Integrated Design and Process Technology, **IDPT-2003**
Printed in the United States of America, June, 2003

SYSTEM-LEVEL TESTS WITH TTCN-3

Ina Schieferdecker, Fraunhofer FOKUS, Berlin, Germany, schieferdecker@fokus.fhg.de

Abstract

System-level testing considers functionality and load aspects to check how a system performs for single service requests and scales as the number of service requests accessing/using it increases. This paper presents a flexible test framework including functional, service interaction and load tests. It is generic in terms of being to a large extend independent of the system to be tested. The paper discusses the automation of the test framework with the Testing and Test Control Notation TTCN-3. The test framework is exemplified for Web service tests.

INTRODUCTION

The Testing and Test Control Notation TTCN-3 has been developed by the European Telecommunication Standards Institute (ETSI) to address testing needs of modern Telco and IT technologies and to widen the scope of applicability. TTCN-3 enables systematic, specification-based testing for various kinds of tests including e.g. functional, scalability, load, interoperability, robustness, regression, system and integration testing. TTCN-3 is a language to define test procedures to be used for black-box testing of distributed systems. It allows an easy and efficient description of complex distributed test behavior in terms of sequences, alternatives, and loops of stimuli and responses. The test system can use a number of test components to perform test procedures in parallel. TTCN-3 is a modular language that 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 test campaigns 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.

This paper discusses the application of TTCN-3 for system-level tests. It describes a test framework with predefined test scenarios and test setups, which can be adapted to different system under tests by exchanging the modules for the basic functional tests only. The basic idea is to define a hierarchy of tests for service interaction, scalability and load tests by reusing basic functional tests for the system under test. Test components are used to emulate system clients. These test components perform the basic functional tests to evaluate the reaction of the system to selected service requests or complex service request scenarios. The combination of test components performing different basic functional tests and being executed in parallel leads to different test scenarios for the system under tests and support the evaluation of various system aspects. Parameterization of this test framework enables flexible test setups with varying functional and performance load.

The test framework is based on a set of basic functional tests for the individual services of a system. In separate functional tests, each of those basic functional tests is performed. A service interaction test checks the simultaneous request of different services by applying several basic functional tests concurrently. A separate load test for individual services checks for scalability and load aspects of selected services by using several test components with the same basic functional test, while combined load test checks for a mixture of requests for different services. These combined load tests use test components performing different functional tests. All the tests return not only a test verdict but also the response times for the individual requests. A key element of this test framework is its genericity of being to a large degree independent of the concrete system to be tested.

The application of this test framework to an example Web service is presented. At first, an overview on Web services, XML and SOAP and a discussion on testing Web services are given. The test framework is presented next. Selected details of the test framework are discussed. Conclusions finish the paper.

WEB SERVICES

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.

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. 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. XML schemas [2] are defined using the same basic XML syntax of tags and end tags and actually follow a well-defined DTD. 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. XML Schemas are very flexible and allow describing the same rules in many different ways depending on the use of type inheritance, restrictions and extensions, global and local definitions, embedded, flat catalog and named type structuring constructs.

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 1).

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.

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>
```

Figure 1. XML Schema for the Weather Service

SOAP is a simple mechanism for exchanging structured and typed information between peers in a decentralized distributed environment using XML [5][4]. 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.

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 fulfillment of established system features in the new release. First approaches towards automated testing with proprietary test solutions exist [10], 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 cases 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.

THE TEST FRAMEWORK

We have developed a hierarchy of tests for evaluating Web services for functional and load aspects. The basic idea is to define service interaction, scalability and load tests by reusing basic functional tests for the Web service. Test components are used to emulate Web service clients. These test components perform basic functional tests to evaluate the reaction of the Web service to their requests. The combination of test components performing different basic functional tests and being executed in parallel leads to different test scenarios for the Web service. Parameterization of this test framework enables flexible test setups with varying functional and performance load. The basis of the test framework (see Figure 2) is a set of basic functional tests for the individual services of a Web service. In separate functional tests, each of those basic functional tests is performed. A service interaction test checks the simultaneous request of different services by applying several basic functional tests concurrently. A separate load test for individual services checks for scalability and load aspects of selected services by using several test components with the same basic functional test, while combined load test checks for a mixture of requests for different services. These combined load tests use test components performing different functional tests. All the tests return not only a test verdict but also the response times for the individual requests.

An important aspect of this test framework is its genericity of being to a large degree independent of the concrete Web service to be tested. Besides the basic functional tests, fixed test case definitions for the separate functional, service interaction, and separate load as well as for the service mixture load test can be given. Further test patterns can be envisaged.

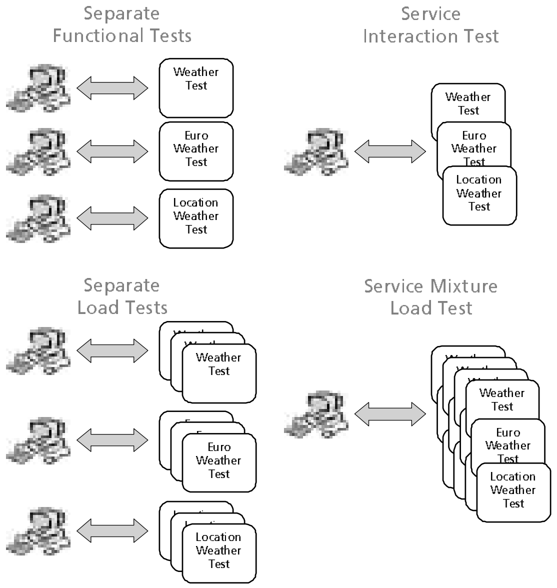

Figure 2. Test hierarchy for Web services

The test framework has been realized with the Testing and Test Control Notation TTCN-3 [7], 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.

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 behavior 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 behavior 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.

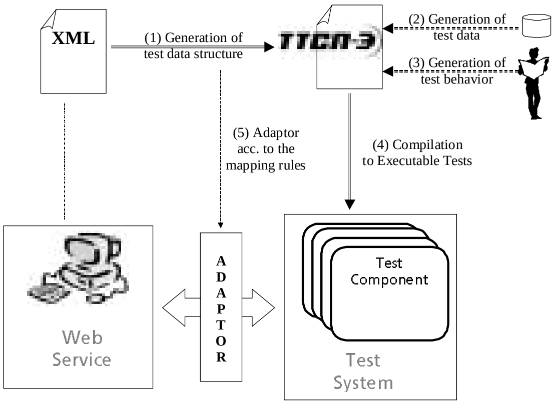
XML
(1) Generation of test data structure
TTCN-3
(2) Generation of test data
(3) Generation of test behavior
(5) Adaptor acc. to the mapping rules
(4) Compilation to Executable Tests
Web Service
ADAPTOR
Test Component
Test System

Figure 3. Testing of Web services with TTCN-3

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 behavior of the service being tested that consist of sequences of requests and responses.

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

1. The structure of the test data is derived from the XML definition with a set of mapping rules from XML to TTCN-3.
2. Test data (i.e. the concrete values for test stimuli and observations) is generated.
3. Test configuration (i.e. the communication structure between test system and system under test) that respects the structure of the Web service to be tested.
4. Test behavior (i.e. the sequences of test stimuli and observations) is generated.
5. The resulting TTCN-3 module is compiled to executable code.
6. 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), (4), and (5) can be automated with the help of tools. The automation for step (2) and (3) requires further work: for this step mainly test generation approaches based on finite state machines or labeled transition systems will be used. The test adaptor for step (6) 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).

THE TESTS OF THE TEST FRAMEWORK

The tests of the test framework follow all the same procedure: the main test component (MTC) creates parallel test components (PTCs) according to the services to be tested (the create statement) and according to the load to be generated (the for loop). Every PTC gets a concrete test function assigned and is started (the start statement). Afterwards, the MTC awaits the termination of all PTCs (the all component.done statement). The overall test verdict is the accumulated test verdict of the local test verdicts of the PTCs.

The generic test cases can be controlled with a general test case control mechanism like shown in Figure 4. Firstly in the control part, the functionality of each service offered by a Web service is tested. The results of the tests are recorded and are used as a basis to guide the further execution of the test campaign. If for example, a functional test for a service fails, it is meaningless to test for service interaction and load aspects for this service. Subsequent to the functional tests, 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.

Another aspect of this test framework is the evaluation of the final verdict: in functional and conformance testing every failure detected by a single test component will lead to an overall fail of the complete test. This is also the built-in verdict mechanism of TTCN-3. However, in load tests this is not applicable: a load tests checks whether certain thresholds like “99% request are successful” are fulfilled. Therefore, a specific verdict type has to be used to handle the collection of the local PTC verdict and to accumulate them according to the requirements of specific tests.

```
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 4. Execution Control for the Test Framework

For that, the MTC was extended to handle the arbitration of PTC verdict to the overall test verdict.

REALIZATION WITH TTCN-3

This section completes the discussion of automating the test framework with TTCN-3. A core element of the automation is the definition of a XML to TTCN-3 mapping, which supports the derivation of test data types from XML schema definitions, and is therefore the basis for testing of XML interfaces with TTCN-3. The mapping rules from XML to TTCN-3 have been provided in [13].

Test data

Templates are used to define the concrete test data to be used for requests to and responses from the Web service. Figure 5 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 [11] 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 5. Test data for the Weather service

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 6). 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 6. Test components

Basic test function for the weather service

The basic test function for the weather service is depicted in Figure 7.

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.

```
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 7. Basic test function for the Weather service

The map operation at the beginning enables the communication of the PTC to the Weather service. The if statement allows to differentiate the test behavior 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.

CONCLUSION

TTCN-3 is the new test specification and implementation technique being applicable to a wide range of test kinds for various system technologies [14]. It is also suited for system level testing. This paper discusses system level tests for Web services with TTCN-3. Beyond the functional and load aspects, aspects like security, privacy, availability, accuracy and usability have to be tested.

The paper presents a flexible test framework for Web services realized in TTCN-3. The tool environment supporting this test framework consists of a TTCN-3 to Java compiler TTthree [12], an XML to TTCN-3 conversion tool and a test adaptor for XML/SOAP interfaces. 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 from XML to TTCN-3 have to be respected by the tests being defined in 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. Which aspect of a Web service is tested, is defined by basic test functions: a functional test will check for the request/response behavior, a security test will check for data integrity, authorization, encryption, etc.

The provided test framework with its test hierarchy 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 a Web service. For that, XML DTDs and Schemas have been analyzed 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 principles of the test framework can be applied to other systems and system components such as other middleware or Internet technologies as well. However, if the data specification technique changes, another mapping to TTCN-3 data structures and a corresponding test adaptor will be needed.

```
testcase SeparateFunctionalTest
(integer Service)
runs on MTCType system SUTType
{
  var PTCType PTC:= PTCType.create;
  PTC.start(SeparateFunctional(Service));
  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 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 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 8. Test cases for the different kinds of tests in the Test Framework

While the paper concentrates on functional and load tests, more work on the basic test functions to address additional aspects is needed. Furthermore, test patterns beyond the presented functional, service interaction, and load tests should be investigated. In any case, test automation will be essential to a sound and efficient automated system level test process, for the assessment of the functionality, performance and scalability of systems.

Future work will further elaborate methods for test data and test behavior generation. In particular, the classification tree method will be investigated for potential extension towards the generation of TTCN-3 templates. The generation of test behavior skeletons from Message Sequence Chart (MSC) specifications is under development. Special emphasis will be given to distributed test configurations with appropriate coordination and synchronization between test components.

The development of the UML Testing Profile at OMG [15] will ease the integrated design and development of test systems together with the system itself – system level tests can be developed on an abstract level on the basis of use cases and use scenarios. The mapping of the UML Testing Profile to TTCN-3 enables the direct execution of such tests on TTCN-3 infrastructures.

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 0,1,2: Primer, Structures, Datatypes*, W3C Recommendations, 2 May 2001, http://www.w3.org/TR/2001/REC-xmlschema-{0,1,2}-20010502

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

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

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

[6] Don Box MSDN magazine on the Web: *A Young person's guide tot the simple object access protocol: SOAP increases interoperability across platforms and languages.*

[7] 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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] I. Schieferdecker, B. Stepien: *Automated Testing of XML/SOAP based Web Services*. Proc. Of the GI Fachtagung “Kommunikation in Verteilten Systemen”, KIVS 2003, Leipzig, Germany, Febr. 2003.

[14] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles, C. Willcock: *An Introduction into the Testing and Test Control Notation (TTCN-3).* Accepted to Appear in Computer Networks Journal, 2003.

[15] I. Schieferdecker, Z. R. Dai, J. Grabowski, A. Rennoch: *The UML 2.0 Testing Profile and its Relation to TTCN-3*. IFIP 15th Intern. Conf. on Testing Communicating Systems - TestCom 2003, Cannes, France, May 2003.