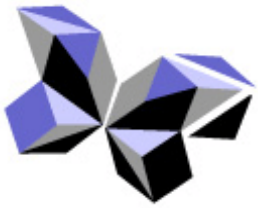

Digital Business Ecosystem

Contract n° 507953

WP 10: Test Automation

Del 10.1: Test Automation Environment

Information Society Technologies

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: D10.1

Due date: 30/05/2005

Delivery Date: 27/05/2005

Short Description:

This document provides an overview of the test automation environment for DBE. This covers both the automated generation of tests, as well as the automated execution of test cases. It also distinguishes between those test services that will be provided as quality support during development of the DBE infrastructure, and those test services that will be provided as part of the DBE execution environment.

Partners owning: UniS Computing

Partners contributed: UniS Computing

Made available to: Consortium and EC

VERSIONING		
VERSION	DATE	AUTHOR, ORGANISATION
DRAFT	20 MAY 2005	PAUL KRAUSE, YONGYAN ZHENG, LEON MAK, UNIS COMPUTING
V^n 1.0	17 JUN 2005	PAUL KRAUSE – EXECUTIVE SUMMARY ADDED; INTRODUCTION SECTION ADDED; REVIEWER'S CORRECTIONS MADE

Quality check

1st Internal Reviewer: Paul Malone, Waterford Institute of Technology

2nd Internal Reviewer: Ruben Gonzalez, Sun Microsystems

Executive Summary

This document summarises the current status of work on the DBE test automation technology. We aim to support two goals with our test automation work. Firstly we wish to use test automation to support the development of high quality software within the DBE project itself. Secondly, we wish to support the deployment of a range of test case generation and execution tools as services to assist in the deployment of high-quality software services by SMEs, once the DBE itself is established.

This document discusses three aspects of test automation:

1. Non-intrusive user acceptance testing using the native user interface of desktop, handheld and mobile devices;
2. Test execution services that may be bound to published DBE services to test for functionality and conformance to the DBE environment, upon deployment by an SME (or collaboration of SMEs);
3. A range of test case generation tools that are in the process of being adapted to support test case generation from the various DBE modelling languages.

Contents

1 Introduction

DBE is creating a next generation infrastructure for supporting the creation and evolution of virtual communities in an e-business environment. In order to succeed, both the services published on the DBE infrastructure, and the infrastructure itself, must be tested to ensure high quality of service standards can be maintained. It is therefore important that we develop software testing technology that can be effectively deployed within the context of DBE to:

a) support the testing of the DBE infrastructure together with all its associated components, and;
b) provide testing services that can be used to evaluate the functionality provided by an SME (or collaboration of SMEs), and its conformance to the DBE infrastructure, once a service (or composition of services) has been published into the ecosystem.

This document summarises the current status of the development of the DBE test technology. Our goal is to automate both the generation and execution of test cases as much as possible, and to do this in a way that supports the evolution of test cases as business requirements change. As a result, this report covers a wide range of different technologies. We summarise them briefly here in order to provide an overview of the scope and purpose of the different technologies that are being provided.

1. Non-intrusive test automation for user acceptance testing. Currently we are working with a commercial tool vendor for this aspect of the test environment. Using this work, we are able to stimulate and evaluate results of software under test by interacting directly through the native used interface of a PC, PDA or 3G phone. We can automate extensive functional and performance tests in a 24x7 simulation of direct user interaction without any intervention into the system under test being needed. This work builds on a long-standing collaboration between the University of Surrey and TestQuest Inc, and so we were able to produce a configuration for DBE quite quickly. This test configuration will be used extensively throughout the second phase of DBE for user testing of the software infrastructure. However, because this work is based around usage of a commercial tool, it is not appropriate (at least in its current form) for inclusion in the DBE workbench for use by DBE service developers. Hence the focus on open-source tools in the next point;
2. Development of a test automation environment for inclusion in the DBE workbench. The goal here is to be able to search for the service description of a published service (upon request), generated a test skeleton from the service description, and then bind to the published service and generate a comprehensive test report. Work here has involved the evaluation and selection of open source tools for test execution, based around JUnit. Although this work fits within the open-source spirit of DBE, it is not yet able to address automated testing at the native user interface.

3. Development of test-case generation from UML models. Currently, we focus on test case generation from Use Cases, Sequence Charts and State models. This work contains some foundational research contributions in software testing. Nevertheless, more work is needed here in order to move these results into the context of MDA and, more particularly, to establish their relevance to test case generation from the modelling languages that are now stabilising in DBE. Our experience to date indicates that this can be achieved through the use of appropriate transformation strategies and then using the most appropriate test case generation techniques from the pool covered in this report.

2 DBE Test Execution Environment

2.1 Introduction

In this section, we will introduce different types of testing required for testing web services, and the experiments we have performed on functional testing of web services. An automated test execution environment (TEE) allows testers to execute sets of test cases largely without human intervention. TEE is most effective when it is set up to support an automated *data driven* execution of test scripts. Automated test execution tools that merely support capture replay demonstrate poor return on investment. As a consequence, our work focuses to a very large extent on the development of robust, maintainable, and configurable test scripts as a basic requirement. But the full benefit of test automation can only be realized if automated test execution is augmented with automated test generation. The latter will be addressed in the later stages of this report.

Testing a system can be categorized by three levels: unit testing, integration testing, and system testing. When the categorization is applied for web services, testing the interfaces of individual web services is a kind of unit-level testing, verifying the execution of BPEL itself is a kind of integration-level testing, and finally testing the interface of a composite web service is a kind of system-level testing. A GUI could be created for each web service so that users can invoke the service's methods, but it is obviously not flexible and scalable to create a different GUI for every web service under test. Since web services do not inherently have a user interface for the user to interact with, a proper way for testing web services is to test them directly at the published interface.

There are many different types of test purpose for a general application, such as functional testing, regression testing, load/performance testing. Similarly, we need to consider these different sorts of testing for web services. As many users may invoke the web services at the same time, for example, load testing is important for evaluating the scalability of the DBE architecture.

1) Functional testing: This is to ensure that the functionality of the web service is as expected. It includes checking the correctness of expected responses for the requests such as security/authentication, and communication protocols.
2) Regression testing: This is to ensure that the web service is still working between builds or releases. Given that some area of functionality was working in the past, the regression testing is to check the web service still works well after changing the requirements. Regression testing must be automated due to its repetitive nature.
3) Load/performance testing: This is to find how the web service scales when increasing the number of users. Functional or regression testing only ensures that the web service works well for a single

user. Hence we need load testing to evaluate whether the web service still responds correctly for multiple users, such as the ability to cope with 10,000 request a second. For the load testing, we need to consider factors such as the number of potential clients of the web service, what is normal access pattern, what is the request protocol (e.g. SOAP requests or HTTP requests), and also what is the acceptable performance.

For testing the DBE web services, we start from installing the whole DBE infrastructure in a local PC. A summary for setting up the DBE infrastructure of Jan-2005 version can be found in Appendix A.

2.2 Review of WSDL4J Engine and TestMaker

There exist various open source tools in the literature for testing web services. We evaluated the two main tools in the market: Apache Axis's WSDL4J and PushToTest's TestMaker, and compare their suitability in the context of DBE.

Apache Axis's Web Services Description Language for Java Toolkit (WSDL4J) allows parsing and understanding the WSDL document to generate a Java server skeleton, test case stub, and a set of supporting Java classes. The generated JUnit test case acts as a service client, which helps the tester to implement a run-able test script in Java. The supporting Java files take care of locating the service, getting the service, and binding the test stub and the service. JDK is required with the WSDL4J engine.

PushToTest's TestMaker provides a framework with GUI for a user to import WSDL. Its 'Agent' Wizard for WSDL uses Apache Axis's WSDL4J engine to parse methods and parameters of the web service. The wizard will generate test case stub in Jython scripting language, which helps the tester to implement a run-able test script in Jython. Jython is a Java implementation of the Python language. JDK is also required for TestMaker.

2.3 Experiment on WSDL4J Engine and TestMaker

2.3.1 Testing Individual Web Services

TestMaker

In TestMaker, the Agent Wizard for WSDL can generate a test case stub in the Jython scripting language. The generated test case stub cannot be re-used directly, because the parameter order of each method is not the same as the parameter order in the target WSDL. Also, when reusing the generated code, we met the "de/serialization" problem, but this problem can by simply avoided by constructing the XML request explicitly. However, the shortcoming of this explicit way is that the test script

almost does not make use of the automated generated code. JDOM is required to parse the returned XML response. The Jython test script for HotelReservation service in the DBE is listed as an example in Appendix B.

WSDL4J Engine

We use WSDL4J Engine as a plug-in for Eclipse to invoke and get responses from an external web service. Here is our process:

1) Create a new Java project for testing, e.g. TestDBE.
2) Add the following jar files to the Java Build Path's library (right click Project->Properties): junit.jar (in DBE infrastructure, this is located in %eclipse_home%\plugins\) jaxrpc.jar,axis.jar,commons-logging.jar,commons-discovery.jar, saaj.jar (in DBE infrastructure, they are located in %tomcat_home%\shared\lib\ )
3) Save the external web service's wsdl file in the created Java project. For instance, we can save the HotelReservation.wsdl file from http://gaudi.techideas.info:8080/axis/servlet/AxisServlet to the TestDBE project.
   - Note that it's not an ideal way to import the wsdl file by saving it manually into a project. If a wizard for importing WSDL is installed in Eclipse, such as the WSDL2JavaWizard, the WSDL file can be imported directly from Eclipse's GUI by clicking "Import->Add web reference" option. However, the WSDL2JavaWizard tool only supports Eclipse2.x, but Eclipse3.x is currently used in DBE. Hence the target WSDL file has to be saved locally for WSDL4J engine to use. A WSDL import wizard suitable for Eclipse3.x can be plug-in in a later stage.
4) For the imported HotelReservation.wsdl, use the built-in wsdl4j engine to generate test case stub, five java files will be generated: HotelReservation.java, HotelReservationLocater.java, HotelReservationService.java, HotelReservationServiceSoapBindingStub.java, HotelReservationTestCase.java. The only file we need to care about is HotelReservationTestCase.java. See the Java test script for HotelReservation service in the Appendix C.
5) Build the TestDBE project, and run the test script.

Both Jython and Java test scripts can generate a result like the following:

```
Now sending simpleAvailabilityCheckingRequest...
simpleChecking available = false
Now sending advancedAvailabilityChecking request...
dailyPice = 0
newRoomType = -1
UnavailabilityReson = There are free rooms (2) we would try with -1 type rooms
```

```
available = false
Now sending makeReservationRequest...
failureReason =  There are no available rooms
reservationNumber =  0
makerReservation successful = false
Now sending canccelReservationRequest...
Cannot cancelReservation, the reservationNumber is not valid
```

2.3.2 Testing Composite Web Services

DBE uses BPEL to model composite services. Since a composite service itself is a web service, it's always the case to create a new WSDL interface for the BPEL model (i.e. composite service). With the WSDL interface, users can send requests to the composite service by invoking the methods in the WSDL interface. Therefore, we can use exactly the same approach of testing individual web services for composite web services. However, this approach seriously limits the points of observation that are available for checking execution of the test cases. We will return to this point later when discussing how the DBE interceptor architecture may be used to enhance the test services that we can provide.

As we introduced in section 1.1, in the web service world, testing composite services is system-level testing, executing/verifying BPEL itself is integration-level testing (e.g. checking whether the individual web services coordinate correctly), and finally testing individual web services is unit-level testing. At this stage, we only concern testing composite services and testing individual services in our experiment.

For executing/verifying BPEL itself, IBM's BPEL4J engine has been delivered as Eclipse's plug-in to generate Java code from BPEL models. There also exist academic tools to verify BPEL models. The main idea is to take advantage of the popular verification technique- model checking. BPEL models can be transformed into state-based models, which will in turn be inputted to a model checker for the verification. In the context of DBE, these open-source tools are candidates for testing integration of individual web services. They will be discussed in more detail in a later report which will discuss the generation of test cases from work flow models.

TestMaker

As with testing individual web services, the Agent Wizard for WSDL will generate a test case stub in Jython scripting language from the WSDL interface. For each method invocation, an XML request needs to be constructed for the request, and JDOM parser needs to be created to parse the returned

XML response. The Jython test script for BookingService service in the DBE demo is listed in Appendix D.

WSDL4J Engine

The WSDL interface for BPEL is created manually at the moment, so WSDL4J cannot generate test case stub when the WSDL interface contains syntax errors. Also, when the WSDL for a composite service contains "import individual wsdls" parts (like below), the WSDL4J engine cannot generate test case stub until it can locate the individual WSDL files. The simplest solution is to delete the import parts, because the imported wsdls will not influence the generated code.

<import location="http://localhost:8080/active-bpel/wsdl-catalog?entry=bpr:/wsdl/HotelReservation.wsdl" namespace="urn:soapimpl.hotelreservation.demos.dbe.org" />

The Java test script for BookingService service is listed in Appendix E. Both Java and Jypthon test scripts can get the result from the booking service such as the following.

```
Now sending requests to bookingservice request...
confirmation = true
hotelReservationNumber = 9MB8KIJ
taxiReservationNumber = null
failureReason = null
```

2.4 Comparison of WSDL4J Engine and TestMaker

After having experimented with TestMaker and WSDL4J in Eclipse for functional testing, here we compare them by showing pros and cons. TestMaker's code generation engine makes use of WSDL4J to generate a test case stub, but its output test case is in Jython instead of JUnit test case. Both tools require JDK support.

TestMaker

1) Pros
   - It provides GUI to import the WSDL's URL.
   - It provides a light execution environment. A test script does not need to be compiled, and it can run after being saved.
2) Cons:
   - In the generated test case stub, the parameter order of a method may not conform to the parameter order shown in the target WSDL. We have to change the parameter order following the WSDL manually.

- We need to write low-level code in the test script to build xml request and parse xml response. It means that in order to write a test script, we need to look at both the API of TestMaker script language and the API of JDOM.

WSDL4J Engine in Eclipse:

1) Pros:
   - Being a plug-in in Eclipse, the WSDL4J engine can generate JUnit test case stub and supporting Java classes from the target WSDL. In the generated code, the parameter order of each method always conforms to the parameter order in the WSDL.
   - We can concentrate on writing code to test the web service itself, and no need to care about the low-level message exchanges.
2) Cons:
   - We need to save the target WSDL to the local project manually. There are some wizards for importing WSDL available as Eclipse plug-ins, but a wizard supporting Eclipse3.x still needs to be identified, although this is a minor issue.
   - A test script needs to be complied before running.

As a summary, WSDL4J engine is the preference during functional testing. It has advantage of providing supporting Java classes to handle low-level message exchanges.

2.5 Interceptor Architecture in Testing

According to the DBE core architecture document, the run-time interceptors will define pre and post hooks to method invocation. The interceptor architecture will act as a monitor for the sending requests and receiving responses. In the current instance of the DBE infrastructure, the Axis Handler architecture is being utilized, but this will change in the next evolution. In the next phase, we will investigate the writing of generic “test interceptors” that can be attached to the services under test.

3 TestQuest: Acceptance Test Execution Environment

TestQuest is a powerful test execution environment for non-intrusive testing. When the system under test is a PC, TestQuest is similar to other GUI test execution tools. However, when the software or service under test is embedded in, or accessed through a device such as a mobile phone or a PDA, TestQuest has advantage at providing non-intrusive test environment by simulating the system under test and verifying results through its native interface. In the context of DBE, TestQuest can be used for the user acceptance test environment. In addition to supporting non-intrusive testing, TestQuest supports data driven test script execution and an architecture, TVT (Text Verb Technology), for the development of maintainable and configurable test scripts. UniS has an active research program in the use of User Interface specifications to transform "abstract test cases" into executable test cases. This involves transforming the "logical events" used in a functional specification into the actual physical sequences of user actions needed to trigger these events. This work is used in a number of projects, and so has not been charged to DBE. However, it does have an important impact on our ability to develop advanced test case generation techniques that can be fully exploit the use of non-intrusive testing in DBE.

Non-intrusive Testing

Imagine a client uses a mobile phone to send 100 requests to DBE for some services, and the responses from DBE will be sent to the mobile phones directly. Since TestQuest can simulate and capture the screen of the mobile device, we need not verify all the responses manually by looking at the mobile phone itself. Instead the test script running on TestQuest can automatically verify the response against the request without human intervention.

Data Driven Test Script Execution

Data-driven testing is to separate data from the functions of test scripts, so that test scripts can be reused. The data is stored in data files to provide both the input events and expected outputs, and the data files should be easily maintainable like spreadsheet, access, or oracle. There may be multiple data files required for a test script. TestQuest provides an interface to ODBC databases , e.g Access, SQLServer, and Oracle.

Test Interface

Automated Testing consists of two phases: test case/data generation and test script execution. The generated test cases from the first phase are abstract and platform independent, and the test scripts running on the test execution environment are concrete and platform dependent.

As a result, a test interface is required to fill the gap between the abstract test cases and concrete test scripts. There are two solutions for the test interface:

1) A language specific compiler to transform abstract test cases into concrete test scripts for the language supported by the test execution tool.
2) Define a GUI interface to allow tester to define the mapping between the logical actions (i.e. inputs and outputs) of the abstract test cases to the physical actions of test scripts.

TestQuest provides a Text Verb Technology (TVT), which allows testers to define their action verbs (i.e. TestVerbs). TVT can be the test interface to map the abstract test cases to test scripts. The architecture of TVT is shown in Figure 1. TVT has built-in error handling and logging policies. When a function fails, TVT can roll back to the previous function.

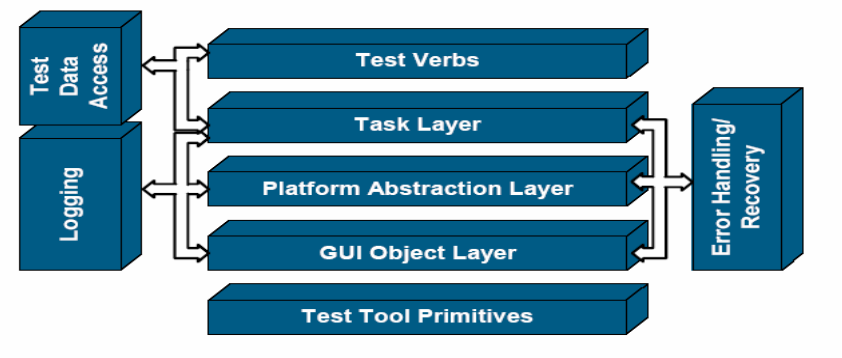

Figure 1 Text Verb Technology (TVT) architecture [TestQuest]

The mapping procedure can be described as:

1) When an abstract test case is generated from the test generator, the input events and output actions in the test case can be the TestVerbs. The TestVerbs will be saved in a data files like spreadsheet or Access. It can be done automatically.
2) The TestVerbs will be added to the Task Layer so that testers can directly use the TestVerbs in test scripts. The Task Layer links the TestVerbs to the functions of the Platform Abstraction Layer (PAL).
3) The implementation of TestVerbs is encapsulated in the PAL. The functions are actual hardware interactions but platform independent. It includes error handling, error recovery, and logging policies, for instance.
4) GUI Object Layers includes external data files (i.e. database) that have information of the system under test such as platforms and languages. Data files can be invoked by the functions in either Task Layer or PAL layer.
5) The Test Tool Primitives Layer includes the lowest level functions that directly interact with hardware. The functions in this layer can be called by the functions of PAL layer.

Within the context of DBE, most of our work on the use of non-intrusive testing has focused on setting up the configuration needed for testing some of the integrated services that could be demonstrated on the DBE infrastructure. Detailed trials will be performed once the June release of the DBE workbench has been installed at UniS.

4 Test Case Generation

Even though an automated test execution environment can improve the efficiency of testing, it cannot improve the quality of testing. The reason is that test scripts still need to be written by testers, which is both time consuming and error prone. An ideal picture is to input the software specification to the test framework, and the test framework can provide the results whether the implementation conforms to the inputted specification. Therefore, when the specification is available, it is desirable to generate test cases automatically from the specification. We will review available techniques for test case generation in this section. We use UML language for the specification, in conformance with DBE's use of the MDA.

4.1 Introduction

Specification based Test Case Generation

Test case generation can be categorized as specification based or code based testing. Code based testing belongs to white box testing which concerns the structure of the system under test (SUT). Specification based testing belongs to black box testing which does not concern the internal structure of the SUT. Black box and white box testing should be used together to complement with each other. The idea of specification based test case generation is shown in Figure2. A test case includes test inputs and expected outputs. Test inputs are generated from the test specification, and will be sent to the SUT to produce test outputs. Ideally the specification should be used as a test oracle to determine whether the actual output from the SUT matches the expected output from the specification.

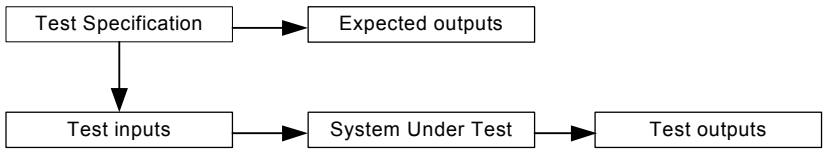

Figure 2 Specification based test case generation

Test Model

Figure 3 shows a classical V test model, where the arrow denotes the time line of a project lifecycle. The test execution lifecycle starts when the code is finished. Unit testing checks whether code meets the detailed design specification, integration testing checks whether previously tested components fit together against the architecture specification. System testing checks whether the fully integrated product meets the requirement specification. This means that there are horizontal dependencies between the respective boxes on the two sides of the V.

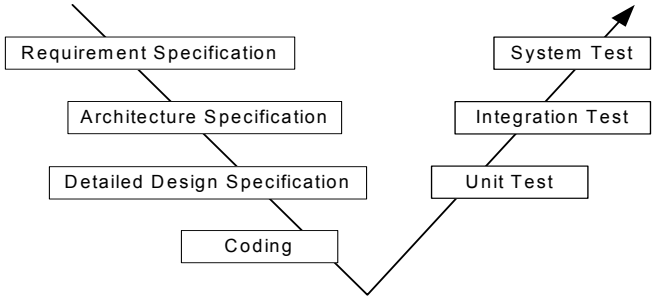

Figure 3 Classical Test Model

According to this V test model, test activities are delayed until the code is implemented. It is a high-risk way to proceed because faults in the specifications will not be detected until the late stage of the development. It is recommended to start the test lifecycle in parallel with the development lifecycle, which means test activities should start when the requirements are available. A refined V test model is shown in Figure 4. When the test specification is derived from the requirement, a test process starts. According to different level test specifications, test cases are generated and executed against the specifications, so that specifications can be verified. When the code is available, the second part of test lifecycle starts. The test cases generated in the early stage will be executed against the SUT, from unit test execution, integration test execution, to system test execution.

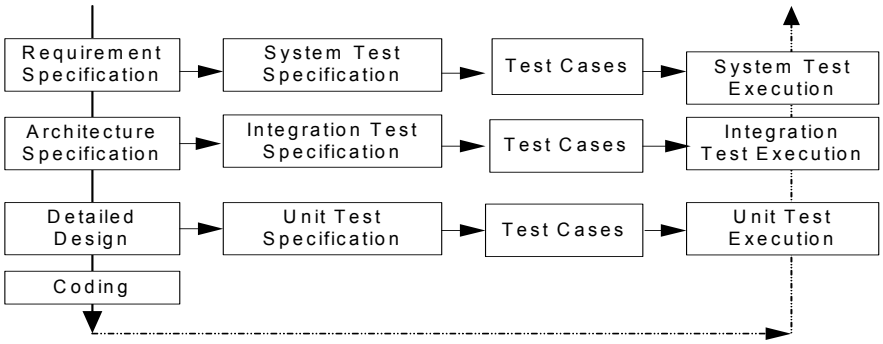

Figure 4 Parallel Test Model

Test Process

Figure 5 shows a test process with automation tool support. The test specification is derived form the system specification, and a generator generates test cases from the test specification. A test interface maps the generated test cases, which are not executable, to test scripts that can execute on the SUT. An execution engine executes the test scripts against SUT, and it also compares the generated test outputs with expected test outputs to generate test results (i.e. pass, fail, inconclusive). It is ideally to automate test case generation (generator), test case execution (execution engine), and test interface.

Automated test case generation can keep test cases synchronized with the updated specification. When software evolves, the tester simply needs to change the test models to regenerate test cases.

Automated test execution helps to improve the mechanical part of testing. Many test execution tools are available and have been used in industries. Test interface automates the mapping from abstract test cases to test scripts.

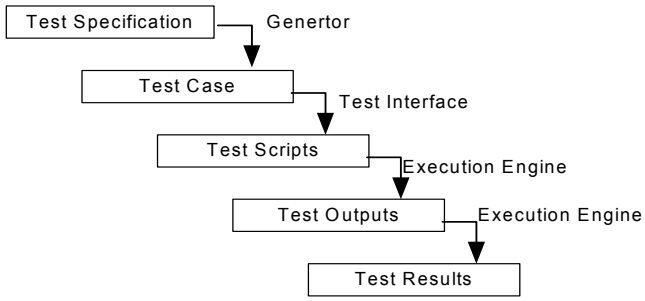

Figure 5 Test Process with automation tool

Test Case and Data Generation

Test generation is composed of test case generation and test data generation. The first phase is test case generation. Given a test specification model, test cases are generated from the model according to some test criteria. A test case corresponds to a selected traversal path on the model. The traversal path is a sequence of input events. The second phase is test data generation. After generating the test cases, the test data must be generated to enable the selected path to be traversed. The test data is selected from the input domain of the specification so that each input is given a value in the generated test case. An ideal test case generation approach should be both efficient and effective.

- Efficiency: the approach can generate short test cases that reach good code coverage.
- Effectiveness: the approach can generate test cases that can detect many faults.

Normally, the more paths the test case covers (i.e. efficient), the better is the fault coverage (i.e. effective), but it is not always necessary the case. A test case generation method that has low code coverage might imply weakness in fault detection. However, a method with high code coverage might not be efficient at detecting faults.

4.2 Use Case based Testing

The complexity of systems causes the creation of a comprehensive state model to derive test case as impractical. Requirement is an alternative source to generate test cases so that the test case explosion problem can be released. Meanwhile, requirement based testing takes advantage of the testers' considerable domain knowledge and insight into the software failure modes. UML use cases are visually represented in use case diagrams and specification template. The template is a table with name, brief description, flow of events (basic or alternatives), special requirements, preconditions, and post-conditions. The most important part of a use case for generating test cases is the flow of events. The flows are normally written as "dialogs" between the system and the actors and each step explains what the actor does and how the system in responds.

[ET03] applies cause effect graph to generate test cases, and proposes use-case relationship matrices to sequence the test cases. The relationship matrices include use case interaction matrix, use case concurrency matrix, and use case data sharing matrix. The approach is scalable but it is not clear how to derive cause effect graph from use case and how to sequence the test cases. [RG99] transforms use cases into finite state machines (FSM), which takes advantage of existing test techniques for FSM. However, FSM is too detailed for requirement based testing. [LBG99] transforms a Use Case Map (UCM) to the LOTOS specification. The UCM describes scenarios in terms of casual relationships between responsibilities. LOTOS provides formal semantics for the UCM, and its existing test techniques can be used. However, the approach needs to model systems using UCM but UCM has not been standardized and integrated with UML. The approach has no consideration of the concurrent relationship of use cases that execute independently. [KC99] proposes use case action matrices to order the importance and reusability for each use case scenario. The method is design oriented as it starts from sequence diagrams to elicit use case scenarios, whilst it can complement other use case based generation techniques to set the priority for use case scenarios. [WP99] proposes a use case flow graph to describe a set of use cases performing in particular orders. The approach integrates category partition method and pair wise coverage criteria for test data generation. However, it does not cover the concurrent relationship of the use cases. [TYLSX03] proposes scenario trees, state-event trees, and an ACDATE to model system behaviors. Using DFS (Depth First Search) traversal, test cases can be generated from the state-event tree. The approach uses partition testing, boundary value testing or random testing for test data generation, and uses the OCL (Object Constraint Language) to specify the constraints. The approach can be an alternative to UML modeling, but UML benefits at linking behavior models with static models and its broadly acceptance by industries.

Given that UML support is a prerequisite, [ET03] is the most appropriate method, and [WP99] can be second selection when it is improved of testing the concurrent relationship of use cases. The comparison of these techniques is shown in table 1.

	UML support	Use Case Interactions	Use Case Concurrency	TestCase Priority	Domain Testing	Use existing test techniques	Tool Support
[ET03]	Y	Y	Y			Y	Y
[RG99]	Y					Y	
[LBG99]		Y	Y			Y	Y
[KC99]	Y			Y			Y
[WP99]	Y	Y			Y		Y
[TYLSX03]		N/A	N/A		Y		Y

Table 1 Comparison of Use Case based Testing

4.3 MSC based Testing

The use of Message Sequence Chart (MSC) as the system models to generate test cases is broadly accepted in telecom industries because it can express the system interactions naturally and explicitly, it is also useful in handling *Feature Interaction (FI*) problem during integration. UML2.0 has adopt MSC as the semantics of Sequence Diagram. MSC has been standardized by ITU (International Telecommunications Union). Also, MSC has richer features than USC, such as co-regions, MSC reference and reference expressions, in-line expressions, high-level MSCs, time constraints, message parameters, and MSC variables. Test case generation approaches for MSCs can be categorized as: 1) Parse the MSC models directly to test cases. 2) Transform MSC to state models, either SDL models or finite state machines.

[BBJKM03] transforms MSCs to test cases. MSCs model the visible interactions between system under test and the environment. The developed tool PTK parses the MSC to a partial graph, and derives test trees (i.e. test cases) from the partial graph. PTK uses various test strategies such as trace testing, branch testing, and completion testing, optimization for test cases, and some rules to support advanced features of MSC2000 such as HMSC, inline, reference expressions, and time constraints. PTK supports executing test scripts either in a single process or in a distributed environment. "Coordinating messages" are used between concurrent test cases to synchronize them. PTK does not concern detecting faults in system internal interactions. [SEGHK96] develops a tool Autolink to generate test case from SDL specification. The method uses MSCs to represent test purposes, and transforms MSCs to SDL. Test case can be derived via exploring state space of the SDL model, according the path selection criteria defined in the test purpose. When the test purpose cannot be represented in the SDL specification, it is not possible to explore state space of SDL to generate test cases. In this case, MSCs have to be directly translated into test cases. The MSCs model both external interactions with the environment and internal interactions of the system. The tool provides a comprehensive solution but only suitable when SDL specification is available. [LC01] transforms MSCs to a global FSM so that existing techniques for FSM can be used. MSCs model both external interactions with environment and internal interactions of the system. The first algorithm is to construct a Behavior Automata (BA) from a set of bMSCs, and the second algorithm is to construct a Global Finite State Machine (GFSM) from the BA. State coverage and transition coverage are used as the test coverage criteria. The approach does not support some new features of MSC2000 such as 'par' (parallel) in-line and timing constraints. When the 'par' in-line is included, it may cause state explosion problem for GFSMs.

Since MSC testing aims to detect faults during integration. It is enough to concern only visible interaction between environment and system under test. [BBJKM03] is the first choice for this purpose. [LC01] can be the second choice when MSCs are in small scale.

4.4 State based Testing

State based testing is suitable to test detailed system behaviors. State machines are well accepted to model system behavior in most industries. Statechart is based on EFSM (Extended Finite State Machine), which is in turn based on FSM (Finite State Machine). Statechart extends the EFSM with hierarchy, concurrency, and the broadcasting communication mechanism, and EFSM in turn extends the FSM with data and guards. Hence, the techniques for testing statecharts are based on testing methods for EFSM with consideration of the extended properties, while testing EFSM is based on methods for FSM with consideration of testing data flow. Thereafter, test case generation techniques for FSM are an obvious place to start.

4.4.1 Testing Finite State Machines

In the literature, there are seven best-known methods to generate test case from finite state machine (FSM): Transition Tour (TT), Distinguishing Sequence (DS), Unique Input Output (UIO), Single UIO (SUIO), Multiple UIO (MUIO), W method, and Wp method. The general assumption of these methods requires the FSM specification is strongly connected, minimum and determinism. Except TT method, other methods use characterization sequences/sets to distinguish each state of FSM, and require the implementation can reset to the initial state correctly.

UIO has advantages over TT, DS, and W methods as it can generate shorter test cases and supported by tools. Thereafter UIO is used in many telecommunication industries. SUIO refines UIO by reducing the test case length. MUIO further refines SUIO by choosing the minimum length UIO among the multiple UIO sequences for each state. However, UIO sequence may not exist for a state, and UIO has no concern about missing/extra state in the implementation. W and Wp methods consider missing/extra-state fault of FSM, whilst Wp method refines W method by reducing the test case length. As a summary, Wp method and MUIO can be the optimum combination. MUIO can be used as the main algorithm, when UIO sequences cannot find for some states, Wp-method will be applied to these states.

4.4.2 Testing Extended Finite State Machines

EFSM extends FSM with variables and conditions, so we need to consider testing data flow as well as control flow. [UY91] [HLJ95] are data flow oriented methods. [UY91] transforms the EFSM to a data flow graph and uses IO-df-chain (input-output dataflow) criterion to generate the do-paths

(definition-observation) from the data flow graph. It does not concern the executable problem. [HLJ95] constructs a Transition Executability Analysis (TEA) tree from EFSM. Using All-do-path criterion to generate do-paths from the TEA tree. Its do-path does not include the path where the value of the input parameter is conveyed to the output parameter (e.g. input (x), temp=x, y=temp, output (y)). It solves the executable problem by reachability analysis. However, constructing the TEA tree may cause state explosion problem as it expands the TEA tree to find all the valid state configurations.

[MP92] transforms the EFSM to a FSM by increasing transitions. The FSM is transformed to a data flow graph for data flow testing. It uses All-do-path criterion. Its do-path only includes the path where the value of the input parameter is conveyed to the output parameter. The method concerns control flow testing by adding a UIO sequence for every state. To solve path executable problem, it extracts a conditional path for each do-path, and checks whether the conditional path is satisfied. However, if the conditional path is not satisfied, it changes the value of influencing value. If such variable cannot be found, it uses a backtracking technique to change the execution order of the conditional path, which is inefficient.

[CZ93] transforms the EFSM to a TDG (Transition Dependence Graph) to show control and data dependency between transitions. It proposes CCS (Cyclic characterizing sequence) to verify the states. CCS concatenates existing characterization sequence (CS) method such as UIO and the sequence that brings the machine back to the state where CS is applied. Testing the TDG graph, it uses all-du-path criterion. It solves the path executable problem by self-loop analysis, symbolic execution, and CSP (Constraint Satisfaction Problem) solving techniques. The self-loop analysis adds self-loops to change the value of the influencing variable. However, not all non-executable paths can be found self-loops to solve the constraint. It only verifies the converging states of the control flow using CCS sequence, instead of verifying all states of control flow.

[BDAR97] transforms the EFSM to a data flow graph using the method of [UY91], and it extends the IO-df-chain criterion by adding cycle analysis. The cycle analysis will analyze self-loops or non-self cycle, and it is done after generating each test case. It solves the path executable problem by cycle analysis, symbolic execution, and CLP-BNR technique. For control flow testing, it uses Wp set when MUIO sequence does not exist, but it only verifies the initial state and the end state of each do-path, without verifying the rest of states.

Comparing these methods, only [MP92] covers data flow testing and verifies all states of the control flow. [BDA97] is benefit at its optimized-IO-df-chain criterion and its MUIO with Wp. To solve the constraints in the path (i.e. path executable problem), symbolic execution, CSP, and CLP-BNR techniques are all useful state techniques. The method comparison is shown in Table 2.

	Transform EFSM to	Data flow test criterion	Control flow testing	Solution to path executable problem	Problems of the methods
[UY91]	Data Flowgraph	All IO-df-chain	No	No	No control flow testing No solution to path executability
[HLJ95]	Transition Executability Analysis (TEA) Tree	All do-path	No	Reachability Analysis	No control flow testing The solution to path executability may cause explosion
[MP92]	FSM & Data flowgrah (from FSM)	All-definition-observation	UIO for all states	Change the value of influencing variable, backtracking	The solution to path executability is inefficient, and hard to automate
[CZ93]	Transition Dependency Graph	All-du-path	CCS for initial state & converging states	Transition self-loop analysis, symbolic execution, CSP	The self-loop to change the influencing variable value may not be found. It does not verify all states of the control flow
[BDAR97]	Data Flowgraph	All Optimized-IO-df chain	MUIO& Wp for initial state & the final state of each du-path	Cycle analysis, Symbolic execution, CLP-BNR technique	It does not verify all states of the control flow

Table 2 Comparison of EFSM based testing methods

4.4.3 Testing CEFSM /CFSM

All approaches for CEFSM testing separately generate test cases from each CEFSM in isolation and generate test cases from a set of communicating transitions. In addition, all approaches assume that:

1) The CEFSMs exchanges messages through channels. A channel exists for each CEFSM and that all messages to a CEFSM go though its channel.
2) Each message identifies its sender and its receiver.

[Hie97] proposes an algorithm to compute minimum test paths to cover all communicating transitions (CTs) of CFSMs. It only considers a CT as a transition that is triggered by an internal event.

Otherwise, a transition is a non-communicating transition (NCT). It proposes a graph with two machines and a node S, where each machines consists of either CT or NCT. A circuit with the arc from a CT to NCT, the arc from NCT to S, and the arc from S to CT represents a test path. Finding shortest test paths is reduced to finding a set of circuit that the total cost is minimum, with a constraint that no circuit may visit a machine more than once. It uses a variant of *the saving algorithm* for a vehicle routing problem (VRP) to solve a set of circuits with the minimum total costs.

[BDAR98] assumes the channel is a bounded storage input FIFO channel. It only considers a CT as a transition that is triggered by an internal event. It proposes algorithms to compute a partial product for each CEFSM. It always computes a partial order for the CEFSM with smallest number of communicating transitions among CEFSMs, and evaluate whether the test cases generated from the partial product satisfy desired transition coverage. It applies the algorithm of [BDAR97] to generate test cases from EFSM, which is equivalent to a CEFSM without communicating transition. The approach is suitable for large systems as it can incrementally compute a partial product of each CEFSM.

[AM03] compute a complete product for all CFSMs to contain all communicating transitions. The complete product is denoted as a CEFSM. It explicitly categorizes communicating transitions (CTs) as transitions that are triggered by broadcast event, internal events, and state-dependent events, so that different CTs can be identified. It proposes an algorithm to identify all communicating transitions, but no algorithm is provided to compute the complete product.

[LW02] proposes a *branching coverage* criterion and to covers control flow of the CEFSM via constructing a *flow diagram* from the CEFSMs. It also proposes three concepts: *dominator, weight, and priority* to derive paths from the flow diagram that satisfy the branching coverage criterion. In a flow diagram, each node corresponds with a branch, and an arc corresponds with a link between the branches. A branch represents a full transition if it does not contain any data-related decision or represents a part of a transition if it contains a data-related decision. The branching coverage criterion includes transition coverage as well as predicate coverage. However, the approach does not concern data flow testing. It does not cover how to identify the communicating transitions. Furthermore, it requires integrating constraint-solving techniques to solve the path executable problem.

As a summary, [Hie97] is good at providing an algorithm to search minimum test paths for testing communicating transitions. [BDAR98] is benefit in its incrementally generation of a partial product for each CEFSM, and the algorithm has been implemented. Its underlying test case generation algorithm for EFSM [BDAR97] is benefit in data flow testing. The algorithm of [BDAR97] is also implemented, even thought it combines MUIO and Wp for control flow testing, it only verifies the

initial state and the final state of each du-path. [AM03] identifies all kinds of communicating transitions with its marking algorithm. [LW02] is simple and ensures transition coverage and predicate coverage for control flow testing.

4.4.4 Testing Statecharts

[HKCBU00] transforms statecharts into data flow graph. It proposes an algorithm to transform statechart into EFSM, and applies the method of [UY91] to transform EFSM to data flow graph. The approach flattens a statechart into an EFSM, where each state of EFSM corresponds to a statechart configuration, and each transition of EFSM corresponds to a step between statechart configurations. It uses all-use as the data flow coverage criterion, and uses symbolic execution to solve the path executable problem. The method suffers from state explosion problem, as it needs to list all statechart configurations. [BHL00] defines a TSL (Test Specification Language) and proposes rules to map from a statechart to its TSL test design framework. The transformation does not support statechart hierarchy or concurrency. The TSL language is based on the category partition method. Categories, partitions, and choices are the main elements of the framework. To map a statechart to a TSL test design, either states/events/call actions are represented by partitions or categories, and each transition is represented by a choice. A directed graph is built from the elements of the framework that has a root category or partition and contains all the different paths of choices. A test case is one possible path from the root to leaf of the reachability tree for the directed graph.

[HSS01] proposes a specification μSZ using statecharts to model system behaviors and using Z to model guards and operations on variables. The approach is only applicable to sequential statecharts, i.e. EFSM. To solve the path executable problem, they refine the EFSM by a data abstraction process, so that more states and transitions will be added to the EFSM and each transition is ensured to be executable. [Bur02] transforms both statechart and domain testing techniques into Z specification. After transforming statecharts into Z schemas, domain-testing techniques are applied to the Z schemas by a theorem prover tool CADiZ, so that new schemas can be derived based on domain partitioning. These new Z schemas are test cases. For testing concurrent statecharts, it applies CFSM approach, but it limits the interactions between two CFSMs to be a single writer/multiple reader relationship. The method only supports synchronous time model of statecharts. To solve the path executable problem, the approach uses CADiZ built in constraint solvers and a non-linear constraint solver Lingo.

[IH96] X- machine is a computational model. Stream x-machine is a refined x-machine that is based on FSMs, together with memory and a set of transition functions Φ that operate on the memory. W and Wp methods are applied to x-machines testing. [Bor98] applies x-machine testing method for statechart, it transform a hierarchical statechart into sub-machine, and flatten a concurrent statechart

into EFSM. The "design for test" assumption of x-machine requires that for each transition in the specification there is a corresponding transition in the implementation and very transition can be tested separately. Such structure conformance is not practical because it requires that the refinement into the implementation must always conform the structure of the specification.

[HLSC01] uses model checking approach to generate executable test cases. Given the system properties written as temporal logics and the system specification, the system properties are model-checked (i.e. mutated) and then input to the model checker. The specification is transformed into a format acceptable to the model checker and then input it. The model checker will automatically generate counterexamples by exploring the state space of the input model. Each counterexample corresponds to a failure of the specification against the property, which can be used as an executable test case. The approach uses temporal logic *CTL* to describe the test coverage criteria and use SMV as the model checker. TGV [JJ02] uses the ioco test theory to generate test cases from IOLTS (Input-Output Labeled Transition Systems) according to the user-defined test purposes. The algorithm uses DFS (Depth First Search) to explore the part of the state space of the specification that is selected by the test purpose. [PJTJJG02] proposes a UMLAUT framework to compile statecharts into IOLTS. [CCD02] proposes an AML (AGEDIS Modeling Language) as an extension to UML to model statechart, and develops a compiler to transform AML models into an IF (Intermediate Format), with existing IF2C compiler that transforms IF to IOLTS, and finally uses TGV tool to generate test cases. [TB02] TorX is a comparable tool with TGV, but it integrates automatic test generation and test execution in an on-the fly manner and it randomly selects test purposes.

As a summary, statechart testing can be divided into three categories: flow graph testing (control flow graph or data flow graph), formal specification (e.g. Z) testing, and state space exploration.

1) Flow graph based testing: It can be used either for control flow testing by traversing the control flow graph (FSM), or for data flow testing by traversing the data flowgraph. For testing control flow graph, characteristic sequences can be used to distinguish each state of FSM, assuming that the FSM is strongly connected, determined, and minimum. This assumption is not suitable for large-scale software.
2) Formal specification: Using formal specification can provide statechart with formal semantics, but using formal specification tool is expensive, also there must have assumptions when transforming a semi-formal model to a formal model.
3) State space exploration with explicit test purposes: Using state space exploration can generate comprehensive test cases by state space exploration, but it may cause test case explosion problem. The problem can be released when using explicit test purposes to narrow the search state space.

Both model checking testing and IOCO testing apply state space exploration as the underlying algorithms. Since model-checking testing is lack of proven test theory, IOCO testing is more suitable because of its formal test theory-ioco. As a result, either TGV or Torx can be choices for state based testing. The differences between TGV and TorX are summarized as:

1) Test theory: the underlying test theories are the same (i.e. *ioco*), but TorX does not consider live-locks.
2) Test purpose: TorX does not allow testers to define test purposes, but selects test purposes from the test specification randomly. TGV allows testers to define test purposes.
3) On-the-fly support: To support on-the-fly testing, TGV needs to be integrated with a test execution environment, while TorX combines test generation and test execution in an integrated manner. However, TGV can support on-the-fly testing when integrated with an external test execution engine.
4) Test Specification Language Support: Both TorX and TGV can be used for different specification languages with operational semantics in terms of LTS or IOLTS, as long as a compiler for this language is available to transform this language to IOLTS.
   - The input format to TorX is LTS (Labeled Transition System). The available compilers include CAESAR for LOTOS, and SPIN for PROMELA.
   - The input format to TGV is IOLTS. The available compilers include ObjectGeode for SDL, CAESAR for LOTOS, UMLAUT for UML, and IF2C with IF.
5) Coverage Criteria: neither TorX nor TGV integrates test coverage criteria.
6) Fault Detection Power: TorX has the same fault detecting power as TGV since they use the same conformance relation ioco.

Based on the above comparison, TGV can the selection for state based testing as it has advantages that it allows testers to define test purpose, it is the only tool that can generate test cases with loops, and it supports timer operations (e.g. start, cancel, timeout). Both UMLAUT and AGEDIS enable using TGV to generate test cases from UML statechart. A summary table is in Appendix-A showing the features of state based testing approaches.

5 Test Data Generation

This section gives a review of test data generation techniques. After the test case generation stage, the test data generation stage is required to traverse the test paths of test cases, so that testers can control how many times the test cases should be executed using different test data. To select test data manually, the equivalent input domain should be identified so that test data can be randomly selected within the equivalent partition, which is the domain partitioning process. To select test data automatically, static or dynamic approaches can be used. The test data generation/selection includes two steps:

1) Derive constraints: domain partitioning can be used together with formal specifications such as Z to generate test cases. Partitioning is applied to derive constraints to divide the input domain. However, domain partitioning has less influence on the test case generation techniques for semi-formal specifications such as UML models, because the constraints on the input domain can be directly derived from the generated test paths (i.e. test cases). In this case, *boundary value analysis* and *mutation testing* can be used to supplement the constraints derived from test paths.
2) Solve the constraints: after solving the constraints, test data can be selected from the range of values that satisfy the constraints.

5.1 Constraint Derivation

A range of partitioning techniques is based on the *Equivalence Partition Method* [GG75], which partitions the input domain as equivalence classes. Equivalence classes are sub-domains of the input domain to represent the same behavior in the specification. Thereafter, only a sample data needs to be selected from each class to verify that the system behaves correctly, given that the uniformity hypothesis assumption on the partitioning is valid. *Boundary Value Analysis* covers the problem that arises at the boundary of equivalence classes. It can be used to complement other partition methods. *Category-Partition Method* [OB88] approach is an application of equivalence class testing where the specification is first decomposed into independently testable functional units and is then further partitioned based on various properties of these units known as categories. Each category is then partitioned into equivalence classes. *Classification Tree Method* [SCE97] partitions the input domain to derive a hierarchy of abstract test cases. Leaves in the tree are combined and instantiated to form the test data. To summarize, domain partitioning is used to derive constraints so that the constraints can be used to partition the input domain.

Mutation Testing [BG81] is a kind of partitioning technique. It identifies an equivalent class in the specification that contains those values that reveal a particular mutant of the specification. Test data is generated to distinguish between the mutated and original specification. Mutation testing has been

used to assess the quality of specification based testing techniques and guide the selection of test approaches to generate more efficient test cases that also achieved a high mutation score. However, the technique to generate data to kill the mutants is left implicit.

[BCL03] proposes a method to derive constraints for a test path that has been generated from statecharts. The approach proposes an algorithm to construct an *Invocation Sequence Tree* (IST) for each test path of statecharts, and it also defines a set of rules to normalize the constraint expressions in the IST tree. The constraints are represented as OCL (Object Constraint Language) notation. The method assumes that the statechart is flattened and deterministic, and each statechart corresponds to an instance of a class. The constraints are the predicates of guards in transitions. *Invocation Sequence Tree (IST):* In order to make the OCL constraint normalization easier, an TST is constructed based on a test path from a statechart, an ITS node denotes an event/action of a transition, and an ITS edge denotes the invocation order for the events/actions. An ITS I s composed of one main tree and one or several sub-trees. A main tree corresponds with the event sequence (associated with guards) of the test path. A sub-tree corresponds to the invoked actions or events (associated with guards) of another statechart (i.e. a statechart of another object). For instance, in a statechart for an object M, t1 is the transition from state A to state B with label e1, and t2 is the transition from B to state C with label e2 [g2]/t3. e2, g2, t3 represents the event, guard, and action of the transition t2, respectively, while t3 in turn is an event of another statechart N. Then main tree of ITS is from t1 to t2 via a g2 predicate, while a sub-tree will be from t2 to t3 via a "true" predicate.

5.2 Constraint Solving Techniques

Static Methods: Symbolic Execution

The static approach uses symbolic execution to statically determine path traversal constraints that will cause the execution of the associated paths. The method works by traversing a given path and builds up symbolic expressions of the context variables in terms of the input variables without actual data involved. Because any test data satisfying the path traversal condition will cause the path to execute, solving the path traversal constraints can derive test data. Linear programming can be used to solve the constraints when the constraint predicates are linear, but further non-linear techniques are required for complex constraint expressions.

Problems of symbolic execution include unbounded loops, array indices, model calls, and constraints solving. Loops are only analyzed on their entry and exit points, using symbol cannot fix how many times a loop should run. Array elements cannot be processed when the array indices are represented as symbols. When calling model calls, it is memory intensive to execute all the sub-programs symbolically.

Dynamic Methods: Simulated Annealing & Genetic Algorithms

The dynamic approach avoids the problems in static methods because it can use the information that is only available at run time. It searches for the desired test data when the software is under test. Optimization techniques are suggested for search improvement. We introduce the ideas of two techniques for solving complex optimization problems: simulated annealing and genetic algorithms.

Simulated Annealing

Simulated annealing originated from the physical process of annealing, a process of cooling a material in a heat bath so that a minimal energy state is reached. In order to find a global optimum, the method controls the acceptance of inferior solutions by changing the value of a *temperature* parameter from high to low gradually. In the extreme, any solution will be accepted at the highest temperature, and no bad solution will be accepted at the lowest temperature. According to the decreasing temperature, a better solution is always accepted to be the current solution.

Genetic Algorithms

Genetic algorithms are adaptive search procedures based on the evolution processes of natural genetics. The method consists of three phases: selection, crossover, and mutation.

1) Firstly, there is a set of potential solutions, and a fitness score is calculated for each solution.
2) Secondly, a pair of parents is selected from the set of solutions base on calculated fitness score.
3) Thirdly, parents are combined (crossover) and then new genetic information is introduced (mutation).
4) Finally, the fitness score for the new generation is calculated and compared with the fitness score of old generation, and then the old generation will be totally or partly replaced by the new generation based on certain strategies. In test data generation, a solution will generally be a set of input parameters.

5.3 Summary

Test data is selected from the input domain of the specification to give actual data to the abstract test cases. Test data generation phase is after generating each test case or after generating all test cases. To select test data automatically, using dynamic approach to search optimized test data is a promising trend. However, the static approach that based on symbolic execution is relatively mature, as several tools are available to solve linear or non-linear constraints. Since our research will focus on the integration and refinement of test case generation techniques, we will use the more mature test data selection technique –symbolic execution approach in our test generation environment.

6 A Test Framework

6.1 Overview

There exist various techniques for generating test cases from different levels of UML specifications. Use case based testing for system-level testing has benefits in testing the overall system without causing the test case explosion problem, MSC based testing for integration-level testing helps in handling *feature interaction* problem during component integration, and state based testing for unit testing is suitable for testing individual components. It is desirable to link techniques together in a comprehensive test case generation environment, where test activities can start from the early phase of the software development lifecycle (i.e. requirement). Hence faults at different levels can be discovered by test cases generated from different specification levels. The test case generation part of our test framework has three layers, where each layer corresponds with a test specification, a test level, and a set of test case generation techniques.

Layer 1 – Use Case Based Testing

The chosen approach is [ET03]. An overview of the test case generation process shows below:

1) Elicit Use Cases.
2) Map the elements of Use Cases into the causes and effects of the Cause Effect Graphs. It's done manually currently.
3) Translate the Cause Effect Graph (CEG) into a Decision Table.
4) Extract test cases from the decision table.
5) Sequence the test cases using the defined use case matrices. It is done manually currently.

There is no formal discussion on the relationship between use cases and cause effect graphs in the literature. The approach uses the Caliber-RT tool to generate test case from CEG. Caliber-RT defines rules to map use cases to the causes and effects of CEG. However, the mapping is simple guideline involving many tradeoffs and is done manually at the moment. The approach also proposes three use case matrices to show the dependencies between use cases. These matrices can be the reference to sequence and combine the generated test cases from the Caliber-RT tool. After some experiment, we conclude that Cause Effect Graph may not be a natural candidate to provide the semantics for use cases. We intend to transform use cases into message sequence charts, and generate test cases from the message sequence charts. Thereafter, the proposed use case matrices can be used to sequence and combine the generated test cases.

Layer 2 – Message Sequence Chart Based Testing

The chosen method is [BBJKM03]. The idea is to parse the message sequence chart into a partial graph, and extracts test cases from the partial graph. The approach provides solutions to detect the feature interaction problem, and supports many advanced features of MSC such as supporting HMSC, inline, reference expressions, and time constraints. Since the sequence chart of UML2.0 adopts MSC as the semantics of the Sequence Diagram, the techniques for MSC can be directly applied to UML sequence diagrams. A non-open source tool PTK was developed for the approach. The tool is valuable for various kinds of integration testing such as for component integration inside DBE or for web service integration of a composite service. In a later stage, an alternative open source tool can be identified or developed for the DBE infrastructure.

Layer 3 – Statechart Testing

State based test techniques are appropriate for testing the behaviors of individual components or individual web services. Statecharts are well accepted to model component behaviors in most industries. Either TGV or SPIN can be our underlying state based test generation engine. UMLAUT [PJTJJG02] can be the compiler to transform UML statechart to TGV's input language IOLTS, and vUML[LP99] can provide the compiler to transform UML statechart to SPIN's input language PROMELA.

6.2 Testing Communicating Objects

6.2.1 Introduction

When a component is treated as a unit, its behaviour can be modelled as a set of communicating state machines. These state machines are test models for the state-based generation techniques. In the literature, there are several ways to test communicating state machines. The first branch of approaches is to flatten them by multiplying the involved state machines into a global state machine. The global state machine as a system configuration reachability graph can be represented as an extended finite state machine (EFSM) [HKCBU00] or a label transition system (LTS)[GLM04]. In a global state machine, each state denotes a system configuration with a set of active states and each transition denotes a change between two configurations. However, it is well known that flattening approaches are not scalable enough to handle complex system. The second branch of approaches is to test each state machine in isolation, and identify those parts with interaction so that further test cases are generated for these interactions [BDAR98] [AM03] [HIE97].

We adopt the second idea to get the benefit from separation. To concrete this idea, we propose an approach that firstly constructs a sliced machine by extracting those communication relevant

transitions, and secondly composes a pair of sliced machines for each communication, and then sequences the compositions. Regarding test models, due to the "comprehensive" nature of UML language, even in UML2, there still exist many variant points of statechart semantics. Executable UML (xUML)[MB02], a precise profile of UML, is chosen as our modelling language.

In order to model the dynamic behaviour precisely, xUML simplify the UML statechart semantics. Here are the main differences: 1) Objects communicate with each other by sending signals; 2) A state machine has no incoming queue or other storage mechanisms to save the incoming events; 3) A state machine has no hierarchy and therefore no concurrency features; 4) An object can be associated with at most one state machine; An event is "consumed" at the moment of detection. Following 3) and 4), the communications between objects can be represented by the communications between the corresponding state machines, and vice versa. Note that in xUML actions (such as sending a signal) take place on entry into a state, not on the transition. However, we have followed the convention that actions (sending events/signals) take place on the transition, to make the diagrams simpler.

6.2.2 Method

The idea of our approach is: based on each communicating event between two state machines, firstly to compute a smaller state machine for each involved state machine where containing only those transitions contributing the communicating event; secondly to compose the pair of state machines; thirdly to sequence the compositions based on the relations between communicating events.

1) Slicing State Machines: slicing disregards pieces of the state machine that are not of interest. Inspired by slicing techniques for programming languages, we defined our slicing criterion and sliced machine for a state machine.
2) Composition Algorithm: The pair of sliced machines for each CE can be composed into a compositional state machine. Since a state machine can be transformed to a state transition table straightforwardly, we make use of state transition table to compute the composition for each CE. A set of rules is proposed for composing sliced state machines. The rules are applied for asynchronous message communication. Note that in xUML, a state machine has no mechanism to store incoming messages, but a new class responsible for scheduling incoming messages can be created to achieve asynchronous communication. For example, in the case that object A sends a message m to object B but B is not on the state to receive m, B can create an object C to store and schedule the incoming messages of B. Thereafter, when B is on the state to receive message m, B can get m from C. We assume that a scheduling class is available, and how to communicate with the scheduling objects is not concerned at the moment.

6.2.3 Experiment Result and Related Works

We applied our approach to the petrol station example from iUML tutorial [iUML02]. The resulting state space of overall compositional state machines is at most 142. The state space for testing these state machines, including state space of individual state machines and the compositional state machines, has at most 163 states. When testing the state machines by simply composing the state machines into a Cartesian product, the overall state space is 700.

Our methodology is inspired by [BDAR98] [AM03] for slicing, and [SA00] for the state machine dependence graph. Both [AM03][BDAR98] aim to separate testing the in the test the transitions related to the communicating events from the individual state machines. [AM03] concerns test case generation from CFSM (Communicating Finite State Machine). The approach slices out the CE(Communicating Event, i.e. event sent across state machines) related transitions and composed all the sliced machines simultaneously. The compositional state machine is a FSM that contains the transitions related to all the communicating events. The state space of the compositional FSM is easily exposed because all the CEs are considered at the same time. However, if two CEs are independent, the transitions related to each CE need not to compose together. Our approach only concerns each CE at a time.

[BDAR98] concerns test case generation from CEFSMs (Communicating Extended Finite State Machine). The approach firstly generates test cases from individual state machines, and secondly computes a compositional state machine for each individual state machine. The composition is between each individual state machine and all test cases of other state machines related to its CEs. However, if a state machine has many transitions unrelated to its CEs, these transitions also need to be composed with the related test cases. In this case, the state space is unnecessarily increased. Our approach further eliminates the CE unrelated transitions for the composition. Also, each of their composition is between a state machine and all of its related test cases. Following state-machine-based compositions, its composing order is least-interaction-start, instead our approach is CE-based composition. We have proved that following CE-based compositions, the compositional state space is not sensitive to the composing order. The compositional state space of the CE-based compositions is at most the same as the state space of state-machine-based compositions. Furthermore, they identified all the CE related transitions from the test cases, which is a benefit. When there is dependency between CT(CEi) and CT(CEj), we cannot use separate test cases directly without manually combining the separate test cases. Alternatively, our approach takes advantage of the collaboration diagram to identify all CEs. Instead of depending on existing test case generation tools, we propose a slicing algorithm to construct sliced machines to only preserve contributing transitions with respect to a CE.

For the sliced machine composition, we adopt the idea of [SA00] to reduce state space by following the dependency graph between state machines. A state machine dependency graph is a directed graph with each node representing a state machine and each directed edge representing a dependency between two state machines. When state machine A has an outgoing edge to B, A is dependent on B. As a result, the reachability analysis for the states of a state machine only needs to check the state machines having dependencies with this state machine. Inspired by this idea, we make use of the collaboration diagram that also shows the dependencies between state machines.

6.2.4 Conclusion

We propose a methodology to test communicating state machines using dependency analysis, static slicing, and reachability analysis techniques. Firstly, a collaboration diagram needs to be designed to show the interactions between objects. In xUML, an object is associated with one state machine without hierarchy, so a collaboration diagram can show the interactions between state machines. Secondly, we proposed a slicing algorithm to compute a pair of sliced machines for each CE (communicating event). Since a CE corresponds an edge in the collaboration diagram, all the transitions contributing the CE can be identified. A sliced machine contains the shortest paths from the initial state to the exist state of the object's state machine that cover all the CE related transitions. Thirdly, we proposed a composition algorithm to compose the pair of sliced machines for each CE. Finally, we proposed rules for composing order of the compositions.

The effectiveness of our approach is dependent on the structure of an individual state machine. However, the effectiveness of our approach is independent of the topology and the composing order of state machines. From the result of the experiment on the petrol station example, our approach can reduce more than a half of the state space, comparing with the state space of the global flattened state machine. A tool is under development to implement the composition algorithm and a future work is to optimize our composition algorithm by integrating partial-reduction techniques.

7 Conclusions

This report has summarizes work on an extensive range of technologies for use in testing DBE services. We believe ii covers most if not all the currently available approaches to both the automated execution and the automated generation of test cases. This provides us with a baseline technology that can be used for both:

1. Independent testing of the DBE platform and its associated components, and
2. The development of a range of DBE testing services that could be offered to SMEs for validation of services once they are ready for deployment.

It should be noted, of course, that the test execution tools have been discussed in the context of Web Services – as was relevant for the first phase of DBE. Hence, as well as aligning our work with the DBE modelling languages now they are stabilising, we are also migrating the test execution tools to be aligned with the use of the DBE Servent in Phase 2.

We are also now ready to begin the development of adaptive techniques, that can evolve the test services in response to changes in the evolutionary landscape.

8 Appendix

8.1 Notes on DBE infrastructure Setup

Components:

- eclipse-3.0.1 (with DBE plug-ins)
- fada-5.2.6.1
- jakarta-tomcat-5.0.28 (with DBE jar library)

Files need to change environment variables:

Notes: If all components are installed in a local computer, the IP address is set to localhost. The port for Tomcat is 8080, and the port for FADA is 2002.

1) For Java, create a system variable "JAVA_HOME" to <java_installed_dir>, e.g. C:\j2sdk1.4.2_07
2) For Tomcat, create a system variable "CATALINA_HOME" to <tomcat_installed_dir>, e.g. C:\servent\ jakarta-tomcat-5.0.28
3) Change the environment variables in <eclipse_installed_dir>\BMLEditor.conf.
4) <tomcat_installed_dir>/webapps/SR/WEB_INF/web.xml
    - <param-value><tomcat_installed_dir>/webapps/SR/</param-value>
5) <tomcat_installed_dir>/webapps/SR/resources/sr.properties
    - org.dbe.kb.serverIP <- Enter the IP of the machine that will run SR service
    - org.dbe.kb.fada.addr <- Enter IP:port of a running FADA node that the SR service will register its proxies
6) <tomcat_installed_dir>/webapps/KB/WEB_INF/web.xml
    - <param-value><tomcat_installed_dir>/webapps/KB/</param-value>
7) <tomcat_installed_dir>/webapps/KB/resources/sr.properties
    - org.dbe.kb.serverIP <- Enter the IP of the machine that will run KB service
    - org.dbe.kb.fada.addr <- Enter IP:port of a running FADA node that the KB service will register its proxies,
8) Copy 'servent.xml' and 'deploy.xml' to your home directory.
    - change the environment variables in the servent.xml
    - change the <tomcat_installed_dir> in the deploy.xml
9) In the service package of the eclipse's workspace, change the environment variables in 'dbeservice.xml'. The dar file will include this xml file.

Related addresses, in the case that all components are installed in a local computer:

1) Check whether FADA is running and the deployed services:
    - http://localhost:2002
2) Deploy the Semantic Registry (SR) to FADA:
    - http://localhost:8080/SR/server
3) Deploy the Knowledge Based (KB) to FADA:
    - http://localhost:8080/KB/server
4) Manage the ActiveBPEL engine:
    - http://localhost:8080/BpelAdmin
    - The ActiveBPEL engine will run the bpr file automatically when the bpr file is copied to <tomcat_installed_dir>\bpr folder.
    - A bpr file normally contains 4 files: a pdd file, META-INT, bpel, wsdl

5) Individual web services can be found at :
    - http://localhost:8080/axis/servlet/AxisServlet
6) Composite web services can be found at :
    - http://localhost:8080/active-bpel/services

Notes:

1) The deployment needs "deploy.wsdd" with a '.dar' file.
    - The dar file of a simple service should include: dbeservice.xml, service.manifest, serviceUIImpl.jar (it's an implementation for the service).
    - The dar file of a composition service should include: dbeservice.xml, a 'bpr' file.
    - See Q & A for further explanation.
2) In order to run the bookingservice (composite service), do the following:
    - Install smtp server
    - Avoid calling accounting interceptor by removing those parts from <requestFlow> to /requestFlow> in the "deploy.wsdd" file.
    - Add "mail.jar" to "\tomcat_installed_dir\shared\lib".

Questions & Answers (Q & A):

Q1: WSDL2Java compilation: Using the HotelReservation service as the example, I could not generate the following java files from WSDL using WSDL2Java compiler, can they be generated automatically or are they hard-coded?

In the 'sun' directory: HotelAdapter.java, HotelRemoteProxy.java
In the 'ui' directory: HotelReservationUI.java, HotelResearvationUIFactory.java, HotelReservationFrame.java

---

A1: The java files have to develop by a DBE developer. The WSDL2Java compiler will only create the server and/or client side SOAP stubs and skeletons that make it easier for the developer to code the implementation of the service or a client side application. At the moment we are only interested in generating the server side skeletons. The service implementations and UI implementations will always most likely have to be developed by a developer.

Q2: When deploying the 'TextReservation' service, this service will be registered to Fada first and then de-registered from Fada, saying that the service is not deployed or is unavailable. Does the demo intend to show this?

A2: The demo did not show undeployment/deregistration of services. This is possible, but it is not fully implemented. We are currently making changes to most tools. Most of the underlining infrastructure (i.e. fada registration and service deployment) will probably be changed, but the studio tools should remain the same in the near future. We will have a newer release in June.

Q3: It seems the content of the 'dar' file for a simple service is different from the 'dar' file for a composition service. Instead of containing jar files for the java classes, when deploying a composition service to Fada, the 'dar' file only needs a '.pbr' file and a 'dbeservice.xml' file, right? However, the entry number of the composition service in the Fada service always displays "INVALID-SM-ID", don't know the reason.

A3: Yes that dars files are different because when you deploy a single service you need to provide its java implementation, but when you deploy a workflow (composition service) it is usually just a workflow (bpel) description and there may not be any java implementation. The composition service dar consists of a .bpr file, which contains a few xml files… a bpel, a pdd, some wsdl files….. which are used to deployed the composition service with the workflow engine. Yes you should be getting an

invalid smid for composed services. The publishing of Service Manifests for composed services was not fully implemented. It will be done in the next release.

8.2 Jython Test Script for HotelReservation

```
# Agent name: TestHotelReservationService.py
# Created on: 10th May 2005
# Author: Yongyan

print "Agent is running: "

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import ProtocolHandler, SOAPHeader,
SOAPBody, SOAPProtocol
from com.pushtotest.tool.response import Response

# This agent also uses JDOM APIs to handle XML data
from org.jdom import Document, Element, JDOMException, Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter
from org.jdom.input import SAXBuilder
from java.io import StringReader

# Variables definitions
checkInDate = "01/07/05"
checkOutDate = "05/07/05"
roomType = "2"
numOfAdults = "1"
numOfChild = "0"

# Helper function to build a request XML document
def addElement( top, name, content, ns ):
    newelement = Element( name, ns )
    newelement.addContent( content )
    top.addContent( newelement )

# Main body of agent
print "Processing..."

# HotelReservation SOAP Script
# HotelReservationService:
urn:soapimpl.hotelreservation.demos.dbe.org:HotelReservationServiceSoapBinding
soap = ProtocolHandler.getProtocol("soap")
soap.setUrl("http://gaudi.techideas.info:8080/axis/services/HotelReservationService
")
body = ProtocolHandler.getBody("soap")
xmlns1 = "urn:soapimpl.hotelreservation.demos.dbe.org"
#-------------------------------------------------------------------------------
-------------

# simpleAvailabilityCheckingRequest
print
print "Now sending simpleAvailabilityCheckingRequest..."
# Build the request XML
elOne = Element( "simpleAvailabilityChecking", xmlns1 )
addElement( elOne, "checkInDate", checkInDate, xmlns1 )
addElement( elOne, "checkOutDate", checkOutDate, xmlns1 )
addElement( elOne, "roomType", roomType, xmlns1 )
addElement( elOne, "numOfAdults", numOfAdults, xmlns1 )
addElement( elOne, "numOfChild", numOfChild, xmlns1 )
doc=Document(elOne)
body.setDocument( doc )

soap.setBody(body)
response = soap.connect()
```

```
#print "response for simpleAvailabilityChecking: ",response

# Use the JDOM SAXBuilder object to create a new DOM object that
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
envelopeElement = doc.getRootElement()
soap_ns = Namespace.getNamespace("soap",
"http://schemas.xmlsoap.org/soap/envelope/")
bodyElement = envelopeElement.getChild("Body", soap_ns)
so_ns = Namespace.getNamespace( "", "urn:soapimpl.hotelreservation.demos.dbe.org")
respElement = bodyElement.getChild( "simpleAvailabilityCheckingResponse", so_ns )
elm = respElement.getContent(0)
if (elm.getAttribute("href") == None):
    available = elm.getValue()
else:
    hrefValue = elm.getAttributeValue("href")
    multiRef = bodyElement.getChild("multiRef")
    #if ("#" + multiRef.getAttribute("id").getValue()) == hrefValue:
    available = multiRef.getValue()
print "simpleChecking available =", available
#-------------------------------------------------------------------------------
-------------

# advancedAvailabilityChecking
print
print "Now sending advancedAvailabilityChecking request..."
# Build the request XML
elOne = Element( "advancedAvailabilityChecking", xmlns1 )
addElement( elOne, "numOfChild", numOfChild, xmlns1 )
addElement( elOne, "numOfAdults", numOfAdults, xmlns1 )
addElement( elOne, "roomType", roomType, xmlns1 )
addElement( elOne, "checkOutDate", checkOutDate, xmlns1 )
addElement( elOne, "checkInDate", checkInDate, xmlns1 )
doc=Document(elOne)
body.setDocument( doc )

soap.setBody(body)
response = soap.connect()
#print "response for advancedAvailabilityChecking: ",response

# Use the JDOM SAXBuilder object to create a new DOM object that
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
envelopeElement = doc.getRootElement()
soap_ns = Namespace.getNamespace("soap",
"http://schemas.xmlsoap.org/soap/envelope/")
bodyElement = envelopeElement.getChild("Body", soap_ns)
so_ns = Namespace.getNamespace( "", "urn:soapimpl.hotelreservation.demos.dbe.org")
respElement = bodyElement.getChild( "advancedAvailabilityCheckingResponse", so_ns )
# get and print the value of each response variable
for i in range(0,4):
    elm = respElement.getContent(i)
    if elm.getAttribute("href") == None:
        print elm.getName(),"=", elm.getValue()
    else:
        hrefValue = elm.getAttributeValue("href")
        multiRefs = bodyElement.getChildren("multiRef")
        for j in range(0, multiRefs.size()):
            multiRef = multiRefs.get(j)
            if ("#" + multiRef.getAttribute("id").getValue()) == hrefValue:
                print elm.getName(),"=", multiRef.getValue()
#-------------------------------------------------------------------------------
-------------

# makeReservationRequest
print
print "Now sending makeReservationRequest..."
# Build the request XML
```

```
elOne = Element( "makeReservation", xmlns1 )
addElement( elOne, "checkInDate", checkInDate, xmlns1 )
addElement( elOne, "checkOutDate", checkOutDate, xmlns1 )
addElement( elOne, "roomType", roomType, xmlns1 )
addElement( elOne, "numOfAdults", numOfAdults, xmlns1 )
addElement( elOne, "numOfChild", numOfChild, xmlns1 )
doc=Document(elOne)
body.setDocument( doc )

soap.setBody(body)
response = soap.connect()
#print "response for makeReservation :",response

# Use the JDOM SAXBuilder object to create a new DOM object that
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
envelopeElement = doc.getRootElement()
soap_ns = Namespace.getNamespace("soap",
"http://schemas.xmlsoap.org/soap/envelope/")
bodyElement = envelopeElement.getChild("Body", soap_ns)
so_ns = Namespace.getNamespace( "", "urn:soapimpl.hotelreservation.demos.dbe.org")
respElement = bodyElement.getChild( "makeReservationResponse", so_ns )
# get and print the value of each response variable
failureReason = respElement.getContent(0)
print "failureReason = ", failureReason.getValue()
reservationNumber = respElement.getContent(1)
print "reservationNumber = ", reservationNumber.getValue()
elm = respElement.getContent(2)
if (elm.getAttribute("href") == None):
    available = elm.getValue()
else:
    hrefValue = elm.getAttributeValue("href")
    multiRef = bodyElement.getChild("multiRef")
    makerReservationSuccessful = multiRef.getValue()
print "makerReservation successful =", makerReservationSuccessful

#-------------------------------------------------------------------------------
-------------
# cancelReservationRequest
print
print "Now sending canccelReservationRequest..."
if (makerReservationSuccessful != "true"):
    print "Cannot cancelReservation, the reservationNumber is not valid"
else:
    #--build xml request
    elOne = Element( "cancelReservation", xmlns1 )
    addElement( elOne, "reservationNumber", reservationNumber.getValue(), xmlns1 )
    doc=Document(elOne)
    body.setDocument( doc )

    soap.setBody(body)
    response = soap.connect()
    #print "response for cancelReservation:",response

    builder = SAXBuilder()
    doc = builder.build( StringReader( response.toString() ) )
    envelopeElement = doc.getRootElement()
    soap_ns = Namespace.getNamespace("soap",
"http://schemas.xmlsoap.org/soap/envelope/")
    bodyElement = envelopeElement.getChild("Body", soap_ns)
    so_ns = Namespace.getNamespace( "",
"urn:soapimpl.hotelreservation.demos.dbe.org")
    respElement = bodyElement.getChild( "cancelReservationResponse", so_ns )
    elm = respElement.getContent(0)
    if (elm.getAttribute("href") == None):
        available = elm.getValue()
    else:
        hrefValue = elm.getAttributeValue("href")
```

```
        multiRef = bodyElement.getChild("multiRef")
        cancelSuccessful = multiRef.getValue()
    if (cancelSuccessful):
        print "The reservationNumber: ",reservationNumber.getValue()," has been
cancelled"
    else:
        print "The reservationNumber: ",reservationNumber.getValue(), " cannot been
cancelled successfully"

#-------------------------------------------------------------------------------
-------------
# Report results
print
print "Agent ended."
```

8.3 Java Test Script for HotelReservation

```
/**
 * mytestCase.java is a full version of HotelReservationTestCase.java
 * Author:Yongyan
 */
package org.dbe.demos.hotelreservation.soapimpl;
import java.util.*;
import java.math.*;
import javax.xml.rpc.holders.*;

public class myTestCase extends junit.framework.TestCase {
    public myTestCase(java.lang.String name) {
        super(name);
    }

    public Boolean test1HotelReservationServiceCancelReservation(String
reservationNumber) throws Exception {

org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub
binding;
        try {
            binding =
(org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub)
                          new
org.dbe.demos.hotelreservation.soapimpl.HotelReservationLocator().getHotelReservati
onService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new junit.framework.AssertionFailedError("JAX-RPC
ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);

        // Test operation
        /* --my comments: This is the original generated test operation
        binding.cancelReservation(new java.lang.String(),
                    new javax.xml.rpc.holders.BooleanHolder());
        */
        BooleanHolder successful = new javax.xml.rpc.holders.BooleanHolder();
        binding.cancelReservation(reservationNumber, successful);

        //      TBD - validate results

        return new Boolean(successful.value);
```

```
    }


    public HashMap test2HotelReservationServiceMakeReservation(String
CheckInDate,String CheckOutDate,
                                    String RoomType, BigInteger NumOfAdults,BigInteger
NumOfChild) throws Exception {

org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub
binding;
        try {
            binding =
(org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub)
                            new
org.dbe.demos.hotelreservation.soapimpl.HotelReservationLocator().getHotelReservati
onService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new junit.framework.AssertionFailedError("JAX-RPC
ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);

        // Test operation
        /* --my comments: This is the original generated test operation
         * binding.makeReservation(new java.lang.String(), new java.lang.String(),
                    new java.lang.String(), new java.math.BigInteger("0"),new
java.math.BigInteger("0"),
                             new javax.xml.rpc.holders.StringHolder(), new
javax.xml.rpc.holders.StringHolder(),
                             new javax.xml.rpc.holders.BooleanHolder());
             */
        StringHolder failureReason = new javax.xml.rpc.holders.StringHolder();
        StringHolder reservationNumber = new javax.xml.rpc.holders.StringHolder();
        BooleanHolder successful = new javax.xml.rpc.holders.BooleanHolder();
        binding.makeReservation(CheckInDate, CheckOutDate, RoomType,
                                NumOfAdults, NumOfChild, failureReason,
                                reservationNumber, successful);

        //      TBD - validate results
        HashMap result = new HashMap();
        result.put("failureReason", failureReason.value);
        result.put("reservationNumber", reservationNumber.value);
        result.put("successful", new Boolean(successful.value));

        return result;
    }

    public HashMap
test3HotelReservationServiceAdvancedAvailabilityChecking(BigInteger
numOfChild,BigInteger numOfAdults,
                                    String roomType,String checkOutDate,String
checkInDate) throws Exception {

org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub
binding;
        try {
            binding =
(org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub)
                            new
org.dbe.demos.hotelreservation.soapimpl.HotelReservationLocator().getHotelReservati
onService();
        }
```

```
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new junit.framework.AssertionFailedError("JAX-RPC
ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);

        // Test operation
        /* --my comments: This is the original generated test operation
        binding.advancedAvailabilityChecking(new java.math.BigInteger("0"), new
java.math.BigInteger("0"),
                        new java.lang.String(), new java.lang.String(), new
java.lang.String(),
                        new javax.xml.rpc.holders.BigDecimalHolder(), new
javax.xml.rpc.holders.StringHolder(),
                        new javax.xml.rpc.holders.StringHolder(), new
javax.xml.rpc.holders.BooleanHolder());
        */
        BigDecimalHolder dailyprice = new javax.xml.rpc.holders.BigDecimalHolder();
        StringHolder newRoomType = new javax.xml.rpc.holders.StringHolder();
        StringHolder unavailabilityReason = new
javax.xml.rpc.holders.StringHolder();
        BooleanHolder available = new javax.xml.rpc.holders.BooleanHolder();


binding.advancedAvailabilityChecking(numOfChild,numOfAdults,roomType,checkOutDate,c
heckInDate,
                        dailyprice, newRoomType, unavailabilityReason, available);

        // TBD - validate results
        HashMap result = new HashMap();
        result.put("dailyprice", dailyprice.value);
        result.put("newRoomType", newRoomType.value);
        result.put("unavailabilityReason", unavailabilityReason.value);
        result.put("available", new Boolean(available.value));

        return result;
    }

    public Boolean test4HotelReservationServiceSimpleAvailabilityChecking(String
checkInDate,String checkOutDate,
             String roomType,BigInteger numOfAdults,BigInteger numOfChild) throws
Exception {

org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub
binding;
        try {
            binding =
(org.dbe.demos.hotelreservation.soapimpl.HotelReservationServiceSoapBindingStub)
                          new
org.dbe.demos.hotelreservation.soapimpl.HotelReservationLocator().getHotelReservati
onService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new junit.framework.AssertionFailedError("JAX-RPC
ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);
```

```
        // Test operation
        /*binding.simpleAvailabilityChecking(new java.lang.String(), new
java.lang.String(),
                    new java.lang.String(), new java.math.BigInteger("0"),
                          new java.math.BigInteger("0"), new
javax.xml.rpc.holders.BooleanHolder());
        */
        BooleanHolder available = new javax.xml.rpc.holders.BooleanHolder();

binding.simpleAvailabilityChecking(checkInDate,checkOutDate,roomType,numOfAdults,nu
mOfChild,available);

        // TBD - validate results
        return new Boolean(available.value);
    }

    public static void main(String[] args) {
        myTestCase mytest = new myTestCase("abc");
        String CheckInDate = "01/07/05";
        String CheckOutDate = "05/07/05";
        String roomType = "2";
        BigInteger numOfAdults = new BigInteger("1");
        BigInteger numOfChild = new BigInteger("0");

        System.out.println("Now sending simpleAvailabilityChecking request...");
        try{
             Boolean
sim_response=mytest.test4HotelReservationServiceSimpleAvailabilityChecking(
                          CheckInDate,CheckOutDate,roomType, numOfAdults,
numOfChild);
             if(sim_response.booleanValue()) System.out.println("The room is
available");
             else System.out.println("The room is not available");
       }catch(Exception e){
             e.printStackTrace();
             }

       System.out.println("Now sending advancedAvailabilityChecking request...");
        try{
             HashMap ad_response =
mytest.test3HotelReservationServiceAdvancedAvailabilityChecking(
                                  numOfChild,
numOfAdults,roomType,CheckOutDate,CheckInDate);

             BigDecimal dailyprice = (BigDecimal) ad_response.get("dailyprice");
             String newRoomType = (String) ad_response.get("newRoomType");
             String unavailabilityReason = (String)
ad_response.get("unavailabilityReason");
             boolean available = ((Boolean)
ad_response.get("available")).booleanValue();

             System.out.println("dailyprice = " + dailyprice.floatValue());
            System.out.println("newRoomType = " + newRoomType);
            System.out.println("unavailabilityReason = " + unavailabilityReason);
            System.out.println("available = " + available);
        }catch(Exception e){
             e.printStackTrace();
        }

        String failureReason, reservationNumber = "";
        boolean successful = false;

        System.out.println("Now sending makeReservation request...");
        try {
            HashMap mr_response =
mytest.test2HotelReservationServiceMakeReservation(
                          CheckInDate,CheckOutDate,roomType, numOfAdults,
numOfChild);
```

```
            failureReason = (String) mr_response.get("failureReason");
            reservationNumber = (String) mr_response.get("reservationNumber");
            successful = ((Boolean) mr_response.get("successful")).booleanValue();

            System.out.println("failureReason = "+failureReason);
            System.out.println("reservationNumber = "+reservationNumber);
            System.out.println("successful = "+successful);
        } catch (Exception e) {
             e.printStackTrace();
        }

        System.out.println("Now sending cancelReservation request...");
        try{
             if (successful){
                   Boolean
cr_response=mytest.test1HotelReservationServiceCancelReservation(reservationNumber)
;
                   if(cr_response.booleanValue()) {
                         System.out.println("The reservationNumber:"+
                                     reservationNumber+ "has been cancelled
successfully");
                   }else {
                         System.out.println("The reservationNumber:"+
                                     reservationNumber+ "cannot been cancelled
successfully");
                   }
             }else{
                   System.out.println("Cannot cancelReservation, the
reservationNumber is not valid");
             }
        }catch(Exception e){
             e.printStackTrace();
        }


    }

}
```

8.4 Jython Test Script for BookingService

```
# Agent name: TestBookingService.py
# Created on: 15th May 2005
# Author: Yongyan

print "Agent is running: "

# Import tells TestMaker where to find Tool objects
from com.pushtotest.tool.protocolhandler import ProtocolHandler, SOAPHeader,
SOAPBody, SOAPProtocol
from com.pushtotest.tool.response import Response

# This agent also uses JDOM APIs to handle XML data
from org.jdom import Document, Element, JDOMException, Namespace, DocType
from org.jdom.output import DOMOutputter, XMLOutputter
from org.jdom.input import SAXBuilder
from java.io import StringReader

# Variables definitions
hotelCheckInDate = "15/07/05"
hotelCheckOutDate = "20/07/05"
hotelRoomType = "1"
hotelNumOfAdults = "1"
hotelNumOfChild = "0"
```

```
receiptType = "email"
contact = "summerzyy@localhost"
taxiArrivalDateTime = "15/12/2005"
taxiDepartureDateTime = "15/12/2005"
originAddress = "Airport"
destinationAddress = "Hotel_Barcelona"
emailSubject = "Automated Booking ServiceL Your Reference Numbers!"


# Helper function to build a request XML document
def addElement( top, name, content, ns ):
    newelement = Element( name, ns )
    newelement.addContent( content )
    top.addContent( newelement )

# Main body of agent
print "Processing..."

soap = ProtocolHandler.getProtocol("soap")
soap.setUrl("http://localhost:8080/active-bpel/services/BookingService")
body = ProtocolHandler.getBody("soap")
xmlns1 = "urn:bookingservice.dbe.org"

print
print "Now sending requests to bookingservice request..."
# Build the request XML
elOne = Element( "makeMultiBooking", xmlns1 )
addElement( elOne, "hotelCheckInDate",hotelCheckInDate, xmlns1 )
addElement( elOne, "hotelCheckOutDate", hotelCheckOutDate, xmlns1 )
addElement( elOne, "hotelRoomType", hotelRoomType, xmlns1 )
addElement( elOne, "hotelNumOfAdults", hotelNumOfAdults, xmlns1 )
addElement( elOne, "hotelNumOfChild", hotelNumOfChild, xmlns1 )
addElement( elOne, "receiptType", receiptType, xmlns1 )
addElement( elOne, "contact", contact, xmlns1 )
addElement( elOne, "taxiArrivalDateTime", taxiArrivalDateTime, xmlns1 )
addElement( elOne, "taxiDepartureDateTime", taxiDepartureDateTime, xmlns1 )
addElement( elOne, "originAddress", originAddress, xmlns1 )
addElement( elOne, "destinationAddress", destinationAddress, xmlns1 )
addElement( elOne, "emailSubject", emailSubject, xmlns1 )
doc=Document(elOne)
body.setDocument( doc )
#print "Here is the XML request document:"
#xo = XMLOutputter()
#print xo.outputString(doc)

soap.setBody(body)
response = soap.connect()
#print "Here is the xml response for makeMultiBooking: ",
#print response

# Use the JDOM SAXBuilder object to create a new DOM object that
builder = SAXBuilder()
doc = builder.build( StringReader( response.toString() ) )
envelopeElement = doc.getRootElement()
soap_ns = Namespace.getNamespace("soap",
"http://schemas.xmlsoap.org/soap/envelope/")
bodyElement = envelopeElement.getChild("Body", soap_ns)
so_ns = Namespace.getNamespace( "", "urn:bookingservice.dbe.org")
respElement = bodyElement.getChild( "makeMultiBookingResponse", so_ns )

elms = respElement.getChildren()
for i in range(0,elms.size()):
    elm = elms.get(i)
    if elm.getAttribute("href") == None:
        print elm.getName(),"=", elm.getValue()
    else:
        hrefValue = elm.getAttributeValue("href")
        multiRefs = bodyElement.getChildren("multiRef")
```

```
        for j in range(0, multiRefs.size()):
            multiRef = multiRefs.get(j)
            if ("#" + multiRef.getAttribute("id").getValue()) == hrefValue:
                print elm.getName(),"=", multiRef.getValue()


# Report results
print
print "Agent ended."
```

8.5 Java Test Script for BookingService

```
/**
 * BookingServiceServiceTestCase.java
 * This file was partially auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 * Author: Yongyan
 * Date: 15 May 2005
 */

package org.dbe.bookingservice;

import java.util.HashMap;
import java.math.*;
import javax.xml.rpc.holders.*;

public class BookingServiceServiceTestCase extends junit.framework.TestCase {
    public BookingServiceServiceTestCase(java.lang.String name) {
        super(name);
    }
    public HashMap test1BookingServiceMakeMultiBooking(
             String hotelCheckInDate, String hotelCheckOutDate, String
hotelRoomType,
                    BigInteger hotelNumOfAdults, BigInteger hotelNumOfChild, String
receiptType,
                    String contact, String taxiArrivalDateTime, String
taxiDepartureDateTime,
                    String originAddress, String destinationAddress, String
emailSubject) throws Exception {
        org.dbe.bookingservice.BookingServiceSoapBindingStub binding;
        try {
            binding = (org.dbe.bookingservice.BookingServiceSoapBindingStub)
                        new
org.dbe.bookingservice.BookingServiceServiceLocator().getBookingService();
        }
        catch (javax.xml.rpc.ServiceException jre) {
            if(jre.getLinkedCause()!=null)
                jre.getLinkedCause().printStackTrace();
            throw new junit.framework.AssertionFailedError("JAX-RPC
ServiceException caught: " + jre);
        }
        assertNotNull("binding is null", binding);

        // Time out after a minute
        binding.setTimeout(60000);

        // Test operation

        BooleanHolder confirmation = new javax.xml.rpc.holders.BooleanHolder();
        StringHolder hotelReservationNumber = new
javax.xml.rpc.holders.StringHolder();
        StringHolder taxiReservationNumber = new
javax.xml.rpc.holders.StringHolder();
        StringHolder failureReason = new javax.xml.rpc.holders.StringHolder();
```

```
        binding.makeMultiBooking(
                    hotelCheckInDate, hotelCheckOutDate, hotelRoomType,
                           hotelNumOfAdults, hotelNumOfChild, receiptType,
                           contact, taxiArrivalDateTime, taxiDepartureDateTime,
                           originAddress, destinationAddress, emailSubject,
                           confirmation, hotelReservationNumber,
                       taxiReservationNumber, failureReason);

        // TBD - validate results
        HashMap result = new HashMap();
        result.put("confirmation", new Boolean(confirmation.value));
        result.put("hotelReservationNumber", hotelReservationNumber.value);
        result.put("taxiReservationNumber", taxiReservationNumber.value);
        result.put("failureReason", failureReason.value);

        return result;

    }

    public static void main(String[] args) {
       String hotelCheckInDate = "15/07/05";
       String hotelCheckOutDate = "20/07/05";
       String hotelRoomType = "1";
              BigInteger hotelNumOfAdults = new BigInteger("1");
              BigInteger hotelNumOfChild = new BigInteger("1");
              String receiptType = "email";
              String contact = "summerzyy@localhost";
              String taxiArrivalDateTime = "15/12/2005";
              String taxiDepartureDateTime = "15/12/2005";
              String originAddress = "Airport";
              String destinationAddress = "Hotel&nbspBarcelona"; //  means space
              String emailSubject = "Automated Booking ServiceL Your Reference
Numbers!";

       BookingServiceServiceTestCase mytest = new
BookingServiceServiceTestCase("abc");
              System.out.println("Now sending requests to bookingservice
request...");
        try{

              HashMap ad_response = mytest.test1BookingServiceMakeMultiBooking(
                           hotelCheckInDate, hotelCheckOutDate, hotelRoomType,
                                  hotelNumOfAdults, hotelNumOfChild, receiptType,
                                  contact, taxiArrivalDateTime,
taxiDepartureDateTime,
                                  originAddress, destinationAddress, emailSubject);

              boolean confirmation = ((Boolean)
ad_response.get("confirmation")).booleanValue();
              String hotelReservationNumber = (String)
ad_response.get("hotelReservationNumber");
              String taxiReservationNumber = (String)
ad_response.get("taxiReservationNumber");
              String failureReason = (String)ad_response.get("failureReason");

              System.out.println("confirmation = " + confirmation);
              System.out.println("hotelReservationNumber = " +
hotelReservationNumber);
              System.out.println("taxiReservationNumber = " +
taxiReservationNumber);
              System.out.println("failureReason = " + failureReason);
        }catch(Exception e){
              e.printStackTrace();
              }

          }
```

```
}
```

9 References

[AM03] Ambrosio, A.M, *Systematic Test Case Generation for Concurrent FSMs*, International Conference on Dependable Systems and Networks, 2003

[BBJKM03] Paul Baker, Paul Bristow, Clive Jervis, David King, Bill Mitchell, *Automatic Generation of Conformance Tests from Message Sequence Charts*, Telecommunications and Beyond: The Broader Applicability of SDL and MSC, LNCS 2599, pp 170-198, 2003

[BDAR97] C. Bourhfir, R. Dssouli, E. Aboulhamid, N. Rico, *Automatic executable test case generation for extended finite state machine protocol* , 1997

[BDAR98] Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico, *A guided incremental test case generation procedure for conformance testing for CEFSM specified protocols*. In IWTCS'98, Tomsk, Russia, Aug. 1998

[BG81] Ajei Gopal and tim Budd, *Program testing by specification mutation*, Technical Report TR 83-17, University of Arizona, November 1983.

[BHL00] Manfred Broy, Jean Hartmann, Heiko Lotzbeyer, *Automatic Test Case Generation from UML statecharts*, 2000.

[CCD02] Charles Crichton, Alessandra Cavarra, and Jim Davies, *Using UML for Automatic Test Generation*, 2002

[CZ93] S. Chanson and J. Zhu. *A Unified Approach to Protocol Test Sequence Generation,* Proc. of IEEE INFOCOM, 1993

[ET03] Elena Pérez-Miñana, Tim Trew, *Requirements Structuring and Analysis for Testing Concurrent Systems*, 2003

[GG75] John B. Goodenough, Susan L. Gerhart, *Towards a theory of test data selection*, IEEE Trans. Volume 1, Number 2, June 1975, pp156-173.

[GLM04] Stefania Gnesi, Diego Latella and Mieke Massink, *Formal Test-case Generation for UML Statecharts*, ICECCS04, Florence, Italy, April, 2004

[HIE97] R. M. Hierons. *Testing from semi-independent communicating finite state machines,* IEE Proceedings In Software Engineering, 144(5-6):291-295, 1997

[HKCBU00] H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae, and H. Ural, *A Test Sequence Selection Method for Statecharts,* Journal of Software Testing, Verification, and Reliability, Vol. 10, No. 4, pp. 203-227, Dec. 2000.

[HLJ95] Chung-Ming Huang, Yuan-Chuen Lin, and Ming-Yuhe Jang, *An Executable Protocol Test Sequence Generation Method for EFSM-specified Protocols,* IFIP Transactions C: Communication Systems - Protocol Test Systems, pp. 29-44, 1995

[HLSC01] H.S. Hong, I. Lee, O. Sokolsky, and S.D. Cha, *Automatic Test Generation from Statecharts Using Model Checking*, Proceedings of the First Workshop on Formal Approaches to Testing of Software (FATES '01), pp. 15-30, Aug. 2001.

[HSS01] R. M. Hierons, S. Sadeghipour, and H. Singh*, Testing a System specified using Statecharts and Z,* Information and Software Technology, 43 2, pp. 137-149, 2001.

[IH96] F. Ipate and M. Holcombe. *An integration testing method that is proved to find all faults*. International Journal of Computer Mathematics, 63:159–178, 1996

[iUML02] www.kc.com*, iUML Tutorial, 2002*

[JJ02] C. Jard, T. Jéron, *TGV: theory, principles and algorithms*, in The Sixth World Conference on Integrated Design & Process Technology (IDPT'02), Pasadena, California, USA, June 2002.

[KC99] Youngchul Kim, C. Robert Carlson *Scenario Based Integration Testing for Object-Oriented Software Development*, Eighth Asian Test Symposium**,** November 16 - 18, 1999 Shanghai, China.

[LBG99] D. Amyot and L. Logrippo, R. J. A. Buhr, and T. Gray, *Use Case Maps for the Capture and Validation of Distributed Systems Requirements*, Proceedings of Fourth International Symposium on Requirements Engineering (RE'99), Limerick, Ireland, June 1999

[LC01] Lee Nam Hee; Cha Sung Deok, *Generating Test Sequences from a set of MSCs,* 2001

[LP99] *Johan Lilius, Ivan Porres Paltor***,** vUML: A Tool for Verifying UML Models, 14th IEEE International Conference on Automated Software Engineering, 1999

[LW02] J. Jenny Li, W. Eric Wong*, Automatic Test Generation from Communicating Extended Finite State Machine (CEFSM)-Based Models***,** Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing**,** April 29 - May 01, 2002

[MB02] Stephen J. Mellor and Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison- Wesley, 2002

[MP92] Raymond E. Miller and Sanjoy Paul, *Generating Conformance Test Sequences for Combined Control and Data flow of Communication Protocols*, Proc. 12th. IFIP International Symposium on Protocol Specification, Testing and Verification, 1992

[OB88] Thomas J. Ostrand and Marc J.Balcer, *The category-partition method for specifying and generating functional tests*, Communications of the ACM, 31(6):676 -- 686, June 1988.

[PJTJJG02] Simon Pickin, Claude Jard, Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, Alain Le Guennec,, *System Test Synthesis from UML models of Distributed Software*, FORTE, 2002

[RG99] Johannes Ryser, Martin Glinz, *A Scenario Based Approach to Validating and Testing Software Systems Using Statecharts*, Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications ICSSEA'99. CNAM, Paris, France.

[SA00]Jorgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jorn Lind-Nielsen, Kim G. Larsen, Gerd Behrmann, Kare Kristoffersen, Arne Skou, Henrik Leerberg, Niels Bo Theilgaard, *Practical Verification of Embedded Software*, *2000 IEEE Vol. 33No. 5*

[SCE97] Harbhajan Singh, Mirko Conrad, Gottfried Egger, and Sadegh Sadeghipour. *Test case design based on Z and the classification-tree method*. First IEEE International Conference on Formal Engineering Methods, November 1997.

[SEGHK96] Michael Schmitt, Beat Koch, Jens Grabowski, Dieter Hogrefe, *Autolink-Putting SDL based test generation into practice*, In A. Petrenko and N. Yevtuschenko, editors, Testing of Communicating Systems, volume 11. Kluwer, 1998.

[TB02] Jan Tretmans, Ed Brinksma, *TORX: Automated Model Based Testing*, 2002

[TYLSX03] W. T. Tsai, L. Yu, X. X. Liu, A. Saimi, Y. Xiao, *Scenario-Based Test Case Generation for State-Based Embedded Systems*, 2001

[UY91] H. Ural and B. Yang. *A test sequence selection method for protocol testing,* IEEE Trans. Commun., 39(4), 1991

[WP99] C. Williams and A. Paradkar, *Efficient Regression Testing of Multi-Panel Systems*, Tenth International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton, FL (November 1999).