Generation of TTCN Test Cases from Use Case Map Scenarios

By: Bryan Mulvihill, 1724192
Supervised by: Dr. Daniel Amyot
April 14, 2003

Table of Contents

Abstract

Use Case Maps Scenarios are used to model the functional requirements of a system. The Testing and Test Control Notation is a language used to write detailed test specifications. Converting Use Case Map Scenarios to Testing and Test Control Notation test cases would enable the generation of test cases directly from function requirements. This report details the conversion from Use Case Map Scenarios to TTCN test cases, and gives a brief overview of some of the technologies used in the conversion.

Introduction

The goal of this project was to convert Use Case Map Scenarios into test cases described in the Testing and Test Control Notation. This goal was accomplished using a Conversion utility written first in C++ then in Java, and an XSLT stylesheet. The Xalan and Xerces libraries, developed by the Apache Software Foundation, were used for the XSLT processing and XML parsing.

Converting Use Case Map Scenarios to TTCN test cases will enable the generation of test cases directly from functional requirements. This would allow easy system verification at each integration phase of a project, or give a good idea of how many requirements are left unfulfilled. Test case generation also allows a proof of concept to be shown directly from a high level design. This would help point out design errors early, potentially saving a great deal of time and effort.

This report will give an overview of Use Case Maps, Use Case Map Scenarios, the Testing and Test Control Notation, a brief explanation of some of the technologies involved, and finally, detail the mapping of UCM Scenarios to TTCN.

Use Case Maps

Use Case Maps (UCMs) are a scenario-based software engineering technique that have a history of application to the description of object-oriented systems and reactive systems in various areas, including wireless, mobile, and agent domains. [1]

UCMs are used as a visual notation for describing causal relationships between responsibilities of one or more use cases, optionally allocated to a structure of abstract components. UCMs are most useful at the early stages of software development and are applicable to use case capturing and elicitation, use case validation, as well as high-level architectural design and test case generation. [1]

UCMs model the functional requirements as scenarios. [1]

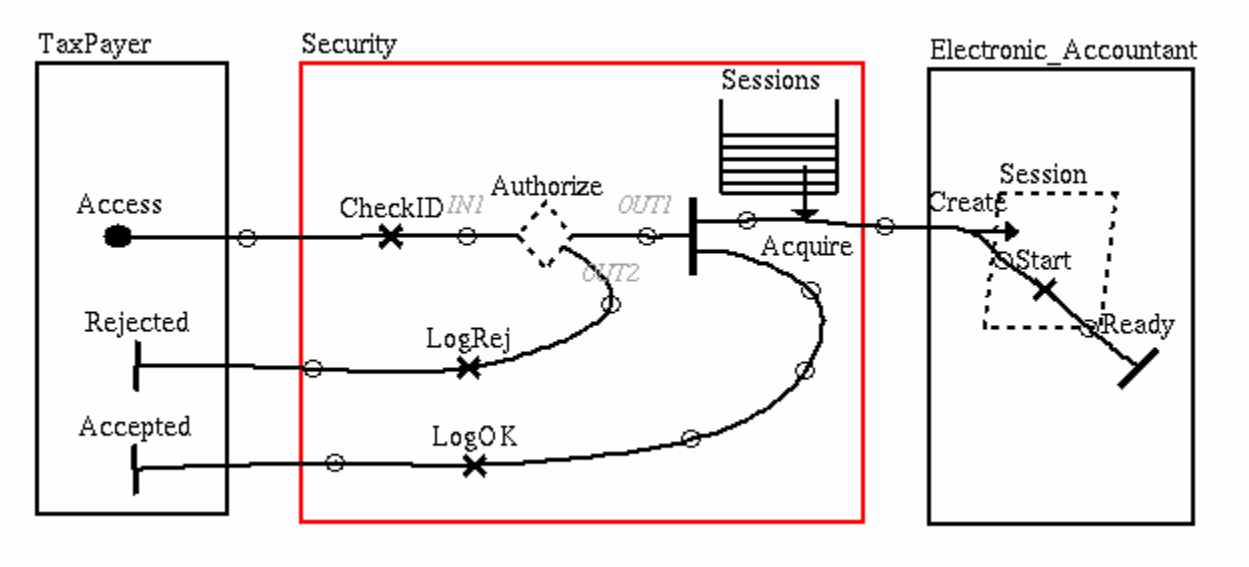

Figure 1 - An example Use Case Map

Use Case Map Scenarios

Modeling functional requirements of complex systems often implies an early emphasis on behavioural aspects such as interactions between the system and its environment (and users), on the cause to effect relationships among these interactions, and on intermediate activities performed by the system. Scenarios represent an excellent and usable way of describing these aspects. [6]

UCM Scenarios define a path through the UCM given a set of conditions. They allow for the extraction of individual scenarios from an integrated collection for visualization,

analysis, and conversion to other representations, such as message sequence charts or TTCN.

Below is an example of a UCM Scenario:

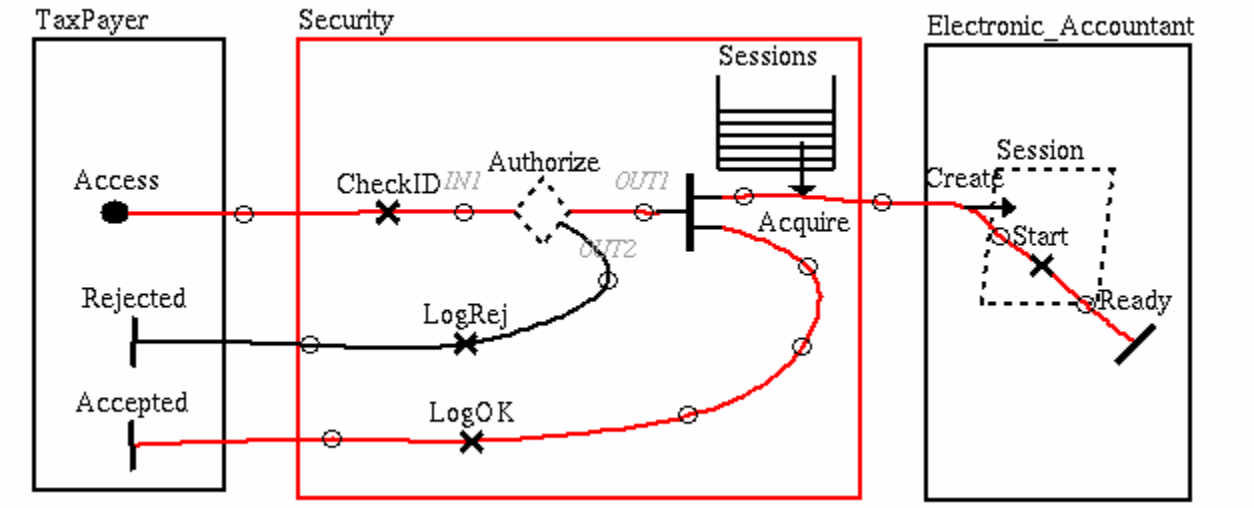

Figure 2 - An example UCM Scenario

The Scenario follows the red line from the start to multiple end points. Using UCM Scenarios in this manner makes it easy for users to easily visualize the flow of data in a system.

UCM Scenarios are described in a simple XML format with seven elements, and have the following structure:

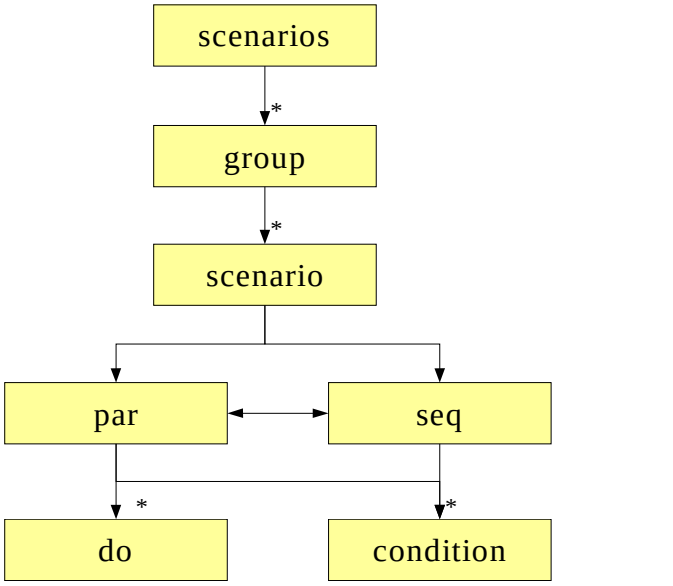

Figure 3 - Structure of a UCM Scenario

At the root of the UCM Scenario is the scenarios element. Scenarios can contain zero or more group elements (or groups of scenarios), which contain zero or more scenario elements. A scenario is a set of sequential (seq) or parallel (par) instructions, which contain do events, conditions, and parallel or sequential set of instructions.

Testing and Test Control Notation

The Testing and Test Control Notation (TTCN), is a language used to write detailed test specifications. TTCN has been used to specify tests for many kinds of applications, including mobile communications (GSM, 3G, TETRA), wireless LANs (Hiperlan/2), cordless phones (DECT), Broadband technologies (B-ISDN, ATM), CORBA-based platforms and Internet protocols such as IPv6, SIGTRAN, SIP and OSP. [2]

TTCN is applicable to the specification of all types of reactive system tests over a variety of communication interfaces. TTCN is intended to be used for the specification of test suites, which are independent of test methods, layers and protocols. TTCN is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. [3]

TTCN programs consist mainly of: modules, components, functions, signature declarations, test cases, and ports.

Modules

Modules are the principal building blocks of TTCN. A module may define a fully executable test suite or just a library. [3] A module consists of an optional definitions part, and an optional control part. The module definitions part specifies the top-level definitions of a TTCN-3 module like test components, communication ports, data types, constants, test data templates, functions, signatures for remote procedure calls, signature templates, named alternatives or test cases. The (optional) control part of a TTCN-3 module can be compared to the *main* function of a C or C++ program. A TTCN-3 module control part executes the test cases specified (or imported) in the module definitions part. A TTCN-3 module without control part can be considered to be a test library. [4]

Modules can import specific component definitions or functions from other modules using the import statement, or import all of a module's definitions part.

Components

A test case is composed of a set of one or more test components. The test case behavior is executed on these components. The **component** type defines which ports are associated with a component. A **component** type definition is also used to define the test system interface, because conceptually component type definitions and test system interface

definitions have the same form, i.e., both are collections of ports defining possible connection points. [4]

Functions

In TTCN-3, functions are used to express test behavior or to structure computation in a module, for example, to calculate a single value or to initialize a set of variables. A function may be parameterized and may return a value. The return value is defined by the **return** keyword followed by a type identifier. If no **return** is specified then the function result is void. An explicit keyword for void does not exist in TTCN-3.

Signatures

Procedure signatures (or signatures for short) are needed for synchronous communication. A procedure may either be invoked in the SUT (i.e., the test system performs the call) or invoked in the test system (i.e., the SUT performs the call). For both procedures called from the SUT and procedures called from the test system the complete procedure signature shall be defined in the TTCN-3 module. [3]

Within a signature definition the parameter list may include parameter identifiers, parameter types and their direction, i.e., in, out, or inout. [4]

Ports

Ports facilitate communication between test components and between test components and the test system interface. TTCN-3 ports are either message-based or procedure-based. Message-based ports are used for asynchronous communication by means of message exchange. Procedure-based ports are used for synchronous communication by means of remote procedure calls. Ports are directional and each port may have an **in** list (for the *in* direction), an **out** list (for the *out* direction) or an **inout** list (for both directions) of allowed messages or procedures. [4]

Ports may be specified as mixed, allowing the port to be both message-based and procedure-based.

Requirements

This project had two main components: the development of a Converter utility that would take as input a UCM Scenario and output TTCN test cases, and an XSLT stylesheet to perform the actual transformation itself.

Implementation

Technologies

The implementation of this project leveraged several different technologies, described briefly below.

XML

The eXtensible Markup Language (XML) is a standard for describing data in XML documents. XML documents contain elements which give meaning to the data contained inside them. XML documents are text based and meant to be easily read by humans and easy to parse by computer programs. Below is an example of a simple XML document:

```
<?xml version="1.0"?>
<memo>
<to>Bob</to>
<from>Jim</from>
<subject>Pizza Night?</subject>
<body>Is there going to be a pizza night on Friday</body>
</memo>
```

From this example, it is clear that the document is a memo regarding a pizza night.

UCM Scenarios are defined by an XML format.

XSLT

The Extensible Stylesheet Language (XSL) is a language for expressing stylesheets. It consists of three parts: XSL Transformations (XSLT): a language for transforming XML documents, the XML Path Language (XPath), an expression language used by XSLT to access or refer to parts of an XML document. (XPath is also used by the XML Linking specification). The third part is XSL Formatting Objects: an XML vocabulary for specifying formatting semantics. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. [5]

Using XSLT, an XML document can be transformed into another XML document, HTML, or a plain text document.

To convert the UCM Scenarios into TTCN, an XSLT stylesheet was used to transform the UCM Scenario defined in an XML format to TTCN in a plain text format.

Document Object Model

The Document Object Model (DOM) is a standard interface for extracting and updating information in a parsed XML or HTML document. When the document is parsing, it is put in a tree structure consisting of document nodes, element nodes, attribute nodes, and text nodes.

The DOM interface was used to extract information from the UCM Scenario and put in extra information in a manner did not change the original UCM Scenario file, but made it easier to process.

Xalan & Xerces

Xalan and Xerces are two software libraries being developed by the Apache Software Foundation. Xalan is an XSLT processor, which takes an XML document and processes it using an XSL stylesheet. Xerces is an XML parser that can be used by Xalan to read in the XML document to be transformed. Xalan and Xerces have been implemented in both Java and C/C++ and run on a wide variety of configurations.

Java vs. C++

The question of whether to use Java or C++ for a project has lead to many a heated argument on the Internet. The decision of using Java or C++ for this project was based on ease of use and support for non-standard XSLT extensions.

Originally, the Converter utility was written in C++ for the reason that it would generate a compiled executable. This worked well for the project until actual validation of the generated TTCN was done. The TTCN parser TTthree required modules to be declared in their own TTCN file. Since a UCM Scenario could require the generation of multiple modules, the Converter would have to parse the UCM Scenario and serialize parts of it to different files that would each be processed, or a method could be found to redirect the output during processing by the stylesheet. The latter approach would be used as it provided a more flexible solution.

Unfortunately, the C++ version of Xalan did not support a method for redirecting the output of a XSLT transformation. This would make the first approach more attractive, except that DOM manipulations using the C++ versions of Xalan and Xerces were difficult to do. The Java version of Xalan did have an extension element for redirecting output, namely xsltc:output, and was easier to perform DOM manipulations with. The conversion from using C++ to Java took little time and provided a more flexible, elegant program.

UCM Scenario to TTCN-3 Mappings

The following sections detail the logical mappings of UCM Scenario elements to their respective TTCN code. There are seven elements defined in a UCM Scenario, but three of them did not require a mapping, namely, the scenarios, condition, and seq elements. A brief explanation is given for each element that was mapped.

Group

Group elements contain a group of scenario elements. Group elements will become modules.

```
<group name=”GroupName” group-id = "1" description=”group description”>
…
</group>
```

```
module GroupName
{
//group description
…
}
```

Since modules had to be in their own TTCN file, the output was redirected to the file “@name.ttcn3”. In the above example, the file GroupName.ttcn3 would be created.

Scenario

The scenario element contained a sequential or parallel set of instructions. It was modelled as a test case.

```
<scenario name='ScenarioName'>
…
</scenario>
```

```
testcase ScenarioName()
runs on DefaultComponent
system DefaultComponent
{
```

```
    map(mtc:thePort, system:thePort);
…
}
```

The component type DefaultComponent is defined in each module with a mixed port called thePort. thePort is an inout port which accepts all procedures and data types.

The map command maps the main test components (MTC) port to the system under tests (SUT) port so that the MTC could conceivable communicate with the test system.

Par

Par elements represent a set of instructions to be executed in parallel. They were modelled as parallel test components (PTC) and called when encountered in the scenario. This required the group template to look for all the seq elements with par elements as their parent node and create functions for each of them. The name of the function is based on the nodes id value, created by the XSLT function generate-id as seq elements do not have an id attribute.

When the par element is encountered in the scenario, a PTC is created and started for each seq element inside the par element. After all PTCs are started, the stylesheet outputs code to wait for all of the created PTCs.

```
<par>
    <seq>
    …
    </seq>
    <seq>
    …
    </seq>
</par>
```

```
//created by group template
function PTCN29() runs on DefaultPTC
{
…
};

function PTCN40() runs on DefaultPTC
{
```

```
…
};

//created by par template
var DefaultPTC vPTCN29:= DefaultPTC.create;
vPTCN29.start( PTCN29() );
var DefaultPTC vPTCN40:= DefaultPTC.create;
vPTCN40.start( PTCN40() );

vPTCN29.done;
vPTCN40.done;
```

Do

The do element represents an instruction in the UCM Scenario. There are eleven different types of do elements: Resp, Start, End_Point, WP_Enter, WP_Leave, Connect_Start, Connect_End, Trigger_End, Timer_Set, Timer_Reset, and Timeout. The Start, End_Point, WP_Enter, WP_Leave, Connect_Start, Connect_End, and Trigger_End types were not necessary and as such had no mapping to TTCN. The types that were necessary are described below.

Resp

The do element with type Resp, short for responsibility, acts like a function call to the system. While TTCN has functions, using this approach would not take advantage of the fact that a component name is specified for the responsibility. Instead the responsibility was modelled as a remote procedure call to the system using a signature declaration, and the SUT would then delegate the call to the proper component.

```
<do hyperedge-id="17" name="Acquire" type="Resp" component-name = "Security"
component-id= "2" />
```

```
signature Security00_00Reject() return integer;
…
thePort.call(Security00_00Acquire);
alt {
    [] thePort.getreply(Security00_00Acquire value 0){ }
```

```
    [] thePort.getreply {   //non zero value, fail
    verdict.set(fail);
    }
}
```

In the above XML, the Resp called Acquire is made of the component named Security. From this, a signature declaration is made using the component name, a delimiter, and the name of the responsibility. When the do element is encountered in the UCM Scenario, a synchronous call on the main test component's port with the proper signature specified. It is assumed that the system will accept the call, parse the signature name, and make a function call to the proper component. Using an alt statement, the return value of the function call is checked. If the reply is a zero, it is assumed the responsibility was executed successfully. A non-zero value is taken to mean a failure, and the verdict of the entire test case is set to fail.

To correctly generate the signature declarations, certain steps had to be taken in the Converter utility. A data structure was created that contained a string for the component name and a set of function names. Since sets can only contain unique elements, no duplicate function names would be created. A list of these structures was created to hold the different components. The Converter utility then looked at all the do elements with type Resp, and checked its component name. If that name were not on the list, a new component would be created. Then the function name would be added to the component's set of functions. This list of components and function names was then added to the documents DOM, and the results processed by the stylesheet, creating a list of unique signature declarations.

An alt statement in TTCN is a like a switch statement in C++. It will execute the first expression that evaluates to true.

Timer_Set

The do element with type Timer_Set represents the start of a named timer. Although a component name is given, that information is not used since timers are used across components.

```
<do hyperedge-id="43" name="Wait4PW" type="Timer_Set" component-name
="TaxPayer" component-id= "5"/>
```

```
Wait4PW.start;
```

Here, the named timer Wait4PW is started. In the generated TTCN, there is a list of timers with global scope in the module. To get a list of unique timer names, when the UCM Scenario is being processed by the Converter utility, all the do elements are

checked for timer names. If one is found, it is put into a set to ensure that only unique names are preserved.

Timer_Reset

The do element with type Timer_Reset represents the stopping of a named timer. It works much like the Timer_Set template.

```
<do hyperedge-id="43" name="Wait4PW" type="Timer_Reset" component-name =
"Security" component-id= "2"/>
```

```
Wait4PW.stop;
```

Timeout

The do element with type Timeout represents a timeout event of a named timer. It works much like Timer_Set and Timer_Reset templates.

```
<do hyperedge-id="43" name="Wait4PW" type="Timeout" component-name = "Security"
component-id= "2"/>
```

```
Wait4PW.timeout;
```

Validation

To ensure that the generated TTCN was valid, a parser was required. The TTthree compiler from Testing Technologies was used for this purpose. It was able to point out where any syntax errors were, and compiled the TTCN into java class files if they parsed perfectly. These java classes were then compiled into a jar file that could be used by another of Testing Technologies products, µTTman, which allows users to execute their test cases.

Generated Java

Having responsibilities modeled as remote procedure calls wouldn't make much sense unless there were remote procedure calls somewhere to be made. Producing Java classes based on the signature declarations was simple given that a list of components with list of their functions was already added to the UCM Scenario DOM. Producing a Java class became a matter of redirecting the output to the appropriate .java file, creating a class,

and processing all of the components functions, which would create public static methods for that class to match the signature declaration.

```
signature Security00_00Acquire() return integer;
signature Security00_00CheckID() return integer;
```

```
//Security.java
public class Security
{
    public static int Acquire()
    {
          return 0;
    }

    public static int CheckID()
    {
          return 0;
    }
}
```

Future Work

While this project accomplished all of its intended goals, there are still other aspects that could be improved or implemented.

The most useful aspect to implement would be to have an actual test case run using µTTman. To accomplish this, a custom TestAdapter would have to be written to interface with µTTman and delegate remote procedure calls to the proper classes. Because of the way the signatures were declared, the default TestAdapter would not have been sufficient for running test cases. A custom test adapter was in fact implemented, but because of a problem with either it or the module loader file (.mlf), the TestAdapter would not load.

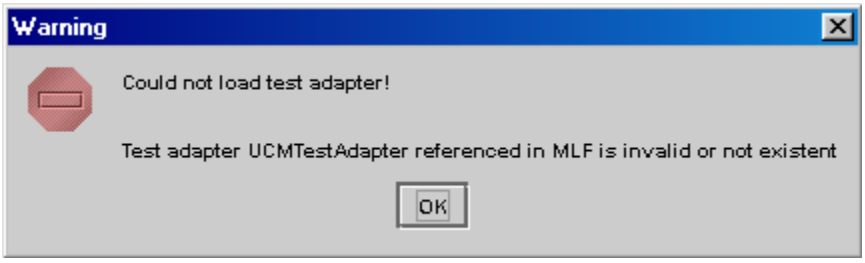

Figure 4 - Error Message displayed by µTTman

Due to time constraints, the cause of this error could not be investigated any further.

One improvement that would not be difficult to implement would be to clean up the component and function names. To accomplish this, when the component names are being collected in the Converter utility, they could be run through a function to take out any characters that would be illegal in TTCN, such as spaces. This would allow a more readable delimiter to be used, generating cleaner TTCN.

Another improvement would be to save the state information from the UCM Scenario and send that information to the system under test at the beginning of the test case. This way the system and all of its components could be reset to a known configuration.

Another improvement would be to implement multiple test components to simulate a distributed test environment. A distributed test environment would be a great asset as distributed architectures are becoming more prevalent. Implementing this type of environment could be done by using arrays of ports in the test components, or by sending an id of the particular test component that is to be communicated with.

Conclusion

This project shows that it is possible to convert Use Case Map Scenarios to test cases in the Testing and Test Control Notation. In doing so, test cases can be generated directly from function requirements. It also allows a proof of concept to be shown directly from a high level design.

This task was accomplished using a variety of technologies that gave an extendable, flexible solution.

Appendix A: Sample UCM Scenario

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">


<scenarios date ="Tue Oct  1 13:31:22 2002" ucm-file = "SAM02.ucm" design-name =
"SAM02" ucm-design-version = "15">
  <group name = "PassWord"  group-id = "2" >
    <scenario name = "PWOK" scenario-definition-id = "1" >
      <seq>
        <par>
          <seq>
            <do hyperedge-id="0" name="Access" type="Start" component-name =
"TaxPayer" component-id= "5" />
            <do hyperedge-id="3" name="CheckID" type="Resp" component-name =
"Security" component-id= "2" />
            <do hyperedge-id="41" name="PW" type="Connect_Start" component-name =
"TaxPayer" component-id= "5"/>
            <do hyperedge-id="43" name="Wait4PW" type="Timer_Set" component-name
= "TaxPayer" component-id= "5"/>
          </seq>
          <seq>
            <do hyperedge-id="59" name="InputPW" type="Start" component-name =
"TaxPayer" component-id= "5" />
            <do hyperedge-id="61" type="Trigger_End" component-name = "Security"
component-id= "2" />
          </seq>
        </par>
        <do hyperedge-id="43" name="Wait4PW" type="Timer_Reset" component-name =
"Security" component-id= "2" />
        <condition hyperedge-id="48" label="[PWOK]" expression ="bv0" />
        <do hyperedge-id="56" name="Continue" type="Resp" component-name =
"Security" component-id= "2"/>
        <do hyperedge-id="42" name="Yes" type="Connect_End" component-name =
"Security" component-id= "2"/>
        <par>
          <seq>
            <do hyperedge-id="17" name="Acquire" type="Resp" component-name =
"Security" component-id= "2" />
```

```
            <do hyperedge-id="19" name="Create" type="Resp" component-name =
"Electronic_Accountant" component-id= "1" />
            <do hyperedge-id="21" name="Start" type="Resp" component-name =
"Session" component-id= "3" />
            <do hyperedge-id="2" name="Ready" type="End_Point" component-name =
"Electronic_Accountant" component-id= "1" />
          </seq>
          <seq>
            <do hyperedge-id="11" name="LogOK" type="Resp" component-name =
"Security" component-id= "2" />
            <do hyperedge-id="8" name="Accepted" type="End_Point" component-name =
"TaxPayer" component-id= "5" />
          </seq>
        </par>
      </seq>
    </scenario>
     <scenario name = "PWNotOK" scenario-definition-id = "2" >
      <seq>
        <par>
          <seq>
            <do hyperedge-id="0" name="Access" type="Start" component-name =
"TaxPayer" component-id= "5" />
            <do hyperedge-id="3" name="CheckID" type="Resp" component-name =
"Security" component-id= "2" />
            <do hyperedge-id="41" name="PW" type="Connect_Start" component-name =
"TaxPayer" component-id= "5"/>
            <do hyperedge-id="43" name="Wait4PW" type="Timer_Set" component-name
= "TaxPayer" component-id= "5"/>
          </seq>
          <seq>
            <do hyperedge-id="59" name="InputPW" type="Start" component-name =
"TaxPayer" component-id= "5" />
            <do hyperedge-id="61" type="Trigger_End" component-name = "Security"
component-id= "2"/>
          </seq>
        </par>
        <do hyperedge-id="43" name="Wait4PW" type="Timer_Reset" component-name =
"Security" component-id= "2"/>
        <condition hyperedge-id="48" label="[PWNotOK]" expression ="!bv0" />
        <do hyperedge-id="53" name="Reject" type="Resp" component-name = "Security"
component-id= "2"/>
```

```
        <do hyperedge-id="47" name="No" type="Connect_End" component-name =
"Security" component-id= "2"/>
        <do hyperedge-id="16" name="LogRej" type="Resp" component-name = "Security"
component-id= "2" />
        <do hyperedge-id="14" name="Rejected" type="End_Point" component-name =
"TaxPayer" component-id= "5" />
      </seq>
    </scenario>
  </group>
</scenarios>
```

Appendix B: TTCN Generated from Sample in Appendix A

```
module PassWord
{
    /*

    */
    //Port type definition
    type port MixedPort mixed
    {
          inout all
    }
    timer Wait4PW;

    signature Security00_00LogRej() return integer;
    signature Security00_00CheckID() return integer;
    signature Security00_00Reject() return integer;

    // Default Component
    type component DefaultComponent
    {
          port MixedPort thePort;
    }

    type component DefaultPTC
    {
          port MixedPort thePort;
    }

    function PTCN12() runs on DefaultPTC
    {
                  //Start

                thePort.call(Security00_00CheckID);
          alt {
                [] thePort.getreply(Security00_00CheckID value 0) {
}
```

```
                [] thePort.getreply {   //non zero value, fail
                      verdict.set(fail);
                }
          }

                 Wait4PW.start;
    }

    function PTCN23() runs on DefaultPTC
    {
                  //Start
    }

    testcase PWNotOK()
    runs on DefaultComponent system DefaultComponent
    {
          map(mtc:thePort, system:thePort);


                var DefaultPTC vPTCN12:= DefaultPTC.create;
          vPTCN12.start( PTCN12() );
          var DefaultPTC vPTCN23:= DefaultPTC.create;
          vPTCN23.start( PTCN23() );
          vPTCN12.done;
          vPTCN23.done;

                Wait4PW.stop;
          thePort.call(Security00_00Reject);
          alt {
                [] thePort.getreply(Security00_00Reject value 0)
    { }
                [] thePort.getreply {   //non zero value, fail
                      verdict.set(fail);
                }
          }

                thePort.call(Security00_00LogRej);
```

```
          alt {
                [] thePort.getreply(Security00_00LogRej value 0)
   { }
                [] thePort.getreply {   //non zero value, fail
                      verdict.set(fail);
                }
          }

          verdict.set(pass);      //by default everything passes
          }
 }
```

References

1. Introduction to the User Requirements Notation, Daniel Amyot, http://www.site.uottawa.ca/~misbah/seg3300b/SEG3300-URN.ppt

2. ETSI – Telecom Standards, http://www.etsi.org/frameset/home.htm?/ptcc/ptccttcn3.htm

3. The Tree and Tabular Combined Notation version 3, ITU, 2001 http://www.itu.int/ITU-T/studygroups/com17/languages/Z.140_0701_pre.pdf

4. TTCN-3 - A new Test Specification Language for Black-Box Testing of Distributed Systems, Jens Grabowski, http://www.fokus.gmd.de/research/cc/tip/ttcn3_tutorial/Papers/Grabowski.pdf

5. The Extensible Stylesheet Language, http://www.w3.org/Style/XSL/

6. URN: Towards a New Standard for the Visual Description of Requirements, Daniel Amyot and Gunter Mussbacher, http://www.usecasemaps.org/pub/sam02-URN.pdf