Automated Testing of XML/SOAP based Web Services

Ina Schieferdecker, FOKUS, Berlin, Germany

Bernard Stepien, University of Ottawa, Canada

Abstract

Web services provide seamless connections from one software application to another over private intranets and the Internet. The major communication protocol used is SOAP being mainly XML over HTTP. The exchanged data follow precise format rules in the form of XML Document Type Definitions or more recently the proposed XML Schemas.

Web service testing considers functionality and load aspects to check how a Web service performs for single clients and scales as the number of clients accessing it increases. This paper discusses the automated testing of Web services by use of the Testing and Test Control Notation TTCN-3. A mapping between XML data descriptions to TTCN-3 data is presented to enable the automated derivation of test data. This is the basis for functional and load tests of XML interfaces in TTCN-3. The paper describes the mapping rules and prototypical tools for the development and execution of TTCN-3 tests for XML/SOAP based Web services.

1 Introduction

Web services are more and more used for the realization of distributed applications crossing domain borders. However, the more Web services are used for central and/or business critical applications, their functionality, performance and overall quality become key elements for their acceptance and wide spread use. Consumers of Web services will want assurances that a Web service will not fail to return a response in a certain time period. Even more, systematic testing of Web services is essential as Web services can be very complex and hard to implement: although the syntax of the data formats is described formally with XML, the semantics and possible interactions with the Web service and use scenarios are described textually only. This encompasses the risk of misinterpretation and wrong implementation. Therefore, testing a final implementation within its target environment is essential to assure the correctness and interoperability of a Web service.

Testing a system is performed in order to assess its quality and to find errors if existent. An error is considered to be a discrepancy between observed or measured values provided by the system under test and the specified or theoretically correct values. Testing is the process of exercising or evaluating a system or system component by manual or automated means to check that it satisfies specified requirements. Testing approves a quality level of a tested system. The need for testing approaches arose already within the IT community: so-called interoperability events are used to evaluate and launch certain XML interface technologies and Web services, to validate the specifications and to check various implementations for their functionality and performance. However, the tests used at interoperability events are not uniquely defined, so that one has to question on which basis implementations are evaluated.

On contrary, there are test-engineering methods and a conformance testing methodology [11] within telecommunication, which have evolved over years, are widely spread and successfully applied to assess the correctness of protocol implementations. The standardized test specification language TTCN-3 [12] with its advanced features for test specification is expected to be applied to many testing applications that were not previously open to TTCN. TTCN-3 has been defined to be applicable for protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, API testing etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing.

The application of TTCN-3 for testing specific target technologies such as for mobile protocol stacks, CORBA based systems or Web service can be made effective by allowing the direct use of the data

definitions of the system to be tested within the TTCN-3 test specification: ASN.1 (Abstract Syntax Notation One) for protocol stacks, IDL (Interface Definition Language) for CORBA (Common Object Request Broker Architecture) and XML (Extended Markup Language) for Web services. TTCN-3 predefines a mapping for ASN.1 to TTCN-3 in the standard itself [12]. A mapping for IDL has been defined in [14]. This paper presents the mapping of XML to TTCN-3 as a basis for automated Web services tests with TTCN-3.

In Section 1, an overview on Web services, XML and SOAP and a discussion on testing Web services are given. Automated testing of Web services with TTCN-3 is presented in Section 2. In particular, the mapping rules for XML DTDs (Document Type Definitions) and for XML Schemas are described. Finally, a tool environment for Web service tests is shown in Section 4. Conclusions finish the paper.

2 Web services, XML and SOAP

A Web service is a URL-addressable resource returning information in response to client requests. Web services are integrated into other applications or Web sites, even though they exist on other servers. So for example, a Web site providing quotes for car insurance could make requests behind the scenes to a Web service to get the estimated value of a particular car model and to another Web service to get the current interest rate.

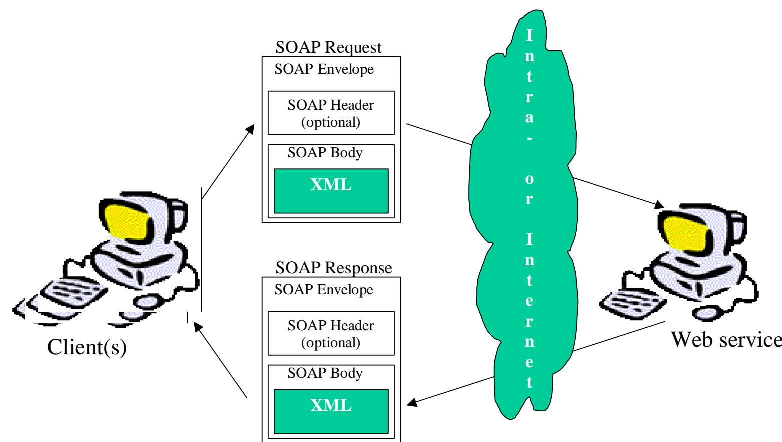


Figure 1. Principal Structure of a Web service

A Web service (see Figure 1) can be seen as a Web site that provides a programmatic interface using the communication protocol SOAP, the Simple Object Access Protocol: operations are called using HTTP and XML (SOAP) and results are returned using HTTP and XML (SOAP). The operations are described in XML with the Web Service Description Language (WSDL). Web services can be located via the Universal Description, Discovery and Integration (UDDI) based registry of services, which will not be considered in this paper.

2.1 XML DTDs and Schemas

XML stands for Extensible Markup Language and as its name indicates, the prime purpose of XML was for the marking up of documents. Marking up a document consist in wrapping specific portions of text in tags that convey a meaning and thus making it easier to locate them and also manipulating a document based on these tags or on their attributes. Attributes are special annotations associated to a tag that can be used to refine a search.

An XML document has with its tags and attributes a self-documenting property that has been rapidly considered for a number of other applications than document markup. This is the case of configuration files for software but also telecommunication applications for transferring control or application data like for example to Web pages.

XML follows a precise syntax and allows for checking well-formedness and conformance to a grammar using a Document Type Description (DTD) that could either be interpreted as a BNF like grammar specification or in some cases as a data type. A DTD consists of a set of production rules for elements that have a name and describe its content as empty, any, mixed, choice or sequence. An element can also contain attributes that are declared separately. The four main concepts in XML DTDs are

- The basic data types are CDATA and PCDATA, i.e. pure text and mixed text with marked up portions of it.
- There are two basic structured types: sequences and choices.
- There is the concept of zero or one, zero or many and one and many elements.
- There are attributes that also carry the concept of enumerated type.

While DTDs are appropriate for marking up text, they are very limited for other applications because the two basic types CDATA and PCDATA are too general for any precise data typing as in other widely used programming languages. Consequently, the new XML data typing model called **Schema** was developed.

First of all, XML schemas [3][4] are defined using the same basic XML syntax of tags and end tags and actually follow a well-defined DTD [7]. Second, XML schemas are true data types and contain many of the data typing features found in most of the recent high level programming languages. The central concept of XML schemas is the building block approach by defining components that consist themselves of type definitions and element declarations. But most important is the fact that XML Schemas are very flexible and allow to describe the same rules in many different ways depending on the use of the following structuring concepts:

- Provide for primitive data typing including byte, date, integer, string, …
- Simple and complex types
- Type inheritance
- Restrictions and extensions
- Global and local definitions
- Embedded, flat catalog and named type structuring constructs

XML schemas have various primitive data types like string, Boolean, decimal, float, double, duration, dateTime, time, date, etc. A simple type definition is used to establish a value space, a lexical space of a type and also to name a specific value space. Complex data types are used to specify sequences, choices and unions.

While XML schemas are close to traditional programming languages data types where a complex type is defined in terms of field type and field names, they also have two other constructs that are not found elsewhere, namely type references and local definitions. These constructs allow for three basic ways to specify a schema: embedded schema, flat catalog, named types. This paper uses a weather service as an example: the weather is given for a location being a city in a country. It is described in terms of the temperature, the barometric pressure and further, textually described conditions (see Figure 2).

Embedded schema

```
<schema>
  <element name="weather">
    <complexType>
     <sequence>
       <element name="location">
        <complexType >
          <sequence>
           <simpleType name="city">
             <restriction base="string">
              <pattern value="[a-zA-Z]"/>
             </restriction>
           </simpleType>
           <element name="country" type="string"/>
          </sequence>
         </complexType>
       </element>
       <element name="temperature" type="integer"/>
       <element name="barometric_pressure" type="integer"/>
      <element name="conditions" type="string"/>
     </sequence>
    </complexType>
  </element>
</schema>
```

Named types

```
<schema>
  <complexType name="weatherType" >
    <sequence>
      <element name="location" type="locationType"/>
       <element name="temperature" type="integer"/>
             <element name="barometric_pressure" type="integer"/>
      <element name="conditions" type="string"/>
    </sequence>
   </complexType>
  <complexType name="locationType" >
    <sequence>
      <element name="city" type="cityType"/>
      <element name="country" type="string"/>
    </sequence>
  </complexType>
  <simpleType name="cityType">
    <restriction base="string">
      <pattern value="[a-zA-Z]"/>
    </restriction>
  </simpleType>
   <element name="weather" type="weatherType"/>
</schema>
```

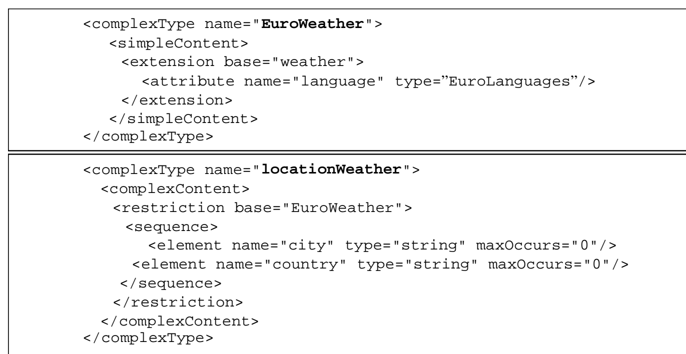
Flat catalog

```
<schema>
  <element name="temperature" type="integer"/>
  <element name="barometric_pressure" type="integer"/>
  <element name="conditions" type="string"/>
  <element name="country" type="string"/>
  <simpleType name="city">
    <restriction base="string">
      <pattern value="[a-zA-Z]"/>
    </restriction>
  </simpleType>
  <element name="location">
    <complexType >
      <sequence>
        <element ref="city"/>
        <element ref="country"/>
      </sequence>
    </complexType>
  </element>
  <element name="weather">
    <complexType>
     <sequence>
       <element ref="location"/>
       <element ref="temperature"/>
       <element ref="barometric_pressure"/>
       <element ref="conditions"/>
     </sequence>
    </complexType>
  </element>
</schema>
```

Figure 2. XML Schema for the Weather Service

The *embedded method* derives from the nested tags mechanism of XML itself. In this method, elements are defined where they are used inside the hierarchy. Consequently there is no need to name a local type - it is called an anonymous type. Eventually the leaves of the tree that constitutes an embedded type definition are composed exclusively of either primitive types or already defined types. This implies that a local definition can be used only once and that there is no need for reusability in a specific application. The *flat catalog* approach uses the concept of substitution. Each element is defined by a reference to another element declaration. *Named types* are the closest to traditional computer languages data typing. Each element has a name and a type name and each subtype is defined separately.

In addition, XML schemas provide two inheritance mechanisms to restrict and extend types. In Figure 3, weather is extended to EuroWeather with an additional attribute for the EuroLanguage. In the restriction, two fields are implicitly removed by setting their maximal occurrences to zero.



```
<complexType name="EuroWeather">
   <simpleContent>
    <extension base="weather">
      <attribute name="language" type="EuroLanguages"/>
    </extension>
   </simpleContent>
</complexType>
```

```
<complexType name="locationWeather">
  <complexContent>
   <restriction base="EuroWeather">
    <sequence>
      <element name="city" type="string" maxOccurs="0"/>
     <element name="country" type="string" maxOccurs="0"/>
    </sequence>
   </restriction>
  </complexContent>
</complexType>
```

Figure 3. Extension and restriction for the Weather Service

2.2 SOAP

SOAP is a simple mechanism for exchanging structured and typed information between peers in a decentralized distributed environment using XML [3][9][8]. SOAP as a new technology to support server-to-server communication competes with other distributed computing technologies including DCOM, Corba, RMI, and EDI. Its advantages are a light-weight implementation, simplicity, open-standards origins and platform independence.

The protocol consists only of a single HTTP request and a corresponding response between a sender and a receiver but that can optionally follow a path of relays called nodes that each can play a role that is specified in the SOAP envelope. A Soap request is an HTTP POST request. The data part consists of:

- the SOAP envelope
- the SOAP binding framework
- the SOAP encoding rules
- the SOAP RPC representation called the body

The SOAP part is encoded again as an XML document like in Figure 4.

```
<Body>
   <getWeather>
    <location>
      <city> Berlin </city>
      <country> Germany </country>
    </location>
      <timeframe>
        <date> 24.12.2001 </date>
        <from> 12:00 </from>
        <to>  20:00 </to>
      </timeframe>
   </getWeather>
</Body>
```

Figure 4. A Weather Service request

2.3 Testing of Web services

Testing of Web services (as for any other technology or system) is useful to prevent late detection of errors (possibly by dissatisfied users), what typically requires complex and costly repairs. Testing enables the detection of errors and the evaluation and approval of system qualities beforehand. An automated test approach helps in particular to efficiently repeat tests whenever needed for new system releases in order to assure the fulfilment of established system features in the new release. First approaches towards automated testing with proprietary test solutions exist [15], however, with such tools one is bound to the specific tool and its features and capabilities.

Specification-based automated testing, where abstract test specifications independent of the concrete system to be tested and independent of the test platform are used, are superior to proprietary techniques: they improve the transparency of the test process, increase the objectiveness of the tests, and make test results comparable. This is mainly due to the fact that abstract test specifications are defined in an unambiguous, standardized notation, which is easier to understand, document, communicate and to discuss.

However, we go beyond “classical” approaches towards specification-based automated testing, which till now mainly concentrate on the automated test implementation and execution: we consider test generation aspects as well as the efficient reuse of test procedures in a hierarchy of tests.

Testing of Web services has to target three aspects: the discovery of Web services (i.e. UDDI being not considered here), the data format exchanged (i.e. WSDL), and request/response mechanisms (i.e. SOAP). The data format and request/response mechanisms can be tested within one test approach: by invoking requests and observing responses with test data representing valid and invalid data formats.

Since a Web service is a remote application, which will be accessed by multiple users, not only functionality in terms of sequences of request/response and performance in terms of response time, but also scalability in terms of functionality and performance under load conditions matters. Therefore we have developed a hierarchy of test settings starting with separate functional tests for the individual services of a Web service, to a service interaction test checking the simultaneous request of different

services, to a separate load tests for the individual services up to a combined load test for a mixture of requests for different services (see Figure 5). All the tests return not only a test verdict but also the response times for the individual requests.

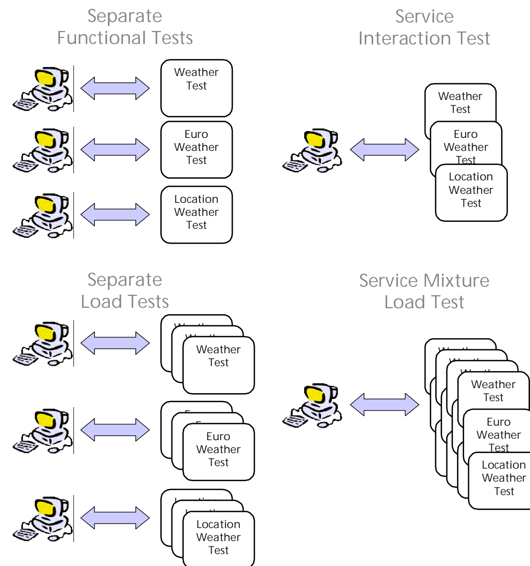


Figure 5. Test hierarchy for Web services

3 Test automation with TTCN-3

Our means to automate Web service testing is the Testing and Test Control Notation TTCN-3 **[11]**, which has been developed by the European Telecommunication Standards Institute ETSI not only for telecommunication but also for software and data communication systems. Like any other communication-based system, Web services are natural candidates for testing using TTCN-3.

3.1 Overview on TTCN-3

TTCN-3 is a language to define test procedures to be used for black-box testing of distributed systems. Stimuli are given to the system under test (SUT), its reactions are observed and compared with the expected ones. On the basis of this comparison, the subsequent test behaviour is determined or the test verdict is assigned. If expected and observed responses differ, then a fault has been discovered which is indicated by a test verdict fail. A successful test is indicated by a test verdict pass.

TTCN-3 allows an easy and efficient description of complex distributed test behaviour in terms of sequences, alternatives, loops and parallel stimuli and responses. Stimuli and responses are exchanged at the interfaces of the system under test, which are defined as a collection of ports. The test system can use a number of test components to perform test procedures in parallel. Likewise to the interfaces of the system under test, the interfaces of the test components are described as ports.

TTCN-3 is a modular language and has a similar look and feel to a typical programming language. However, in addition to the typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load and scalability tests like test verdicts, matching mechanisms to compare the reactions of the SUT with the expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring.

A TTCN-3 test specification consists of four main parts:

- type definitions for test data structures
- templates definitions for concrete test data
- function and test case definitions for test behavior
- control definitions for the execution of test cases

The data type definitions are generated from the corresponding XML schema of the Web service to be tested. The templates are based on the corresponding data types and the behaviour of the service being tested that consist of sequences of requests and responses.

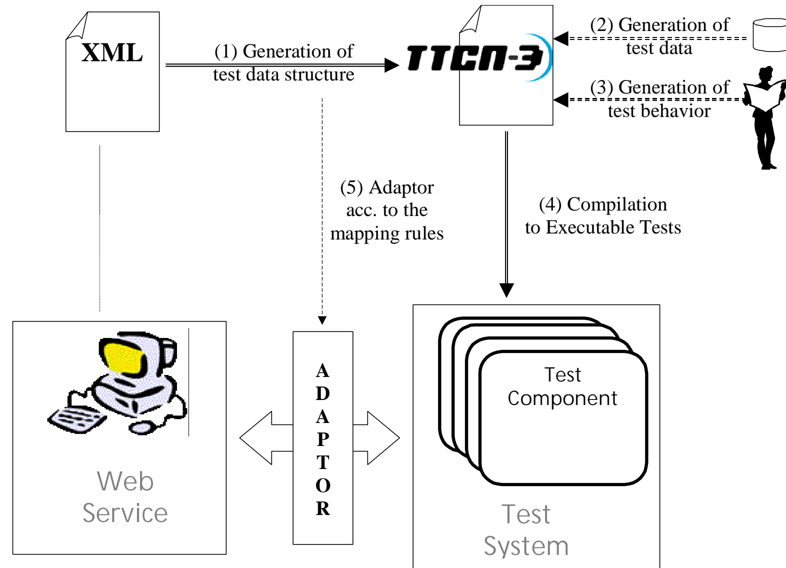


Figure 6. Testing of Web services with TTCN-3

An approach towards automated testing of Web services with TTCN-3 requires therefore the following steps (see Figure 6).

(1) The structure of the test data is derived from the XML definition (see Section 3.2).

(2) Test data (i.e. the concrete values for test stimuli and observations) is generated (see Section 3.4).

(3) Test behaviour (i.e. the sequences of test stimuli and observations) is generated (see Section 3.5).

(4) The resulting TTCN-3 module is compiled to executable code.

(5) The tests are performed using a test adaptor, which follows the mapping rules for test data structure to encode and decode the Web service requests and replies.

Currently, steps (1) and (4) can be automated with the help of tools as described in Section 4. The automation for step (2) and (3) requires further work: for this step mainly test generation approaches based on finite state machines or labelled transition systems will be used. The test adaptor for step (5) has to be developed only once, so that it can be used for any Web service and TTCN-3 test following the mapping rules from step (1).

3.2 Generating Test Data Structure: Mapping XML to TTCN-3

The target of the mapping of XML to TTCN-3 is the integral type system of TTCN-3, which is similar to ASN.1 in terms of availability of basic and structured types. The type system contains basic types (integer, float, boolean), basic string types (bitstring, hexstring, octetstring, charstring, universal charstring) and user-defined structured types (record, record of, set, set of, enumerated, union). XML and TTCN-3 data types are somewhat similar conceptually but because of their differences in purpose and structure the actual mappings require some transformations that are more than pro-format translations. While DTDs and Schemas have common concepts, there are basic differences that need to be addressed separately when defining the mapping.

3.2.1 Mapping XML DTDs

Using data typing concepts, we can divide the mapping problem between mapping predefined and user defined structured data types. Then, when handling user defined data types we can further divide the mapping into mapping attributes and mapping elements. Finally, we need to address the problem of the influence of attributes on the mapping of elements.

The limited set of *predefined types* in DTDs, i.e. CDATA, PCDATA, and token types, map directly to TTCN-3 string types and enumerations.

XML *attributes* can be mapped directly into fields of TTCN-3 record types (see Figure 7). The actual attribute name becomes the field name, while the attribute definition part can be handled in two different ways depending on the nature of the definition, which can either be of the string type or the enumerated type. For string type, we merely used the charstring type while for enumerations we must generate a type name using the attribute name and then create a separate TTCN-3 enumerated type using that name.

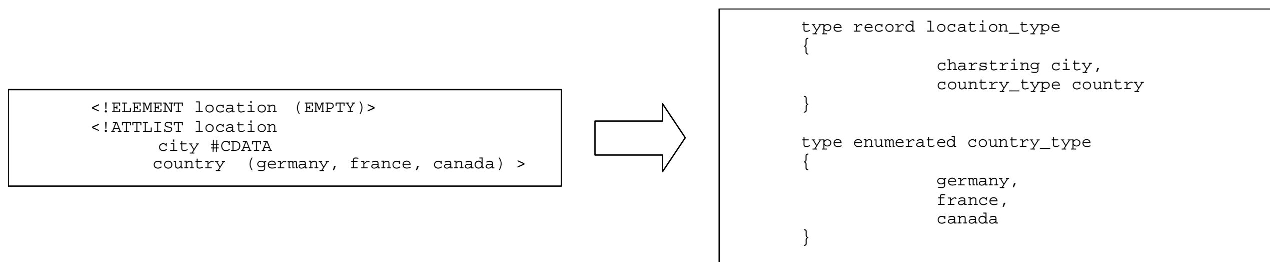


Figure 7. Mapping XML attributes

Another consideration is the fact that XML attributes can contain any values including white space. Since this is not allowed in TTCN-3 white space or illegal characters need to be converted to some legal characters such as underscores or some explicit name.

The mapping of DTDs Element declarations for *User defined types* to TTCN-3 data types needs to address the following problems:

- What is the meaning of an element?
- How do we interpret the children (sequence and choice) of the content specification?
- What is the meaning of a name in the content specification?
- The impact of attributes on the mapping of the DTD choice construct.

Element declarations can be mapped to types while the content specification corresponds to field declarations in a TTCN-3 data type. However, the basic difference between the two representations is that in TTCN-3 field declarations consist of field type-field name pairs while for DTDs the names found in the content specifications consists only of one name that needs to be interpreted either as a field name or field type but obviously not both at the same time. Thus we have decided to retain the content names as field name and to generate a type name by appending the string "_type" to a field name. However, when the content specification of an element consists only of a predefined type we will not generate a type name using the field name but use directly the predefined type name to construct the type name – field name tuple. Also, since the DTD PCDATA and CDATA types are too imprecise, we need to choose more appropriate predefined types for each field.

DTD children can be mapped directly: a DTD sequence can be mapped to a TTCN-3 record type while a choice can be mapped to a TTCN-3 union. However, the later mapping must be further decomposed if the element has also attributes as will be explained in a section further down.

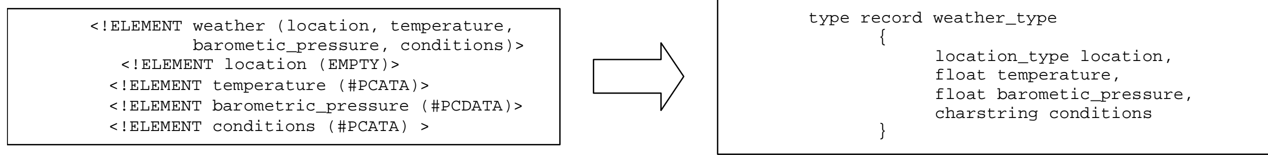


Figure 8. Mapping XML elements

Please note that an element that is defined as a choice of elements cannot always be mapped directly to a TTCN-3 union. This depends whether the element contains also an attribute list. The attribute list can be mapped only to a TTCN-3 record type. Consequently the only way to resolve this conflict is to remove the choice definition and create a separate data type for the choice part of the element declaration. The name of this data type will need to be made up since there is no corresponding name to be found in the DTD itself.

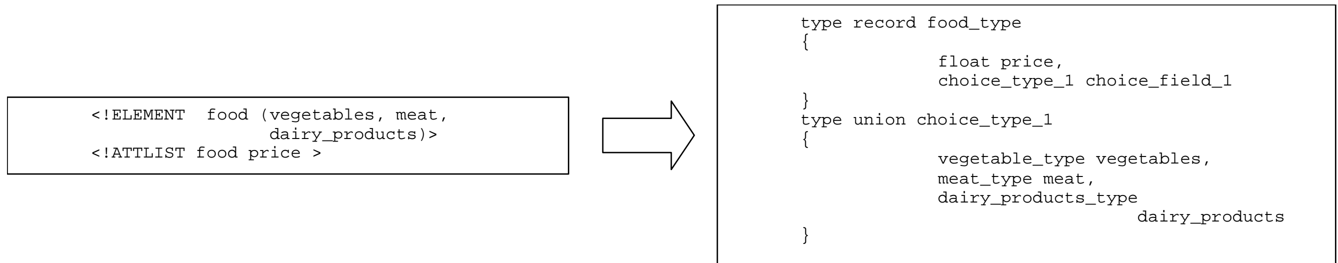


Figure 9. Specific mapping for XML choice elements with attributes

3.2.2 Mapping XML Schemas

Mapping XML schemas to TTCN-3 is different to the mapping of DTDs because in schemas there is an explicit concept of types and there are extension and restriction mechanisms for types. In addition, XML schemas are defined with different approaches (e.g. embedded, flat catalog and named types) that have no equivalent in DTDs.

XML schemas have a wide variety of *predefined types and subtypes*. For example, Schemas have an integer type but also countless variations about integers such as positive integers and negative integers, etc. These map mainly to TTCN-3 basic types together with additional attributes to reflect the specific variation of a basic type, e.g. an attribute to indicate positive or negative integers. Further, some primitive types such as Time and Date are mapped to TTCN-3 records.

Simple types are map to TTCN-3 basic types with the respective lexical restrictions represented by a range of values. The XML list construct is mapped to a TTCN-3 array and enumerations to enumerated types with the same restrictions as for the mapping of DTD enumerations.

Since there is no inheritance mechanism in TTCN-3 data types, XML *extensions and restriction* constructs must be mapped to a duplication of the definition of the inherited type and the potential conversion of its complex kind in the case of choice constructs. This means that if the current type being defined is a sequence and the inherited type is a choice, we need to create a new field with inherited type while if the inherited type is a sequence as well, we merely concatenate the fields of the inherited type with those of the target type. The same situation applies to the case of a defined choice type that inherits a sequence type. The restriction mechanism consists in removing fields in the inheriting type to be mapped.

The *named type* approach has a one to one mapping with TTCN-3 data types since both have the concept of field name and field type name. The element name becomes the field name and the element type becomes the field type name.

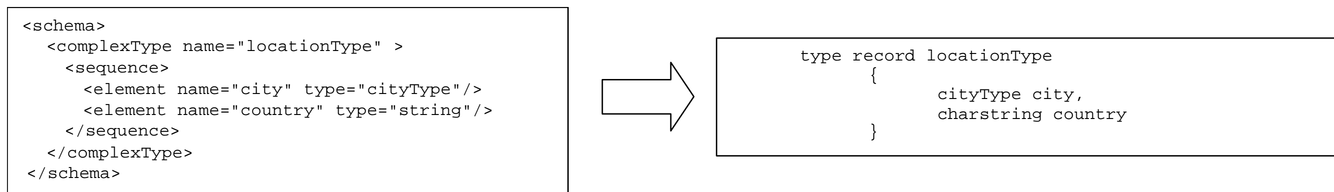


Figure 10. Mapping for named type XML schema

The main construct of XML schema *embedded type* approach is the local type definition. There is no corresponding construct in TTCN-3. Consequently, the local definition must be taken out of the type definition to be defined separately with a new generated type name that is also used as a field type name for the element being mapped to.

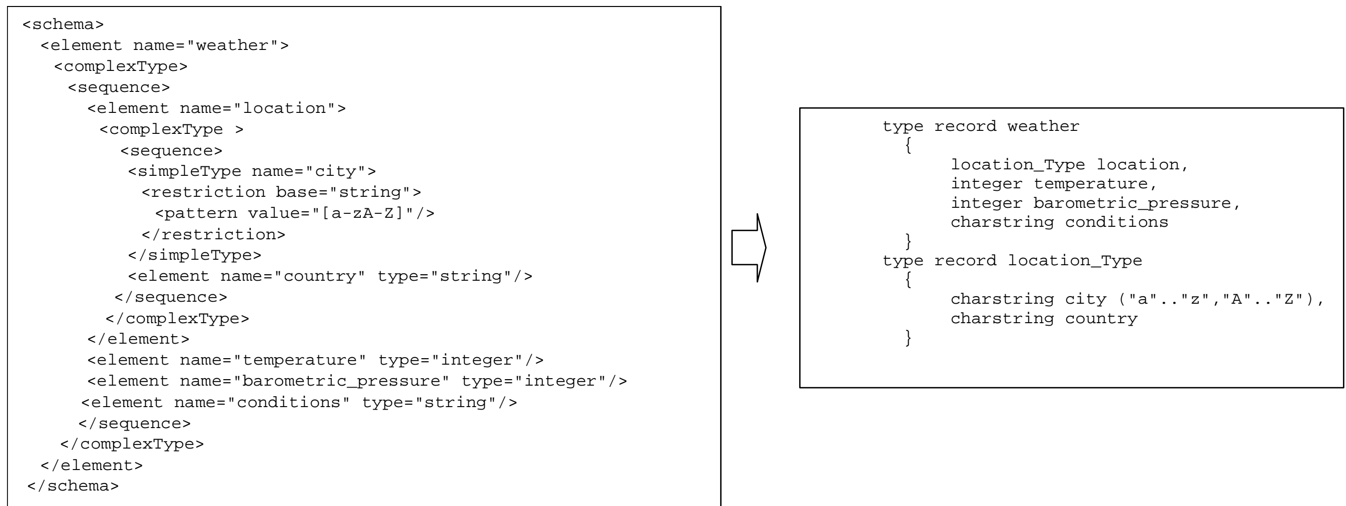


Figure 11. Mapping for embedded XML schema

The *flat catalogue* approach consists in type substitution. This is different from named types and in a way is similar to the DTD approach where each name found in the content specification refers to a separate element declaration. The difference is however that the referenced separate element declaration may be further defined using one of the three different approaches. Consequently, if the separate element declaration is using a named type approach we merely use its type for our current field type name, but if the referenced element uses the flat catalogue or the embedded style we need again to generate a type name.

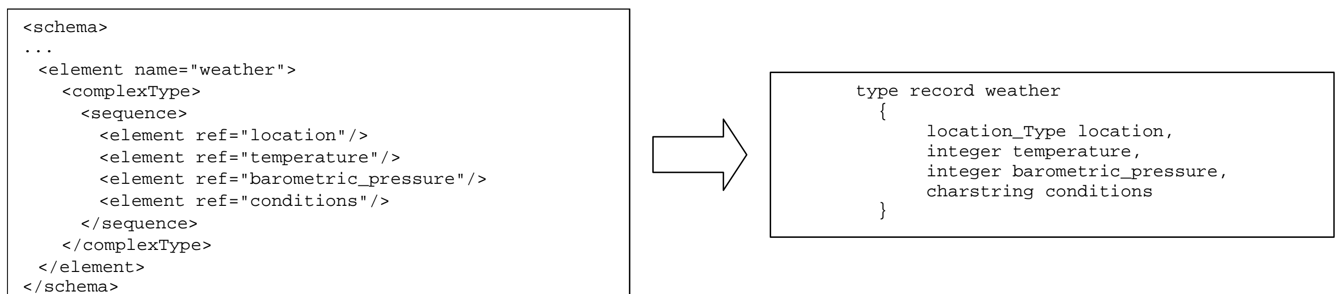


Figure 12. Mapping for flat catalogue XML schema

3.3 Generating test configuration

In addition to the structure of the test data, the test configuration in terms of test components and ports have to be generated (see Figure 13). We use a message port to access a Web service. This port can transfer request and response messages. Furthermore, we use a varying set of parallel test components (PTC) to represent separate functional tests, service interaction tests, separate load tests and load tests for service mixtures. Every PTC like the SUT has a port to represent the Web service interface. The PTCs use the same basic test functions to stimuli requests and observe responses. The main test component (MTC) controls the dynamic creation of the test components according to the kind of tests. The tests with several components are parameterized, so that the actual number of test components emulating the use of a certain service vary depending on the current value of the parameters.

```
type port WeatherService message {
      out weatherRequest;
      in  weatherResponse;
}

type component SUTType {
      port WeatherService weatherservice_port;
}
type component PTCType {
      port WeatherService weatherservice_port;
      timer T_wait := 1.0;
}
type component MTCType {
}
```

Figure 13. Test components

For the main kinds of tests shown in Figure 5 a fixed test case definition being independent of the concrete Web service to be tested can be defined. They follow all the same procedure: the MTC creates PTCs according to the services to be tested and according to the load to be generated. Every PTC gets a concrete test function assigned and is started. Afterwards, the MTC awaits the termination of all PTCs. The overall test verdict is the accumulated test verdict of all the PTCs.

```
testcase SeparateFunctionalTest
(integer Service)
runs on MTCType system SUTType
{
  var PTCType PTC:= PTCType.create;
  PTC.start(SeparateFunctional(Service));
  all component.done
}
```

```
testcase ServiceInteractionTest
(intarray Service)
runs on MTCType system SUTType
{
  var integer serviceno:= sizeof(Service);
  var PTCType PTC[serviceno];
  for (var integer j:=1; j<= serviceno; j:= j+1)
  {
    PTC[j]:= PTCType.create;
    PTC[j].start(SeparateFunctional(Service[j]));
  }
  all component.done
}
```

```
testcase SeparateLoadTest
(integer Service, integer Load)
runs on MTCType system SUTType
{
  var PTCType PTC[Load];
  for (var integer j:=1; j<= Load; j:= j+1)
  {
    PTC[j]:= PTCType.create;
    PTC[j].start(SeparateFunctional(Service));
  }
all component.done
}
```

```
testcase MixedServiceLoadTest
(intarray Service, Load)
runs on MTCType system SUTType
{
  var integer serviceno:= sizeof(Service);
  for (var integer j:=1; j<= serviceno; j:= j+1)
  {
    var PTCType PTC[Load[j]];
    for (var integer k:=1; k<= Load[j]; k:= k+1)
    {
      PTC[k]:= PTCType.create;
      PTC[k].start(SeparateFunctional(Service[j]));
    }
  }
  all component.done
}
```

Figure 14. Test cases for the different kinds of tests – the Test Framework

The generic test cases can be controlled with a general test case control mechanism like shown in Figure 15.

```
module TestFrameWork {
  type record ServiceLoad {
    integer Service,                                              // the service to be tested
    integer Load                                                  // the maximal load for the service
  }
  external const ServiceLoad Services[];                          // array of services to be tested
  external const integer increase;                                // load increase for the load tests
...
  control {
    var integer serviceno:= sizeof(Services);

    var verdicttype ServicesResult[serviceno];                    // test result per service

    for (var integer j:=1; j<=serviceno; j:=j+1) {                // functional test per service
      ServicesResult[j]:= execute(SeparateFunctionalTest(Services[j].Service));
    }
    for (var integer j:=1; j<=serviceno; j:=j+1) {                // load test per service
      if (ServicesResult[j] == pass) {
          for (var integer k:= increase; k <= Services[j].Load; j:= j+increase) {
                                                                  // load tests with increasing load
if (ServicesResult[j] == pass) {
                  ServicesResult[j]:= execute(SeparateLoadTest(Services[j].Service, k));
    } }   }   }


    var verdicttype ServicesMixResult[serviceno][serviceno];      // test result per service pair

    for (var integer j:=1; j<=serviceno; j:=j+1) {                // service interaction test per service pair
      if (ServicesResult[j] == pass) {
          for (var integer k:=1; k<=serviceno; k:=k+1) {
              if (ServicesResult[k] == pass) {
                 const integer ServicePair[2]:= {Services[j].Service, Services[k].Service };
                 ServicesMixResult[j][k]:= execute(ServiceInteractionTest(ServicePair));
    } }   }    }

    for (var integer j:=1; j<=serviceno; j:=j+1) {                // mixture load test per service pair
        for (var integer k:=1; k<=serviceno; k:=k+1) {
           if (ServicesMixResult[j][k] == pass) {
               const integer ServicePair[2]:= {Services[j].Service, Services[k].Service };
               for (var integer l:= increase; l <= Services[j].Load; l:= l+increase) {
                                                                  // load tests with increasing load
                    for (var integer m:= increase; m <= Services[k].Load; m:= m+increase) {
                        const integer PairLoad[2]:= { l, m };
                        ServicesMixResult[j][k]:= execute(MixedServiceLoadTest(ServicePair, PairLoad));
    }   }  }    }    }
  }
}
```

Figure 15. Execution Control for the Test Framework

With the control part at first, the functionality of each service offered by a Web service is tested. Then, load tests for the successfully tested services are performed with an increasing load. Afterwards, service pairs are taken in order to test for service interaction. Finally, the successfully tested service pairs are tested for increasing load. Both, the services to be tested, the maximal load for a service test and the increase for the load tests have to be determined by test execution only – these values are declared as external constants to the TTCN-3 module representing the Test Framework. The control part can be enhanced to reflect other test combinations for e.g. not only tests for service pairs but service sets.

3.4 Generating test data

Templates are used to define the concrete test data to be used for requests to and responses from the Web service. Figure 16 contains example templates to request the weather in Berlin and London and to receive respective responses. The response template uses patterns to indicate ranges of acceptable values. For example, the temperature should be given in the response, but the concrete value is open.

We work on approaches towards the automated generation of test data by using the classification tree method [16] being implemented in the CTE tool. This method enables the generation of exhaustive templates for requests, however, needs to be extended to enable the generation of response templates with patterns as well.

```
template weatherRequest getWeatherBerlin :=
{
      location := {city := "berlin", country := "germany"},
      timeframe := { dateWeather := today,
            fromTime := noon,
            toTime := midnight
      }
};

template weatherRequest getWeatherLondon modifies getWeatherBerlin :=
{
      location := {city := "london", country := "england"}
};

template weatherResponse get_response(charstring theCity, charstring theCountry) :=
{
      location := {city := theCity, country := theCountry},
      timeframe := ?,
      temperature := ?,
      conditions := ?,
      barometric_pressure := ?
};
```

Figure 16. Test data for the Weather service

3.5 Basic test function for the weather service

The basic test function for the weather service is depicted in Figure 17.

```
function SeparateFunctional(integer Service)
runs on PTCType {
      map(self: weatherservice_port, system: weatherservice_port);
      if (Service == 1) //normal weather service
      {
          weatherservice_port.send(getWeatherLondon);
          log(getWeatherLondon); T_wait.start;
          alt
          {
             [] weatherservice_port.receive(get_response("london", "england"))
                   {
                         log(get_response("london", "england")); verdict.set(pass)
                   }
             [] weatherservice_port.receive //unexpected response
                   {
                         log("unexpected response"); verdict.set(fail)
                   }
             [] T_wait.timeout //no response
                   {
                         log("timeout"); verdict.set(fail)
                   }
          }
      }
      else ...
      stop;
}
```

Figure 17. Basic test function for the Weather service

It consists mainly of a pair of request and response to the Weather service. If the expected response is received, a pass is assigned. In addition, unexpected and no response are handled – these cases lead to fail. The log information logs received response or the timeout and the respective time stamp.

The map operation at the beginning enables the communication of the PTC to the Weather service. The if statement allows to differentiate the test behaviour according to the service to be tested.

This basic test function is specific to the Web service to be tested, but has to be developed once and can then be reused for the various types of tests presented above.

4 The tool environment for Web service tests with TTCN-3

The tool environment for automated testing of Web services with TTCN-3 uses the TTCN-3 to Java *compiler* TTthree [17], an XML to TTCN-3 *conversion* tool and a test *adaptor* for XML/SOAP interfaces.

Since there are both XML DTDs and XML schemas it would appear that we would have to build two separate tools to handle the automated mapping of XML type definitions. However, there are at least three reasons to avoid this duplication:

- There exist already DTD to Schema conversion tools [6]
- Most of XML applications where TTCN-3 can be useful as a testing tool use only XML schemas. This is the case for the Simple Object Access Protocol.
- XML schemas can be parsed directly using off the shelf parsers like DOM because an XML schema is defined with the same principle of tags and attributes of XML documents[5].

We have therefore developed a conversion tool using the XML Document Object Model (DOM). The parsing step can be reduced to the following few statements when using the Xerces API:

```
DOMParser parser = new DOMParser();
parser.parse(XMLschemaFileName);
DocumentImpl document = (DocumentImpl)parser.getDocument();
```

The resulting parse tree can be walked through using the `Node class`:

```
Node n;
for(n=rootDoc.getFirstChild(); n != null; n=n.getNextSibling())
{
       if(IsASchemaElement(n.getNodeName()))
              ProcessElement(n, kind, rootDoc);
       else if(IsASchemaComplexType(n.getNodeName()))
              ProcessComplexType(n, null, kind, rootDoc);
       else if(IsASchemaSimpleType(n.getNodeName()))
              ProcessSimpleType(n, kind);
       else if(IsASchemaAttribute(n.getNodeName()))
              ProcessAttribute(n, kind, rootDoc);

}
```

where the methods `IsA…` consist merely in locating the node that has the appropriate tag name like for example:

```
static boolean IsASchemaElement(String theName)
{
       if(StripDomain(theName).equals("element"))
              return true;
       return false;
}
```

For each type of XML schema construct there is a corresponding processing method like for example `ProcessElement(…)` that consists in two main activities: getting the value of attributes and further processing the subtree for more tags:

```
void ProcessElement(Node n, int kind, Node theResultNode)
{
             int nbAttribs;
             Node anAttrib;

             String theFieldName = GetAttributeValue(n, "name");
             String theTypeName = GetAttributeValue(n, "type");
             String theRefName = GetAttributeValue(n, "ref");

             ProcessChildren(n, theFieldName+"_Type", TOP_LEVEL, root);
…
}
```

Due to the fact that a number of types have to be extracted from the body of elements in the case of local definitions or mixed sequence/choice constructs that result from the interference of attributes, the conversion to TTCN-3 is performed in two steps: (1) type extraction and (2) type translation. DOM enables to construct a new tree of XML definitions. This tree can then then be parsed for the translation step.
The *adaptor* for XML/SOAP interfaces realizes the functions of the TTCN-3 runtime interface (TRI [18]) and the TTCN-3 control interfaces (TCI [19]). It is derived from the basic adaptor provided with the runtime environment belonging to TTthree. The adaptor performs the adaptation of the compiled TTCN-3 code to the target test device (in our case a Solaris workstation, Windows or Linux PC) and covers the test system user interface, test execution control, test event logging, as well as communication with the SUT and timer implementation. For the communication with the SUT, i.e. the Web service, SOAP request messages are encoded from and SOAP response messages are decoded to TTCN-3 data used in the test specification. The adaptor is generic and enables the testing of any Web service using XML/SOAP interfaces. In order to use this adaptor the mapping rules provided in Section 3.2 have to be respected by the tests being defined in TTCN-3.

5 Conclusion

Testing Web services presents a variety of new and interesting challenges. In particular, test automation will be essential to a sound and efficient Web service development process, for the assessment of the functionality, performance and scalability of Web services as well as for the approval and acceptance of Web services developed by application providers.

This paper presents a flexible test framework for Web services using the Testing and Test Control Notation TTCN-3. The test framework is developed for Web services with XML/SOAP interfaces and provides functional, service interaction, and load tests with flexible test configurations and varying load.

The provided test hierarchy of predefined kinds of tests is generic as it can be used for arbitrary Web services. The specifics of a concrete Web service are handled within basic test functions emulating the use of the services offered by a Web service. These basic test functions are reused by the kinds of tests provided in the test hierarchy.

A further key element of the test framework is the automated translation of XML data to TTCN-3, so that test skeletons can be generated directly from the specification of the Web service. For that, XML DTDs and Schemas have been analysed and mapping rules have been developed. These rules are realized by a conversion tool from XML to TTCN-3. The conversion tool together with the TTCN-3 compiler and execution environment TTthree provides us a complete tool chain for test data type generation, test development, implementation and execution. The test framework has already been used successfully for selected Web services.

Future work will further elaborate methods for test data generation. In particular, the classification tree method will be investigated for potential extension towards the generation of templates for SOAP responses. In addition, the test framework will be enhanced to deal with further elements of Web services like the specifics of WSDL and UDDI.

References

[1] W3C: *Extensible Markup Language (XML) 1.0,* W3C Recommendation, 6 October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

[2] W3C: *XML Schema Part 0: Primer,* W3C Recommendations, 2 May 2001, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502

[3] W3C: *XML Schema Part 1: Structures,* W3C Recommendation 2 May 2001 http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[4] W3C: *XML Schema Part 2: Datatypes,* W3C Recommendation 02 May 2001 http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[5] R. Jeliffe: *The XML Schema Specification in Context* http://www.ascc.net/~ricko/XMLSchemaInContext.html

[6] W3C: *A Conversion Tool from DTD to XML Schema*, http://www.w3.org/2000/04/schema_hack/

[7] W3C: *XML Schema DTD : DTD for Schemas* (non-normative) http://www.w3.org/TR/2001/PR-xmlschema-1-20010316/#nonnormative-schemaDTD

[8] W3C: *Simple object Access Protocol (SOAP) 1.1*, W3C Note 08 May 2000, http://www.w3.org/TR/SOAP

[9] B. McLaughlin: *Java & XML*, 2nd edition, O'Reilly, Chapter 12: *SOAP*.

[10] Don Box MSDN magazine on the Web: *A Young person's guide tot the simple object access protocol: SOAP increases interoperability accross platforms and languages*, http://msdn.microsoft.com/msdnmag/nettop.asp?page=/msdnmag/issues/0300/soap/soap.asp&ad=ads.ddj.com/msdnmag/premium.htm

[11] ISO/IEC 9646-3 (1998): "Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: *The Tree and Tabular combined Notation* (TTCN)"

[12] ETSI MTS: The Testing and Test Control Notation TTCN-3, Part 1: TTCN-3 Core Language / ETSI ES 201873-1 V2.0.0 (2001-03), http://www.etsi.org

[13] I. Schieferdecker, S. Pietsch, T. Vassiliou-Gioles: *Systematic Testing of Internet Protocols - First Experiences in Using TTCN-3 for SIP. 5th IFIP Africom Conference on Communication Systems*, Cape Town, South Africa, May 2001.

[14] M. Ebner, A. Yin, M. Li: *Definition and Utilisation of OMG IDL to TTCN-3 Mapping*. – 16th Intern. IFIP Conference on Testing Communicating Systems (TestCom 2002), Berlin, March 2002.

[15] *ANTS* (Advanced .NET Testing System), Red Gate Software, http://www.red-gate.com/ants.htm.

[16] Grochtmann, M., J. Wegener and K. Grimm: *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*. Proc. of 8th International Software Quality Week, SanFrancisco, California, USA, pp. 4-A-4/1-11, 1995.

[17] *TTthree* (TTCN-3 to Java compiler), Testing Technologies IST GmbH, http://www.testingtech.de.

[18] ETSI: *The TTCN-3 Runtime Interface TRI*, Technical Report, Sophia Antipolis, Sept. 2001.

[19] ETSI: *The TTCN-3 Control Interfaces TCI*, Draft Technical Report, Sophia Antipolis, June 2002.