_mdb_MovieDatabaseSoap.O_searchMoviesResponse;
   }

   group g_createmovie {
      type record I_createMovieRequest {
         WebServices.SoapBinding soapBinding,
         T_mdb.Element_createMovie mp_parameters
      }

      type record O_createMovieResponse {
         T_mdb.Element_createMovieResponse mp_parameters
      }
   }

   group g_insertMovie {...}

   group g_getMovie {...}

   group g_searchMovies {...}
}
```

Figure 3.19: TTCN-3 output of mapping the WSDL *portType* element

3.5 Mapping of the WSDL *port* and *binding* Elements

The overall mapping rules specify that a WSDL *port* and its associated *binding* element map to a module, which contains TTCN-3 constants with SOAP binding information for every described operation. Figure 3.20 displays an extract from the WSDL document of the .NET implementation of the "Movie Database" Web service that again shows all elements relevant for the current mapping step and will serve as an example along the discussion of the detailed mapping rules. As in Section 3.4, the document is shortened by collapsing some elements whose content is similar to that of expanded elements of the same type.



```
- <wsdl:binding name="MovieDatabaseSoap" type="tns:MovieDatabaseSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  - <wsdl:operation name="createMovie">
      <soap:operation
       soapAction="http://www.troschuetz.de/services/MovieDatabase/createMovie"
       style="document" />
    - <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    - <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:binding>
+ <wsdl:binding name="MovieDatabaseSoap12" type="tns:MovieDatabaseSoap">
- <wsdl:service name="MovieDatabase">
  - <wsdl:port name="MovieDatabaseSoap" binding="tns:MovieDatabaseSoap">
      <soap:address location="http://www.troschuetz.de/MovieDatabase.asmx" />
    </wsdl:port>
  + <wsdl:port name="MovieDatabaseSoap12" binding="tns:MovieDatabaseSoap12">
  </wsdl:service>
```

Figure 3.20: WSDL extract showing *service*, *port*, and *binding* elements

For each WSDL port and its associated binding, a TTCN-3 module is created whose identifier is composed of the prefix specified in Table 3.2 and the value of the *name* attribute on the *port* element. This identifier is ensured to be unique, because of the prefix and the fact that the "*name* attribute provides a unique name among all ports defined within in the enclosing WSDL document" ([37] ch. 2.6). In the current example, two modules named *P_mdb_MovieDatabaseSoap* and *P_mdb_MovieDatabaseSoap12* are generated, whereupon *mdb* denotes a user-defined target namespace qualifier.

Inside the modules, constants are created that hold information, which is common to all operations defined by the respective WSDL port and its associated binding. The first constant called c_*soapVersion* is of the predefined *SoapVersion* type and holds a string that reflects the SOAP binding version used by the WSDL port and binding. The second constant is named *c_location* and of the predefined *anyURI* type. It contains the value of the *location* attribute of the *soap:address* element. The third constant, which is named *c_transport* and of the *anyURI* type, obtains its value from the *transport* attribute on the *soap:binding* element. In case the optional *use* attribute is specified on the *soap:binding* element, the attribute value is assigned to a fourth constant, which is of the predefined *SoapStyle* type and named *c_Style*. In the current example, all constants but the fourth are generated.

After defining the constants, which hold common SOAP binding information, further constants of the predefined *SoapBinding* record type are declared for each operation specified by the WSDL port and its associated binding. In general, their identifier is simply composed of the specified prefix *b_* and the value of the *name* attribute on the *operation* element. Because the "operation element within a binding specifies binding information for the operation with the same name within the binding's portType" ([37] ch. 2.5) and "operation names are not required to be unique" ([37] ch. 2.5), the given naming rule is overridden, if and only if there are equally named operations. In this case, the group identifier is composed of the specified prefix, the operation name, an underscore, and the name of the enclosed *input* element, which is unique and must now be specified to identify the operation unambiguously. In the current example, the four constants named *b_createMovie*, *b_insertMovie*, *b_getMovie*, and *b_searchMovies* are created. An example for a composed identifier is *b_createMovie_createmovieRequest*.

The value for each constant of the *SoapBinding* record type is composed of references to the constants holding common SOAP binding information and the invocation details specified by the attributes on the SOAP binding elements. If an optional SOAP binding detail is not specified, the special TTCN-3 value "`omit`" is assigned to the corresponding field of a constant. The value for the *style* field of the *SoapBinding* record type poses a special case. It becomes the value specified by the *style* attribute on the *soap:operation* element, if present. In case the *soap:operation* element specifies no *style* attribute but the *soap:binding* element does, it becomes a reference to the previously created *c_style* constant. If none of the preceding cases applies, the value for the *style* field becomes the assumed default value "`document`" (s. [37] ch. 3.4 and [41] ch. 3.2).

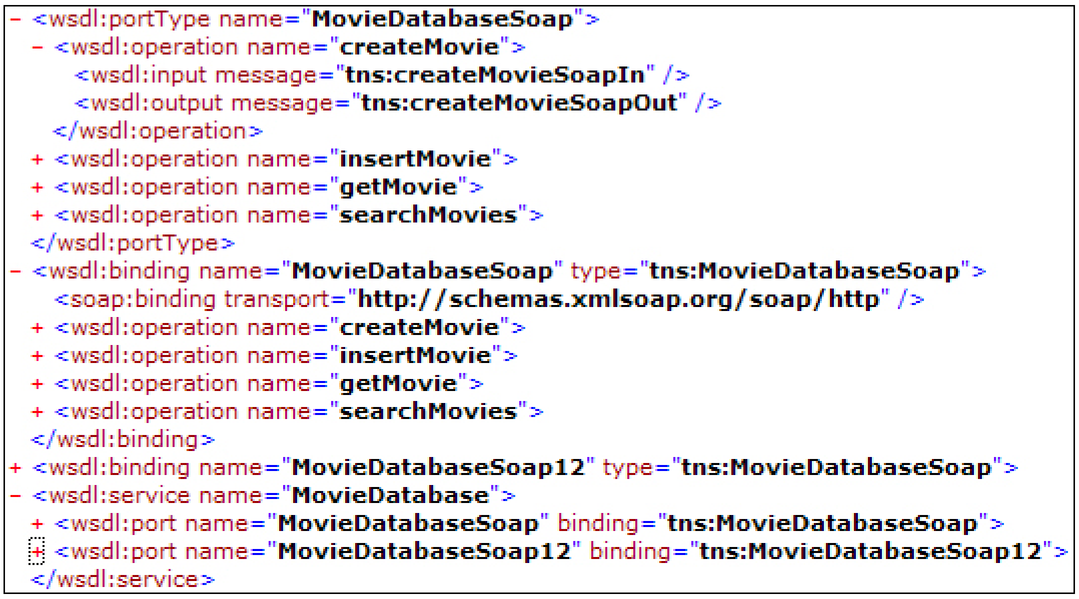
```
module P_mdb_MovieDatabaseSoap {
   import from XSDAUX all;
   import from WebServices all;

   const WebServices.SoapVersion c_soapVersion := "V_11";

   const XSDAUX.anyURI c_location := "http://www.troschuetz.de/MovieDatabase.asmx";

   const XSDAUX.anyURI c_transport := "http://schemas.xmlsoap.org/soap/http";

   const WebServices.SoapBinding b_createMovie := {
      operationName :=
         "{http://www.troschuetz.de/services/MovieDatabase}createMovie",
      soapVersion := P_MovieDatabaseSoap.c_soapVersion,
      location := P_MovieDatabaseSoap.c_location,
      transport := P_MovieDatabaseSoap.c_transport,
      style := "document",
      soapAction := "http://www.troschuetz.de/services/MovieDatabase/createMovie",
      soapActionRequired := omit,
      inputBody := {
         use := "literal",
         parts := omit,
         encodingStyles := omit,
         namespace := omit
      },
      inputHeaders := omit,
      outputBody := {
         use := "literal",
         parts := omit,
         encodingStyles := omit,
         namespace := omit
      },
      outputHeaders := omit,
      faults := omit
   };

   const WebServices.SoapBinding b_insertMovie := {...};

   const WebServices.SoapBinding b_getMovie := {...};

   const WebServices.SoapBinding b_searchMovies := {...};
}
```

Figure 3.21: TTCN-3 output of mapping the WSDL *port* and *binding* elements

Figure 3.21 lists the mapping result for the exemplary WSDL port *MovieDatabaseSoap* and its binding *MovieDatabaseSoap*, which are both expanded in the initially shown Figure 3.20. To shorten the figure, only the constant corresponding to the *createMovie* operation, which is expanded in Figure 3.20, is completely shown.

3.6 Mapping of the WSDL *service* Element

The overall mapping rules specify that the WSDL *service* element maps to a collection of modules. Each module contains basic test behavior and execution control for testing a WSDL port type referenced by the service via port and binding. Figure 3.22 displays an extract of the WSDL description of the .NET implementation of the exemplary "Movie Database" Web service containing all elements that are relevant for the current mapping step. It exemplifies the following discussion of the detailed mapping rules. As in the Sections 3.4 and 3.5, the document is shortened by collapsing some elements.



```
- <wsdl:portType name="MovieDatabaseSoap">
  - <wsdl:operation name="createMovie">
      <wsdl:input message="tns:createMovieSoapIn" />
      <wsdl:output message="tns:createMovieSoapOut" />
    </wsdl:operation>
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:portType>
- <wsdl:binding name="MovieDatabaseSoap" type="tns:MovieDatabaseSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  + <wsdl:operation name="createMovie">
  + <wsdl:operation name="insertMovie">
  + <wsdl:operation name="getMovie">
  + <wsdl:operation name="searchMovies">
  </wsdl:binding>
+ <wsdl:binding name="MovieDatabaseSoap12" type="tns:MovieDatabaseSoap">
- <wsdl:service name="MovieDatabase">
  + <wsdl:port name="MovieDatabaseSoap" binding="tns:MovieDatabaseSoap">
  + <wsdl:port name="MovieDatabaseSoap12" binding="tns:MovieDatabaseSoap12">
  </wsdl:service>
```

Figure 3.22: WSDL extract showing *service*, *port*, *binding*, and *portType* elements

For every WSDL port type, which is referenced by the *service* via the enclosed *port* and their associated *binding* elements, a TTCN-3 module is created. The name of a module is the value of the *name* attribute on the respective *portType* element prepended with the prefix specified in Table 3.2. The identifiers are guaranteed to be distinctive, because of the prefix and the fact that the "*name* attribute provides a unique name among all port types defined within in the enclosing WSDL document" ([37] ch. 2.4). In the current example, a single module named *TPT_mdb_MovieDatabaseSoap* is generated, whereat *mdb* denotes a user-defined target namespace qualifier.

Each module defines a single test component with a single port on which the behavior for testing the WSDL port type, to which the module corresponds, will be executed. The identifiers of both become the name of the WSDL port type appended to the respective prefix specified in Table 3.2. The single port describes the test component's interface and is of the TTCN-3 port type that allows transmission and receipt of the messages defined by the WSDL port type and that is already defined in the module representing the WSDL port type (s. ch. 3.4). Besides the port, the test component also contains a timer named *t_timeoutguard*, which will be used to safeguard the test behavior against inactivity of the Web service under test. In the current example, the test component and its port are named *TC_MovieDatabaseSoap* and *pt_MovieDatabaseSoap*.

Apart from the test component, there are two more definitions on the module level. On the one hand, there is the *c_timeout* constant, which defines a common timeout for communication with the Web service to be tested and is therefore referenced whenever the previously mentioned timer is started. On the other hand, there is an altstep named *alt_ReceiveAnyOrTimeout*, which defines common reactions on faulty communication with the Web service to be tested e.g. the receipt of unexpected responses or inactivity. The altstep is defined to run on the already defined test component, whose port and timer can therefore be used in the altstep definition. It specifies two alternatives that set the test verdict to "`fail`" after catching either the receipt of any (unexpected) message on the port or a timeout of the timer.

Next, a TTCN-3 group is defined for every operation specified by the WSDL port type. Each group contains a test case and message templates that allow testing the operation. In general, the group identifier is simply the value of the *name* attribute on the *operation* element appended to the specified prefix *g_*. Because the WSDL 1.1 Standard allows WSDL operations to have identical names, the given naming rule is overridden, if and only if there are equally named operations. In this case, the group identifier consists of the prefix *g_*, the operation name, an underscore, and the name of the enclosed *input* element, which has to be unique among all *input* elements within the enclosing WSDL port type. In the current example, the *g_createMovie*, *g_insertMovie*, *g_getMovie*, and *g_searchMovies* groups are specified. A hypothetical example for a compound group identifier is *g_createMovie_createMovieRequest*.

The message templates defined inside each group correspond to the input, output, and fault messages specified by the respective WSDL operation. The type of each template is the TTCN-3 record type that is already defined for its corresponding input, output, or fault message in the module representing the WSDL port type (s. ch. 3.4). From the type of a template, its identifier can be easily derived by appending the type name to the prefix specified for templates. Because the mapping presented in this thesis does not cover the generation of test data, the values assigned to the message templates are only required to be valid with regard to the template type. Real test data has to be specified by hand before executing the test behavior that is defined by the mapping. However, the template values should preferably follow the rules listed in Table 3.6. The template corresponding to the input message of a WSDL operation poses a special case, because it possesses an input parameter, which is named *p_soapBinding* and of the predefined *SoapBinding* type. The parameter value is assigned to the *soapBinding* field, which is specified by every record type corresponding to a WSDL input message (s. ch. 3.4), and accompanies the message with the SOAP binding information that is necessary to make a call to the Web service upon test execution. In the current example, the two templates *a_I_createMovieRequest* and *a_O_createMovieResponse* are defined.

TTCN-3 type	**Template value**
boolean	`true`
integer	`0`
float	`0.0`
charstring	`""`
octetstring	`''O`
record set	A record/set value whose fields are set to • "`omit`", if the field is optional (this aims at avoiding values that are nested infinitely deep, because recursive fields i.e. those of the defining record/set type are expected to be optional for this reason) • A value of the field type that follows these rules, otherwise
set-of	If the set-of type has no length restriction, a set-of value containing one value of the element type that follows these rules. Otherwise, a set-of value containing the minimum number of elements required by the length restriction. Every element becomes a value of the element type that follows these rules.
union	A union value with its first variant set. The value assigned to the variant follows these rules.
subtype	A value according to the rule that first matches • The first value from the value list, if present • The minimum range value, if present and not "`-infinity`" • The maximum range value, if present and not "`infinity`" • A value of the base type that follows these rules

Table 3.6: Default values of TTCN-3 message templates

The test case defined in each group runs on the already specified test component, so the port and timer owned by the component can be used within the test case definition. The naming of the test cases follows the same rules as the naming of their defining groups, except the test case-specific prefix *tc_* is used. Furthermore, every test case has an input parameter equal to the one specified by an input message template. As aforementioned, the test cases only define behavior for basic testing of a WSDL operation. At first, the template for the input message of the operation is transmitted via the port owned by the test component. In doing so, the test case parameter *p_soapBinding*, which contains the SOAP binding information for the operation, is passed to the parameterized template. If the operation is a one-way operation i.e. the Web service to be tested will not respond, the test case ends with setting the test verdict to "`pass`". Otherwise, the timer owned by the test component is started to safeguard the test behavior against inactivity of the Web service under test. The timer will run for the number of seconds defined by the value of the *c_timeout* constant, which is passed to the *start* operation as duration parameter. After starting the timer, the test case defines an *alt* statement with several alternatives. The first alternative is executed, if the template defined for the output message of the WSDL operation is received on the port owned by the test component. It will stop the timer and set the test verdict to "`pass`". If the WSDL operation specifies fault messages, further alternatives are defined for the receipt of their corresponding templates. Each of those alternatives will also stop the running timer, but set the test verdict to "`fail`". At last, the *alt* statement refers to the *alt_ReceiveAnyOrTimeout* altstep in order to catch the receipt of unexpected messages or a timeout.

With the test behavior being specified, it can finally be executed in the module control part. Concretely, every test case specified for an operation of the WSDL port type is executed once for each WSDL port, which refers to the WSDL port type via its binding. When a test case is executed for a specific WSDL port, the value, which is passed to its *p_soapBinding* parameter, is the constant that is defined in the module representing the WSDL port (s. ch. 3.5) and that contains SOAP binding information for the operation to which the test case corresponds. In the current example, the test cases specified for the port type *MovieDatabaseSoap* are executed once for the port *MovieDatabaseSoap* and once for the port *MovieDatabaseSoap12*.

To conclude this section, Figure 3.23 lists the mapping result for the exemplary WSDL service *MovieDatabase*, which is displayed in Figure 3.22. To aid display, the figure is shortened as in the previous sections. Moreover, the module prefixing of the references to templates and test cases is removed to improve the readability of the listed code.

```
module TPT_mdb_MovieDatabaseSoap {
   import from WebServices all;
   import from PT_mdb_MovieDatabaseSoap all;
   import from P_mdb_MovieDatabaseSoap all;
   import from P_mdb_MovieDatabaseSoap12 all;

   type component TC_MovieDatabaseSoap {
      port PT_mdb_MovieDatabaseSoap.TP_MovieDatabaseSoap pt_MovieDatabaseSoap;
      timer t_timeoutGuard;
   }

   const float c_timeout := 5.0;

   altstep alt_ReceiveAnyOrTimeout ()
   runs on TPT_mdb_MovieDatabaseSoap.TC_MovieDatabaseSoap {
      [] pt_MovieDatabaseSoap.receive {
         t_timeoutGuard.stop;
         setverdict(fail);
      }
      [] t_timeoutGuard.timeout {
         setverdict(fail);
      }
   }

   group g_createMovie {
      template PT_mdb_MovieDatabaseSoap.I_createMovieRequest a_I_createMovieRequest(
            in WebServices.SoapBinding p_soapBinding) := {
         soapOpBinding := p_soapBinding,
         mp_parameters := {
            eq_id := -2147483648,
            eq_name := omit,
            eq_director := omit
         }
      };

      template PT_mdb_MovieDatabaseSoap.O_createMovieResponse a_O_createMovieResponse
            := {
         mp_parameters := {
            eq_createMovieResult := true
         }
      };

      testcase tc_createMovie (
         in WebServices.SoapBinding p_soapBinding)
      runs on TPT_mdb_MovieDatabaseSoap.TC_MovieDatabaseSoap {
         pt_MovieDatabaseSoap.send(a_I_createMovieRequest(p_soapBinding));

         t_timeoutGuard.start(TPT_mdb_MovieDatabaseSoap.c_timeout);

         alt {
            [] pt_MovieDatabaseSoap.receive(a_O_createMovieResponse) {
               t_timeoutGuard.stop;
               setverdict(pass);
            }
            [] TPT_mdb_MovieDatabaseSoap.alt_ReceiveAnyOrTimeout();
         }
      }
   }

   group g_insertMovie {...}

   group g_getMovie {...}

   group g_searchMovies {...}

   control {
      execute(tc_createMovie(P_mdb_MovieDatabaseSoap.b_createMovie));
      ...

      execute(tc_createMovie(P_mdb_MovieDatabaseSoap12.b_createMovie));
      ...
   }
}
```

Figure 3.23: TTCN-3 output of mapping the WSDL *service* element

3.7 Mapping of the WSDL *documentation* Element

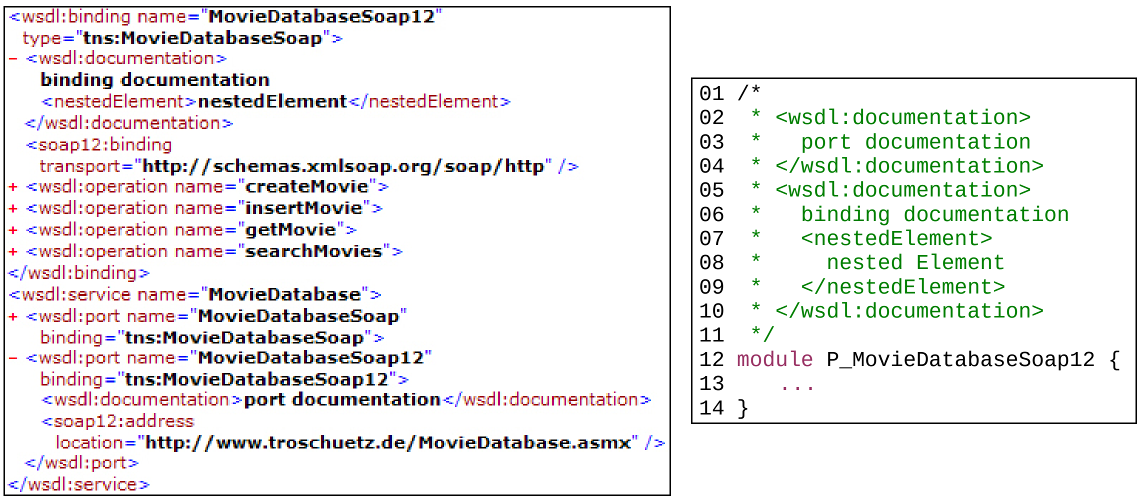
The optional *documentation* element is used by the WSDL 1.1 Standard as a container for human readable documentation and thus allowed to occur as the first child element of any WSDL element other than the *part* element. If it is present, the *documentation* element is mapped to a block comment in front of the TTCN-3 construct to which the enclosing WSDL element maps. The precise mapping rules are given in Table 3.7.

Documented WSDL element			**TTCN-3 construct that gets commented**
definitions			None
types			Each module representing a target namespace that is declared by the contained XML Schemas
message			The record type, which represents the WSDL *input*, *output*, or *fault* element (nested inside a WSDL port type) that refers to this WSDL message
portType			The module, which represents the WSDL port type, as well as the TTCN-3 port type defined in the module
	operation		The group that represents the WSDL operation
		input output fault	The record type that represents the *input*, *output*, or *fault* element
binding			The module that represents the WSDL port referring to this binding
	operation		The constant of the *SoapBinding* type to which the WSDL operation maps
		input output fault	The constant of the *SoapBinding* type, which represents the WSDL *operation* element that embodies the *input*, *output*, or *fault* element
port			The module that represents the WSDL port
service			Each module that defines basic test behavior and execution control for testing a WSDL port type referenced by the WSDL service via port and binding

Table 3.7: Mapping rules for the WSDL *documentation* element

The block comment, to which a WSDL *documentation* element maps, is nearly its XML representation, because "the content of the element is arbitrary text and elements ("mixed" in XSD)" ([37] ch. 2.1.4) and thus improper for a more sophisticated mapping. The start and end tags are copied without any changes into subsequent lines. The value of every nested text node is put into a new line between the element tags, whereat the value is indented by two space characters (*#x20*). If the text node value contains any line breaks, the following white space characters are replaced with two space characters, so the text node value is consistently indented in the TTCN-3 comment. Any child element of a *documentation* element is processed in the same manner as the element itself. The resulting string representation is also inserted into a new line between the element tags and indented by two space characters. If multiple *documentation* elements are mapped to a block comment before the same TTCN-3 construct, all resulting comments should preferably be merged into a single block comment.

The preceding explanations are illustrated in Figure 3.24, which shows the mapping of WSDL *documentation* elements on a WSDL port and its associated binding.



```
<wsdl:binding name="MovieDatabaseSoap12"
  type="tns:MovieDatabaseSoap">
- <wsdl:documentation>
    binding documentation
    <nestedElement>nestedElement</nestedElement>
  </wsdl:documentation>
  <soap12:binding
    transport="http://schemas.xmlsoap.org/soap/http" />
+ <wsdl:operation name="createMovie">
+ <wsdl:operation name="insertMovie">
+ <wsdl:operation name="getMovie">
+ <wsdl:operation name="searchMovies">
</wsdl:binding>
<wsdl:service name="MovieDatabase">
+ <wsdl:port name="MovieDatabaseSoap"
    binding="tns:MovieDatabaseSoap">
- <wsdl:port name="MovieDatabaseSoap12"
    binding="tns:MovieDatabaseSoap12">
    <wsdl:documentation>port documentation</wsdl:documentation>
    <soap12:address
      location="http://www.troschuetz.de/MovieDatabase.asmx" />
  </wsdl:port>
</wsdl:service>
```

```
/*
 * <wsdl:documentation>
 *   port documentation
 * </wsdl:documentation>
 * <wsdl:documentation>
 *   binding documentation
 *   <nestedElement>
 *     nested Element
 *   </nestedElement>
 * </wsdl:documentation>
 */
module P_MovieDatabaseSoap12 {
    ...
}
```

Figure 3.24: Mapping of the WSDL *documentation* element

4 The WSDL2TTCN Utility

The mapping of a Web service description to TTCN-3, which is discussed in detail in Chapter 3, can be done manually but it is also well capable of being automated. Because the automated mapping of a WSDL description is less error-prone and most notably less time-consuming, the WSDL2TTCN utility was developed in the context of this thesis. The utility is a simple console application written in the Java programming language, which was chosen for the following reasons:

- Applications written in Java are platform-independent i.e. they can be run on any platform for which a Java Virtual Machine exists. The WSDL2TTCN utility requires the J2SE (Java Platform, Standard Edition) Runtime Environment 1.5.0 that amongst others is available for Windows, Linux, and Solaris platforms.
- There exists the "Web Services Description Language for Java Toolkit (WSDL4J)" ([25]) that is the reference implementation of the "JSR110 Java™ APIs for WSDL" ([17]) and allows the creation, representation, and manipulation of WSDL documents in Java code. The WSDL2TTCN utility uses WSDL4J version 1.6.2.
- Java is a well-known and widely-used object-oriented programming language, which has evolved continuously since the first public implementation in 1995. Furthermore, there are a variety of (non-)commercial tools for Java such as the open source IDE (Integrated Development Environment) Eclipse ([5]). Version 3.2.0 of Eclipse was used for the implementation of the WSDL2TTCN utility.

The WSDL2TTCN utility is introduced in this chapter, which is structured as follows: At first, Section 4.1 presents the architecture of the utility by giving an overview of the created packages and their respective main classes. Moreover, the overall program flow is introduced. Afterwards, Section 4.2 discusses two implementation aspects in detail. On the one hand, it presents the generation of target namespace qualifiers for TTCN-3 module identifiers, whose format is not specified by the mapping. On the other hand, the translation of XML Schema regular expressions provided by the *pattern* facet to TTCN-3 patterns is described. Thereby, the focus lies on the regular expressions that actually perform the translation and that may be re-used by other implementations of the mapping. Next, Section 4.3 introduces an option of the WSDL2TTCN utility, which allows changing the mapping in certain points in order to generate TTCN-3 code fully compatible to TTworkbench Basic ([28]). In the context of this thesis, this TTCN-3 test system implementation is used to execute a TTCN-3 abstract test suite derived from a WSDL description, but it lacks support for some of the used TTCN-3 features regarding subtypes and attributes. Finally, Section 4.4 exemplifies how the WSDL2TTCN utility is invoked and presents the available command line options.

4.1 Architecture

The classes, interfaces, and enumerations implemented for the WSDL2TTCN utility are organized in three main packages as shown in Figure 4.1.

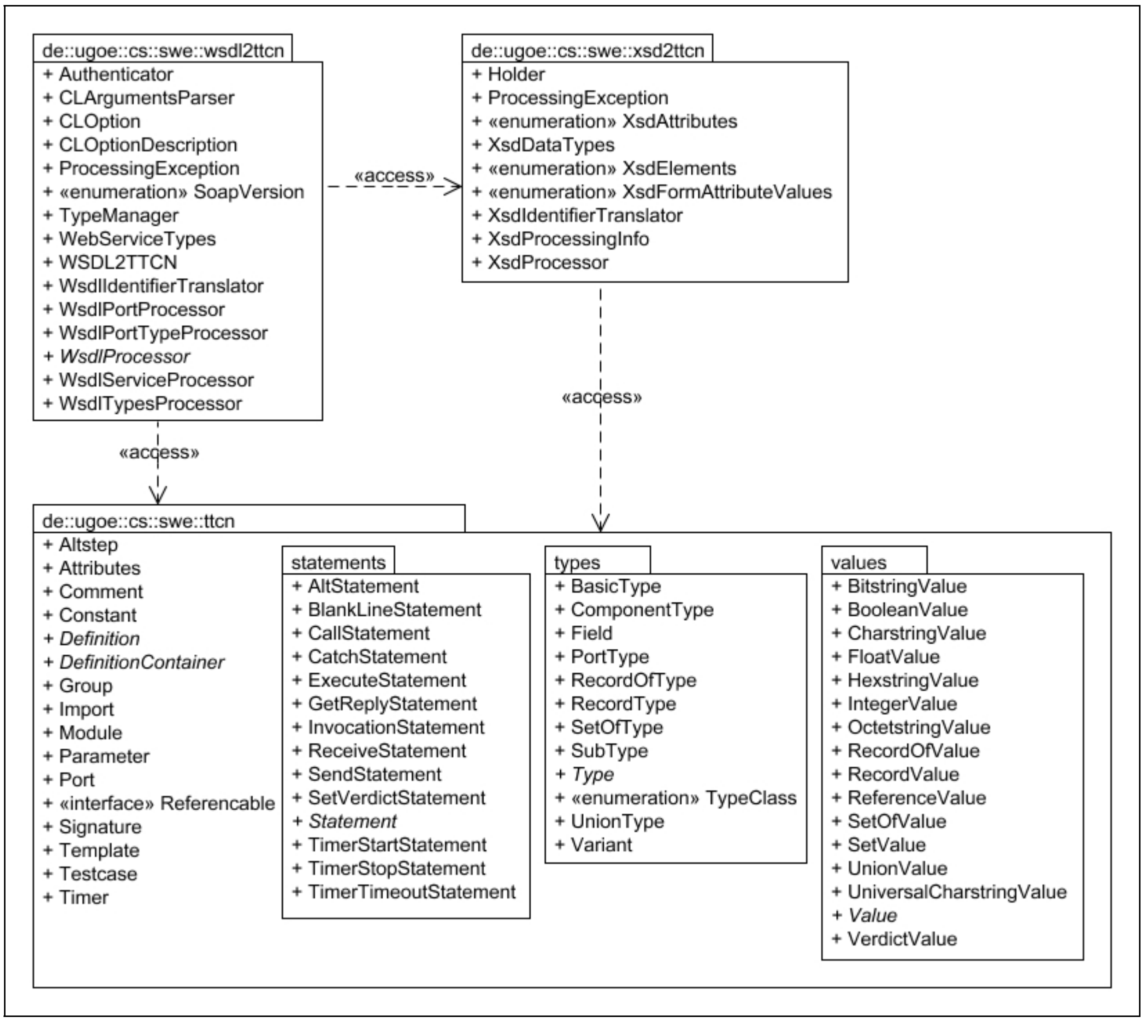


Figure 4.1: Packages of the WSDL2TTCN utility

The greatest importance has the *wsdl2ttcn* package. It contains all implementations that are directly related to the mapping of a WSDL description to a TTCN-3 abstract test suite. Most notably, the package contains the *WSDL2TTCN* class that defines the main entry point for the utility. The main classes of the *wsdl2ttcn* package are discussed in more detail in Section 4.1.2.

The *xsd2ttcn* package contains the implementations that allow mapping XML Schemas to TTCN-3. This functionality is organized in a distinct package to separate it from the domain of mapping WSDL to TTCN-3 and therewith ease its re-use in other fields of application. In order to map the XML Schema data type definitions given by the WSDL *types* element to TTCN-3, the *xsd2ttcn* package is imported by the *wsdl2ttcn* package. The main classes of the *xsd2ttcn* package are discussed in more detail in Section 4.1.1.

The third main package named *ttcn* and its three subpackages contain implementations for the representation of TTCN-3 in Java. The usage of subpackages thereby provides a better structuring of the classes. Because the focus of this thesis is on the testing of Web services, only those TTCN-3 constructs, which are mandatory for the mapping between WSDL and TTCN-3, can be created, modified, and written to the file system by Java code. Like the mapping of XML Schemas to TTCN-3, the representation of TTCN-3 in Java is organized in a distinct package to ease its re-use in other fields of application. The *ttcn* package and its subpackages are imported by the other two main packages in order to perform their respective mappings.

4.1.1 The *xsd2ttcn* Package

The main classes of the *xsd2ttcn* package and their relations to each other are visualized in the class diagram shown in Figure 4.2. The central element is the *XsdProcessor* class that encapsulates the functionality of mapping XML Schemas to Java representations of TTCN-3 modules. The other classes and enumerations displayed by the class diagram are used by the *XsdProcessor* to perform this mapping.

The *XsdDataTypes* class exposes class attributes with the complete Java representations of the predefined TTCN-3 types for the XML Schema built-in types and their defining module *XSDAUX*, which are described in Section 3.2.2. The enumerations *XsdAttributes* and *XsdElements* define the attributes and elements that are allowed to occur in an XML Schema and aid the process of parsing an XML Schema. The *XsdIdentifierTranslator* class is very important, because it provides class operations for the translation of XML Schema to TTCN-3 identifiers. Those operations are not defined in the *XsdProcessor* itself to separate the translation functionality from the overall functionality of mapping XML Schemas to TTCN-3. Furthermore, the translation operations are defined on the class level to make them easily accessible for the *XsdProcessor* and any other class that needs to know the names of the TTCN-3 types representing XML Schema components and those of their defining modules. For example, the mapping of a WSDL description to TTCN-3 requires such knowledge, because the described WSDL messages consist of parts whose types are by default defined in the XML Schema language.

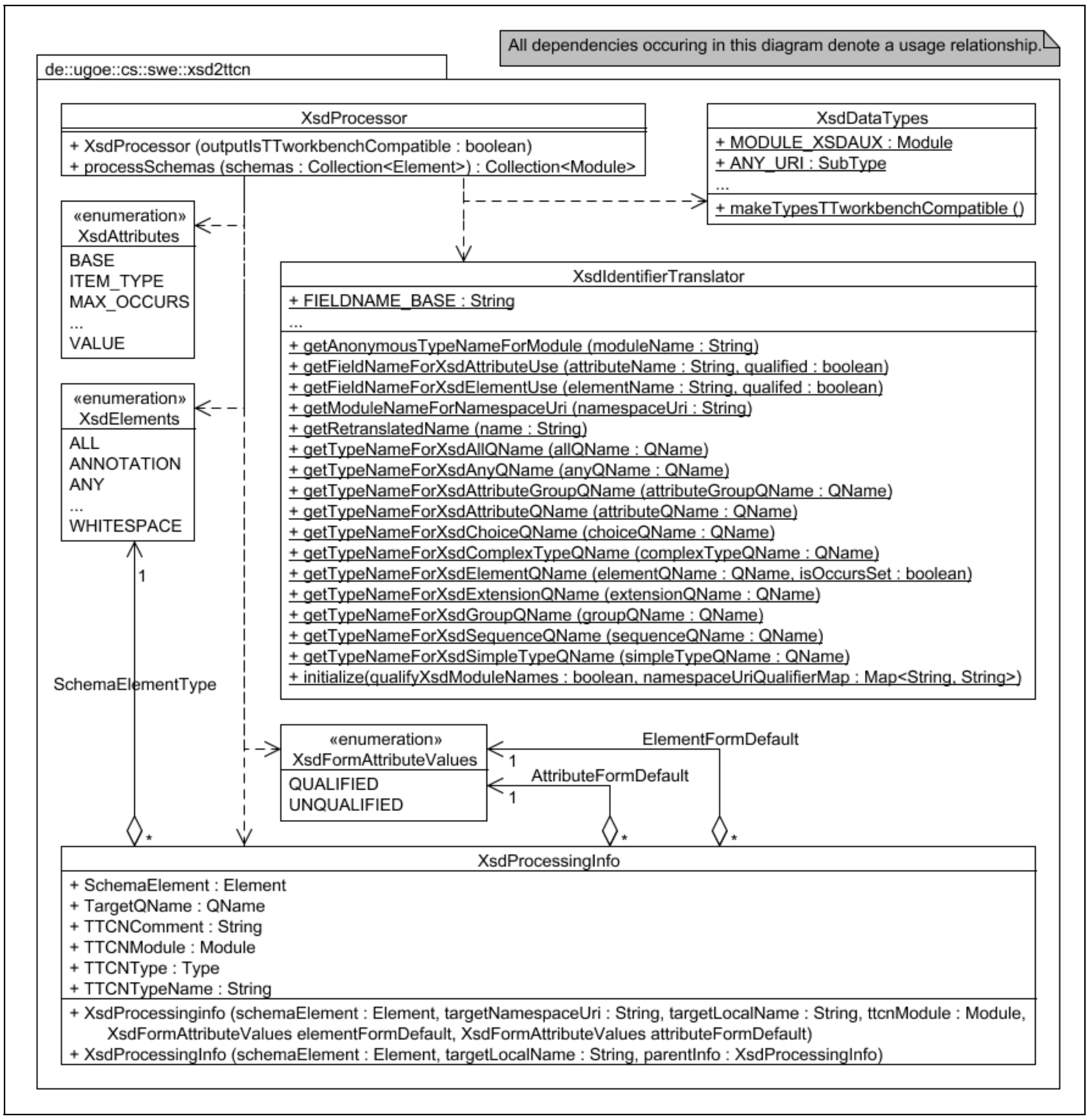

All dependencies occuring in this diagram denote a usage relationship.
de::ugoe::cs::swe::xsd2ttcn
XsdProcessor
+ XsdProcessor (outputIsTTworkbenchCompatible : boolean)
+ processSchemas (schemas : Collection<Element>) : Collection<Module>
XsdDataTypes
+ MODULE_XSDAUX : Module
+ ANY_URI : SubType
...
+ makeTypesTTworkbenchCompatible ()
«enumeration» XsdAttributes
BASE
ITEM_TYPE
MAX_OCCURS
...
VALUE
«enumeration» XsdElements
ALL
ANNOTATION
ANY
...
WHITESPACE
XsdIdentifierTranslator
+ FIELDNAME_BASE : String
...
+ getAnonymousTypeNameForModule (moduleName : String)
+ getFieldNameForXsdAttributeUse (attributeName : String, qualified : boolean)
+ getFieldNameForXsdElementUse (elementName : String, qualifed : boolean)
+ getModuleNameForNamespaceUri (namespaceUri : String)
+ getRetranslatedName (name : String)
+ getTypeNameForXsdAllQName (allQName : QName)
+ getTypeNameForXsdAnyQName (anyQName : QName)
+ getTypeNameForXsdAttributeGroupQName (attributeGroupQName : QName)
+ getTypeNameForXsdAttributeQName (attributeQName : QName)
+ getTypeNameForXsdChoiceQName (choiceQName : QName)
+ getTypeNameForXsdComplexTypeQName (complexTypeQName : QName)
+ getTypeNameForXsdElementQName (elementQName : QName, isOccursSet : boolean)
+ getTypeNameForXsdExtensionQName (extensionQName : QName)
+ getTypeNameForXsdGroupQName (groupQName : QName)
+ getTypeNameForXsdSequenceQName (sequenceQName : QName)
+ getTypeNameForXsdSimpleTypeQName (simpleTypeQName : QName)
+ initialize(qualifyXsdModuleNames : boolean, namespaceUriQualifierMap : Map<String, String>)
1
SchemaElementType
«enumeration» XsdFormAttributeValues
QUALIFIED
UNQUALIFIED
ElementFormDefault
AttributeFormDefault
1
1
*
*
*
XsdProcessingInfo
+ SchemaElement : Element
+ TargetQName : QName
+ TTCNComment : String
+ TTCNModule : Module
+ TTCNType : Type
+ TTCNTypeName : String
+ XsdProcessinginfo (schemaElement : Element, targetNamespaceUri : String, targetLocalName : String, ttcnModule : Module, XsdFormAttributeValues elementFormDefault, XsdFormAttributeValues attributeFormDefault)
+ XsdProcessingInfo (schemaElement : Element, targetLocalName : String, parentInfo : XsdProcessingInfo)

Figure 4.2: Main classes of the *xsd2ttcn* package

Finally, the most important helper for the *XsdProcessor* is the *XsdProcessingInfo* class, which encapsulates all information that is necessary to map a specific XML Schema component to TTCN-3 e.g. the XML element representing the component or the name and the defining module of the resulting TTCN-3 type definition. At the beginning of the mapping process, the *XsdProcessor* class creates an instance of this class for every top-level component found in the XML Schemas to be mapped. Subsequently, those instances can easily be passed to operations involved in the mapping process.

4.1.2 The *wsdl2ttcn* Package

The most relevant classes of the *wsdl2ttcn* package and their relations to each other are visualized in the class diagram shown in Figure 4.3.

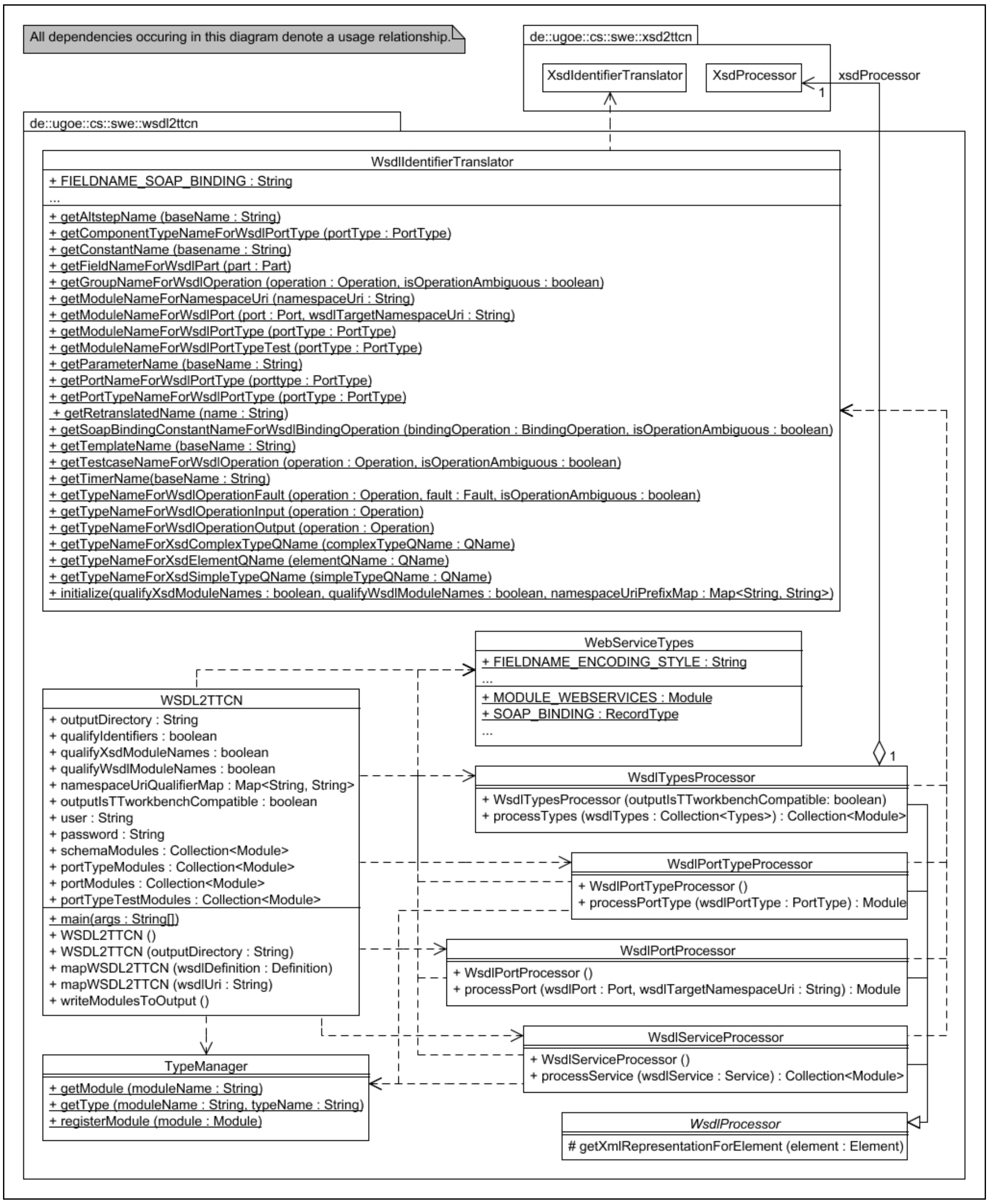

All dependencies occuring in this diagram denote a usage relationship.
de::ugoe::cs::swe::xsd2ttcn
XsdIdentifierTranslator
XsdProcessor
xsdProcessor
1
de::ugoe::cs::swe::wsdl2ttcn
WsdlIdentifierTranslator
+ FIELDNAME_SOAP_BINDING : String
...
+ getAltstepName (baseName : String)
+ getComponentTypeNameForWsdlPortType (portType : PortType)
+ getConstantName (basename : String)
+ getFieldNameForWsdlPart (part : Part)
+ getGroupNameForWsdlOperation (operation : Operation, isOperationAmbiguous : boolean)
+ getModuleNameForNamespaceUri (namespaceUri : String)
+ getModuleNameForWsdlPort (port : Port, wsdlTargetNamespaceUri : String)
+ getModuleNameForWsdlPortType (portType : PortType)
+ getModuleNameForWsdlPortTypeTest (portType : PortType)
+ getParameterName (baseName : String)
+ getPortNameForWsdlPortType (porttype : PortType)
+ getPortTypeNameForWsdlPortType (portType : PortType)
+ getRetranslatedName (name : String)
+ getSoapBindingConstantNameForWsdlBindingOperation (bindingOperation : BindingOperation, isOperationAmbiguous : boolean)
+ getTemplateName (baseName : String)
+ getTestcaseNameForWsdlOperation (operation : Operation, isOperationAmbiguous : boolean)
+ getTimerName(baseName : String)
+ getTypeNameForWsdlOperationFault (operation : Operation, fault : Fault, isOperationAmbiguous : boolean)
+ getTypeNameForWsdlOperationInput (operation : Operation)
+ getTypeNameForWsdlOperationOutput (operation : Operation)
+ getTypeNameForXsdComplexTypeQName (complexTypeQName : QName)
+ getTypeNameForXsdElementQName (elementQName : QName)
+ getTypeNameForXsdSimpleTypeQName (simpleTypeQName : QName)
+ initialize(qualifyXsdModuleNames : boolean, qualifyWsdlModuleNames : boolean, namespaceUriPrefixMap : Map<String, String>)
WebServiceTypes
+ FIELDNAME_ENCODING_STYLE : String
...
+ MODULE_WEBSERVICES : Module
+ SOAP_BINDING : RecordType
...
WSDL2TTCN
+ outputDirectory : String
+ qualifyIdentifiers : boolean
+ qualifyXsdModuleNames : boolean
+ qualifyWsdlModuleNames : boolean
+ namespaceUriQualifierMap : Map<String, String>
+ outputIsTTworkbenchCompatible : boolean
+ user : String
+ password : String
+ schemaModules : Collection<Module>
+ portTypeModules : Collection<Module>
+ portModules : Collection<Module>
+ portTypeTestModules : Collection<Module>
+ main(args : String[])
+ WSDL2TTCN ()
+ WSDL2TTCN (outputDirectory : String)
+ mapWSDL2TTCN (wsdlDefinition : Definition)
+ mapWSDL2TTCN (wsdlUri : String)
+ writeModulesToOutput ()
1
WsdlTypesProcessor
+ WsdlTypesProcessor (outputIsTTworkbenchCompatible: boolean)
+ processTypes (wsdlTypes : Collection<Types>) : Collection<Module>
WsdlPortTypeProcessor
+ WsdlPortTypeProcessor ()
+ processPortType (wsdlPortType : PortType) : Module
WsdlPortProcessor
+ WsdlPortProcessor ()
+ processPort (wsdlPort : Port, wsdlTargetNamespaceUri : String) : Module
WsdlServiceProcessor
+ WsdlServiceProcessor ()
+ processService (wsdlService : Service) : Collection<Module>
TypeManager
+ getModule (moduleName : String)
+ getType (moduleName : String, typeName : String)
+ registerModule (module : Module)
WsdlProcessor
# getXmlRepresentationForElement (element : Element)

Figure 4.3: Main classes of the *wsdl2ttcn* package

The central element of the *wsdl2ttcn* package is the equally named *WSDL2TTCN* class, which implements the overall behavior of mapping a WSDL description to TTCN-3 and therefore contains the main entry point for the WSDL2TTCN utility namely the class operation *main*. In addition to this operation, the *WSDL2TTCN* class exposes various attributes that allow certain mapping options to be configured e.g. the directory where the resulting TTCN-3 ATS will be created. The mapping process is invoked by calling an overload of the *mapWSDL2TTCN* operation. One overload takes the URI of a WSDL document and resolves it to a Java representation of the WSDL description by using the WSDL4J API. This representation is subsequently passed to the second overload that executes the actual mapping to TTCN-3. The representations of the resulting TTCN-3 modules are accessible via the *SchemaModules*, *PortTypeModules*, *PortModules*, and *PortTypeTestModules* attributes and can be written to the configured output directory by calling the operation *writeModulesToOutput*.

In order to execute the mapping, the *WSDL2TTCN* class uses the *WsdlTypesProcessor*, *WsdlPortTypeProcessor*, *WsdlPortProcessor*, and *WsdlServiceProcessor* classes. Every class encapsulates the mapping of an individual, major element of a WSDL description. The four classes are derived from the abstract *WsdlProcessor* class that provides the *getXmlRepresentationElement* operation, which is utilized by all inheritors during the mapping of WSDL *documentation* elements. A special case is the *WsdlTypesProcessor* class. It does not perform the complete mapping by itself, but contains an instance of the *XsdProcessor* class from the *xsd2ttcn* package that is used to map the XML Schemas contained in the WSDL *types* element to TTCN-3.

All processor classes use the class operations provided by the *WsdlIdentifierTranslator* in order to translate WSDL to TTCN-3 identifiers. Those operations are not defined by the classes themselves for two reasons. First, some are utilized by multiple processor classes so that duplicate code is avoided. Second, the encapsulation in a dedicated class separates the translation operations from the overall functionality of mapping WSDL to TTCN-3. The *WsdlIdentifierTranslator* class utilizes the *XsdIdentifierTranslator* class from the *xsd2ttcn* package to be capable of resolving the identifiers of TTCN-3 types representing XML Schema components and those of their defining modules. This is mandatory, since the WSDL messages are composed of parts whose types are by default defined in the XML Schema language.

Finally, the *WebServiceTypes* and *TypeManager* classes provide supporting functions. The former exposes class attributes with the Java representations of the TTCN-3 types for encapsulation of SOAP binding information and their defining module *WebServices*, which are described in Section 3.2.2. The *TypeManager* class exposes class operations that allow registering all types defined by a module as well as retrieving the registered types by their name. The functionality is for example used by the *WsdlServiceProcessor* to retrieve the TTCN-3 port type, which has been created for a specific WSDL port type by the *WsdlPortTypeProcessor* class.

4.1.3 Overall Program Flow

The overall program flow of mapping a WSDL document to TTCN-3 is depicted by the sequence diagram shown in Figure 4.4. The focus of the diagram lies on the interaction of the classes, so that the invocation of non-public operations is omitted to aid display.

As mentioned in Section 4.1.2, the overall behavior of mapping a WSDL description to TTCN-3 is implemented in the *mapWSDL2TTCN* operation of the *WSDL2TTCN* class. Thus, the sequence diagram starts with a call to this operation on an already configured instance of the *WSDL2TTCN* class.

First, an instance of the *WsdlTypesProcessor* class is created that in turn instantiates an object of the *XsdProcessor* class. On the former object, the *processTypes* operation is called, whereupon the Java representations of all *types* elements of the currently mapped WSDL description are passed in. The XML Schemas contained in those elements are retrieved and put into a collection, which is then passed to the *XsdProcessor* object by calling its *processSchemas* operation. In this operation, the XML Schemas are mapped to TTCN-3 modules whose Java representations are returned to the *WsdlTypesProcessor* object and in the end to the *WSDL2TTCN* instance. The WSDL *types* elements or more precisely the contained schemas are mapped all at once, because this eases dealing with possible interdependencies of the XML Schema components.

Next, the WSDL *portType* elements are mapped to TTCN-3. Therefore, an instance of the *WsdlPortTypeProcessor* class is created whose *processPortType* operation is called for every WSDL port type defined in the currently mapped WSDL document. Thereby, the Java representation of the respective *portType* element is passed in and that of the resulting TTCN-3 module is returned.

Finally, the WSDL *service* and *port* elements contained in the currently mapped WSDL document are processed. In order to do so, instances of the classes *WsdlPortProcessor* and *WsdlServiceProcessor* are constructed. The *mapWSDL2TTCN* operation afterwards iterates over the WSDL *service* elements and inside of a nested loop over their enclosed WSDL *port* elements. The Java representations of the elements are thereby passed to the *processService* or the *processPort* operation and those of the resulting TTCN-3 modules are returned.

When the *mapWSDL2TTCN* operation ends, the current WSDL document is completely mapped to TTCN-3 and the Java representations of the generated TTCN-3 modules are exposed via the respective class attributes of the *WSDL2TTCN* instance.

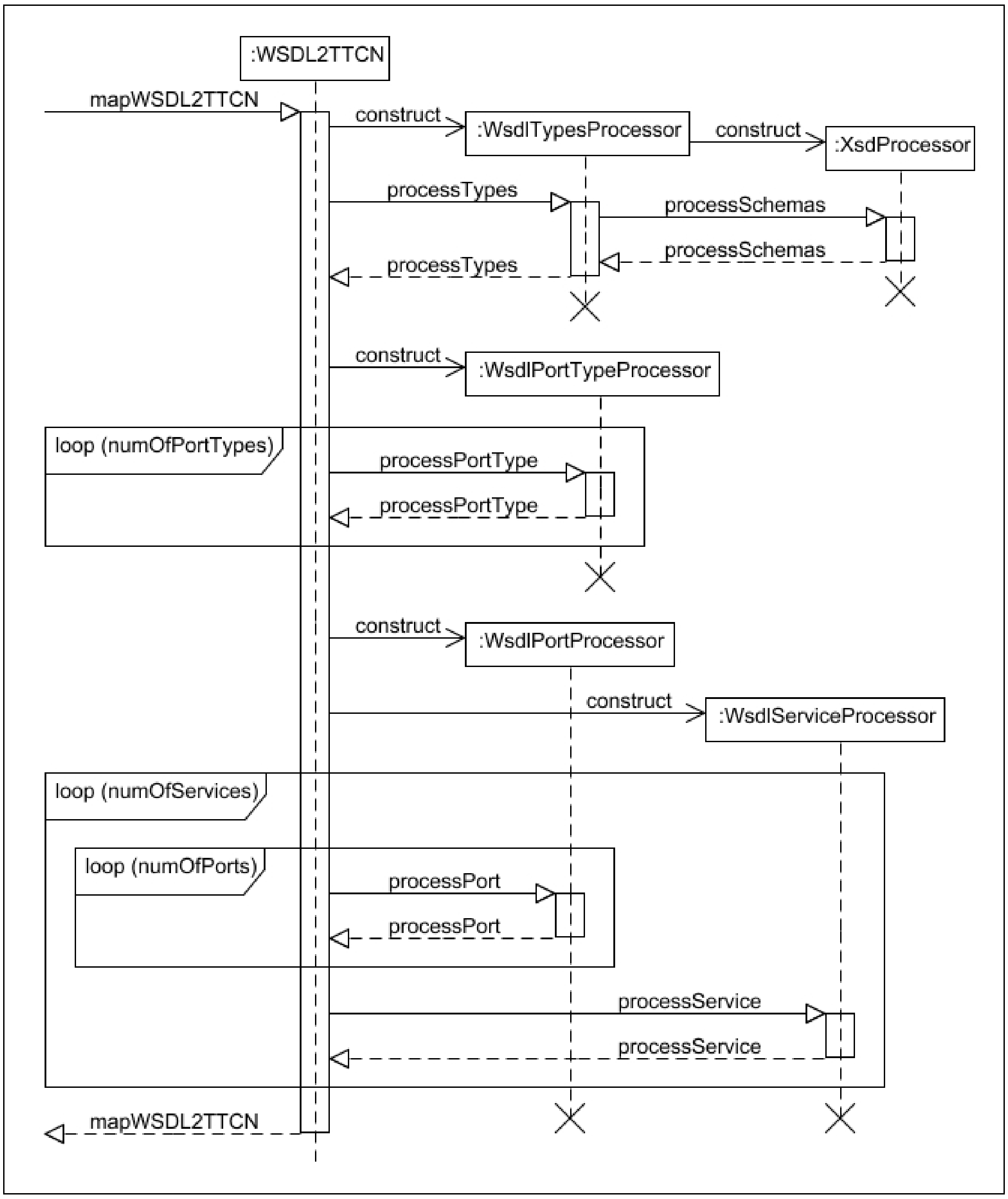


Figure 4.4: Overall program flow

4.2 Implementation Details

4.2.1 Creation of Target Namespace Qualifiers

In the mapping presented in this thesis, it is specified that every identifier of a TTCN-3 module shall incorporate a qualifier for the target namespace of either the XML Schema that maps to the module in question or the WSDL description that encloses the WSDL element mapping to the module in question (s. ch. 3.2.1). Although the concrete format of the target namespace qualifiers is not specified, the mapping suggests the usage of user-defined qualifiers or hashes of the namespace URIs, since those are able to ensure unambiguous identifiers even for the subsequent mapping of WSDL descriptions. The WSDL2TTCN utility follows this suggestion and implements a combination of both.

By default, a target namespace qualifier becomes the hash of the namespace URI. This hash is created by calling the *hashcode* method of the *String* instance, which holds the namespace URI and converting the returned integer into a string. Since the return value of the *String.hashcode* method can be negative but the "-" character is not allowed in TTCN-3 identifiers, the conversion to a string has to replace occurrences of the minus character without breaking the uniqueness of the hash. In order to do so, the absolute value of the hash is prefixed with "`1`", if the hash is negative, or "`0`" otherwise.

Because the usage of hashes results in target namespace qualifiers that can be quite long and are not very expressive, the WSDL2TTCN utility allows the user to define specific qualifiers for the namespaces. Concretely, the *WSDL2TTCN* class exposes the attribute *namespaceUriQualifierMap*, which stores the correlation between namespace URIs and user-defined qualifiers. This map can be configured as needed in front of starting the mapping process, at whose beginning the map is passed to the *WsdlIdentifierTranslator* and the *XsdIdentifierTranslator* classes that are responsible for the creation of TTCN-3 identifiers. In order to allow the specification of user-defined qualifiers not only in code, the WSDL2TTCN utility provides a command line option, which is interpreted by the main entry point and then used to properly configure an instance of the *WSDL2TTCN* class (s. ch. 4.4).

Finally, the WSDL2TTCN utility also facilitates disabling the incorporation of target namespace qualifiers into TTCN-3 module identifiers, since it is not always necessary but nevertheless increases the complexity of the identifiers. If for example the data type definitions of a WSDL description are contained in a single XML Schema, there is no imperative need for the incorporation of a target namespace qualifier into the identifier of the resulting TTCN-3 module. The use of target namespace qualifiers can be disabled selectively for modules mapping to an XML Schema target namespace and/or modules that map to WSDL elements. In Java code, this can be achieved by setting the attributes *qualifyXsdModuleNames* and *qualifyWsdlModuleNames* of the *WSDL2TTCN* class. In order to disable the incorporation of target namespace qualifiers on external invocation of the WSDL2TTCN utility, the corresponding command line options have to be used (s. ch. 4.4).

To conclude this section and illustrate the previous explanations, Figure 4.5 shows the implementation of incorporating target namespace qualifiers into identifiers of TTCN-3 modules as defined in the *XsdIdentifierTranslator* class.

```java
public class XsdIdentifierTranslator {
   public static final String PREFIX_MODULE = "T_";

   ...

   private static boolean QUALIFY_XSD_MODULE_NAMES;
   private static Map<String, String> NAMESPACEURI_QUALIFIER_MAP;
   private static Map<String, String> ANONYMOUS_TYPES_COUNTER;

   static {
      QUALIFY_XSD_MODULE_NAMES = true;
      NAMESPACEURI_QUALIFIER_MAP = new HashMap<String, String>();
      ANONYMOUS_TYPES_COUNTER = new HashMap<String, String>();
   }

   public static void initialize(boolean qualifyXsdModuleNames, Map<String, String>
         namespaceUriQualifierMap) {
      QUALIFY_XSD_MODULE_NAMES = qualifyXsdModuleNames;
      NAMESPACEURI_QUALIFIER_MAP = namespaceUriQualifierMap;
   }

   public static String getModuleNameForNamespaceUri(String namespaceUri) {
      if (namespaceUri.equals(XMLConstants.W3C_XML_SCHEMA_NS_URI)) {
         return XsdDataTypes.MODULE_XSDAUX.getName();
      } else {
         String qualifier = "";
         if (QUALIFY_XSD_MODULE_NAMES) {
            qualifier = NAMESPACEURI_QUALIFIER_MAP.get(namespaceUri);
            if (qualifier == null) {
               int hash = namespaceUri.hashCode();
               if (hash < 0) {
                    qualifier = String.format("ns1%d", -1 * hash);
               } else {
                    qualifier = String.format("ns0%d", hash);
               }
               NAMESPACEURI_QUALIFIER_MAP.put(namespaceUri, qualifier);
            }
         }
         return PREFIX_MODULE + qualifier;
      }
   }

   ...
}
```

Figure 4.5: Creation of target namespace qualifiers

4.2.2 Translation of XML Schema Regular Expressions

The implementation of the WSDL2TTCN utility uses regular expressions to translate the XML Schema regular expressions provided by the *pattern* facet to TTCN-3 patterns. Table 4.1 lists the translations defined by the mapping (s. Table 3.5) together with the used regular expressions split into matching and replacement expression.

XSD regular expression	**TTCN-3 Pattern**	**Matching Expression**	**Replacement Expression**
`?`	`#(0,1)`	`([^\\](\\\\)*)\?`	`$1#(0,1)`
`+`	`#(1,)`	`([^\\](\\\\)*)\+`	`$1#(1,)`
`*`	`#(0,)`	`([^\\](\\\\)*)\*`	`$1#(0,1)`
`{n}`	`#(n)`	`([^\\](\\\\)*)\{(\d+)\}`	`$1#($3)`
`{n,}`	`#(n,)`	`([^\\](\\\\)*)\{(\d+),\}`	`$1#($3,)`
`{n,m}`	`#(n,m)`	`([^\\](\\\\)*)\{(\d+),(\d+)\}`	`$1#($3,$4)`
`.`	`?`	`(^|[^\\](\\\\)*)\.`	`$1?`
`\s`	`[ \t\n\r]`	`(^|[^\\](\\\\)*)\\s`	`$1[ \\t\\n\\r]`
`\S`	`[^ \t\n\r]`	`(^|[^\\](\\\\)*)\\S`	`$1[^ \\t\\n\\r]`
`\D`	`[^\d]`	`(^|[^\\](\\\\)*)\\D`	`$1[^\\d]`
`\W`	`[^\w]`	`(^|[^\\](\\\\)*)\\W`	`$1[^\\w]`
`\i`	`[a-zA-Z_:]`	`(^|[^\\](\\\\)*)\\i`	`$1[a-zA-Z_:]`
`\I`	`[^a-zA-Z_:]`	`(^|[^\\](\\\\)*)\\I`	`$1[^a-zA-Z_:]`
`\c`	`[\w.\-_:]`	`(^|[^\\](\\\\)*)\\c`	`$1[\\w.\\-_:]`
`\C`	`[^\w.\-_:]`	`(^|[^\\](\\\\)*)\\C`	`$1[^\\w.\\-_:]`
`\.`	`.`	`(^|[^\\](\\\\)*)\\\.`	`$1\.`
`"`	`\"`	`"`	`\\"`
`&#xgprc;`	`\q{g,p,r,c}`	`&#x([0-9a-fA-F])([0-9a-fA-F])([0-9a-fA-F])([0-9a-fA-F]);`	`\\q{$1,$2,$3,$4}`
`&#xprc;`	`\q{0,p,r,c}`	`&#x([0-9a-fA-F])([0-9a-fA-F])([0-9a-fA-F]);`	`\\q{0,$1,$2,$3}`
`&#xrc;`	`\q{0,0,r,c}`	`&#x([0-9a-fA-F])([0-9a-fA-F]);`	`\\q{0,0,$1,$2}`
`&#xc;`	`\q{0,0,0,c}`	`&#x([0-9a-fA-F]);`	`\\q{0,0,0,$1}`
Note: The back references `$n` (n=1, 2 …) contained in the replacement expressions may have a different syntax in other implementations of regular expressions.			

Table 4.1: Translation of XML Schema regular expressions (cp. Table 3.5)

The terms "`(^|[^\\](\\\\)*)`" and "`([^\\](\\\\)*)`" appearing at the beginning of most matching regular expressions ensure that the term to be matched either occurs at the beginning of a line or is preceded by zero or an even number of backslashes. In addition, it is important that the translation of "`?`" to "`#(0,1)` " is executed before translating "`.`" to "`?`", since otherwise each dot character appearing in an XML Schema regular expression will become first a question mark and in a second step the quantifier "`#(0,1)`".

Besides the above translations, the mapping presented in Chapter 3 also specifies that hexadecimal characters inside the Unicode character representation of TTCN-3 patterns must be replaced by their respective decimal character (s. ch. 3.3.3). This translation is performed by the regular expressions listed in Table 4.2.

Matching Expression	**Replacement Expression**
`(^|[^\\](\\\\)*)(\\q\{[0-9a-fA-F,]*)[aA]([0-9a-fA-F,]*\})`	`$1$310$4`
`(^|[^\\](\\\\)*)(\\q\{[0-9b-fB-F,]*)[bB]([0-9b-fB-F,]*\})`	`$1$311$4`
`(^|[^\\](\\\\)*)(\\q\{[0-9c-fC-F,]*)[cC]([0-9c-fC-F,]*\})`	`$1$312$4`
`(^|[^\\](\\\\)*)(\\q\{[0-9defDEF,]*)[dD]([0-9defDEF,]*\})`	`$1$313$4`
`(^|[^\\](\\\\)*)(\\q\{[0-9efEF,]*)[eE]([0-9efEF,]*\})`	`$1$314$4`
`(^|[^\\](\\\\)*)(\\q\{[0-9fF,]*)[fF]([0-9fF,]*\})`	`$1$315$4`
Note: The back references `$n` (n=1, 2 …) contained in the replacement expressions may have a different syntax in other implementations of regular expressions.	

Table 4.2: Translation of Unicode character representations of TTCN-3 patterns

There is a regular expression for each hexadecimal character, which replaces its upper and lower case occurrences at any place in a Unicode character representation. The term "`(^|[^\\](\\\\)*)`" guarantees that the expression to be matched either occurs at the beginning of a line or is preceded by zero or an even number of backslash characters. Furthermore, it is important to execute the translations in the given order, because the lower matching expressions become less complex under the assumption that specific hexadecimal characters have already been replaced.

4.3 Optional Mapping Changes for TTworkbench Compatibility

As initially mentioned, the WSDL2TTCN utility offers an option that allows changing the mapping between WSDL and TTCN-3 in certain points in order to create TTCN-3 output fully compatible to TTworkbench Basic. This tool is used in the context of this thesis to execute the abstract test suite resulting from the mapping, but does not support some of the used TTCN-3 features regarding subtypes and attributes.

Foremost, there are three problems with the pattern-subtyping of character string types. First, TTworkbench Basic requires all occurrences of the hyphen character that do not denote a range of characters to be escaped, though the TTCN-3 Standard Part 1 defines that the hyphen has its metacharacter meaning only inside a set expression (s. [6] Table B.1). For example, the pattern "`-option`" is valid according to the standard but marked as erroneous by TTworkbench Basic. Second, Table B.1 of the TTCN-3 Standard Part 1 specifies a metacharacter meaning for the plus character namely that it matches the preceding expression one or several times. Hence, a plus character needs to be escaped to loose its special meaning, but this is not permitted by TTworkbench Basic. For example, the pattern "`1\+2`" expressing the addition of two numbers is marked as erroneous. TTworkbench Basic only accepts the pattern without the backslash i.e. "`1+2`", but following the standard such a pattern matches one or more "`1`" characters followed by a single "`2`" character. Third and most significant, TTworkbench Basic marks values of a pattern-restricted subtype as erroneous, although they are valid. For example, the module displayed in Figure 4.6 is objected, because TTworkbench Basic claims that the value "`Max`" is not of the preliminarily defined *Name* type.

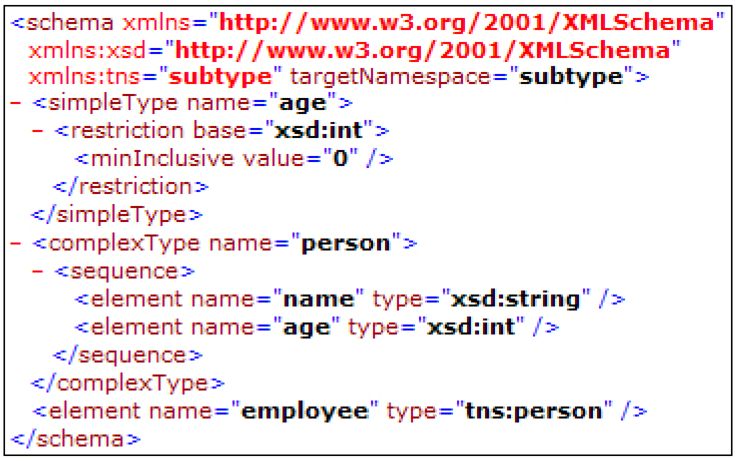
```
module Pattern {
   type charstring Name (pattern "[A-Z][a-z]#(1,)");

   const Name c_name := "Max";
}
```

Figure 4.6: Pattern-restricted TTCN-3 subtype

Since many TTCN-3 representations of the XML Schema built-in data types are pattern-restricted subtypes and the XML Schema constraining facet *pattern* maps to a pattern-restricted subtype, the third limitation of TTworkbench Basic would effectively prevent the specification of valid templates for the execution of an abstract test suite derived from a WSDL description. Hence, the activation of the option discussed in this section suppresses the mapping of XML Schema *pattern* facets as well as the pattern-restriction of the TTCN-3 types representing XML Schema built-in data types.

In contrast to the mentioned problems, where TTworkbench Basic marks the generated TTCN-3 code as erroneous, the remaining problem arises during test execution. More precisely, it regards the encoding and decoding of proper SOAP messages from or to the generated TTCN-3 definitions. The mapping of WSDL descriptions to TTCN-3 makes use of the TTCN-3 *encode* attribute in order to allow codecs keep track of the original

XSD nature of a TTCN-3 definition and thus allow an easy and proper construction of the corresponding XML fragment. In TTworkbench Basic, if the codec retrieves the *encode* attribute of a subtype whose base type is not a TTCN-3 built-in type, it does not get the *encode* attribute of the subtype as expected, but that of the base type. Because this would cause the encoding of invalid SOAP messages, such subtype definitions are replaced by a copy of the base type definition, whereat the identifier, *encode* attribute, and present restriction of the subtype are taken over. This replacement is illustrated in Figure 4.7. It shows the general as well as the changed mapping of the simple type definition named *age*, which restricts the XML Schema built-in data type *integer*, and of the element declaration *employee*, which is of the globally defined complex type named *person*.



```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="subtype" targetNamespace="subtype">
- <simpleType name="age">
  - <restriction base="xsd:int">
      <minInclusive value="0" />
    </restriction>
  </simpleType>
- <complexType name="person">
  - <sequence>
      <element name="name" type="xsd:string" />
      <element name="age" type="xsd:int" />
    </sequence>
  </complexType>
  <element name="employee" type="tns:person" />
</schema>
```

```
module T_subtype {
    import from XSDAUX all;

    type XSDAUX.integer_ SimpleType_age
           (0 .. 2147483647) with {
        encode "{subtype}age";
    }

    type record ComplexType_person {
        XSDAUX.string e_name,
        XSDAUX.int e_age
    } with {
        encode "{subtype}person";
        encode (e_name) "name";
        encode (e_age) "age";
    }

    type T_subtype.ComplexType_person
           Element_employee with {
        encode "{subtype}employee";
    }
}
```

```
module T_subtype {
    import from XSDAUX all;

    type integer SimpleType_age
           (0 .. 2147483647) with {
        encode "{subtype}age";
    }

    type record ComplexType_person {
        XSDAUX.string e_name,
        XSDAUX.int e_age
    } with {
        encode "{subtype}person";
        encode (e_name) "name";
        encode (e_age) "age";
    }

    type record Element_employee {
        XSDAUX.string e_name,
        XSDAUX.int e_age
    } with {
        encode "{subtype}employee";
        encode (e_name) "name";
        encode (e_age) "age";
    }
}
```

Figure 4.7: Changed mapping of TTCN-3 subtypes

In Java code, the WSDL2TTCN utility is configured to create a TTCN-3 ATS fully compatible to TTworkbench Basic by enabling the *outputIsTTworkbenchCompatible* attribute of the *WSDL2TTCN* class before starting the mapping process. In order to activate the option on external invocation of the WSDL2TTCN utility, the proper command line option specified in the Section 4.4 has to be used.

4.4 Invocation and Command Line Options

An invocation of the WSDL2TTCN utility has the form given in Figure 4.8, whereupon "`wsdl_uri`" denotes the URI of the WSDL document that should be mapped to TTCN-3.

```
java de.ugoe.cs.swe.wsdl2ttcn.WSDL2TTCN [options] wsdl_uri
```

Figure 4.8: Invocation of the WSDL2TTCN utility

The available command line options are listed in Table 4.3. For each option, there are two names: a short one represented by a single character and a long, more descriptive form. The usage of either short or long names does not affect the result and is up to the users personal preferences.

Option	Description
-h --help	Print a list of all command line options.
-o <arg> --output <arg>	Specify the directory for the TTCN-3 abstract test suite generated by the WSDL2TTCN utility.
-i --identifiers	The identifiers of TTCN-3 definition usages are unqualified i.e. they are not prefixed with the name of the defining module. Be aware of the fact that the activation of this option may lead to name ambiguities.
-t <arg1> <arg2> --targetnamespace <arg1> <arg2>	Specify a qualifier (<arg1>) that is used to qualify module names with regard to the given target namespace URI (<arg2>).
-s --schemamodules	The identifiers of modules representing an XML Schema target namespace are unqualified with regard to the target namespace i.e. all modules are named the same. When activating this option, be sure that the WSDL document uses only types from a single XML Schema target namespace.

Option	Description
-w --wsdlmodules	The identifiers of modules representing major WSDL elements are unqualified with regard to the target namespace of the enclosing WSDL document. Be aware of the fact that the activation of this option may lead to name ambiguities.
-T --ttworkbench	The TTCN-3 ATS will miss some features as described in Section 4.3 to be fully compatible to TTworkbench Basic.
-U <arg> --user <arg>	Specify the username, which is used, if retrieval of the WSDL description requires authentication.
-P <arg> --password <arg>	Specify the password, which is used, if retrieval of the WSDL description requires authentication.

Table 4.3: Command line options of the WSDL2TTCN utility

5 Extension of TTworkbench Basic for Web Service Testing

Similar to any other programming language, TTCN-3 code is not executable by itself. An abstract test suite, which has been generated for Web service testing according to the mapping specified in Chapter 3, needs to be interpreted or executed by a TTCN-3 test system implementation. In the context of this thesis, TTworkbench Basic ([28]) is used for that purpose.

TTworkbench Basic is a TTCN-3 IDE designed and marketed by Testing Technologies IST GmbH and was used in version 1.0.13 under an educational license. It is the basic format of the TTworkbench product line, which is based upon the Eclipse platform and consequently written in the Java programming language. TTworkbench Basic supports a broad spectrum of test development, ranging from the text-based specification to the compilation and the execution of tests. The provided functionality is split into three distinct features:

- CL Editor: TTCN-3 Core Language Editor
- TTthree: TTCN-3 Compiler
- TTman: Test management, execution, and analysis

This chapter discusses the extension of TTworkbench Basic for Web service testing. It is structured as follows: First, Section 5.1 presents the test adapter and codec that enable the execution of a TTCN-3 abstract test suite, which has been created according to the mapping specified in Chapter 3, against the corresponding Web service. Subsequently, Section 5.2 presents a plug-in for TTworkbench that contains three supportive wizards. Those provide dialog-based means to either use the WSDL2TTCN utility from within TTworkbench or define new, more complex test cases based upon TTCN-3 abstract test suites derived from WSDL descriptions.

5.1 Test Adapter and Codec enabling Test Execution

Before the test adapter and codec enabling Web service testing are discussed, the overall TTworkbench test implementation process is briefly introduced. The process, the files involved, and the interrelation between them are exemplified in Figure 5.1. It shows the test implementation for the TTCN-3 abstract test suite, which is developed throughout Chapter 3 for testing the .NET implementation of the "Movie Database" Web service.

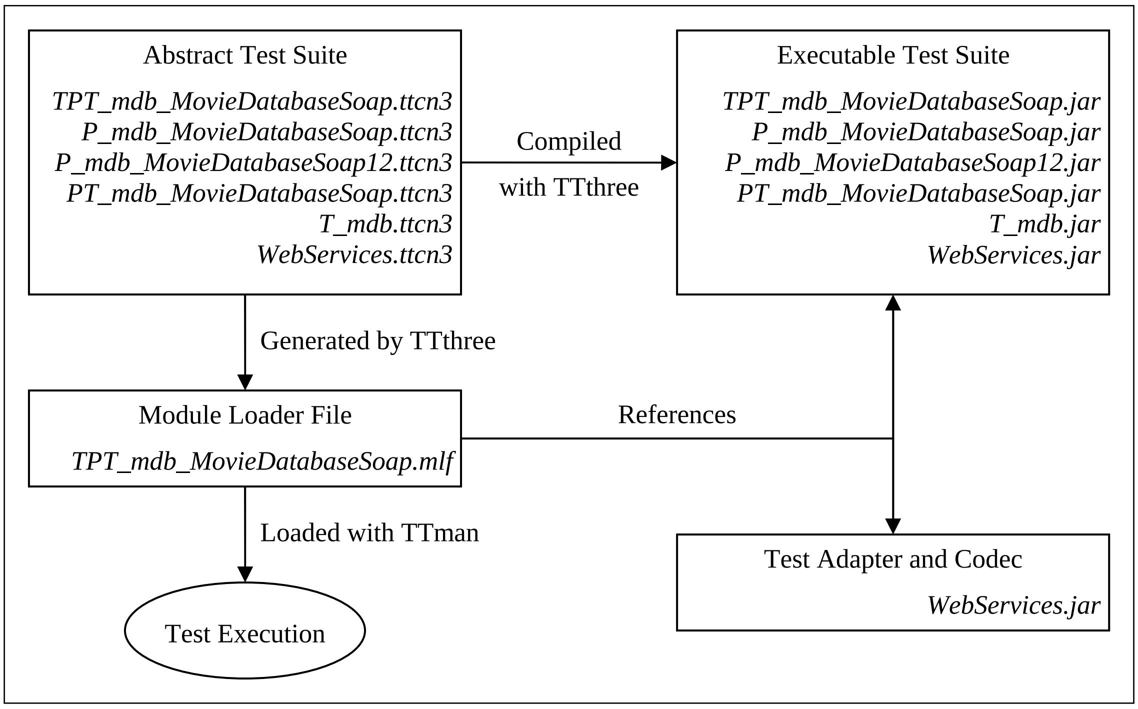

Abstract Test Suite
TPT_mdb_MovieDatabaseSoap.ttcn3
P_mdb_MovieDatabaseSoap.ttcn3
P_mdb_MovieDatabaseSoap12.ttcn3
PT_mdb_MovieDatabaseSoap.ttcn3
T_mdb.ttcn3
WebServices.ttcn3
Compiled with TTthree
Executable Test Suite
TPT_mdb_MovieDatabaseSoap.jar
P_mdb_MovieDatabaseSoap.jar
P_mdb_MovieDatabaseSoap12.jar
PT_mdb_MovieDatabaseSoap.jar
T_mdb.jar
WebServices.jar
Generated by TTthree
Module Loader File
TPT_mdb_MovieDatabaseSoap.mlf
References
Loaded with TTman
Test Adapter and Codec
WebServices.jar
Test Execution

Figure 5.1: TTworkbench test implementation process (cp. [29] Figure 2.1)

The TTthree compiler reads module definitions written in the TTCN-3 core notation and translates them into Java sources. Those are then compiled into byte code class files and combined into a single JAR (Java archive) file for each module. After compilation, the JAR files generated by the TTthree compiler correspond to the TTCN-3 Executable i.e. the part of the TTCN-3 test system that is responsible for the interpretation or execution of TTCN-3 code.

In order to run the generated executable test suite against an actual system under test for example a Web service, a specific test adapter and codec have to be implemented in the Java programming language and provided by means of a single Java archive. The test adapter thereby combines the SA and PA entities of the TTCN-3 test system and the codec corresponds to the CD entity. TTworkbench provides basic implementations for both test adapter and codec from which custom adapters and codecs can derive.

Finally, test parameterization is performed with the so-called module loader file (MLF). It informs the test management, which compiled TTCN-3 module should be executed, which test adapter and codec should be used, and where the respective JAR files can be found. Furthermore, it tells the test management about the test cases that are located in the compiled module and the parameters that are used. A module loader file is an XML file that can be generated automatically during compilation and/or authored manually.

As indicated in the previous paragraphs and depicted in Figure 5.1, the test adapter and codec enabling test execution against Web services are written in the Java programming language and combined into the Java archive *WebServices.jar*. Both test adapter and codec inherit from the basic implementations provided by TTworkbench and override the operations of the TRI and TCI interfaces that are invoked by the TE when executing a test suite generated according to the mapping specified in Chapter 3. Concretely, the test adapter implements the *triSend* operation and the codec implements the *encode* and *decode* operations. In order to deal with SOAP messages and communicate with a Web service under test, the test adapter and codec make use of the APIs provided by the Apache Axis distribution ([1]) version 1.4. Besides, the codec uses the implementation of the WSDL2TTCN utility for a proper encoding and decoding of SOAP messages e.g. for the recovery of element and attribute names from TTCN-3 identifiers. In Figure 5.2, the *WebServicesAdapter* and *WebServicesCodec* classes are depicted together with the most important supportive classes and the interrelations between them. To aid display, all constructors and the methods providing access to private attributes are omitted.

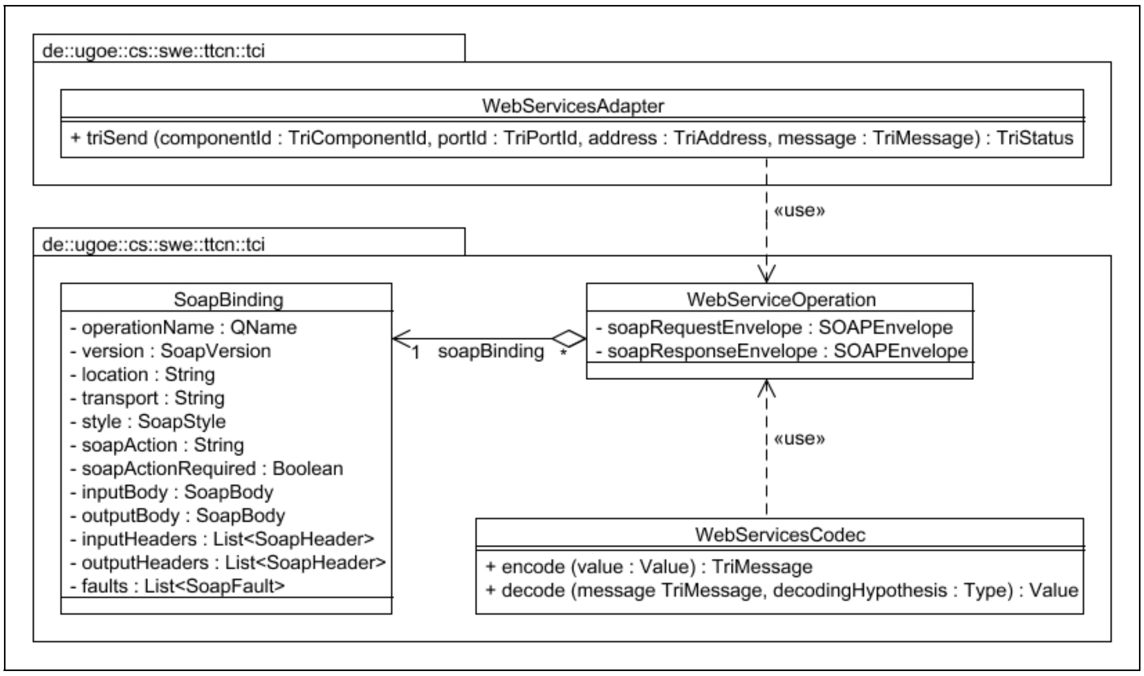

de::ugoe::cs::swe::ttcn::tci
WebServicesAdapter
+ triSend (componentId : TriComponentId, portId : TriPortId, address : TriAddress, message : TriMessage) : TriStatus
«use»
de::ugoe::cs::swe::ttcn::tci
SoapBinding
- operationName : QName
- version : SoapVersion
- location : String
- transport : String
- style : SoapStyle
- soapAction : String
- soapActionRequired : Boolean
- inputBody : SoapBody
- outputBody : SoapBody
- inputHeaders : List<SoapHeader>
- outputHeaders : List<SoapHeader>
- faults : List<SoapFault>
1
soapBinding
*
WebServiceOperation
- soapRequestEnvelope : SOAPEnvelope
- soapResponseEnvelope : SOAPEnvelope
«use»
WebServicesCodec
+ encode (value : Value) : TriMessage
+ decode (message TriMessage, decodingHypothesis : Type) : Value

Figure 5.2: Main classes of the test adapter and codec implementation

When a message is send to a Web service, it is first encoded from a structured TTCN-3 value to an appropriate SOAP message by the *encode* method of the *WebServicesCodec* class. The value, which is passed into the method, is thereby expected to be a TTCN-3 record value that adheres to the rules specified in Section 3.4. That is, the first field is named *soapBinding* and it contains SOAP binding information for the current Web service operation, whereas the remaining fields represent the message parts. The *encode* method first transfers the SOAP binding information from the *soapBinding* field to an instance of the *SoapBinding* class, which allows to access it in a more convenient way. In the following, the retrieved SOAP binding information is used to construct a SOAP envelope from the remaining fields of the TTCN-3 record value. On the one hand, the SOAP binding information tells the codec whether a certain message part has to appear under the SOAP body or the SOAP header. On the other hand, the codec evaluates the message style to know whether message parts appearing under the SOAP body need to be enclosed by a wrapper element. The further encoding of the fields, which represent message parts, to respective XML fragments in the SOAP envelope is straightforward and therefore not discussed. For the time being, the *WebServicesCodec.encode* method supports only construction of the SOAP body and only the rpc/literal, document/literal, and wrapped message styles. After the SOAP envelope has been constructed, it is stored along with the SOAP binding information in an instance of the *WebServiceOperation* class. This object is finally serialized into a binary string representation and returned to the TE, which passes it on to the *WebServicesAdapter* via the *triSend* method.

The *triSend* method recovers the *WebServiceOperation* instance and uses the contained SOAP binding information to configure an object of the *Call* class. In detail, the SOAP binding information provides the location of the Web service, the transport that has to be used and possibly a SOAP action that has to be specified. The *Call* class is provided by the Axis distribution and it facilitates communication with a Web service. After the *Call* instance has been configured, its *invoke* method is executed in order to send the SOAP request contained in the *WebServiceOperation* instance to the Web service under test and receive its response. The *invoke* method, which blocks until the receipt of a response, is thereby executed on a separate thread, so that the *triSend* method is able to return immediately. Inside the new thread, a SOAP response that is received from the Web service is assigned to the *soapResponseEnvelope* field of the *WebServiceOperation* object. Thereafter, the latter object is again serialized into a binary string and passed to the TE via the TRI operation *triEnqueueMsg*.

In order to match the received SOAP response against a specific message, it has to be decoded into a structured TTCN-3 value. For that purpose, the TE passes the serialized

WebServiceOperation object containing the SOAP response as well as the assumed type of the message, the so-called decoding hypothesis, to the *WebServicesCodec.decode* method. The latter argument is thereby very important for the decoding process, because the SOAP response cannot be decoded deterministically without it. This is due to the fact that the TTCN-3 types representing parts of the SOAP envelope are derived from XML Schema descriptions, but there are many, possibly an unlimited number of XML Schema descriptions for the same XML fragment. Hence, it is not possible to determine to which structured TTCN-3 value a SOAP fragment should be decoded without the decoding hypothesis. As a result of this, the TTCN-3 value is constructed according to the decoding hypothesis, whereupon it is checked whether the SOAP envelope contains the assumed elements and attributes. If they are found, their content is copied to the TTCN-3 value. Otherwise, the *decode* method fails and the distinct *null* value is passed back. During the decoding process, the SOAP binding information contained in the *WebServiceOperation* object is again used to determine whether a message part appears under the SOAP body or the SOAP header and whether message parts appearing under the SOAP body have to be enclosed by a wrapper element. As the *encode* method, the *WebServicesCodec.decode* method supports only the rpc/literal, document/literal, and wrapped message styles and for the time being only message parts appearing under the SOAP body are decoded.

The decoding of SOAP envelopes according to the decoding hypothesis is problematic at two points. The first problem arises when the assumed message type contains fields or variants that are of a TTCN-3 union type or the *anytype*. In those cases, the decoding hypothesis provides no information which variant of the union or *anytype* is expected, so that the codec has to loop over all variants and try to decode the current fragment of the SOAP envelope according to one of them. As soon as the decoding according to a variant succeeds, it is assumed that the current SOAP fragment was decoded correctly, but this is not necessarily the case. Assuming there is a TTCN-3 union type with two variants that are of different name aliases for the TTCN-3 *charstring* type, it would be possible to decode a SOAP fragment with simple text content according to both variants without knowing which one is actually expected.

The second problem occurs when the decoding hypothesis contains a field or variant that is of a TTCN-3 set-of type. In such cases, no information is provided on how many elements are expected and how many elements are actually allowed by the set-of type. The former restriction is circumvented by simply trying to decode as many elements as possible from the current SOAP fragment. The second restriction is more problematic as it eliminates the possibility to distinguish between valid and invalid empty lists. That is, if no elements can be decoded from the current SOAP fragment, it is not possible to determine whether the resulting empty set-of value is valid according to its type. Since assigning zero to the length of a set-of value, whose type defines a greater minimum length, causes a test execution error i.e. the abortion of the test case, empty lists are generally not supported. Consequently, when the decoding hypothesis contains a field or variant of a TTCN-3 set-of type but no elements can be decoded from the current SOAP fragment, the *decode* method fails by returning the distinct *null* value.

5.2 Plug-in bundling Supportive Wizards

Since TTworkbench is based on the Eclipse platform, it inherits their high extensibility. The Eclipse platform is build around the concept of plug-ins, structured bundles of code and/or data that contribute function to the system. The platform itself is structured as set of subsystems that provide key functionality and run on top of a small runtime engine. Every subsystem is implemented in one or more plug-ins that have so-called extension points, well-defined places where other plug-ins can add functionality to the platform. Those other plug-ins are in turn allowed to define their own extension points for further customization, so the Eclipse platform can be seen as a layered system of plug-ins.

At its heart, TTworkbench comprises a few plug-ins, which extend the Eclipse platform with functionality for TTCN-3 test specification, compilation, test case execution, and analysis. The plug-in that is discussed in this section furthermore extends TTworkbench by contributing three wizards. Two of the wizards provide dialog-based utilization of the WSDL2TTCN utility from within TTworkbench. The first one simply executes the automated mapping of a WSDL document to a TTCN-3 abstract test suite as introduced in section 5.2.1. The second wizard presented in Section 5.2.2 performs not only the mapping but also creates a TTCN-3 project for the generated abstract test suite. The project is readily configured for Web service testing for what the test adapter and codec introduced in Section 5.1 are re-used. Section 5.2.3 finally discusses the third wizard that facilitates the dialog-based specification of new, more complex test cases based on TTCN-3 abstract test suites generated according to the mapping specified in Chapter 3.

Plug-in	Description
org.eclipse.core.resources *org.eclipse.core.runtime* *org.eclipse.jdt.ui* *org.eclipse.ltk.core.refactoring* *org.eclipse.text* *org.eclipse.ui* *org.eclipse.ide*	These plug-ins are included in the Eclipse SDK (Software Development Kit), which bundles the Eclipse platform with major tools useful for plug-in development. They also ship with TTworkbench Basic and provide basic functions for the *de.ugoe.cs.swe.webservices* plug-in.
com.testingtech.ttworkbench.core	This plug-in ships with TTworkbench Basic and it is required to generate TTCN-3 projects and configure them for Web service testing.
de.ugoe.cs.swe.trex.antlr *de.ugoe.cs.swe.trex.core*	These two plug-ins are part of TRex ([48]), a TTCN-3 Refactoring and Metrics Tool. They are used in version 0.5.3 to parse TTCN-3 sources when a new test case is defined based upon an existing abstract test suite.

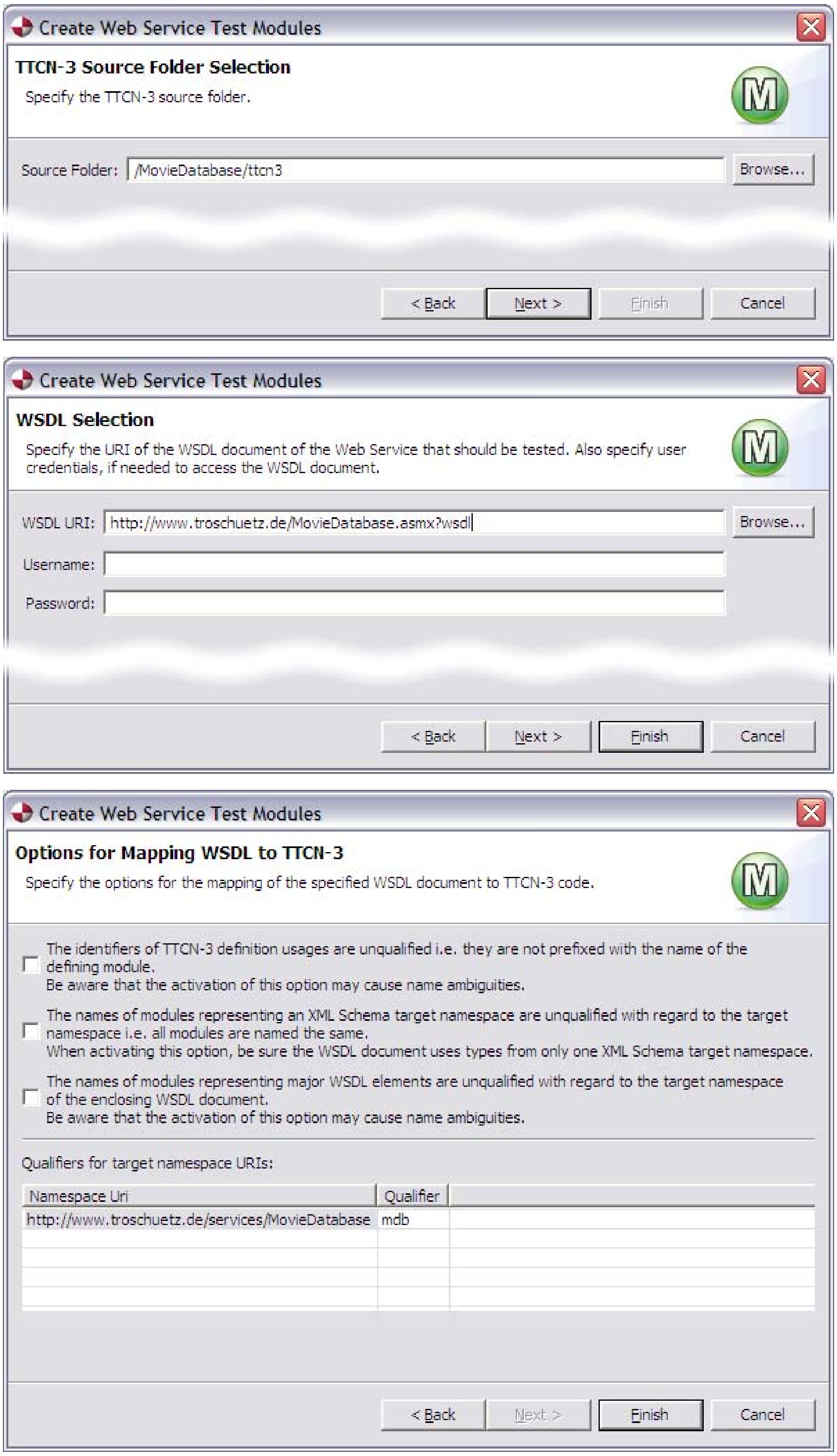
Table 5.1: Plug-ins required by the *de.ugoe.cs.swe.webservices* plug-in

The plug-in bundling the three wizards has the id "de.ugoe.cs.swe.webservices" and it is packaged into a single JAR file. In order to use the three wizards, the Java archive has to be copied into the "plugins" directory of a TTworkbench installation. Furthermore, it is required that the plug-ins listed in Table 5.1 are also present.

5.2.1 "Web Service Test Modules" Wizard

The "Web Service Test Modules" wizard is the first of two wizards that provide dialog-based utilization of the WSDL2TTCN utility from within TTworkbench. It comprises three dialog pages, which are shown in Figure 5.3. The wizard requires that at least one project is already created in the workspace and it can be invoked either via the main menu or via the context menu that appears after performing a right-click on a project or folder in the "TTCN-3 Projects" view.

On the first wizard page, the user is requested to specify the source folder to which the TTCN-3 abstract test suite resulting from the mapping between WSDL and TTCN-3 should be written. The user can either directly specify the source folder in a text field or alternatively browse the workspace to select a source folder.



TTCN-3 Source Folder Selection

Specify the TTCN-3 source folder.

Source Folder: /MovieDatabase/ttcn3

WSDL Selection

Specify the URI of the WSDL document of the Web Service that should be tested. Also specify user credentials, if needed to access the WSDL document.

WSDL URI: http://www.troschuetz.de/MovieDatabase.asmx?wsdl

Username:

Password:

Options for Mapping WSDL to TTCN-3

Specify the options for the mapping of the specified WSDL document to TTCN-3 code.

- [ ] The identifiers of TTCN-3 definition usages are unqualified i.e. they are not prefixed with the name of the defining module. Be aware that the activation of this option may cause name ambiguities.
- [ ] The names of modules representing an XML Schema target namespace are unqualified with regard to the target namespace i.e. all modules are named the same. When activating this option, be sure the WSDL document uses types from only one XML Schema target namespace.
- [ ] The names of modules representing major WSDL elements are unqualified with regard to the target namespace of the enclosing WSDL document. Be aware that the activation of this option may cause name ambiguities.

Qualifiers for target namespace URIs:

Namespace Uri	Qualifier
http://www.troschuetz.de/services/MovieDatabase	mdb

Figure 5.3: Pages of the "Web Service Test Modules" wizard

With a source folder being selected, the second wizard page requests the user to specify the URI of the WSDL description, which should be mapped to TTCN-3. The user can either directly specify the URI in a text field or browse the file system to select a WSDL document. If the retrieval of the WSDL description requires authentication, a user name and password can be specified in two additional text fields. In case the wizard is able to obtain the specified WSDL document, for what the Web Services Description Language for Java Toolkit ([25]) version 1.6.2 is used, the user can either finish the wizard or proceed to the optional third wizard page.

The last wizard page allows the user to configure the mapping of the obtained WSDL description to a TTCN-3 abstract test suite. On the one hand, this includes three options regarding the qualification of module identifiers and identifiers of TTCN-3 definition usages. On the other hand, the user can specify qualifiers for the URIs of all WSDL and XML Schema target namespaces that appear in the retrieved WSDL description.

After the user has finally finished the "Web Service Test Modules" wizard, the already retrieved WSDL description is mapped to TTCN-3 using the functionality provided by the WSDL2TTCN utility. At first, an instance of the *WSDL2TTCN* class is created and configured according to the specified mapping options. Subsequently, the mapping is executed by passing the obtained WSDL4J representation of the WSDL document to the appropriate overload of the *mapWSDL2TTCN* operation. After the mapping is executed, the generated TTCN-3 modules, whose Java representations are exposed via attributes of the *WSDL2TTCN* object, are written to the TTCN-3 source folder initially specified by the user.

If a module resulting from the mapping already exists in the source folder, the wizard opens a modal dialog to ask the user whether the existing module should be overwritten. Besides the provision of "Yes" and "No" options, the specially implemented dialog also allows the user to specify that all or none of the TTCN-3 modules already existing on the file system should be overwritten. Those additional options are provided, since the mapping possibly generates a great number of modules that already exist on the file system. Exceptions to this overwriting procedure are the creation of the *XSDAUX* and *WebServices* modules. Because they are commonly used for the testing of any Web service, those TTCN-3 modules are never overwritten.

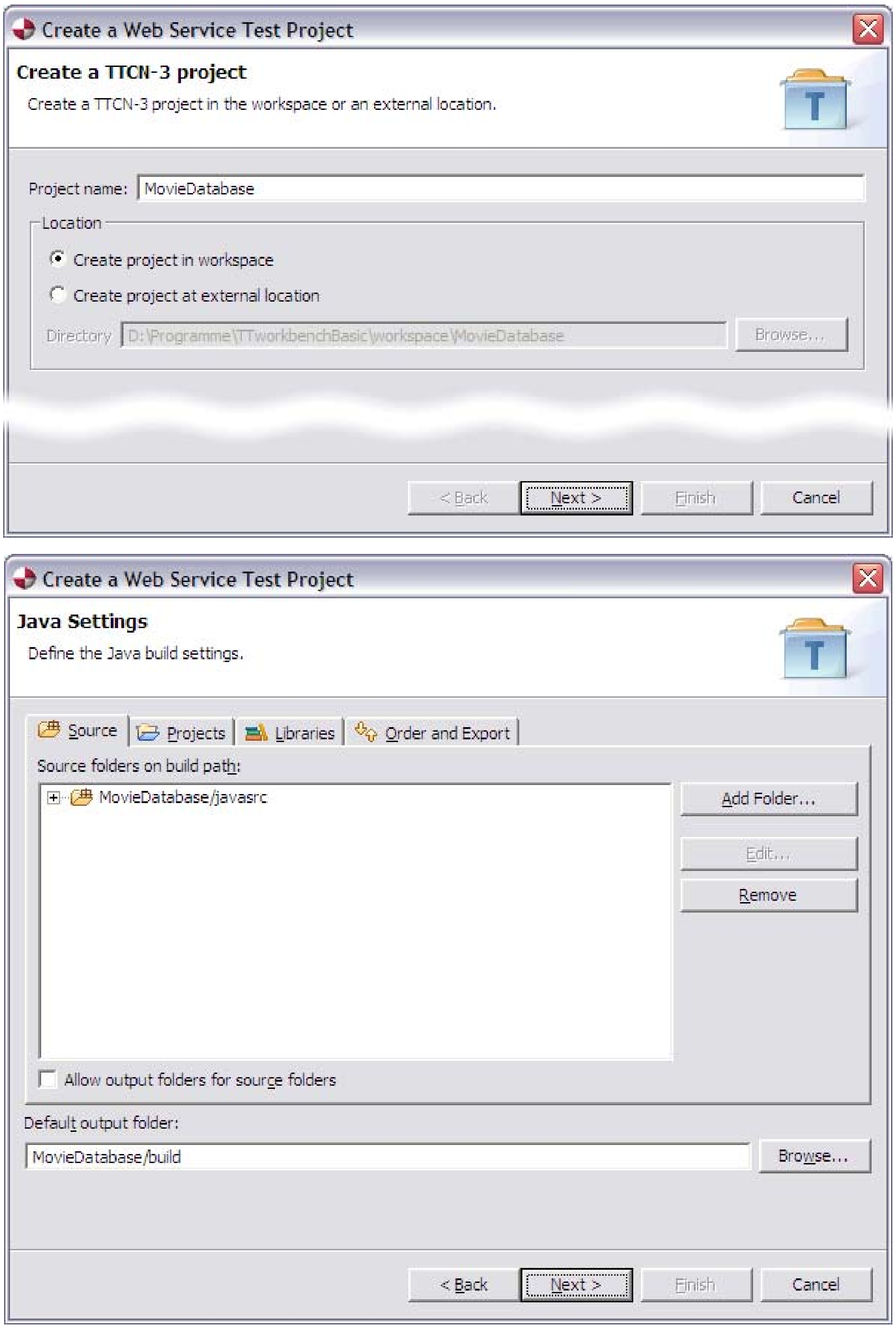
5.2.2 "Web Service Test Project" Wizard

The "Web Service Test Project" wizard is the second wizard that provides dialog-based usage of the WSDL2TTCN utility from within TTworkbench. Besides the execution of the automated mapping between WSDL and TTCN-3, the main purpose of the wizard is the creation of a TTCN-3 project that takes up the generated abstract test suite and that is readily configured for Web service testing. For the latter purpose, the test adapter and codec introduced in Section 5.1 are re-used. Similar to the "Web Service Test Modules" wizard, the "Web Service Test Project" wizard can be invoked either via the main menu or via the context menu that appears after performing a right-click on a project or folder in the "TTCN-3 Projects" view.

The "Web Service Test Project" wizard includes five pages. The first three wizard pages resemble the TTworkbench wizard for the creation of a new TTCN-3 project as shown in Figure 5.4 and Figure 5.5. For that purpose, the implementations of the TTworkbench wizard pages, which are included in the *com.testingtech.ttworkbench.core* plug-in, are re-used or extended, respectively. The remaining two pages of the "Web Service Test Project" wizard enable the selection of a WSDL description as well as the specification of mapping options. They are re-used from the "Web Service Test Modules" wizard, so that they are used as described in Section 5.2.1 and except for the wizard title and icon appear as shown in Figure 5.3.

The first "Web Service Test Project" wizard page is re-used from the TTworkbench wizard and it allows the user to create a TTCN-3 project in the workspace or an external location. When the user proceeds to the next wizard page, a project with the specified name is immediately created at the specified location and the project is pre-configured for Java development.

This pre-configuration can be edited by the user on the second page of the "Web Service Test Project" wizard. The individual Java settings are not relevant in the current context and thus not discussed. When the user advances to the next page, the project is pre-configured for TTCN-3 development. Since the "Web Service Test Project" wizard aims at the creation of a project that is configured for Web service testing, the behavior of the TTworkbench wizard needs to be extended at this point. Thus, the implementation of the second TTworkbench wizard page is not simply re-used but inherited and enhanced. After execution of the base behavior, the extended implementation of the second wizard page copies the *WebServices.jar* file that packages the test adapter and codec introduced in Section 5.1 into the project directory. Thereafter, the pre-configuration for TTCN-3 development is changed, so that the TTthree compiler generates a module loader file on compilation of TTCN-3 modules and these MLFs reference the test adapter packaged in the *WebServices.jar* file.



Create a Web Service Test Project

Create a TTCN-3 project

Create a TTCN-3 project in the workspace or an external location.

Project name: MovieDatabase

Location

[✓ Create project in workspace]

Create project at external location

Directory: D:\Programme\TTworkbenchBasic\workspace\MovieDatabase

Create a Web Service Test Project

Java Settings

Define the Java build settings.

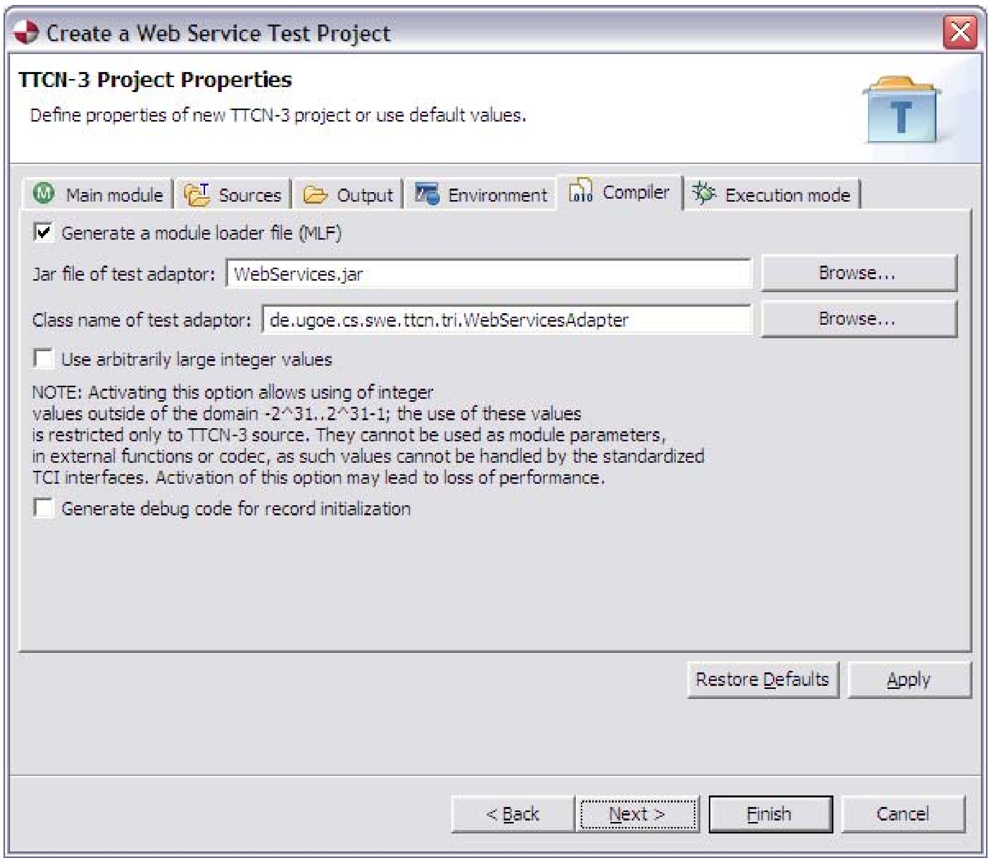
Source folders on build path:

MovieDatabase/javasrc

Allow output folders for source folders

Default output folder: MovieDatabase/build

Figure 5.4: First and second page of the "Web Service Test Project" wizard



Create a Web Service Test Project

TTCN-3 Project Properties

Define properties of new TTCN-3 project or use default values.

[✓] Generate a module loader file (MLF)

Jar file of test adaptor: WebServices.jar

Class name of test adaptor: de.ugoe.cs.swe.ttcn.tri.WebServicesAdapter

[ ] Use arbitrarily large integer values

NOTE: Activating this option allows using of integer
values outside of the domain -2^31..2^31-1; the use of these values
is restricted only to TTCN-3 source. They cannot be used as module parameters,
in external functions or codec, as such values cannot be handled by the standardized
TCI interfaces. Activation of this option may lead to loss of performance.

[ ] Generate debug code for record initialization

Figure 5.5: Third page of the "Web Service Test Project" wizard

The pre-configuration of the project for TTCN-3 development can be edited by the user on the third page of the "Web Service Test Project" wizard, which is again simply re-used from the TTworkbench wizard. Figure 5.5 shows the third wizard page with its "Compiler" tab being selected and the compiler settings being configured as described. The remaining TTCN-3 project properties are not relevant in the current context and thus not discussed. After possibly adjusting the TTCN-3 properties to his needs, the user can finish the wizard immediately, leaving the TTCN-3 project readily configured for Web service testing but empty i.e. no TTCN-3 source files are created.

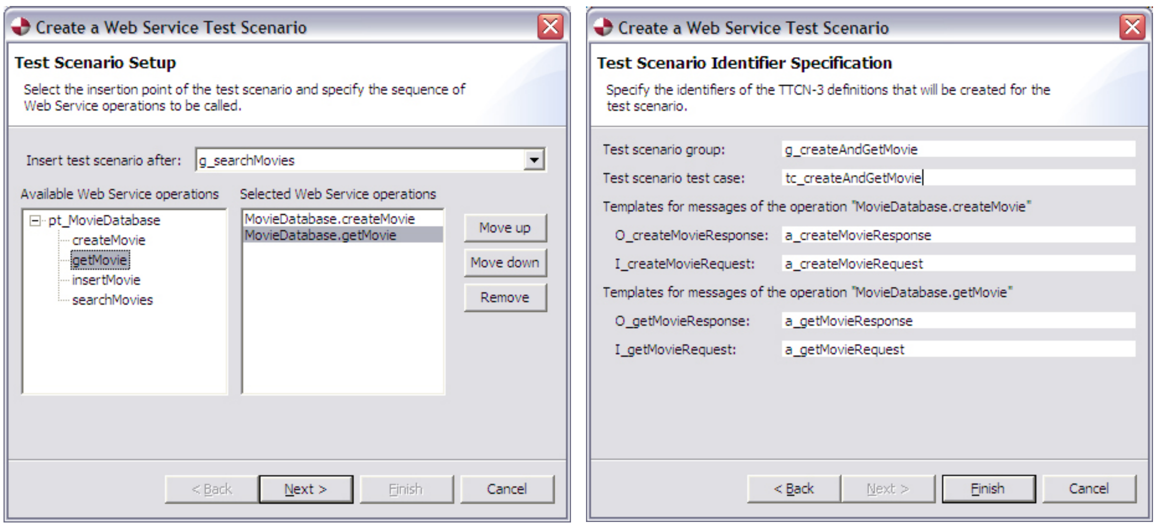
Alternatively, the user can advance to the last two wizard pages that provide means to select a WSDL description and configure the mapping between WSDL and TTCN-3. If the user selects a WSDL document and finishes the wizard, the Web service description is mapped to a TTCN-3 abstract test suite using the functionality of the WSDL2TTCN utility. The use of the wizard pages and the execution of the mapping are almost the same as in the context of the "Web Service Test Modules" wizard introduced in Section 5.2.1. The only difference is that the generated TTCN-3 abstract test suite is not written to a user-specified directory but the standard TTCN-3 source folder named "ttcn3".

5.2.3 "Web Service Test Scenario" Wizard

A TTCN-3 abstract test suite generated according to the mapping specified in Chapter 3 contains only simple test cases for testing the Web service operations individually. This is due to the fact that a Web service description provides no semantics for the described operations and thus makes it hard or even impossible to derive complex test scenarios automatically. The "Web Service Test Scenario" wizard provides dialog-based means to ease the specification of new, more complex test cases that base on a TTCN-3 abstract test suite derived from a WSDL description and may involve numerous calls to Web service operations. The wizard comprises two pages, which are displayed in Figure 5.6, and it can be invoked via the "Web Service Testing" menu item, which is added to the TTworkbench main menu by the *de.ugoe.cs.swe.webservices* plug-in.

After invocation of the "Web Service Test Scenario" wizard, it is first verified whether a TTCN-3 source file is opened and selected for editing. If not, the wizard immediately aborts with an error message. Otherwise, the TTCN-3 source file and all other TTCN-3 sources in the same project are analyzed using the parser functionality provided by the TRex plug-ins. In the course of the analysis, the source code written in the TTCN-3 core notation is lexed and parsed using an ANTLR (Another Tool for Language Recognition) grammar and the resulting syntax tree is used to generate a symbol table. The resulting syntax tree and symbol table constitute representations of TTCN-3 in Java, which are less comfortable to work with than the API implemented for the WSDL2TTCN utility. Nevertheless, in the context of the "Web Service Test Scenario" wizard the effort to work with the syntax tree and symbol table is less than adding parser functionality to the Java representations of TTCN-3 implemented for the WSDL2TTCN utility. For future work, it may be an interesting option to extend the TTCN-3 API of the WSDL2TTCN utility and combine it with the TRex parser functionality. With all TTCN-3 sources files being analyzed, it is subsequently verified whether the selected source file contains a TTCN-3 module for testing a WSDL port type as described in Section 3.6 and depicted in Figure 3.23. If no appropriate module is found, the wizard immediately aborts with an error message. Otherwise, the found module is used as basis and final target of the new test scenario.

The initial page of the "Web Service Test Scenario" wizard enables the user to select the TTCN-3 definition already existing on the top level of the target module after which the new test scenario should be inserted. Furthermore, the user can specify the sequence of Web service operations that should be invoked in the course of the test scenario. For that purpose, the TTCN-3 test component, on which the behavior for testing a WSDL port type is executed, is determined and analyzed. Thereby, all Web service operations that can be invoked via ports owned by the test component are listed inside the tree view positioned at the left edge of the first wizard page. The operations are grouped per port and can be selected for the test scenario by performing a double-click on the respective tree view item. A selected Web service operation can be moved in or removed from the ordered sequence of operations by selecting the respective list item and clicking on the appropriate button positioned at the wizard's right edge. In order to advance to the next wizard page at least one Web service operation has to be selected.



Test Scenario Setup

Select the insertion point of the test scenario and specify the sequence of Web Service operations to be called.

Insert test scenario after: g_searchMovies

Available Web Service operations	Selected Web Service operations
pt_MovieDatabase	MovieDatabase.createMovie
createMovie	MovieDatabase.getMovie
getMovie	
insertMovie	
searchMovies	

Test Scenario Identifier Specification

Specify the identifiers of the TTCN-3 definitions that will be created for the test scenario.

Test scenario group:	g_createAndGetMovie
Test scenario test case:	tc_createAndGetMovie

Templates for messages of the operation "MovieDatabase.createMovie"

O_createMovieResponse:	a_createMovieResponse
I_createMovieRequest:	a_createMovieRequest

Templates for messages of the operation "MovieDatabase.getMovie"

O_getMovieResponse:	a_getMovieResponse
I_getMovieRequest:	a_getMovieRequest

Figure 5.6: Pages of the "Web Service Test Scenario" wizard

The second page of the "Web Service Test Scenario" wizard requires the user to specify the identifiers of the TTCN-3 definitions that will be created for the test scenario. These include a test case, templates for all input, output, and fault messages of the selected Web service operations, and a group that encloses the former test scenario definitions. All specified identifiers are required to adhere to the naming conventions specified in Section 3.2.1 and it is verified whether they would cause name ambiguities in the target module. That is, the specified identifiers must be unique among each other and among the identifiers of all top-level definitions in the target module. In case an identifier is invalid, the corresponding text field is highlighted and an error message is shown. When all specified identifiers are valid, the wizard can be finished.

After the user has finally finished the "Web Service Test Scenario" wizard, the new test scenario is created and inserted into the target module. For that purpose, a syntax tree representing the test scenario group and all enclosed TTCN-3 definitions is constructed, transformed into TTCN-3 core notation by using the TRex pretty printer, and finally inserted into the target module after the already existing definition initially specified by the user.

The test case of the test scenario is similar to those created during the mapping between WSDL and TTCN-3. In fact, it is like a sequence of the test behavior defined by those test cases that test the individual Web service operations selected for the test scenario. For each operation in the test scenario sequence, first the template for the input message is transmitted via the port owned by the test component. In case the operation is a one-way operation i.e. the Web service under test will not respond, the test verdict is set to "`pass`". Otherwise, the timer owned by the test component is started to safeguard the test behavior against inactivity of the Web service to be tested. The timer will run for the number of seconds defined by the value of the *c_timeout* constant, which is passed to the *start* operation as duration parameter. After starting the timer, the test case defines an *alt* statement that specifies several alternatives. The first alternative is executed, if the template defined for the output message of the Web service operation is received on the port owned by the test component. It will stop the timer and set the test verdict to "`pass`". If the operation specifies fault messages, further alternatives are defined for the receipt of their corresponding templates. Each of those alternatives will also stop the running timer, but set the test verdict to "`fail`". At last, the *alt* statement refers to the *alt_ReceiveAnyOrTimeout* altstep in order to catch the receipt of unexpected messages or a timeout.

The values of the message templates created for the input, output, and fault messages of the selected Web service operations do not follow the rules that are suggested by the mapping between WSDL and TTCN-3 (s. Table 3.6). For the time being, all message templates have the distinct value "`omit`" and a test user is necessarily required to specify real test data by hand. This is due to the fact that with the syntax tree and symbol table representations of TTCN-3 definitions, the generation of values following the suggested rules would have been to complex.

To conclude this section, Figure 5.7 shows an exemplary test scenario generated by the "Web Service Test Scenario" wizard. The scenario tests the .NET implementation of the "Movie Database" Web service and includes subsequent calls to the *createMovie* and the *getMovie* operations. In order to generate the exemplary scenario, the "Web Service Test Scenario" wizard was invoked on the module for testing the WSDL port type of the .NET implementation of the exemplary "Movie Database" Web service (s. Figure 3.23) and configured as depicted in Figure 5.6.

In the displayed test scenario, the references to the component type *TC_MovieDatabase* (Figure 5.7 line 6) and the altstep *alt_ReceiveAnyOrTimeout* (Figure 5.7 lines 14 and 23) are not prefixed with the module identifier. This is because the used version of the TRex pretty printer would not generate the delimiting point character after the module identifier and thus would create erroneous TTCN-3 code.

```
01 group g_createAndGetMovie {
02    template PT_mdb_MovieDatabase.O_createMovieResponse a_createMovieResponse := omit;
03    template PT_mdb_MovieDatabase.I_createMovieRequest a_createMovieRequest(
            in WebServices.SoapBinding p_soapBinding) := omit;
04    template PT_mdb_MovieDatabase.O_getMovieResponse a_getMovieResponse := omit;
05    template PT_mdb_MovieDatabase.I_getMovieRequest a_getMovieRequest(
            in WebServices.SoapBinding p_soapBinding) := omit;
06    testcase tc_createAndGetMovie(in WebServices.SoapBindings p_soapBindings)
            runs on TC_MovieDatabase {
07       pt_MovieDatabase.send(
            TPT_mdb_MovieDatabase.a_createMovieRequest(p_soapBindings[0]));
08       t_timeoutGuard.start(TPT_mdb_MovieDatabase.c_timeout);
09       alt {
10          [] pt_MovieDatabase.receive(TPT_mdb_MovieDatabase.a_createMovieResponse) {
11             t_timeoutGuard.stop;
12             setverdict(pass);
13          };
14          [] alt_ReceiveAnyOrTimeout();
15       }
16       pt_MovieDatabase.send(
            TPT_mdb_MovieDatabase.a_getMovieRequest(p_soapBindings[1]));
17       t_timeoutGuard.start(TPT_mdb_MovieDatabase.c_timeout);
18       alt {
19          [] pt_MovieDatabase.receive(TPT_mdb_MovieDatabase.a_getMovieResponse) {
20             t_timeoutGuard.stop;
21             setverdict(pass);
22          };
23          [] alt_ReceiveAnyOrTimeout();
24       }
25    }
26 }
```

Figure 5.7: TTCN-3 output of the "Web Service Test Scenario" wizard

6 Conclusion

6.1 Summary

In this thesis, a framework for Web service testing with TTCN-3 has been presented. It bases on an automated, specification-based testing approach developed in related work ([23], [24], and [46]). From the WSDL description of a Web service, an abstract test suite expressed with TTCN-3 is derived that is independent of the test platform and the concrete system to be tested. This abstract test suite is subsequently executed against the Web service by a TTCN-3 test system implemented in a native programming language. Although the existing research work includes case studies that prove the concept of the testing approach, it lacks a thorough specification of the mapping between WSDL and TTCN-3.

Thus, detailed rules have been presented for the mapping of a Web service description given by means of a WSDL document to a TTCN-3 abstract test suite that allows basic testing of the Web service. In addition to the specification of mapping rules for the individual major WSDL elements, the mapping between TTCN-3 and XML Schema, which forms the default type system of WSDL descriptions, has been examined. The discussion of the proposed mapping between WSDL and TTCN-3 has been exemplified through the derivation of a TTCN-3 abstract test suite from the WSDL description of an exemplary "Movie Database" Web service.

The proposed mapping between WSDL and TTCN-3 can be done manually but it is also well capable of being automated. Because an automated mapping is less error-prone and most notably less time-consuming, the WSDL2TTCN utility was implemented in the context of this thesis. The architecture, two implementation details, the invocation, and the available command line options of the Java console application have been presented. Furthermore, an option has been discussed that allows changing the mapping between WSDL and TTCN-3 in certain points in order to generate a TTCN-3 abstract test suite fully compatible to TTworkbench Basic.

Lastly, the extension of TTworkbench Basic, a TTCN-3 test development and execution environment, for Web service testing has been presented. At first, the test adapter and codec, which facilitate the execution of TTCN-3 abstract test suites generated according to the mapping between WSDL and TTCN-3, have been discussed. Subsequently, three wizards have been introduced, which provide dialog-based means to either use the WSDL2TTCN utility from within TTworkbench or specify new, more complex test cases based upon TTCN-3 abstract test suites derived from WSDL descriptions.

6.2 Outlook

Regarding the mapping between WSDL and TTCN-3, it would be desirable to adapt it to WSDL 2.0. This version has been disregarded in the context of this thesis, because it is still being standardized at the time of this writing and hence not widely supported or used. Nevertheless, WSDL 2.0 will presumably displace WSDL 1.1 bit by bit after becoming a W3C Recommendation.

Besides an adaptation of the proposed mapping to WSDL 2.0, it could be worthwhile to add support for the document/encoded and the rpc/encoded message styles. They have not been considered in the context of this thesis, because their utilization is forbidden by the WS-I Basic Profile Version 1.0 and 1.1 due to interoperability concerns. However, there are some Web services that do not conform to the WS-I Basic Profile and that make use of those message styles.

All changes that will be made to the mapping between WSDL and TTCN-3 should also be adapted by the WSDL2TTCN utility as well as the test adapter and codec that enable test execution with TTworkbench Basic. In case of the latter, it would furthermore be desirable to add support for the encoding and decoding of the *Header* element of SOAP envelopes.

Finally, it is certainly worthwhile to enhance the "Web Service Test Scenario" wizard in order to better support the test user in defining new, more complex test cases based on TTCN-3 abstract test suites derived from WSDL descriptions. A possible extension may be to allow composing a new test case from the operations of multiple WSDL port types. Moreover, it would be useful to provide better support in defining the message templates, possibly by dialog-based means. As a part of extending the wizard, it may be an interesting and useful option to enhance the Java representations of TTCN-3, which were implemented in the context of the WSDL2TTCN utility, and for example combine them with the TRex parser functionality.

Bibliography

The validity of all listed URIs was lastly checked on June 6, 2007.

[1] Apache <Web Services /> Project: Web Services - Axis
http://ws.apache.org/axis/

[2] Alonso, G., F. Casati, H. Kuno, and V. Machiraju: Web Services
First Edition, Springer Verlag, Berlin Heidelberg (Germany), 2004

[3] Butek, R.: Handle namespaces in SOAP messages you create by hand
IBM developerWorks article, 2005-05-03
http://www-128.ibm.com/developerworks/webservices/library/ws-tip-namespace.html

[4] Butek, R.: Which style of WSDL should I use?
IBM developerWorks article, 2005-05-24
http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[5] Eclipse Foundation: Eclipse
http://www.eclipse.org/

[6] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language
ETSI ES 201 873-1 V3.1.1, ETSI, Sophia Antipolis Cedex (France), 2005-06

[7] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)
ETSI ES 201 873-5 V3.1.1, ETSI, Sophia Antipolis Cedex (France), 2005-06

[8] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)
ETSI ES 201 873-6 V3.1.1, ETSI, Sophia Antipolis Cedex (France), 2005-06

[9] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML Schema with TTCN-3
ETSI ES/MTS-201 873-9 V1.1.1, ETSI, Sophia Antipolis Cedex (France), 2006-10

[10] ETSI: TTCN-3 Home page
http://www.ttcn-3.org/

[11] Ewald, T.: The Argument Against SOAP Encoding
Microsoft Developer Network SOAP Technical Article, 2002-10
http://msdn2.microsoft.com/en-us/library/ms995710.aspx

[12] Goldberg, C.: WebInject - (HTTP) Web Application and Web Services Test Tool
http://www.webinject.org/

[13] Grabowski, J., D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock: An introduction to the testing and test control notation (TTCN-3)
In: Computer Networks Volume 42 Issue 3, Elsevier, Amsterdam (Netherlands), 2003-06

[14] Graham, S., D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamura, P. Fremantle, D. König, and C. Zentner: Building Web Services with Java
Second Edition, Sams Publishing, Indianapolis (USA), 2005

[15] International Telecommunication Union: The Evolution of TTCN
ITU-T Study Group 7 Special Issue, 2002-09-30
http://www.itu.int/ITU-T/studygroups/com07/ttcn.html

[16] Jadhav M., and M. Ahmed: Automate Web service testing, Part 2: Test a Web service with XMLUnit
IBM developerWorks article, 2007-03-26
http://www-128.ibm.com/developerworks/edu/ws-dw-ws-soa-autotest2.html

[17] Java Community Process: JSR-000110 Java™ APIs for WSDL
Maintenance Release 2, 2006-09-12
http://jcp.org/aboutJava/communityprocess/mrel/jsr110/index2.html

[18] Jeaca D.-M: XML Schema to TTCN-3 Mapping: Importing XMLSchema datatypes into TTCN-3
Diploma Thesis, Politehnica University of Bucharest (Department of Computer Science), Bucharest (Romania), 2004-09

[19] Jeaca D.-M., G. Din, and A. Rennoch: Importing XML Schema datatypes into TTCN-3
In: Proceedings of the 3rd Workshop on System Testing and Validation (SV04), Paris (France), 2004-12-02

[20] McCaffrey, J.: Test Run: Automate Your ASP.NET Web Services Testing
In: MSDN Magazine, 2005-03
http://msdn.microsoft.com/msdnmag/issues/05/03/TestRun/

[21] Microsoft: .NET Framework Home
http://msdn2.microsoft.com/netframework/default.aspx

[22] Puro, Vesa-Matti: TTCN-3 Basics Course Slides
http://www.ttcn3basics.com/

[23] Schieferdecker, I., and B. Stepien: Automated Testing of XML/SOAP based Web Services
In: Proceedings of the 13th Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe Kommunikation in verteilten Systemen (KIVS), Leipzig (Germany), 2003-02-26/28

[24] Schieferdecker, I., D. Vega, and C. Renta: Import of WSDL Definitions in TTCN-3 Targeting Testing of Web Services
In: Proceedings of the 9th International Conference on Integrated Design and Process Technology (IDPT 2006), San Diego (USA), 2006-06-25/30

[25] SourceForge.net: Web Services Description Language for Java Toolkit (WSDL4J)
http://sourceforge.net/projects/wsdl4j

[26] Sumra, R., and R. Venkatvaradan: Web Service's Test Harness: A Functional, Load, and Performance Testing Framework for Web Services
developer.com article, 2003-06-06
http://www.developer.com/services/article.php/2229161

[27] Sun: Java™ 2 Platform Standard Edition 5.0 API Specification
http://java.sun.com/j2se/1.5.0/docs/api/index.html

[28] Testing Technologies: TTworkbench Basic
http://testingtech.de/products/ttwb_basic.php

[29] Testing Technologies: TTworkbench Developer's Guide
Revision 1.2, 2006-02
http://www.testingtech.de/download/TTwb_UsersGuide.zip

[30] W3C: Extensible Markup Language (XML) 1.0 (Fourth Edition)
W3C Recommendation, 2006-08-16 (edited in place 2006-09-29)
http://www.w3.org/TR/2006/REC-xml-20060816

[31] W3C: Namespaces in XML 1.0 (Second Edition)
W3C Recommendation, 2006-08-16
http://www.w3.org/TR/2006/REC-xml-names-20060816/

[32] W3C: Simple Object Access Protocol (SOAP) 1.1
W3C Note, 2000-05-08
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[33] W3C: SOAP Version 1.2 Part 0: Primer
W3C Recommendation, 2003-06-24
http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[34] W3C: SOAP Version 1.2 Part 1: Messaging Framework
W3C Recommendation, 2003-06-24
http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[35] W3C: SOAP Version 1.2 Part 2: Adjuncts
W3C Recommendation, 2003-06-24
http://www.w3.org/TR/2003/REC-soap12-part2-20030624/

[36] W3C: Web Services Architecture
W3C Working Draft, 2002-11-14
http://www.w3.org/TR/2002/WD-ws-arch-20021114/

[37] W3C: Web Services Description Language (WSDL) 1.1
W3C Note, 2001-03-15
http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[38] W3C: XML Schema Part 0: Primer Second Edition
W3C Recommendation, 2004-10-28
http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[39] W3C: XML Schema Part 1: Structures Second Edition
W3C Recommendation, 2004-10-28
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[40] W3C: XML Schema Part 2: Datatypes Second Edition
W3C Recommendation, 2004-10-28
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[41] W3C Member: WSDL 1.1 Binding Extension for SOAP 1.2
W3C Member Submission, 2006-04-05
http://www.w3.org/Submission/2006/SUBM-wsdl11soap12-20060405/

[42] Wang, D., T. Bayer, T. Frotscher, and M. Teufel: Java Web Services mit Apache Axis
First Edition, Software & Support Verlag, Frankfurt (Germany), 2004

[43] Willcock, C., T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz: An Introduction to TTCN-3
First Edition, John Wiley & Sons Ltd, West Sussex (England), 2005

[44] WS-I: Basic Profile Version 1.0
Final Material, 2004-04-16
http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

[45] WS-I: Basic Profile Version 1.1
Final Material, 2006-04-10
http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html

[46] Xiong, P., R. L. Probert, and B. Stepien: An Efficient Formal Testing Approach for Web Service with TTCN-3
In: Proceedings of the 13th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2005), Split (Croatia), 2005-09-15/17

[47] Zeiß, B.: A Refactoring Tool for TTCN-3
Master's Thesis, Georg-August-University Göttingen (Institute for Informatics), Göttingen (Germany), 2006-03-15

[48] Zeiß, B., and H. Neukirchen: TRex - the TTCN-3 Refactoring and Metrics Tool
http://www.trex.informatik.uni-goettingen.de/trac

Appendix A Implementations of the "Movie Database" Web Service

A.1 Apache Axis Framework

```java
package services.moviedatabase;

import java.util.HashMap;
import java.util.Map;

public class MovieDatabase {
   private Map<Integer, Movie> database;

   public MovieDatabase()
   {
      this.database = new Map<Integer, Movie> ();
      this.insertMovie(new Movie(4, "Kill Bill: Vol. 1", new Person("Quentin",
            "Tarantino")));
      this.insertMovie(new Movie(12, "Kill Bill: Vol. 2", new Person("Quentin",
            "Tarantino")));
   }

   public boolean createMovie(int id, String name, Person director) {
      return this.insertMovie(new Movie(id, name, director));
   }

   public boolean insertMovie(Movie movie) {
      if (movie == null || movie.getName() == null || movie.getDirector() == null ||
            movie.getDirector().getFirstName() == null ||
            movie.getDirector().getLastName() == null) {
         return false;
      } else {
         // Does not persist, because the Web Service itself is stateless.
         this.database.put(movie.getId(), movie);

         return true;
      }
   }
      return true;
   }

   public Movie getMovie(int id) {
      return this.database.get(id);
   }

   public Movie[] searchMovies(String[] keywords) {
      if (keywords == null) {
         return null;
      } else {
         // Instead of executing a search, the two default movies are returned.
         Movie[] movies = new Movie[this.database.values().size()];
         this.database.values().toArray(movies);

         return movies;
      }
   }
}
```

Figure A.1: MovieDatabase.java

```java
package services.moviedatabase;

public class Movie {
   private int id;
   private String name;
   private Person director;

   public Movie() {
      this(0, "", new Person());
   }

   public Movie(int id, String name, Person director) {
      this.id = id;
      this.name = name;
      this.director = director;
   }

   public void setId(int id) {
      this.id = id;
   }
   public int getId() {
      return id;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }

   public void setDirector(Person director) {
      this.director = director;
   }
   public Person getDirector() {
      return director;
   }
}
```

Figure A.2: Movie.java

```java
package services.moviedatabase;

public class Person {
   private String firstName;
   private String lastName;

   public Person() {
      this("", "");
   }

   public Person(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public String getFirstName() {
      return firstName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   public String getLastName() {
      return lastName;
   }
}
```

Figure A.3: Person.java

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

   <service name="MovieDatabase" provider="java:RPC" style="wrapped" use="literal">
      <requestFlow><handler type="soapmonitor"/></requestFlow>
      <responseFlow><handler type="soapmonitor"/></responseFlow>

      <parameter name="className" value="services.moviedatabase.MovieDatabase" />
      <parameter name="allowedMethods" value="createMovie, insertMovie, getMovie,
         searchMovies" />
      <namespace>http://www.troschuetz.de/services/MovieDatabase</namespace>

      <beanMapping qname="md:Movie"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Movie"/>
      <beanMapping qname="md:Person"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Person"/>
   </service>

   <service name="MovieDatabaseDoc" provider="java:RPC" style="document"
         use="literal">
      <requestFlow><handler type="soapmonitor"/></requestFlow>
      <responseFlow><handler type="soapmonitor"/></responseFlow>

      <parameter name="className" value="services.moviedatabase.MovieDatabase" />
      <parameter name="allowedMethods" value="insertMovie, getMovie, searchMovies" />
      <namespace>http://www.troschuetz.de/services/MovieDatabase</namespace>

      <beanMapping qname="md:Movie"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Movie"/>
      <beanMapping qname="md:Person"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Person"/>
   </service>

   <service name="MovieDatabaseRpc" provider="java:RPC" style="rpc" use="literal">
      <requestFlow><handler type="soapmonitor"/></requestFlow>
      <responseFlow><handler type="soapmonitor"/></responseFlow>

      <namespace>http://www.troschuetz.de/services/MovieDatabase</namespace>
      <parameter name="className" value="services.moviedatabase.MovieDatabase" />
      <parameter name="allowedMethods" value="createMovie, insertMovie, getMovie,
         searchMovies" />
      <beanMapping qname="md:Movie"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Movie"/>
      <beanMapping qname="md:Person"
         xmlns:md="http://www.troschuetz.de/services/MovieDatabase"
         languageSpecificType="java:services.moviedatabase.Person"/>
   </service>

</deployment>
```

Figure A.4: MovieDatabase.wsdd

A.2 Microsoft .NET Framework

```
<%@ WebService Language="C#" Class="Services.MovieDatabase.MovieDatabase" %>

using System.Web.Services;
using System.Collections.Generic;

namespace Services.MovieDatabase
{
   [WebService(Namespace = "http://www.troschuetz.de/services/MovieDatabase")]
   [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
   public class MovieDatabase : System.Web.Services.WebService
   {
      private Dictionary<int, Movie> database;

      public MovieDatabase()
      {
         this.database = new Dictionary<int, Movie>();
         this.insertMovie(new Movie(4, "Kill Bill: Vol. 1", new Person("Quentin",
            "Tarantino")));
         this.insertMovie(new Movie(12, "Kill Bill: Vol. 2", new Person("Quentin",
            "Tarantino")));
      }

      [WebMethod]
      public bool createMovie(int id, string name, Person director)
      {
         return this.insertMovie(new Movie(id, name, director));
      }

      [WebMethod]
      public bool insertMovie(Movie movie)
      {
         if (movie == null || movie.Name == null || movie.Director == null ||
            movie.Director.FirstName == null || movie.Director.LastName == null)
         {
            return false;
         }
         else
         {
            // Does not persist, because the Web Service itself is stateless.
            this.database.Add(movie.Id, movie);

            return true;
         }
      }

      [WebMethod]
      public Movie getMovie(int id)
      {
         return this.database[id];
      }

      [WebMethod]
      public Movie[] searchMovies(string[] keywords)
      {
         if (keywords == null)
         {
            return null;
         }
         else
         {
            // Instead of executing a search, the two default movies are returned.
            Movie[] movies = new Movie[this.database.Values.Count];
            this.database.Values.CopyTo(movies, 0);

            return movies;
         }
      }
   }
}
```

Figure A.5: MovieDatabase.asmx

```
namespace Services.MovieDatabase
{
   public class Movie
   {
      public int Id
      {
         get { return id; }
         set { id = value; }
      }
      private int id;

      public string Name
      {
         get { return name; }
         set { name = value; }
      }
      private string name;

      public Person Director
      {
         get { return director; }
         set { director = value; }
      }
      private Person director;

      public Movie()
         : this(0, "", new Person())
      {
      }

      public Movie(int id, string name, Person director)
      {
         this.id = id;
         this.name = name;
         this.director = director;
      }
   }
}
```

Figure A.6: Movie.cs

```
namespace Services.MovieDatabase
{
   public class Person
   {
      public string FirstName
      {
         get { return firstName; }
         set { firstName = value; }
      }
      private string firstName;

      public string LastName
      {
         get { return lastName; }
         set { lastName = value; }
      }
      private string lastName;

      public Person()
         : this("", "")
      {
      }

      public Person(string firstName, string lastName)
      {
         this.firstName = firstName;
         this.lastName = lastName;
      }
   }
}
```

Figure A.7: Person.cs

Appendix B TTCN-3 Representations of the XML Schema Built-in Data Types

```
/* This module defines TTCN-3 representations of XML Schema types basing on
 * the types presented in the ETSI standard ES 201 873-9 V1.1.1 Annex A
 */
module XSDAUX {
   // String types
   type charstring token (pattern "([^ \t\r\n]#(1,)( [^ \t\r\n]#(1,))#(0,))|") with {
      encode "{http://www.w3.org/2001/XMLSchema}token";
   }

   type charstring string with {
      encode "{http://www.w3.org/2001/XMLSchema}string";
   }

   type octetstring hexBinary with {
      encode "{http://www.w3.org/2001/XMLSchema}hexBinary";
   }

   type charstring base64Binary (pattern "[0-9a-zA-Z+/= ]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}base64Binary";
   }

   type charstring anyURI with {
      encode "{http://www.w3.org/2001/XMLSchema}anyURI";
   }

   type charstring QName with {
      encode "{http://www.w3.org/2001/XMLSchema}QName";
   }

   type charstring normalizedString (pattern "[^\n\r\t]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}normalizedString";
   }

   type charstring languageXSD (pattern "[a-zA-Z]#(1,8)(-[\w]#(1,8))#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}language";
   }

   type charstring NMTOKEN (pattern "[\w_.:-]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}NMTOKEN";
   }

   type charstring ENTITY (pattern "[a-zA-Z_][\w_\-.]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}ENTITY";
   }

   type charstring Name (pattern "[a-zA-Z_:][\w_:\-.]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}Name";
   }

   type charstring NCName (pattern "[a-zA-Z_][\w_\-.]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}NCName";
   }

   type charstring ID (pattern "[a-zA-Z_][\w_\-.]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}ID";
   }

   type charstring IDREF (pattern "[a-zA-Z_][\w_\-.]#(0,)") with {
      encode "{http://www.w3.org/2001/XMLSchema}IDREF";
   }

   // Integer types
   type integer integer_ with {
      encode "{http://www.w3.org/2001/XMLSchema}integer";
   }

   type integer positiveInteger (1 .. infinity) with {
      encode "{http://www.w3.org/2001/XMLSchema}positiveInteger";
   }

   type integer nonPositiveInteger (-infinity .. 0) with {
      encode "{http://www.w3.org/2001/XMLSchema}nonPositiveInteger";
   }

   type integer negativeInteger (-infinity .. -1) with {
      encode "{http://www.w3.org/2001/XMLSchema}negativeInteger";
   }

   type integer nonNegativeInteger (0 .. infinity) with {
      encode "{http://www.w3.org/2001/XMLSchema}nonNegativeInteger";
   }

   type integer long_ (-9223372036854775808 .. 9223372036854775807) with {
      encode "{http://www.w3.org/2001/XMLSchema}long";
   }

   type integer unsignedLong (0 .. 18446744073709551615) with {
      encode "{http://www.w3.org/2001/XMLSchema}unsignedLong";
   }

   type integer int (-2147483648 .. 2147483647) with {
      encode "{http://www.w3.org/2001/XMLSchema}int";
   }

   type integer unsignedInt (0 .. 4294967295) with {
      encode "{http://www.w3.org/2001/XMLSchema}unsignedInt";
   }

   type integer short_ (-32768 .. 32767) with {
      encode "{http://www.w3.org/2001/XMLSchema}short";
   }

   type integer unsignedShort (0 .. 65535) with {
      encode "{http://www.w3.org/2001/XMLSchema}unsignedShort";
   }

   type integer byte_ (-128 .. 127) with {
      encode "{http://www.w3.org/2001/XMLSchema}byte";
   }

   type integer unsignedByte (0 .. 128) with {
      encode "{http://www.w3.org/2001/XMLSchema}unsignedByte";
   }

   // Float types
   type float decimal with {
      encode "{http://www.w3.org/2001/XMLSchema}decimal";
   }

   type float float_ with {
      encode "{http://www.w3.org/2001/XMLSchema}float";
   }

   type float double with {
      encode "{http://www.w3.org/2001/XMLSchema}double";
   }

   // Time types
   type charstring duration (pattern "-?P([\d]+Y)#(0,)([\d]+M)#(0,1)([\d]+D)#(0,1)
         (T([\d]+H)#(0,1)([\d]+M)#(0,1)([\d]+(.[\d]+)#(0,1)S)#(0,1))#(0,1)") with {
      encode "{http://www.w3.org/2001/XMLSchema}duration";
   }

   type charstring dateTime (pattern "-?[\d]#(4)-((0[1-9])|[10-12])-((0[1-9])|[10-
         31])T((0[1-9])|[10-23]):((0[1-9])|[10-59]):((0[1-9])|[10-
         59])(.[\d]+)#(0,1)([+-]?[\d:Z]+)#(0,1)") with {
      encode "{http://www.w3.org/2001/XMLSchema}dateTime";
   }

   type charstring time (pattern "((0[1-9])|[10-23]):((0[1-9])|[10-59]):((0[1-
         9])|[10-59])(.[\d]+)#(0,1)([+-]?[\d:Z]+)#(0,1)") with {
      encode "{http://www.w3.org/2001/XMLSchema}time";
   }
```

```

  type charstring date (pattern "-?[\d]#(4)-((0[1-9])|[10-12])-((0[1-9])|[10-31])([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}date";
  }

  type charstring gYearMonth (pattern "[\d]#(4)-((0[1-9])|[10-12])([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}gYearMonth";
  }

  type charstring gYear (pattern "-?\d#(4)([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}gYear";
  }

  type charstring gMonthDay (pattern "--((0[1-9])|[10-12])-((0[1-9])|[10-31])([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}gMonthDay";
  }

  type charstring gDay (pattern "---((0[1-9])|[10-31])([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}gDay";
  }

  type charstring gMonth (pattern "--((0[1-9])|[10-12])([+-]?[\d:Z]+)#(0,1)") with {
     encode "{http://www.w3.org/2001/XMLSchema}gMonth";
  }

  // Sequence types
  type set of XSDAUX.NMTOKEN NMTOKENS with {
     encode "{http://www.w3.org/2001/XMLSchema}NMTOKENS";
  }

  type set of XSDAUX.IDREF IDREFS with {
     encode "{http://www.w3.org/2001/XMLSchema}IDREFS";
  }

  type set of XSDAUX.ENTITY ENTITIES with {
     encode "{http://www.w3.org/2001/XMLSchema}ENTITIES";
  }

  // Boolean types
  type boolean boolean_ with {
     encode "{http://www.w3.org/2001/XMLSchema}boolean";
  }

  // XSI attribute declarations
  type XSDAUX.QName Attribute_type with {
     encode "{http://www.w3.org/2001/XMLSchema-instance}type";
  }

  type boolean Attribute_nil with {
     encode "{http://www.w3.org/2001/XMLSchema-instance}nil";
  }

  type set of XSDAUX.anyURI Attribute_schemaLocation with {
     encode "{http://www.w3.org/2001/XMLSchema-instance}schemaLocation";
  }

  type XSDAUX.anyURI Attribute_noNamespaceSchemaLocation with {
     encode "{http://www.w3.org/2001/XMLSchema-instance}noNamespaceSchemaLocation";
  }
}
```

Figure B.1: XSDAUX.ttcn3