uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Pierre Séguin
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

An Integration Test Framework for Service-Oriented Architecture

TITRE DE LA THÈSE / TITLE OF THESIS

Liam Peyton
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Daniel Amyot

Michael Weiss

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

An Integration Test Framework for Service-Oriented Architecture

Pierre Séguin

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the M. Sc. Degree in Computer Science

University of Ottawa

Ottawa, Ontario, Canada

November 17, 2008

Your file Votre référence
ISBN: 978-0-494-48626-9
Our file Notre référence
ISBN: 978-0-494-48626-9

Abstract

Increasingly organizations are leveraging service oriented architectures (SOA) as the core infrastructure on which their business applications and processes are organized.

The testing and debugging of applications within a SOA presents special challenges. A defect observed at the level of user interaction with the application could be

- a fault (performance, security, scalability, etc.) in the application or process logic
- a fault in any of the services used by the application
- an unintended interaction in combining services

The key challenge to be faced is how to ensure that organizations can manage the complexity of their composite applications and service-oriented architecture in a manner that addresses the engineering requirements of testing while providing tools that are efficient, productive, and reasonable for testers to learn.

In our thesis we address these issues by introducing a 'grey box' test agent architecture and a model-driven approach to express 'grey box' test campaigns.

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Liam Peyton, without whom this thesis would have been impossible to complete. His breadth of knowledge, foresight, and support has been invaluable.

Special thanks are dedicated to my colleague, Bernard Stepien, who has been a great source of insight and ideas. His breadth of knowledge in TTCN-3 has been an invaluable contribution to this thesis.

I would also like to thank my family. It is their love and unconditional support through all of my undertakings that have made me who I am. This thesis would not have been possible without your constant encouragement.

Finally, I would like to thank the Ontario Graduate Scholarship Program (OGS,) the Natural Sciences and Engineering Research Council of Canada (NSERC) and SpotOn Systems Inc. for supporting this research.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Problem Statement

Increasingly organizations are leveraging service oriented architectures as the core infrastructure on which all their business applications and processes are organized. A service-oriented architecture enables composite applications that support business processes to be defined and built dynamically from a loosely coupled and interoperable set of distributed web services. It also ensures that core business services such as security, order processing and inventory management, are provided a single overall architecture so that individual applications do not re-invent core infrastructure and business logic.

However, in such an architecture, the testing and debugging of composite applications presents special challenges. A defect observed during a test could be

- a fault (performance, security, scalability, etc.) in the application or process logic;
- a fault in any of the services used by the application;
- an unintended interaction in combining services.

The situation is further complicated by the distributed nature of the service oriented architecture and the large volumes of user interactions that must be handled simultaneously. As well, individual services may be replaced or updated independently of the application.

The key challenge to be faced is how to ensure that organizations can manage the complexity of their composite applications and service-oriented architecture in a manner that addresses the engineering requirements of testing while providing tools that are efficient, productive, and reasonable for testers to learn.

1.2. Thesis Motivation and Contributions

Currently, there is a lack of test environments in industry geared towards testing the choreography and composition of scenarios in a service-oriented architecture. The creation of such an environment would improve the quality of service-oriented architectures by allowing testers to more easily manage the testing of such architectures and more efficiently identify and locate faults within them.

The objective of this thesis is to enhance composite application development with additional techniques for creating and managing tests, focusing on integration issues. Our framework should support testing the orchestration and composition of services in composite applications in an intuitive fashion. We want a declarative approach for systematic quality assurance of enterprise applications and SOA. The contributions of this thesis are:

1. Grey-box test agent architecture for integration testing of composite applications.
2. Evaluation of the Testing and Test Control Notation 3 (TTCN-3) as a standards based foundation for our architecture.
3. A set of generic adaptor codes for testing web applications (HTML, web services).

4. A declarative model and notation for defining SOA test frameworks and scripts.
5. A generator of executable or run-time TTCN-3 test frameworks from declarative test models.

Some aspects of the above contributions have been published in the following papers:

- L. Peyton, B. Stepien, **P. Seguin**, "Integration Testing of Composite Applications", Proceedings of the 41st Hawaii International Conference on System Sciences (HICSS 2008), 10 pages, 2008. ISSN:1530-1605.
- **P. Seguin**, L. Peyton, B. Stepien, "Open Source Integration Testing", Workshop on Integration of Open Source Components into Large Software Systems, OOPSLA 2007, Montreal, 4 pages, October 2007.

1.3. Thesis Methodology

The methodology we employed in this thesis was analysis of existing problems, followed by an iterative analysis, evaluation and improvement of our proposed approach to solve the problem. In particular we took the following steps.

1. Identify and analyze problem (integration testing composite applications)
2. Review literature and industry approaches to problem
3. Identify shortcomings and define objectives

4. Create initial grey box test agent architecture to address shortcomings and meet objectives
5. Perform case study (CD Store) to evaluate and compare architecture to traditional industry approaches (combination of white and black box testing).
6. Identify shortcomings of previous work and revise objectives for thesis
7. Extend architecture with generic web-service and HTML adapter/codecs to address shortcomings and meet objectives
8. Use case study to evaluate and compare hard-coded framework with custom adapter/codes.
9. Create model-based framework approach to address shortcomings and meet objectives
10. Perform case study to evaluate and compare model based approach with hard-coded framework and generic adapter/codec approach.
11. Draw conclusions and identify future work.

The thesis is organized as follows:

We set the context for the problem we intend to solve in Chapter 2 by giving a background on service oriented architecture (SOA) and composite applications. We then discuss different testing methodologies and the challenges that arise when applying them to composite applications. We then review other work that has been done in the field and identify how it differs from our goals and where it can be improved upon.

In chapter 3, we identify the key requirements for our framework and then introduce the basic building blocks of our approach: a grey box test agent architecture, generic adapter/codecs for HTML and Web Services, and a model-driven notation for specifying test campaigns.

In chapter 4, we follow this up with a detailed description of our case study including the architecture of the CD Store and the various steps we took to implementing and evaluating our approach.

In chapter 5, we report the results of our evaluation comparing the different versions of our implementation as well as comparing it to other approaches.

In chapter 6, we summarize our conclusion and identify future work that could build on this thesis.

Chapter 2. Background

2.1. Service Oriented Architecture

Service-Oriented Computing (SOC) is a computing paradigm that utilizes services as fundamental elements for developing applications and enterprise software systems [Papazoglou03]. SOC can be implemented by the use of a Service Oriented Architecture (SOA). A service-oriented architecture is built from a set of loosely coupled services which interoperate through the use of predefined protocols of information exchange. The two protocols most widely used in SOA have been put forth by the World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment [W3C07] and the Web Services Description Language (WSDL) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information [W3C01]. Together, these protocols allow services to publish their description (WSDL) and based on the description consumers can invoke them using the SOAP protocol. Since services may be offered by different enterprises and communicate over the Internet, they provide a distributed computing infrastructure for both intra and cross-enterprise application integration and collaboration.

2.2. Composite Applications

A composite application is a piece of software that composes functionality drawn from services to perform operations or tasks on behalf of a user, usually in the context of

a service oriented architecture and often within the context of a well defined business process. The OASIS organization has developed a framework [Oasis07] of related standards to address issues related to composite applications and business processes beyond the initial set of standards for service oriented architecture defined by the W3C [W3C04]. Composite applications support dynamic run-time configuration and the collaboration of services.

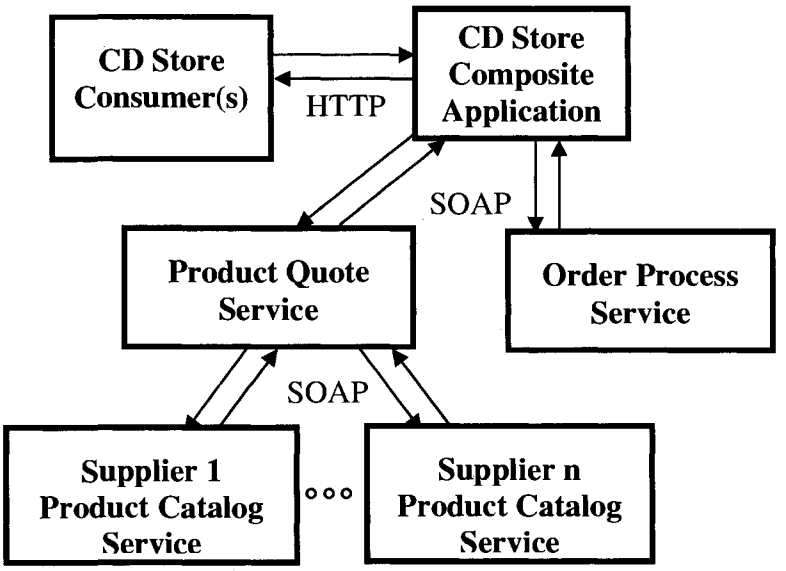

Figure 1 – Example Composite Application

Figure 1 gives a simple example of a composite application that orchestrates a set of services available in an SOA to execute business processes. In this case, the composite application is an on-line CD Store.

There are several services available that are consumed by the composite application including an Order Process service that creates and processes orders for fulfillment and a Product Quote service that provides information on product availability and pricing. The Product Quote service in turn communicates with several Supplier

Product Catalog services, corresponding to the different suppliers that can provide CDs. Each of the Supplier Catalog services may have an implementation that was developed completely independently of each other and also of the CD Store composite application. Furthermore, each of the Supplier Catalog services can reside in other organizations. This is problematic for traditional testing approaches since the organization developing the composite application may not have access to the source code of these services and in many cases may even have to do testing using the 'live' service which resides at the other organization when a suitable test service is not available.

2.3. Testing Methodologies

In this section we introduce the different testing methodologies that are most often used at different stages of development of composite applications.

2.3.1 White Box Testing

White box testing is done in conjunction with the primary development activities of each component in the SOA and is done by the component developer. By 'white box' we mean that test cases have access to the source code of the component being tested and are written in parallel to the component under test. Although unit testing can be used in other situations it is the most popular methodology used in conjunction with the white box testing approach. A single unit test tests a particular behaviour within the production code [Hamill04]. In the case of object oriented programming behaviours are expressed as methods. Test cases are set up to test each method in isolation. Other dependent parts of the system such as domain objects or external entities such as services can be tested through the use of mock objects [Mackinnon00]. Mock objects define the messages they

expect to receive and their proper sequence [Fowler07]. The part of the system the method is dependent on is initialized and the method is executed. The state of the system after the method is executed is compared against an expected state. If the expected end state and the actual end state match then the test is deemed to be successful. Unit testing frameworks are available for most general purpose languages (JUnit for Java [JUNIT08], NUnit for .NET [NUNIT08], etc.).

2.3.2 System Testing

System testing is concerned with testing an entire system based on its specifications, and involves several activities such as functional testing (testing from behavioral descriptions of the system) and performance testing (response time and resource utilization) [Binder99]. System testing is a form of integration testing where the group of subsystems being tested is in reality the system as a whole. As opposed to 'white box' testing, system testing is a form of 'black box' testing where the internal structure of system under test is hidden and it is only accessible through its public interfaces.

There are tools available to perform this type of testing with respect to composite applications, OpenSTA being the most widely used. OpenSTA [OpenSTA08] is a distributed software testing architecture designed around CORBA. The current toolset has the capability of performing scripted HTTP and HTTPS heavy load tests with performance measurements. It is also able to do some basic validation of responses from the system under test. Other model based approaches have also been proposed. In [Amyot05] a model based approach to acceptance testing of web applications was proposed in which acceptance tests expressed in FitNess [FITNESS08] were derived

from Use-Case Maps. However it was not applied to web-services or multi component systems.

2.3.3 Integration Testing

Integration testing is concerned with verifying functional, performance and reliability requirements placed on major subsystems or groups of subsystems [Lucas00] and focuses on testing subsystem interfaces and their interaction with other subsystems. It is usually done after the individual sub systems have been completed and integrated together to form groups of sub systems. The subsystem or group of subsystems under test is traditionally treated as a black box and tested through its interfaces. Pre determined error and success conditions are set and tests consists of verifying responses from the system under test (functional) and the time it takes the system to generate the response (performance) against the expected conditions and under varying loads. Multi user scenarios can also be tested in this way. To do integration testing on a complete system it may be necessary to do integration testing on multiple individual sub systems first, treating each as a black box and doing additional iterations on assemblages of sub systems that can be grouped as a black box and so on until the largest unit can be tested.

An example of this approach is presented below. Figure 2 shows a composite test agent that emulates the behavior of a composite application consumer, communicating with the composite application via HTTP requests and responses. The composite test agent not only emulates a consumer, but it also verifies the responses received based on pre-defined test cases. This enables one to test the actual flow of composite application responses and their presentation elements. It also stresses the overall system under the

actual combination (orchestration and choreography) of web service calls that the system employs.

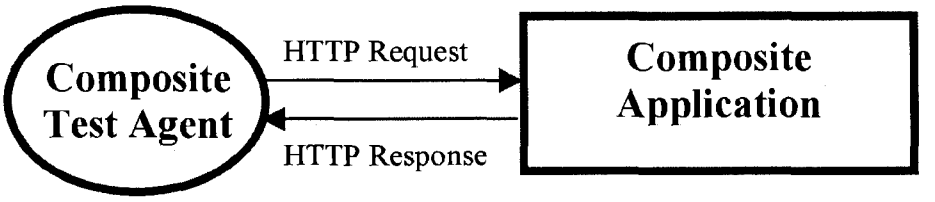

Figure 2 – Black-box Testing

This is the most natural approach but it does not allow one to pinpoint the reasons for a failure with precision, because all the messages between the web application and the underlying services are not visible. There are also complications in that the expected response from the composite application will incorporate many volatile presentation and formatting details related to the browser interface [Stepien08].

A more complete black box test of the composite application, shown in figure 3 below, consists of testing both the composite application's interaction with the consumer and the flow of messages that are occurring between the composite application and the web services.

For each service that the Composite application interacts with, a service stub is created to emulate the service and validate the interaction of the composite application with it. The environment is configured so that the SOAP requests that the composite application makes when calling a web service are redirected to the appropriate service stub instead of the real service. The service test stub consists in receiving and matching expected SOAP requests from the composite application and if satisfied sending the corresponding responses back to the composite application as the real service would do.

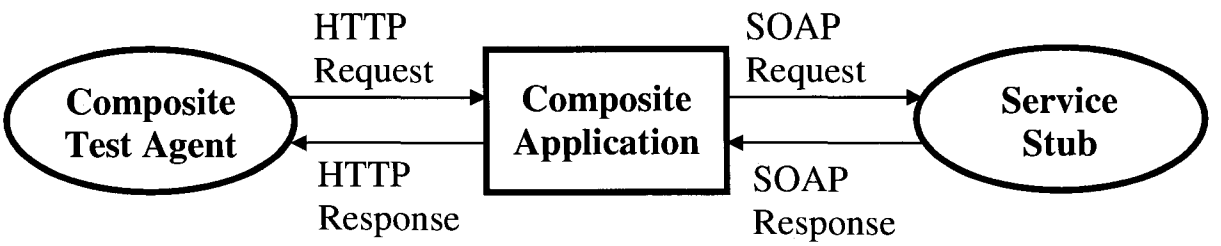

Figure 3 - Application test with service emulation

This approach verifies that the composite application sends the expected requests to the services. Once this is verified, if the response to the user is incorrect, one can conclude with confidence that the problem is located in the composite application processing as long as each service test agent is accurately emulating its service. However, this methodology does not address the following issues. Each of the services in the service oriented architecture may be evolving independently of the composite application so we need some mechanism of integrating verification of the composite application with verification of each service. Also, this methodology does not address how the composite application will behave when in conjunction with the actual services it will be using in production. Finally, this methodology does not address how difference scenarios will impact overall performance of the composite application since the performance characteristics of the underlying services are not considered.

Figure 4 shows an example of how each service can have its own black box test in which a service test agent emulates the behavior of the composite application by sending SOAP requests and receiving SOAP responses that one would expect a composite application to send/receive when orchestrating an interaction with the services in the SOA.

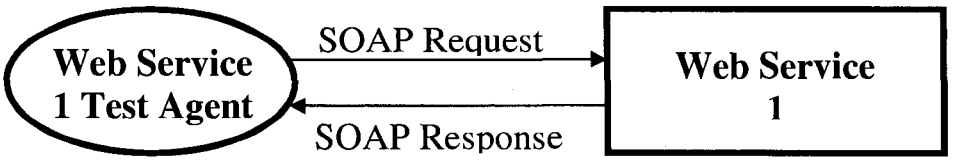

Figure 4 - Service Test with Application Emulation

Note, however, that this service agent is completely different from the service test agent in Figure 3 and completely independent of the black box text of the composite application in Figure 2. It provides a simple unit test of the service narrowly focused on the perspective of the composite application (ignoring the types of interactions other composite applications may invoke) and completely ignoring any possible interactions or dependencies with other services. The web service call may function as designed and pass tests but fail in combination with other web service calls in the full application. There still needs to be some mechanism of integrating verification of the composite application with verification of each service. . This thesis presents one possible mechanism for doing this.

2.3.4 Regression Testing

The purpose of regression testing is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change [Wong97]. Regression test cases can be drawn from the other testing methodologies described above. In a composite application regression testing takes on an even more important role since the underlying services may be evolving independently or being substituted for other implementations independently of the composite application itself. When a change is made to the underlying services

regression testing should support the ability to verify that the substituted service behaves in the same way as the original service.

2.4. TTCN-3

TTCN-3 [ETSI08] is a test specification and test implementation language for testing distributed systems developed by the European Telecommunications Standards Institute (ETSI). It provides powerful abstraction mechanisms for interfacing to different data and presentation formats. It also enables one to define test cases at different levels of abstraction, much as developers use modeling languages to specify the design of a system at different levels of abstraction. This allows one to define functional tests in terms of the essential application logic and its management of information independent of volatile implementation and presentation details. It also allows for reuse across different levels of test activities [Probert04]. In particular, it allows testers to start working in parallel to developers from the same system requirements and specifications. This would be useful for our methodology since many of the components in a composite application already exist at design time.

TTCN-3 is based on the concept of sending a message to a system under test and receiving a response that it will attempt to match against a very flexibly structured template that serves as an oracle to define the possible outcomes. The central concept of the TTCN-3 testing language is a separation of concerns in the architecture of a test framework. This separation of concerns is performed at two different levels:

First, TTCN-3 defines an Abstract Test Suite separate from the concrete implementation of coding and decoding of requests and responses and all related communication with the system under test.

Second, TTCN-3 presents an Abstract Test Suite as a system behavior tree that displays sequences of requests to and alternative responses from the system under test. The switching of paths through that tree is achieved via templates that are combinations of test data and matching rules. Thus, the tree and templates represent a separation of concerns between behavior and conditions governing behavior. The shear volume of test cases to manage in a composite application would make it beneficial to abstract test case logic from the implementation details.

Test behavior is displayed using the concept of a hierarchal tree where the child nodes indicate branching. The tree specifies the sequence of requests and responses to the various services composing a system. The tree shows all the possible alternative behavior paths a system can follow during a specific test. TTCN-3 templates are used to determine which alternative path the system takes. It is by matching a given template against an incoming response that the test execution tool can determine which path to follow. Eventually, a path will lead to a leaf where the test verdict is set according to the tester's test purpose. This is analogous to the choreography and orchestration that a composite application uses to function in conjunction with its underlying services. The composite application makes sequences of requests to services to fulfill its goals. It would be advantageous to have a test system which can efficiently mirror this functionality.

A test case consists of a sequence of requests and responses encoded as a tree as described previously. A test case can be parameterized to make it re-usable with different test data templates. A test case is always declared to run on a specific test component and system test component. Normal computations can be inserted anywhere in the behavior tree.

TTCN-3's main characteristic is to separate the abstract test suite from lower level activities such as the communication management and the coding and decoding of messages. For example, HTTP requests arrive in the form of text that needs to be decoded to obtain the relevant information for a test. However, this coding/decoding activity is of no interest at the abstract specification of behavior. Consequently it is an advantage to separate it from the abstract layer and all we need is some mechanism to populate the abstract data structures with the values obtained from the messages. This adaptation layer is most efficiently programmed using a traditional programming language and depends mostly of the APIs provided by TTCN-3 test execution tools. Since composite applications which make use of SOA have clearly defined interfaces (WSDL, SOAP) with its underlying components it would be advantageous to have a test system which can also communicate with these components in a systematic and concise way.

Figure 5 shows the TTCN-3 stack. Abstract test cases are defined at the top level and are translated into concrete test cases by the selected TTCN-3 compiler. The executable code is then linked to the test adapter and codec. The codec is responsible for encoding and decoding of requests and responses from the Test Adapter which sends messages to the SUT or CUT. Requests and responses are in a raw form that needs to be

decoded into TTCN-3 data structures to obtain the information relevant to the abstract test suite for validating the requests and responses.

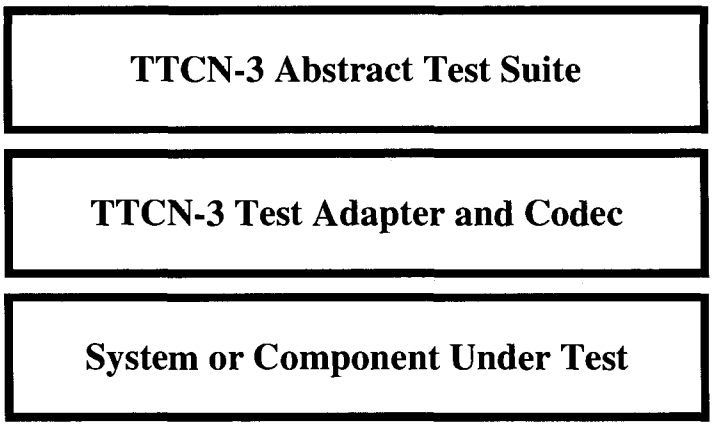

Figure 5 – Separation of Concerns

Work has been done to apply TTCN-3 in web service testing. In particular in [Troschütz07] a generator of TTCN-3 data structures from WSDL was proposed. The authors found that a 1-to-1 mapping of WSDL to TTCN-3 data structures was possible and implemented a program to accomplish it. However, they did not take into account the communication and parsing mechanisms used to communicate with the SUT and the ability to test web-services within a larger composite application.

2.5. Related Work for Integration Testing Composite Applications

2.5.1 Service Log Files

Typically, each service in a SOA generates log files [Rankin02]. In [Peyton08] a prototype framework was proposed for collecting log files into a data warehouse in order to analyze quality assurance issues. However this proved problematic for several reasons.

- There is no standardization in the industry for the format of log files therefore they are often processed individually with a custom built processor.
- It can be difficult to configure the services to log all necessary information.
- Log files for services not under the control of your organization are often impossible to retrieve.
- Multi-user scenarios are hard to process since there is usually little correlation between the log files of different services.

A large part of the effort was on the processing of the files and the efforts to organize them after the fact. Their main QA focus is on performance & reliability testing and not on integration testing and validation of request handling.

2.5.2 Formal Verification

Several approaches have focused on supporting formal verification of services against defined protocols. In [Xiong05] formal verification of web services using TTCN-3 was presented, while [Bertolino06] leveraged the UML 2.0 protocol state machine to define the expected protocol for web service conformance. In [Tsai06], ebXML dynamic collaboration protocols from OASIS are extended with temporal logic and timing constraints and showed how a distributed framework of test agents [Bai06] could be used for dynamically verifying completeness and consistency of service invocations in compliance with the protocol. These methodologies only consider a service's interface definition and do not confirm that a particular service implementation actually behave

properly within the interface's definition. Also, they do not address multi-user issues associated with performance when service implementations are combined under load.

2.6. Model-Based Testing

Declarative models have been used to successfully describe artifacts of software systems as in with the Unified Modeling Language (UML) [OMG08] and Business Processes as in with Use Case Maps (UCM) [Buhr98] and in many other areas. Figure 6 shows how declarative models can be used in the software design process in defining abstract models of system behaviour and architecture and in the generation of domain specific source code. Modeling environments can be used to edit the declarative model using graphical representations of the declarative model. Declarative models are either generated by the modeling environment or edited directly and are created in a standard readable format such as XML following a XML schema. Based on the declarative model many model translators can be created (iii) for domain specific executable implementations of the model (iv)

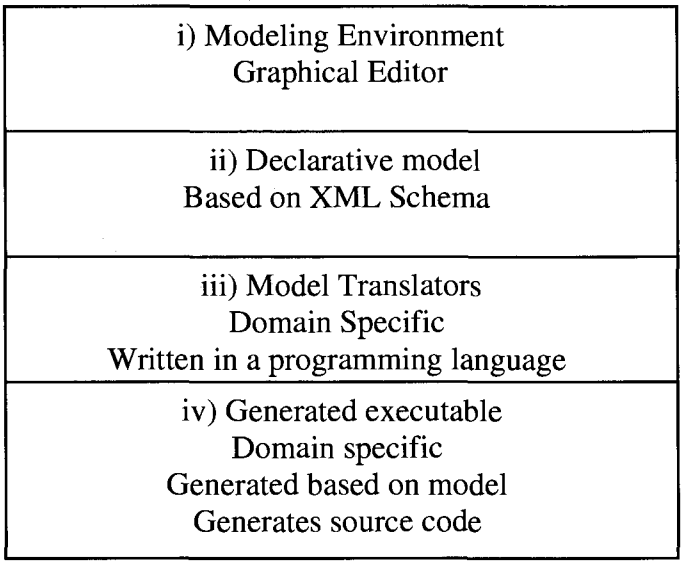

Figure 6 – Declarative Model

With regard to testing with TTCN-3 based on declarative models work has been done on translating the declarative models of Message Sequence Charts (MSC) [MSC99] and Specification and Description Language (SDL) [SDL96] into executable TTCN-3 functional test cases [Probert99]. In [Amyot03] MSCs are derived from higher level UCM models. Combining these two approaches would result in a UCM to TTCN-3 test case translation mechanism. A similar approach to ours is presented in [Mulvihill03]. In this paper the generation of TTCN-3 test cases based on UCM models was considered. As future work the authors' suggested the support of multi-component systems. Also executable test cases were not considered, only the abstract definition of test cases in TTCN-3. Finally, in [HEC03] the notion of model-driven testing is proposed in which the authors create platform independent models for unit testing and based on these they generate platform specific test drivers. However these approaches consider only the testing of single components and do not go into detail on how these models can be extended to generate tests that take into account the multi component nature of composite applications.

Chapter 3. Integration Testing Framework for Service Oriented Architecture

In section 3.1 we outline the issues or requirements that an integration testing framework should address. In section 3.2 we introduce the grey box test agent architecture we propose to address these goals and in section 3.3 we discuss challenges associated with implementing our architecture. Finally, in section 3.4 we present a generic model based approach to implementing the architecture detailed in section 3.3.

3.1. Integration Testing Requirements

In our context, the purpose of integration testing is to ensure composite applications run on a stable common infrastructure even as different components of that infrastructure can be dynamically changing. In particular, a framework for integration testing must provide a comprehensive and comprehensible set of tools for managing integration testing in such a complex environment. Below, each of the following subsections identifies a major requirement or criteria that an integration test framework should support or address.

3.1.1 Testing Individual Component Interfaces

The ability to test individual component interfaces is the basis for all other requirements of integration testing. Many unit testing tools can accomplish this such as (HTTPUnit [HTTPUnit08], OpenSTA [OpenSTA08], JMeter [JMeter08], JUnit [JUnit08], NUnit [NUnit08] and many others. These tools fall into two categories; GUI or abstract test script based tools which can easily create test scripts for web based interfaces

(OpenSTA, JMeter) used to do black-box system testing and tools that assist writing test scripts in general purpose languages (JUnit, NUnit, and HTTPUnit) usually to test individual methods or function calls to do white-box unit testing.

3.1.2 Component Interaction

The goal of testing component interaction is to be able to localize faults within a composite application to a particular component involved in the test case. In large systems there may be many components, all of which may be updated or added independent of the composite application. The ability to localize faults is necessary to efficiently address them. In a composite application, when a user interacts with it, a single user request may result in a complex set of interactions between components in the SOA in order to satisfy the request. One must be able to analyze each component interaction along the way to satisfying a user request, in order to verify that each component is performing as expected in the context of the overall user request. One must be able to localize a fault in satisfying a user request down to the specific component that failed. The failure could either be that a specific component passed in the wrong parameters to another component, or that a component supplied an incorrect response to a valid request.

3.1.3 Performance

Similar to component interactions, poor performance in handling user requests can be analyzed in terms of the component interactions which make up a user request to identify or localize which component interactions are not performing as expected. When testing the integration of components in a composite application an important aspect is testing its performance characteristics. When components are combined to form a

composite application, unexpected performance issues can arise from unexpected areas. The requirement is to measure the actual performance of individual components during tests cases and compare the results to pre defined success/failure criteria.

3.1.4 Multi user scenarios

Enterprise scale composite applications usually have to support large volumes of users interacting with the system in parallel. In addition, many composite applications may be sharing components in parallel as well. Component interactions which execute correctly in the context of a single user request, may not execute correctly under multi-user scenarios due to side effects of being invoked in parallel. Similarly, system performance may be acceptable and scale well for single user requests to a single composite application executed in serial, but suffer significant performance problems under multi-user scenarios. Issues of performance, security, and validity are all affected by multiple concurrent users accessing the composite application. For this reason our integration testing framework must be able to support simulation of these types of scenarios and the ability to localize faults and performance issues to particular component interactions.

3.1.5 User competencies

Testing professionals that concentrate on integration testing are usually not primarily software developers. Therefore, for an integration testing platform to be successfully adopted it must be simple enough to express the test and manage the scripts for testers whose main competencies do not lie in programming in a general purpose programming language.

3.1.6 Effort for New SUT

When applying a test strategy to a new system it is important that the initial application of the strategy not have high start up time. The first aspect of this is that it must not be intrusive to integrate the testing strategy into the test system by placing restrictions on the system design. Secondly, the test system must be as efficient as possible to apply to the new system.

3.1.7 Extensibility of Test Framework

New or unforeseen testing strategies may arise as the composite applications become more complex over time. By extensibility we mean that when an unforeseen testing strategy must be implemented we would like to know how much effort is needed to implement it. There are two aspects to this.

- How flexibly can the given test framework be adapted to new test types by simply adding new test cases within the bounds of the test framework's current underlying structure.
- If the test framework itself must be extended how simple is it to extend the test framework.

3.1.8 Ease of Expression for Test Campaigns

In order to reduce human error in constructing test campaigns, the framework must have a clear and simple model for interaction and definition of campaigns. If the language used to express campaigns, or the framework itself, is complex, then users will make mistakes. Test scripts must have low error rates.

3.1.9 Effort to Debug Test Framework and Scripts

There are two considerations for debugging the test framework. The first is efficiently identifying when there is in fact a bug in the test framework. The second is the ability to localize and correct bugs within the test framework.

3.2. Test Framework Architecture

In order to address the complexity inherent in a service oriented architecture, our framework uses a grey box test agent architecture that mirrors the service oriented architecture in which composite applications are run. This enables support for localizing faults and performance issues as well as addressing multi-user scenarios in a scalable fashion. Typically, each component of the system being tested is paired with a test agent specific to that component which can monitor and test the component it is paired with. A master test component is used to coordinate the activities of all test agents.

3.2.1 Addressing the Limitations of Black Box Testing

As seen in section 2.3.3 in black box integration testing, the composite application and each service used is tested as a separate "black box" in which only the inputs and outputs of the black box are tested. This architecture does not address how the various test agents and test cases are kept in synch as the different components evolve independently. It also does not address the most difficult aspect of integration testing which are the possible interactions between web services as the composite application choreographs and orchestrates its use of the web services.

In our grey box architecture, there is a test agent for each component which combines both a black box test driver functionality in order to call the component and check its responses, but also a "mock" object functionality in which the test agent can be called in lieu of the component with an expected set of requests and provide the appropriate responses.

3.2.2 Grey Box Test Agent Example

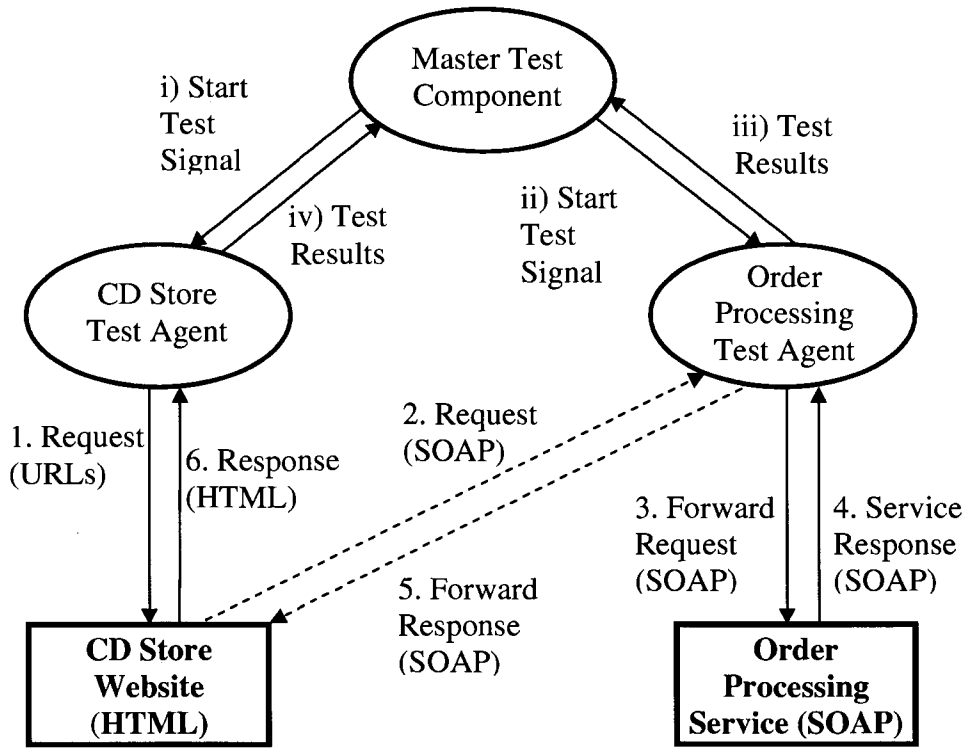

Figure 7 – Test Agent Example (CD Store)

Figure 7 introduces our 'grey box' testing architecture by using a simple example of how the test agents interact during a simple test case. The test case is based on a simple CD Store which consists of a CD Store composite application component and an underlying order processing service component (bottom left and bottom right

respectively). The test case verifies the behaviour of the CD Store under the use-case of a user completing an order to buy some CDs.

Initialization: Starting from the top left portion of the figure, a start test signal is sent to the Order Processing Test Agent first (i). This is done to start the test agent and prepare it to accept requests on behalf of the Order Processing Service. Once the Order Processing Test Agent has been started the CD Store Test Agent is then sent the Start Test Signal (ii) and the integration test starts.

1. The CD Store Test Agent actively sends the appropriate Request URLs to the CD Store Website.
2. During processing the CD Store Website must make a SOAP request to the Order Processing Service to complete the transaction. In our test framework, the request must be redirected to the Order Processing Test Agent and the test agent verifies that the request is one expected by the test script.
3. The Order Processing Test Agent then forwards the request to the Order Processing Service, so it can process the request.
4. The Order Processing Service processes the request and sends the response back to the Order Process Test Agent.
5. The Order Processing Test Agent verifies that the response is what was expected and forwards the response back to the CD Store Website.
6. The CD Store Website builds the html page that comprises the response to the user and forwards it back to the CD Store Test Agent. The CD Store Test Agent then verifies the response and sets its test verdict.

End of test behaviour: In the mean time the Order Processing test agent has also completed processing and has set its test verdict. The test verdicts at each test agent are forwarded to the master test component (iii, iv). Finally, the overall test verdict is set at the master test component and the test completes.

3.2.3 Grey Box Test Agent Architecture

In this section we generalize the scenario introduced in section 3.2.2 to form our grey box test agent architecture.

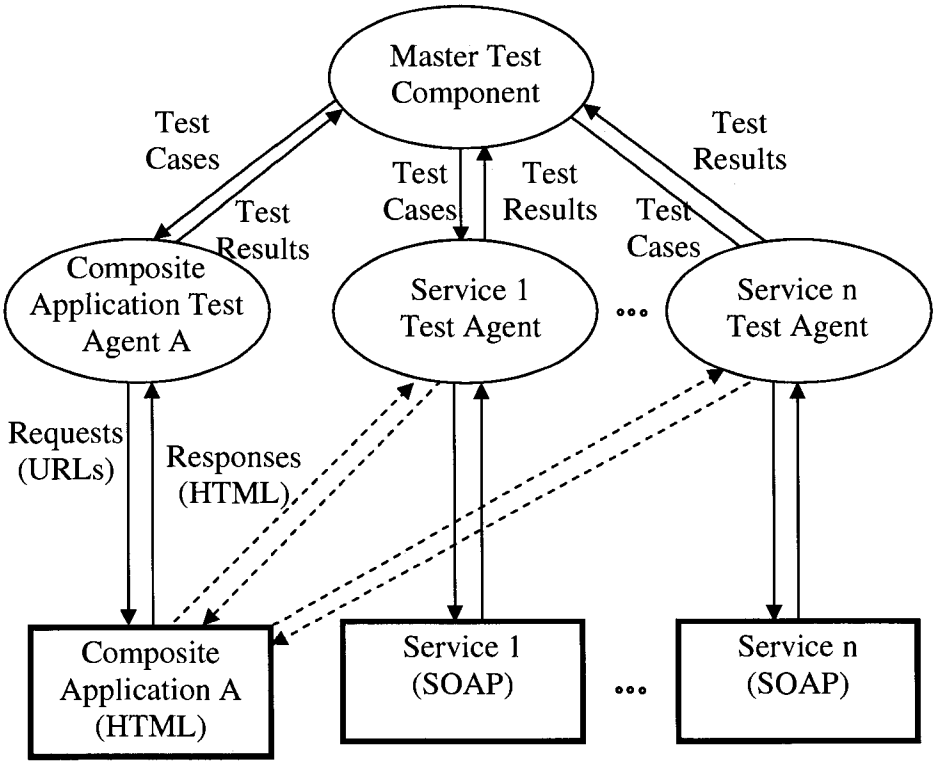

Figure 8 – Grey Box Test Agent Architecture

Figure 8 shows a generalized grey box test agent architecture in which each of the subsystems (in the form of a composite application and a set of underlying services) are

tested together in a single integrated test framework. We refer to this as "grey box" testing because the system we are testing is in effect the overall service oriented architecture and it is not treated as a black box, rather it is treated as a "grey" box in which we are aware of all of its components and can monitor and test the interactions between these components. As you can see in figure 8, each service test agent is placed between the service consumer(s) and the service itself by acting as a proxy which relays messages to/from the actual service. For example, when a request made to *Service 1* from *Composite Application A* it is routed through *Service 1 Test Agent* which acts as a proxy. *Service 1 Test Agent* verifies that that the request from *Composite Application A* is expected at *Service 1* and that *Service 1* returns the correct response to *Composite Application A* according to the test script. More generally, each of the service test agents has the ability to both validate the requests from service consumers and the responses generated from the actual services. These test agents are passive and wait for requests from service consumers. The active test agents in our architecture are composite application test agents (*Composite Application Test Agent A* in figure 8). These agents emulate user interaction with the composite application. For example *Composite Application Test Agent A* will send requests to *Composite Application A* in the form of URLs and receive and verify responses in the form of HTML web pages. The Master Test Component is able to correlate precisely where faults are occurring. This test architecture also stresses the overall system under the actual combination (orchestration and choreography) of web service calls that the system must support, testing the actual responses that are returned by each service and pinpointing where faults (validation, performance, security) are occurring. Careful design of the service test agents should

also make it possible for them to be implemented in such a way that their core functionality is reusable by any composite application.

3.3. Implementation Requirements for an Integration Test Architecture

There are some significant implementation challenges associated with this test agent architecture, especially if the composite test agent is simulating many users making multiple simultaneous requests to the composite application. In particular, we highlight two challenges that our Integration Test Architecture must be able to address in its implementation.

- Caching: Previous responses from an underlying service may be cached so that identical requests to the composite application may not result in the same requests to underlying services, even when performed on behalf of different users.
- Correlation Gap: The sequencing and interleaving of requests and responses may vary significantly making it difficult to correlate service requests and responses to a particular user request made to the composite application.

Here we explain the challenges and what features are needed address them. In section 4, we go into more details as to how our implementation using TTCN-3 provided these features.

3.3.1 Correlation Gap

The correlation of request/responses at the composite application and possible corresponding request/responses to underlying services is difficult. By their nature, each

service in a composite application may or may not be under direct control of the development team in charge of the composite application. In fact, they may even be controlled by other organizations (see Figure 1). In this case, it is not feasible to simply include unique IDs to each message to solve this problem. Another approach is needed. The correlation gap is a temporal ordering problem. The composite application may place its requests to the service in a different order then what was received from the users. Similarly, services may return responses in a different order from the order in which it receives requests. Figure 9 shows an interaction diagram of two users (simulated by the composite test agent) interacting with a composite application. Request 1 is submitted first by User1 however Request2 from User2 is fulfilled first by the composite application. The interleaving of requests and responses makes it so that requests cannot simply be correlated by their order of arrival/departure from the test agents. Therefore, in the general case of composite applications, simple end to end tracking does not work.

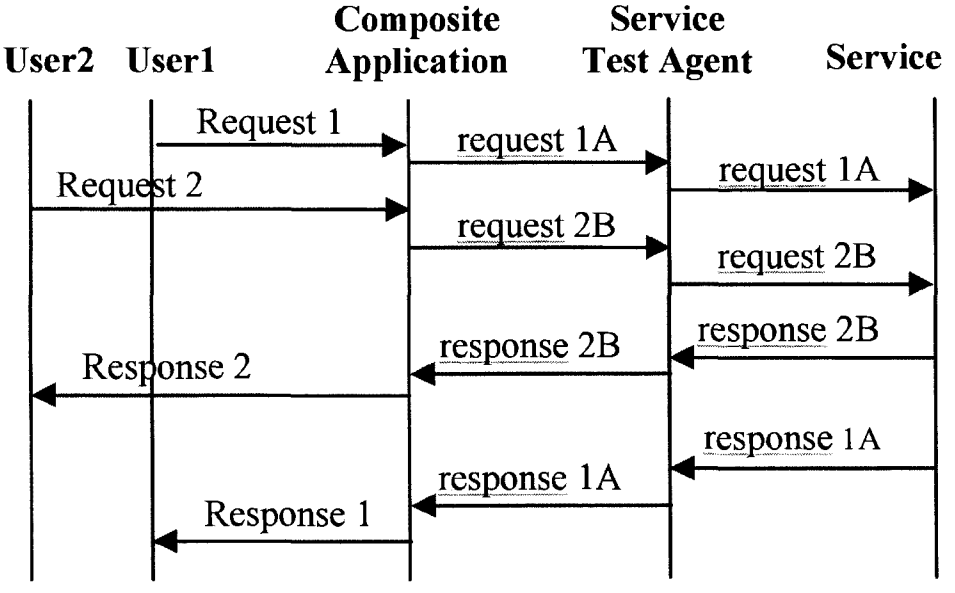

Figure 9 – Correlation Gap for Multi-user Requests

To handle the correlation gap, we must use sets of requests/responses to handle the verification of messages agnostic of arrival time. For each service request received, the service test agent performs two types of checking:

- Verifies that the message was expected for a specific test campaign, if yes, then forwards it to the service.
- Enforces the expected response from the service and if successful forwards the service response to the web application.

The master test component tells the service handler what requests to expect based on some internal logic gathered from the user test components, but not the order in which they will be received. At the end of the test, the service handler checks if the set of messages it was told to expect by the master test component completely matches the set of actually received messages.

3.3.2 Caching

Composite applications that consume services often cache responses from services for future use. This usually happens when the response is known to be valid across a certain time interval or transaction e.g. for a consumer's session. It is important that the caching mechanism should be well documented by the composite application designers since it typically is based on assumptions of how the service it is using behaves. In order to test caching, we need to verify that a (non-event) has occurred. If a request to a service that should be cached does not occur then the test can pass, and if it does occur, the test should fail. This requires three mechanisms:

- A mechanism for representing a caching mechanism
- A mechanism for representing the non-event detection
- A mechanism to distinguish messages that are subject to caching from others that never can be cached because they contain only one time user data such as invoice content.

3.4. Model-Driven Testing

In addition to our grey-box test agent architecture, we defined a model-driven test generator to address the requirements outlined in section 3.1 related to ease of maintenance and debugging. The error rate of test scripts should be lower if we do not have to manually create them from scratch each time in a general purpose language. The effort for implementing the test system on a new SUT should also be lowered since the foundation of the test framework (in terms of test agents and message handling) is already created and the details of the specific components and test cases only need to be specified. The ease of maintenance should be higher as well, since the test scripts are easier to understand at a high level and easer to manage and organize. Finally, the level of user competency required should be lowered since it is no longer necessary to implement the test system in a general purpose language. One can concentrate on the abstract definition of tests.

In addition to the potential advantages described above, we were motivated to take a model-driven approach in an attempt to address the two main implementation problems we encountered in simply hard-coding our grey box test agent architecture (which is described in chapter 4):

- Decoupled test agents
- Correlation gap and Caching

In our hard-coded architecture the behavior of each service test agent is defined separately. This makes it confusing to implement new test cases since the behaviour of a single test case is distributed across several behaviour functions, namely the behaviour functions of each of the involved components related to the test agent. For example the test case presented in section 3.2.2 can be abstractly defined as:

1. User sends request to submit order to the CD Store.
2. CD Store sends request to the order process service to fulfill the submitted order request.

However, using our basic architecture this abstract test case must be broken down into the requests made to each service. (1) is associated with the CD Store application and (2) is associated with the Order Processing service. Therefore, when implementing this test case there is no concrete connection in the implementation that ties these two aspects of the test case together.

The correlation gap and caching problems are partly responsible for this issue and compound it. To handle the correlation gap and caching, a list of expected messages must be defined for each service so that the completeness of the messages exchanged with a particular component be verified. Each message request/response expected at a component is added to this list and once it gets large it can be hard if not impossible to manage it as new test cases are added and existing test cases are updated.

3.5. A Declarative Model for Integration Testing

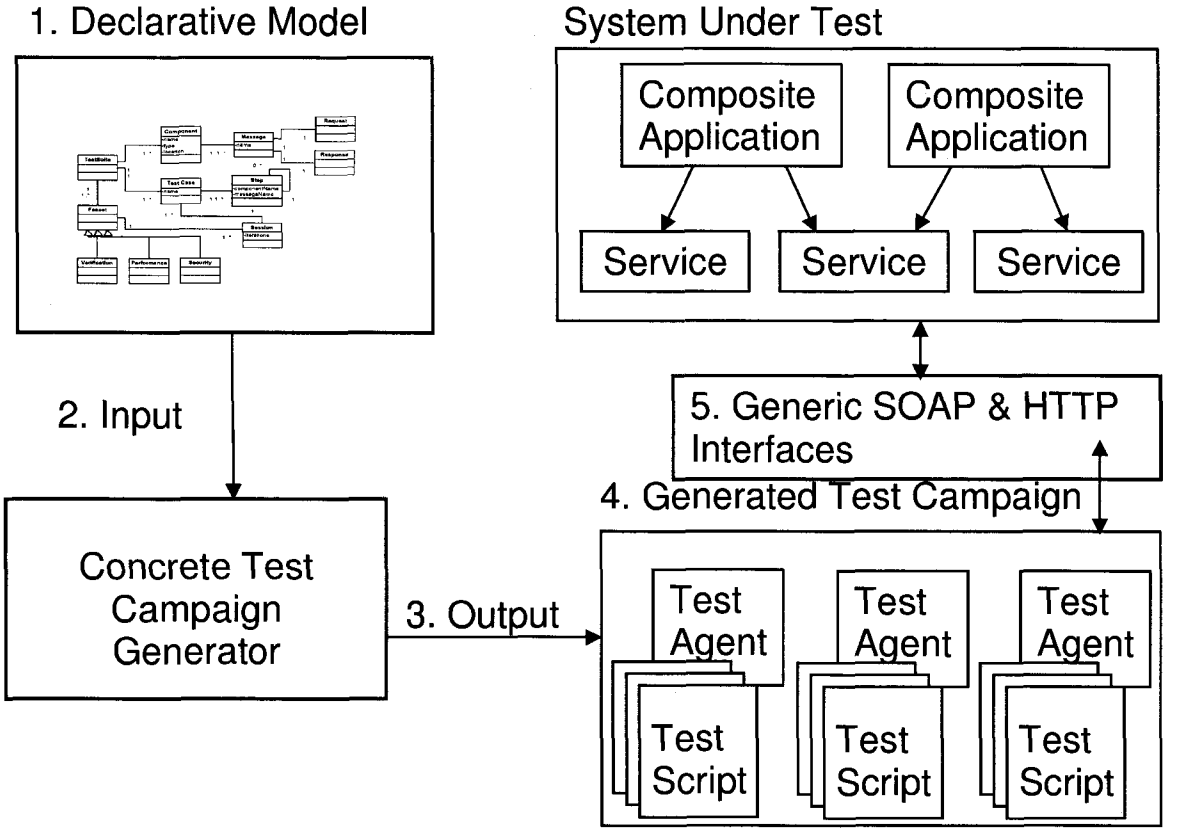

Figure 10 – Model-Driven Testing

In this section we give an overview of the workflow involved in using a model based approach to our testing architecture. Starting from the top left of figure 10,

1. The tester defines a declarative model which describes the interacting components of the system under test and the test scripts that will be run against it.
2. This model is used as the input for the concrete test campaign generator.
3. The generator translates the model into a concrete test campaign which is composed of test agents and test scripts in a grey box test agent architecture

4. The generated test campaign can be executed against the SUT by the tester.
5. The generated concrete test campaign interfaces with the SUT through the use of a generic set of predefined SOAP and HTTP interfaces to the SUT.

3.6. Model Specification

This section details our proposed declarative model. The discussion is aided by a Declarative UML Class diagram below (Figure 11). The diagram is declarative in the sense that objects, attributes and relationships are specified with no mention of methods or behavior. An example model for our case study in chapter 4 can be found in Appendix A.

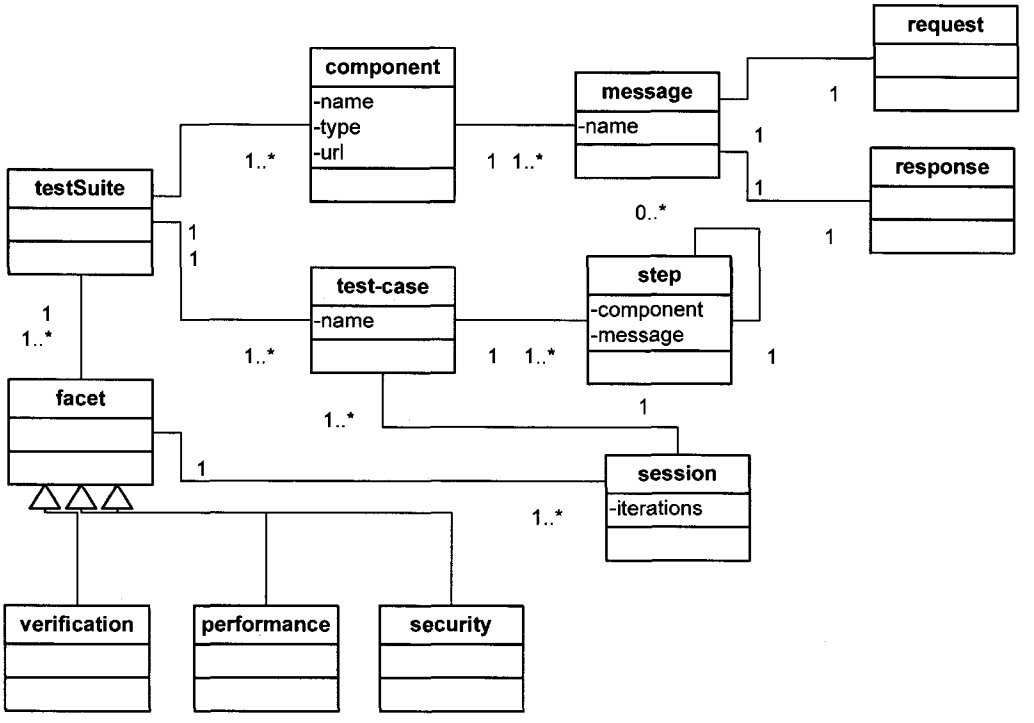

Figure 11 – Declarative Object Oriented Model for 'Grey Box' Testing Framework

3.6.1 Components

The first section of the test suite definition is to define the set of interacting components. Each component is uniquely identified by a name attribute. Each component has one or more messages which it can handle. The messages are uniquely identified by a name attribute. Each message is composed of a request and a corresponding response.

3.6.2 Test Cases

The test suite is composed of a set of test cases. Each test case is composed of a set of test case steps. Each step references a message as defined in the component section. Steps can also be nested within other steps. The nested list of steps defines the order in which messages will be exchanged with the SUT for a particular test. In this way test cases can be specified in a central location with the interconnections of the components explicitly defined.

3.6.3 Facets

A test suite may have facets for validation testing, performance testing, security testing, and other types of testing. In this way, as the test strategy evolves, new testing goals can be defined as new test facets while still making use of the other components of the model. Facets are composed of a set of user sessions. In each session there is a set of test cases which will be executed by that session. The number of times a session is iterated is defined.

3.7. Test Generation

Based on the above model a generative step must be performed which translates the model into a grey box architecture implementation and a concrete set of tests that can actually be run against the SUT.

3.7.1 Generic SUT Interfaces

For a model based approach to 'grey box' integration testing to be successful there must be a generic way to interface with the system under test. In particular, for our CD Store example, there must be a generic HTTP interface and a generic SOAP interface. There are two aspects to this: the communication mechanism and the message validation mechanism.

A generic communication mechanism means that we have a generic way for sending and receiving messages with components of the SUT. A generic message validation mechanism means we have a consistent way to process messages into standard data structures that are possible to verify in a systematic way.

3.7.2 Advantages of the Approach

This approach will address some of the requirements stated in section 3.1. User competencies would be lowered since the testers would not have to be concerned with the concrete test architecture. The maintenance effort would be lowered since the components and test cases are more explicitly defined and not dispersed across multiple sections of testing code. The effort for setting up this framework in a new SUT would be lowered since the test scripts could be written more quickly and concisely. Finally the

error rate of test scripts would be lower since there is no need to code the more complex and error prone concrete test campaign at the expense of flexibility in the test definitions.

Chapter 4. Case Study

To validate our approach we conducted a case study on a simple but representative example composite application (CD Store) in a service oriented architecture in which we compared our model-driven integration test framework with a simple black box integration testing approach, and with a hard-coded grey box test agent architecture using TTCN3. In section 4.1 we introduce this application and its architecture. In section 4.2 we test the composite application using a hard-coded TTCN-3 grey box test agent architecture. In section 4.3 we improve on the TTCN-3 framework by adding a generic set of HTTP and SOAP codec/adapters to communicate with the SUT. Finally, in section 4.4, we refactor and generalize our architecture to provide our full model-driven integration test framework. The effectiveness and efficiency of these approaches (and their different components) are evaluated and summarized in chapter 5.

4.1. Case Study: Composite Application Architecture

The example composite application used to evaluate the architecture is an online CD store web application. Visitors can browse a product catalog of CDs, select, add and move CDs to a shopping cart and check out by providing credit card and shipping information to purchase CDs.

The CD Store application is composed of three components which form a three tiered architecture.

- Web application front end

- Underlying services
- Relational database

The web application front end uses a Model-View-Controller architecture and its purpose is to interact with the users through web pages and coordinate the orchestration of requests to the underlying services. There are two underlying services; the product catalog service, which supplies information on products available for purchase at the CD Store and the order processing service which validates and processes orders made by users. The services are implemented as standard web services adhering to the SOAP protocol. The services interact with an underlying relational database to fulfill their roles.

The Eclipse IDE is used as the development and test environment for the CD Store application and Tomcat 5.5 is the server instance in which the CD Store is run. DB2 Enterprise Edition V8.2 is used as the relational database engine. We use a deployment diagram, a package diagram, and a sequence diagram to describe the application's design in more detail in the following subsections.

4.1.1 Deployment Diagram

The deployment diagram in figure 12 shows the overall architecture of the CD Store composite application. Starting from the top left; a user connects to the application using a web browser on their personal computer. The web browser connects over HTTP to the controller servlet which serves jsp pages to the user. The user's current state is also held on this component. When a request for product information or order processing is received by the CD Store web server requests are made to relevant web service(s). These two services are able to make requests to a database server to fulfill their roles.

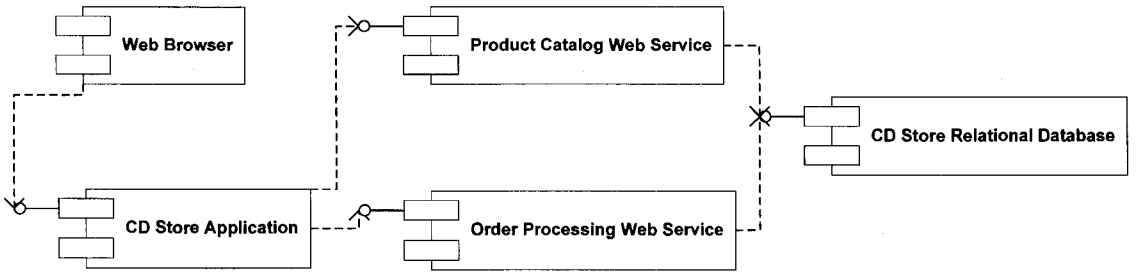

Figure 12 – Deployment Diagram

4.1.2 Package Diagram

The package diagram in figure 13 gives a more detailed description of the internal architecture of the individual components that make up the CD Store composite application. It is most relevant to section 4.2 which details our experiences unit testing this application.

The CD Store implements a Model View Controller architecture. Starting from the top left the view package of the MVC architecture is a set of Java Servlet Pages (JSP) that format the output for users. The controller package consists of a Java servlet which handles incoming requests and a set of request handlers which the servlet delegates incoming requests to. Once the request is handled the servlet forwards the request to the proper jsp page which formats the response to the user. The model is a set of classes that represent the users' state while interacting with the composite application. It also interacts with the underlying services through the use of proxy classes. In the bottom right the DomainModel package is shown. It is simply a standardized format for the exchange of domain information between the various packages that need access to it.

The second row in the diagram outlines the service layer. Starting from the top left the product catalog service handles queries about the products available to buy and

the order process service handles the processing of orders. Both the service packages have access to the DB Agent package which handles interaction with the underlying relational database.

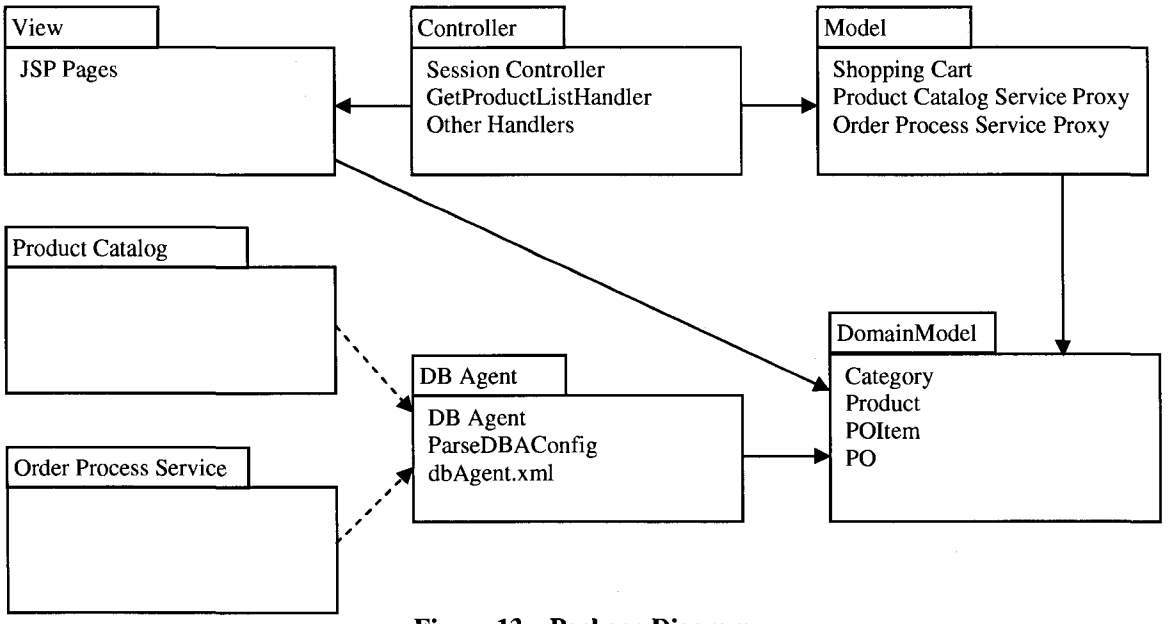

Figure 13 – Package Diagram

4.1.3 Interaction Diagram

This section shows a typical interaction of components of the CD Store application. The discussion is aided by figure 14. It only considers the interactions between the components of the three tiered architecture and not the interactions within each component. In this interaction diagram a user requests to view a list of all the categories of CDs available on the CD Store website (ex. Jazz, Classical, etc.). The user makes the request by clicking on the proper link on the website. The CD Store application processes the request and makes a request to the product catalog service for the category list so that it can to fulfill it. The product catalogue is accessed through a proxy class which handles interaction with the product catalog service. The product

catalog service queries the database agent to retrieve the list of categories. The database agent returns the list of categories to the product catalog service and the response is returned back up the chain to the user (with appropriate processing done at each step in the chain).

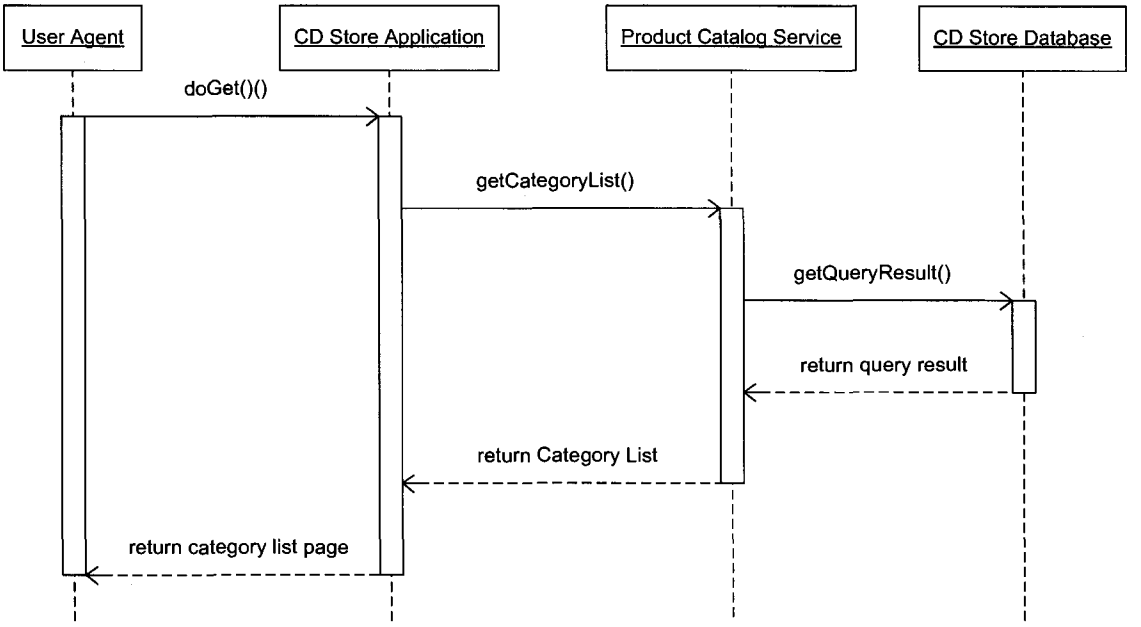

Figure 14 – Interaction Diagram

4.2. Hard Coded TTCN-3 Test Agent Architecture

In this section we outline our manual approach to grey-box integration testing. With this approach, test scripts are written specifically for each test case. We first describe how test cases are written using this approach. We then address the problems for grey-box integration testing outlined in chapter 3 by illustrating how they are handled, namely caching and the correlation gap.

4.2.1 Test Case Definition

The first step to define a test case in TTCN-3 is to create the Master Test Component for it. The Master Test Component coordinates the individual test agents and collects their verdicts once the test is complete.

```
1   testcase CDStoreMTC() runs on MTCType
2   system SystemComponentType {
3      var ServiceAgentType theOrderServiceTest;
4      var CompositeAgentType theUserTest [2];
5      theOrderServiceTest:=
6         ServiceAgentType.create;
7      theOrderServiceTest.start(
8         serviceEventsTest(
9            theExpectedOrderRequests));
10      theUserTest [0] :=
11        CompositeAgentType.create;
12      theUserTest [1] :=
13        CompositeAgentType.create;
14      theUserTest[0].start(User_1_behavior());
15      theUserTest[1].start(User_2_behavior());

16     theUserTest[0].done;
17     theUserTest[1].done;

18     servCoordPort.send("end of test");

19     all component.done;
20  }
```

Figure 15 – Master Test Component

The example in Figure 15 shows the specification of the Master Test Component for a CD Store test case. First, the CD Store Master Test Component is declared (line 1) as a test case. The Order Processing Service Test Agent is then created and started (line

5-9) so that it can be ready to process requests when the User Test Agents start sending requests to the Composite Application. Once the underlying service test agent is started, the User Test Agents can be created and started (line 10-15). These agents simulate users performing behaviors defined in User_1_Behaviour() and User_2_Behaviour() respectively (line 14-15). The Master Test Component then waits for the User Test Agents to complete (line 17-18) then notifies the Order Processing Service Test Agent that the test case has completed (line 18). It then waits until the Order Processing Service Test Agent has completed its work (line 19). The individual test verdicts are implicitly combined by the Master Test Component to form an overall verdict for the pass/failure of the test case as a whole.

Each Test Agent has its own behaviour function. In figure 16 the behaviour function for the Product Catalog Service Test Agent is detailed. We concentrate on the verification of a request for the category list to the Product Catalog Service. The behaviour function runs on the ServiceAgentType (line 2). The ServiceAgentType is the basis for all Service Test Agents. The Test Agent must receive the exact number of messages that are expected. This is done by iterating through its message receiving code block once for each expected message (line 3). For each type of message expected there is a different *alt step* to handle it. Here we show the *alt step* for receiving the categoryList_request (lines 5-18). For each message received by the Service Test Agent, a corresponding response from the underlying service is also expected. This is specified by adding a sub *alt step* for the message. In this case, when the categoryList_request is received a corresponding categoryList_response is expected (line 8) to be received from the underlying service. If no response is received, or an unknown response is received

then the test is stopped and the test verdict is set to failed (line 11-17). If the service receives the correct amount of expected messages without failing then the behaviour is finished and the verdict is set to pass (line 26).

```
1     function productCatalogServiceBehaviour() runs on
2             ptcServiceComponentType {
3             for(i:=0; i < NUM_OF_MESSAGES; i:=i+1) {
4               alt {
5                 [] webPort.receive(categoryList_Request) {
6                       localTimer.start;
7                       alt {
8                         [] servicePort.recieve(categoryList_Response) {
9                             localTimer.stop;
10                        }
11                        [] servicePort.receive {
12                            localTimer.stop;
13                            setverdict(fail);
14                        }
15                        [] localTimer.timeout {
16                            setverdict(fail);
17                        }
18                      }
19                }
20                ...
21                [] webPort.receive {
22                     setverdict(fail);
23                }
24              }
25            }
26            setverdict(pass);
27        }
```

Figure 16 – Service Test Agent Behavior for an Example Request.

4.2.2 Correlation Gap

To handle the correlation gap, the service test agent must verify that the received messages match the expected messages but not in what order they are expected. This is handled as two TTCN-3 templates that represent sets of messages. One template for the expected messages and another for received messages. Using the set matching mechanisms in TTCN-3 we can verify that the proper set of messages has been received without worrying about the order of their reception. In this way two considerations need to be addressed:

- Check if a request arriving at the service test agent was expected for a given test case.
- Check if all requests that are expected for a given test case have actually been received by the test service agent.

In figure 17 we present an updated Product Catalog Service Test Agent which takes the correlation gap into account. The new sections are bolded for clarity. The first consideration is actually addressed naturally by the content of the function productCatalogServiceBehaviour() shown in figure 16. This function is composed of a TTCN-3 alt construct that contains all of the messages that are expected regardless of their order of arrival. Any message that is not in these alternatives would fall into the generic receive statement where a verdict of fail can be set as discussed above.

```
1.  function productCatalogServiceRequestBehaviour() runs on ptcServiceComponentType {
2.    for(i:=0; i < NUM_OF_MESSAGES; i:=i+1) {
3.      alt {
4.        [] webPort.receive(categoryList_Request) -> value msg {
5.            localTimer.start;
6.            receivedMessages := {receivedMessages, msg};
7.            alt {
8.               [] servicePort.receive(categoryList_Response) {
9.                   localTimer.stop;
10.              }
11.              [] servicePort.receive {
12.                   localTimer.stop;
13.                   setverdict(fail);
14.              }
15.              [] localTimer.timeout {
16.                   setverdict(fail);
17.              }
18.           }
19.       }
20.       [] servCoordPort.receive("end of test"){
21.           if(match(expectedMessages, receivedMessages)){
22.               setverdict(pass);
23.             } else {
24.               setverdict(fail);
25.             }
26.           }
27.       }
28.       [] webPort.receive {
29.               setverdict(fail);
30.       }
31.     }
32.   }
33. }
```

Figure 17 – Correlation Gap Handling

The second consideration consists in updating a set of received messages as the messages arrive at the Service Test Agent. Once the test is completed, a final match of the expected versus received sets of messages suffice to conclude that the test has passed or failed. Each time a message is received its value is stored and added the list of received messages (line 6). Once the User Test Agents finish their processing the Master Test Component sends a signal the Service Test Agents to end the test (line 20). At this time the Service Test Agent matches the expected messages against the received messages (line 21) and sets its test verdict appropriately (line 22-25).

4.2.3 Caching

In our implementation caching is handled with the same mechanism as the correlation gap. The list of expected messages used by the Service Test Agents is custom built by the tester. The tester takes into account the particular caching mechanism by editing the list of expected messages appropriately.

For example, the CD Store caches subsequent requests for the category list made during a user session. Therefore a test case which emulates a user making multiple requests for the category list would have only one instance of the categoryList_request in its set of expected messages. This is illustrated in Figure 18 and19 below.

```
expectedMessages := {categoryList_request, categoryList_request};
```

Figure 18 – Expected Requests without Caching Considered

```
expectedMessages :={categoryList_request};
```

Figure 19 – Expected Requests with Caching Considered

4.2.4 Maintenance of Framework as Test Strategy evolves

The hard coded framework we have presented here needs to be manually updated as the test strategy evolves and new tests are added. The structure of the master test component (shown in figure 15) remains the same for different tests. However, when defining new test cases, there are four differences that must be addressed.

1. Specify TTCN-3 templates corresponding to requests/responses
2. Specify custom behaviour functions
3. Edit list of expected requests/responses
4. Specify custom codecs/adapters.

The first step is to define TTCN-3 templates that correspond to the messages involved in the new test case. These data structures are in turn built in two steps:

- First, for each individual user interaction with the composite application, define the request and expected response templates directly associated with it.
- Second, for each corresponding expected request and response at each service test agent, define the message templates associated with them.

The second step is to define the custom behaviour functions. Each test case has a different set of user interactions with the SUT and as a result their behavior must be coded separately. The test case behaviour functions are also built in two steps:

- First, build the user behavior that specifies their interaction with the composite application only.
- Second, build behavior trees for each kind of request/response expected at a given service test agent. Then compose a behavior tree with these individually defined request/response behavior trees.

The third step is to edit the list of expected requests and responses that is used at the end of the test to verify completeness (Section 4.3.3).

Finally, for each message involved in the new test case a new adapter/codecs must be created to handle parsing it into appropriate TTCN-3 templates. The decoders/encoders are written in the GPL that the implementation of TTCN-3 is based on (in our case they are written in Java). This, of course, makes it necessary for the tester to be familiar with the GPL the adapter/codecs are written in. The messages are received from the SUT in raw format and the relevant matching information must be extracted from the raw message and parsed into the relevant TTCN-3 templates. In our experience this can take between 20 and 400 lines of code to accomplish, depending on the complexity of the data that must be extracted from the raw message.

4.3. TTCN-3 Standardized Test Adapter and Codecs

We were able to improve on the approach of section 4.2 and significantly reduce the effort in maintaining the integration framework, through the creation of a generic set of adapters and codes. The benefit of using generic adapter/codecs is that when new test cases are created, the time consuming extra step of creating new custom adapter/codecs does not have to be done. Our solution is based on the XPath query language. XPath is a

language for selecting nodes in an XML document. In addition, XPath may be used to compute values from the content of an XML document. Since both SOAP messages and HTML are based on XML, XPath is a natural way to verify elements of documents in this format and has in fact been used for this purpose in industry. SOAPUI is an open source project to test the SOAP interfaces of web services [SOAPUI08]. They present a black box testing tool similar to OpenSTA where SOAP messages are validated by, among other techniques, the use of XPath expressions. In our case we use XPath to verify the existence of elements in the HTML/SOAP messages involved in the test cases. More robust matching rules can be implemented in XPath syntax since calculations can also be done although this is not covered in this thesis.

For the grey box testing of SOA applications, we have implemented two different types of generic adapter/codecs. The first type is used to communicate with the SUT during the user behaviour section of test cases in conjunction with the Composite Test Agents. This adapter/codec standardizes request/responses for HTML documents and keeps session and HTTP cookie information for each emulated user. The second set is used to interface with the underlying services in the SUT and are used in conjunction with the Service Test Agents. Both requests to services and responses from services are considered.

4.3.1 HTML Adapter/Codec

The generic HTML adapter/codec we created is 800 lines of code and took roughly 80 hours to complete. It has two aspects. The first aspect is the ability to retrieve web pages. This is accomplished by supporting the sending of requests for web pages to

the composite application in the form of URLs. The second aspect is the standardized format for verifying responses, in the form of HTML web pages, from the SUT. Work has been done by others which use htmlUnit in combination with TTCN-3 to test HTML based web applications [Stepien08]. This method takes into account client side scripting. Test agents can perform such actions as invoking JavaScript methods on the client which could result in asynchronous requests to the web application. However, the adapter/codecs which they use are not standardized. Therefore custom ones must be built for each new message exchanged with the SUT. In this thesis we use a different approach which does not take client side scripting into account but lends itself more easily to a generic approach. We do, however, take into account user session state and cookie information by using HTTPUnit [HTTPUnit08] as the base for our generic adapter/codec implementation.

Our adapter/codec uses XPath expressions to verify messages from the SUT. Messages are exchanged with the SUT by using HTTPUnit and messages received from the SUT are forwarded to an XPath validation component of the adapter/codec. This component parses messages from the SUT and matches them against expected messages provided by the abstract TTCN-3 layer. Figure 20 shows the TTCN-3 types that support the generic codec/adapter. The first type, xPathType, (line 1-3) holds a single XPath expression (line 2) to be used in verifying the response from the SUT. The second type, xPathListType (line 4), contains a list of XPath expressions so that multiple expressions can be evaluated to match a single response returned from the SUT. The third type, xPathRequestType, (line 8-11) wraps the actual request that is sent to the codec/adapter layer. This includes the URL which is being requested and the XPath expressions used to

validate the response. The *with* keyword (line 5-7 and line 12-14) specifies which adapter/codec will be used to parse the messages.

```
1   type record xPathType {
2         charstring xPathValue
3   }
4   type set of xPathType xPathListType
5   with {
6         encode "XPATH"
7   }
8   type record xPathRequestType {
9         urlType url,
10        xPathListType xPathList
11  }
12  with {
13        encode "XPATH"
14  }
```

Figure 20 – Generic Types

Figure 21 shows an example TTCN-3 template set based on this format. The first template, categoryListURL (line 1-5), specifies the URL to request from the composite application; in this case the category list page of the CD Store. The second template, categoryListBlues (line 6-8), specifies an XPath to match in the response. In this case we want to verify that 'BLUES' is one of the categories in the html page returned by the SUT. The final template, categoryListRequest (line 10-13), encapsulates both the URL that is being requested and the XPath expressions used to validate the response.

```
1   template urlType categoryListURL := {
2         protocol := "http://",
3         host   := "localhost:8080",
4         file   := "/eStore/Store?action=showCategoryList"
5   }
6   template xPathType categoryListBlues := {
7         xPathValue := "//table/tr[td='BLUES']"
8   }
9   template xPathListType categoryList := {categoryListBlues};
10  template xPathRequestType categoryListRequest := {
11        url := categoryListURL,
12        xPathList := categoryList
13  }
```

Figure 21 – Example Generic Template

4.3.2 Web Service Adapter/Codec

The generic web service adapter codec is roughly 1000 lines of code and took approximately 80 hours to complete. It is similar to the one used for the HTML adapter/codec described in section 4.3.1. XPath is again used to verify the messages from the SUT in much the same way as with the HTML adapter/codec. The main difference is that the Service Tests Agents are passive and simply relay requests/responses between services and service clients. To help accomplish this, we use the open source TCP connection listening tool tcpmon [Tcpmon08] as the base for our service adapter. Tcpmon is an open-source utility for monitoring the data flowing on a TCP connection. It is used by placing it between a client and a server. The client is made to connect to tcpmon, and tcpmon acts as a proxy between the client and server. Tcpmon also handles

http header translation much like a proxy server does. Raw messages are received by tcpmon and the SOAP messages are parsed from this raw format. The parsed messages are then put in a message queue and processed by an XPath based codec similar to the one used in section 4.3.1.

Figure 22 shows the Web Service adapter/codec architecture. First, we intercept messages from the client component (1) and forward the messages to the server component (2). The response from the service is stored and the response is forwarded to the service client (3). Once the message sequence has completed we forward the request/response to the codec (4) to parse it into appropriate TTCN-3 data structures. In this way we do away with building SOAP messages entirely, which can be considerably more complex, and simply verify both the actual requests and responses exchanged with the service.

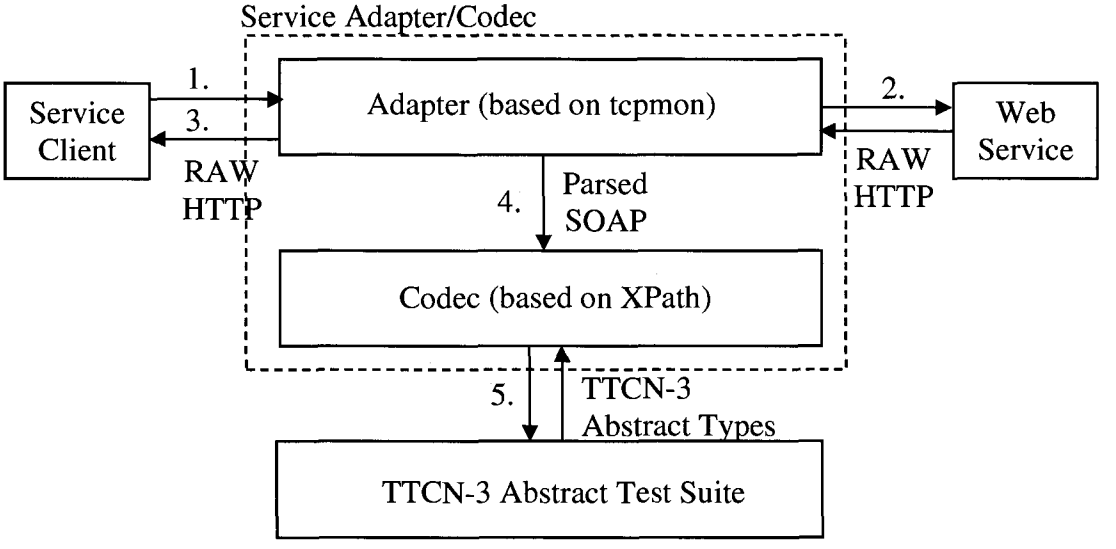

Figure 22 – Web Service Adapter/Codec Architecture

4.4. Model-Driven Test Agent Framework

Generic adapter/codes addresses the fourth point listed in 4.3.4 which contributes to our integration testing requirements (Section 3.1). Specifically, it addresses the issues of lowering user competencies, since the tester does not have to learn the GPL the adapter/codec is written in. The ease of expression for test campaigns is also improved since the tester can express test cases in TTCN-3 only instead of having test cases spread over two languages (TTCN-3 and a GPL). The generic adaptor/codecs relieve some of the burden of maintaining the test framework but the hard-coded test campaigns are complex to express test cases in and hard to maintain. With the manual approach the behaviour of a test case is spread over multiple behaviour sections corresponding with each component in the composite application. This makes it difficult to trace the overall behaviour of test cases and to add new test cases to a test campaign. In order to leverage the framework for different types of applications and make it easier to edit and extend test campaigns it is necessary to create a more direct way to specify the test cases and test campaigns which is clear to understand and edit. In the following sections we outline our abstracted model of the test campaign and discuss the TTCN-3 generation step based on the model.

In figure 23 we introduce an abbreviated model based on the CD Store composite application to illustrate how the model introduced in section 3.6 was implemented in this case study. A more complete one is available in Appendix A. The model is written in XML.

```
<testSuite>
  <component name = "cdStore" type = "HTMLWebService" url =
  "http://localhost:8080/eStore/">
    <messages>
      <message name = "getCategoryList" url="Store?action=showCategoryList">
        <xpath value="/html/body//table/tr/td[a='Blues']"/>
        …
      </message>
    </messages>
  </component>
  <component name = "ProductCatalogService" type = "SOAPWebService" url =
  "http://localhost:8080/eStore/services/ProductCatalogService">
    <messages>
      <message name = "getCategoryList" >
        <request>
          <xpath value = "//getCategoryList"/>
        </request>
   <response>
          <xpath value = "//getCategoryListReturn[catId='BLUES']"/>
          …
   </response>
      </message>
      …
    </messages>
  </component>
  <test-cases>
    <test-case name = "categoryListTest">
      …
      <step component = "cdStore" message = "getCategoryList">
      …
        <step component = "ProductCatalogService" message = "getCategoryList"/>
   …
      </step>
      <step component = "cdStore" message = "getCategoryList"/>
      …
    </test-case>
    …
  …
  </test-cases>
  <facets>
    …
    <validation>
```

```
      <sessions>
        <session iterations = "1">
          <test-case name = "categoryListTest"/>
        </session>
        <session iterations = "1">
          <test-case name = "categoryListTest"/>
        </session>

        ...
   </sessions>
     </validation>
     ...
  </facets>
</testSuite>
```

Figure 23 – Example Declarative Model

4.4.1 Test Suite Node

The root node of our model is the testSuite node. The testSuite node encapsulates all of the elements of the model.

4.4.2 Component Node

The first section of the test suite definition is to define the set of interacting components. In an SOA a component can have two basic types; an HTML based service which acts as a user interface or a SOAP based web service which can be consumed by other interacting components. Each component is uniquely identified by a *name* attribute. Both the HTML and SOAP based component types have the *url* attribute which identifies where they can be reached at. Each component has a set of messages which it can handle. The messages are uniquely identified by a *name* attribute. Each message is composed of a request and a corresponding response. In the case of HTML based components the

request is simply a URL which corresponds to the message denoted by a *url* attribute in the *message* tag. The valid response for an HTML message is defined by a set of XPATH expressions which will be used to validate the response from the SUT.

The *url* attribute of each SOAP based component declares where the web service is located. As with the HTML based components the SOAP based components have a set of messages which it can handle. The main difference is that in a SOAP based component message both the *request* and *response* are defined as a set of XPATH expressions.

4.4.3 Test Cases

Now that each component along with their allowable messages has been defined, we can specify test cases based on these components and messages. The *test-cases* node encapsulates the set of *test-case* nodes we would like to define. The *test-case* node defines a particular test case. Each test case has a set of one or more *step* nodes. Each *step* node has the attributes *component* and *message*. *Step* nodes can be nested within other *step* nodes. The nested list of *step* nodes defines the order in which messages will be exchanged with the SUT for a particular test. For example, the test case *categoryListTest* specifies that the first step is to send the getCategoryList message to the cdStore component. A nested *step* within this *step* specifies that there is an expected request made to the ProductCatalogService to retrieve the category list. Notice that the *categoryListTest* defines the caching mechanism for the getCategoryList request. In this case each user session only retrieves the category list once from the ProductCatalogService. Subsequent calls for getCategoryList do not involve a corresponding call to the underlying service. In

this way test cases can be specified in a central location with the interconnections of the components explicitly defined.

4.4.4 Facets

A test suite may have facets for validation testing, performance testing, security testing, and other types of testing. In this case study we concentrate only on validation testing. The validation test facet is composed of a set of user sessions which correspond to browser sessions (emulated by HTTPUnit in our adapter (section 4.5)). Each session is run concurrently against the SUT and within each session the test cases are run sequentially. Each session defined in the validation facet can be run any number of times to verify that responses are valid at different server loads.

4.5. Generator

In order to leverage the model, a generator was built. The generator has roughly 1400 lines of code and took one month to build. However, once the generator was completed, the entire integration test framework for the CD store case study was generated in a matter of seconds.

The generator takes as its input a model file conforming to the structure outlined in section 4.4 and its output is TTCN-3 code which is based on the generic adapter/codec implementation introduced in section 4.4. The outputted TTCN-3 has the same basic structure as the hard coded approach detailed in section 4.3.

Chapter 5. Evaluation

The results of our case study are used to evaluate our thesis contributions along a number of different dimensions. In section 5.1 and 5.2 we compare our grey box test agent framework against other approaches outlined in the background section as well as our "strawman" white box integration testing. Then in section 5.3 and 5.4 we compare the three evolutions of our grey box test agent framework (hard-coded, generic adaptors and model driven).

5.1. Comparison with Other Approaches

In this section, our grey box test integration testing approach is compared to the other testing approaches. Firstly, we briefly recap the approaches we are comparing ours to which were introduced in section 2.3. We then present a table containing the comparison criteria followed by a description of the table's contents.

5.1.1 White Box Testing

White box testing involves the testing of the smallest units of code in a computer program. In the case of object oriented programming the smallest units of code are taken to be individual methods. Test cases are set up to test each method. The part of the system the method is dependent on is initialized and the method is executed. The state of the system after the method is executed is tested against an expected state. If the expected end state and the actual end state match then the test is deemed to be successful.

Each method is tested in isolation. Other dependent parts of the system such as other objects or calls to outside entities are handled through the use of mock objects. These objects emulate the expected behaviour of the actual objects. With respect to integration testing, the white box approach first tests individual functionalities in the form of methods, then groups these tests together to test individual classes, then class tests are grouped to perform tests on larger and larger sub components until whole components can be tested.

We examine white box testing and how it relates to the integration testing requirements found in section 3.1. It is not intended to be an analysis of the overall abilities of white box testing frameworks.

5.1.2 Black Box Testing

By black box testing we mean to group together the system and integration testing types that are outlined in section 2.3.2 and 2.3.3 respectively. Black box testing refers to testing a system from its bounds. As opposed to 'white box' testing, black box testing ignores the internal structure of system under test is only accessible through its public interfaces. With respect to this analysis we again concentrate on the integration issues found in section 3.1 and how well black box testing methodologies address them.

Comparison Summary

Table 1 summarizes our evaluation in comparing our Grey Box Test Agent Architect to White Box Testing and Black Box Testing specifically as it relates to

integration testing requirements. Each feature listed in the table is explained in the subsections following the table.

Table 1 – Comparison with Other Approaches

Approach / Feature	White Box Testing (JUnit, NUnit)	Black Box Testing (OpenSTA, JMeter, HTTPUnit)	Grey Box Test Agent Architecture
Testing Individual Component Interfaces	Yes	Yes	Yes, but you have to provide a calling component
Component Interaction	Good for testing a single component	Unable to isolate faults/issues	Yes
Performance	Yes, but no ability for cross component	Yes, overall. But not fine grained to each component	Yes, built into the methodology
Multi user scenarios	No	Yes, but unable to isolate faults/issues	Yes
Testing Components internal logic	Yes	No	No
User competencies	Must be able to program in the GPL of the SUT (e.g. Java for JUnit)	Flexible toolsets, low requirements for user knowledge. Minimal knowledge of HTTP+SOAP requests.	Minimal knowledge of XPATH and XML for configuration of test scripts* and TTCN-3 execution environment.

*Future work on IDE would eliminate the configuration of XML test scripts.

5.1.3 Individual Component Interfaces

Both white box and black box are capable of testing individual component interfaces 'out of the box' whereas with grey box test agent architecture requires a master test component and test agents to be built. However, with our model driven test generator, these are automatically generated.

5.1.4 Component Interaction

By testing component interaction we mean that when components are interacting we can test that they behave to specification. With white box testing, we can test a single component. By design, underlying components are emulated by mock objects so that individual components can be thoroughly tested in isolation. Therefore, component interaction is not considered and underlying components are assumed to behave properly. However, in multi component systems where underlying components can change independently of the system as a whole this does not always hold true. Using black box testing we can similarity verify the correctness of a single component interface. However, with this approach, even if we do discover a fault with the component we cannot locate the fault to a particular component or component interaction in the system under test and more testing has to be done to locate the fault. Also, it is not guaranteed that a fault will manifest itself at the interface of the component being tested. The grey box test agent architecture enables one to locate precisely in which component or component interaction a fault has occurred, thus saving effort. Furthermore, faults which manifest only during the interaction with underlying components and not at the interface which would have been used for the equivalent black box test can also be found with our grey box approach.

5.1.5 Performance

To measure performance we measure how long a particular request takes to complete. Only the total length of a request can be measured with both white box and black box testing. During multi component interactions the grey box test agent architecture can accurately capture the time taken to process requests at each component involved in the test. However, because of the correlation gap it is not possible to measure the performance of a particular request/response with respect to corresponding request/responses at other components.

5.1.6 Multi-user Scenarios

By multi user scenarios we mean that the system is tested based on the emulation of multiple simultaneous users. Since unit tests are written in a GPL it is theoretically possible to write multi user scenarios in most unit testing frameworks. In practice, though, this is a difficult and complex task that cannot be done reliably. With black box testing tools such as OpenSTA, simulating multi user scenarios is well supported. However, faults cannot be isolated to any particular component. The grey box test agent architecture supports multi user scenarios more completely. It has the same ability to test multi user scenarios as the black box approach with the added feature of being able to localize faults to a particular component or component interaction. Furthermore, it can uncover faults which are only apparent during the interaction with underlying services.

5.1.7 Component Internal Logic

By internal logic we mean the verification of algorithms and classes that an individual component implements to fulfill its purpose. Unit testing is designed for just this. Fine grain tests are written to verify each algorithm/class that is implemented within a particular component. Black box testing cannot verify the internal logic of components. If a fault with the system under test is found, further testing is needed to locate the fault to a particular component and then to a particular piece of internal logic of that component. Grey box testing also cannot verify the internal logic of components although the step that occurs in black box testing of locating which component a fault is located is not needed.

5.1.8 User Competencies

In white box testing the tester must by nature be able to program in the GPL of the SUT (e.g. Java for JUnit). Ideally, the tester would be the same developer writing the code and developing their tests in parallel. Black box testing on the other hand has many tools to assist the tester. Black box integration testers have a low requirement for knowledge. They only need minimal knowledge of HTTP and SOAP protocols. To use our model-driven grey box test agent architecture for integration testing the testers must currently have minimal knowledge of XPATH and XML to write test scripts although future work on an IDE which generates test scripts based on a graphical interface would eliminate this. The user also must have competencies with the test execution environment used to run TTCN-3 test campaigns.

5.2. Critical Integration Testing Issues

Table 2 – Integration Issues

Approach / Critical Issue	White Box Testing (JUnit, NUnit, HTTPUnit)	Black Box Testing (OpenSTA, JMeter)	Grey Box Test Agent Architecture
Correlation Gap	No	No	Yes
Fault Isolation	No	No	Yes
Caching	No	No	Yes

Both black box and white box testing do not address the most critical integration testing issues, namely fault isolation and multi-user and multi-component issues like Correlation Gap and Caching. Black box testing does not address the correlation gap, fault isolation, and caching issues since the system under test is treated as a black box and each of these issues originate from some aspect of the internal structure of the system and cannot be deduced from the outside of the system. White box testing does not address these issues since it is designed to test components, methods, classes, etc. in isolation of the system as a whole. The grey-box test agent architecture can isolate faults to a particular component and can verify that caching mechanisms are behaving properly while taking into account the correlation gap. It does this by keeping track of the messages exchanged between components and verifying both that individual messages are valid and that the complete set of messages exchanged with a component is what was expected.

5.3. Evaluation of Efforts and Complexity

The integration testing framework described in this thesis evolved from a manual or hard-coded TTCN-3 implementation of a grey box test agent architecture to a more general framework that leveraged generic adapter/codecs to a declarative model-driven integration testing framework. In the next two sections we analyze and compare each of these three stages in the evolution of our framework. The model driven stage is not necessarily better than the other approaches in every case. There is a set of trade offs detailed in this section that give each approach its own audience of users and purpose.

Table 3 – Evaluation of Efforts and Complexity

Approach / Metric	Manual or Hard-Coded	Generic Adapter/Codec	Model Driven Generated Framework
Competencies of User	Must be versed in TTCN-3 and the GPL that the Adapters/Codecs are written in	Must be versed in TTCN-3 + minimal knowledge of XPath	Must be versed in the modeling language used by the approach + minimal knowledge of XPath
Extensibility of Test Framework	Most flexible approach, can be adapted to any multi component system.	Flexible test design, but interface to SUT is limited to those implemented by the adapter/codec.	Inflexible within a specific model test generation and adapter/codec are well defined and extensible.
Effort for New SUT	High	Lower	Lowest
Maintenance Effort as SUT Evolves	High – Ad-hoc creation of TTCN-3 test scripts and adapter/codecs	Lower – Ad-hoc creation of TTCN-3 test scripts	Lowest – Editing of model file and regeneration of test campaign.

5.3.1 Competencies of User

To write test scripts with the manual approach the tester must be competent with writing TTCN-3 test scripts and also the general purpose language the adapter/codecs are written in. With the generic adapter/codecs the tester must only be competent in writing TTCN-3 scripts and in basic XPath syntax. With the model driven approach the tester does not need to be competent with either TTCN-3 or the general purpose language for the adapter/codecs. However, they must be competent in the modeling language used to express the test scripts and in basic XPath syntax. Our choice of XML as the model language was natural for our implementation because of our experience with it, however a domain-specific language could be developed which would be more specific to the application and simpler to learn. With all of the approaches the user must also be familiar with the TTCN-3 test campaign execution environment to actually run the tests.

5.3.2 Extensibility of Test Framework

By extensibility we mean that when a test case is needed which cannot be completed within the bounds of the already existing test framework, we would like to know how much effort is needed to implement that test case. As an example, if the testers would like to evolve the test strategy into new domains such as security testing.

First we consider the manual approach. Since the adapter/codecs and the TTCN-3 test scripts are both manually written for new test cases this is the most adaptable approach. A new adapter/codec and a new TTCN-3 test script would have to be written that satisfy the requirements of the new test case. With the generic adapter/codec approach consideration would have to be taken to implement a new generic codec/adapter

that could support the new test case or goal. This would be more difficult but once done it would be reusable for similar tests in the future. With the model based approach first a generic adapter/codec would have to be created then if the test cases must also incorporate new TTCN-3 test script types an addition to the generator would have to be made that supports the generation of the new TTCN-3 test scripts and finally an update to the model would also have to be made to support the new test type.

In general an iterative approach would be the best way to incorporate new test case types into the system. First a manual approach would have to be taken and tested. Once verified to be working the adapter/codecs should be generalized if possible, and finally the model and generator would have to be updated to handle the new test type in a systematic way.

5.3.3 Effort for New SUT

When designing a new system we would like to know the effort needed to create the new test scripts. With the manual approach, the entry level is high. New adapters must be created for every known component of the system and new codecs must be added for each known message. After this, TTCN-3 test scripts must be written to apply the known use cases in the system. With the generic adapter/codec approach this effort is reduced since new adapters and codecs do not have to be created. With the model based approach, new adapter/codecs do not have to be created. The tester must describe the system in the modeling language and set up the test scripts in an abstract sense. The use cases written to describe the system can be transformed into the modeling language easily.

5.3.4 Maintenance Effort as SUT Evolves

As the system under test evolves to handle new use cases the test scripts must reflect these changes. New use cases may also include the addition of new components to the system architecture. With the manual approach this involves writing new TTCN-3 test scripts to describe the test case and new adapter/codecs to handle the new messages involved in the test case. With the generic adapter/codec it is only necessary to create the new TTCN-3 test scripts. With the model based approach it is necessary to add the new test case and new involved components to the model and regenerate the TTCN-3 test scripts.

5.4. Effort to Debug Tests

Table 4 – Effort to Debug Tests

Approach / Metric	Regular TTCN-3 Debugging	Regular TTCN-3 + Generic Adapter/Codec	Model based
Effort to Debug Test Scripts	Hard	Hard	Easier
Effort to Debug Test Framework	Easiest	Harder	Hardest
Error Rate of Test Scripts	Often in both TTCN-3 Scripts and adapters/codecs	Often in TTCN-3 Scripts but rare in adapters/codecs	Rare in both TTCN-3 scripts and adapter/codecs

5.4.1 Effort to Debug Test Scripts

When an error or bug in a test script is uncovered we would like to know how much effort is needed to debug the test script assuming there is no problem with the

underlying test system and the error is located somewhere in the logic of the definition of the test case itself.

To debug test scripts written both with the regular method and the generic adapter/codec method it is difficult to locate where an error in a test script is located. This is because the test case definition is spread across the behaviours of each component involved in the test case. The method of finding the error in a test script involves

- Verifying that the expected request/response data is properly specified in TTCN-3
- Verifying that the behaviours of each component adhere to the intent of the test case.
- Verifying the complete list of messages expected at each service is in accordance with the test case.

On top of this, with the regular method it must also be verified that the codec/adapter is properly formatting the request/response objects used by the TTCN-3 test case.

On the other hand, with the model based approach, the tester must verify that

- The messages are properly specified in the model
- The test case in the model properly states the interaction between the components of the SUT

Most importantly, the test case behaviour is located in a single definition that takes into account the behaviour of each component involved in the test. This makes it much easier to follow the definition of a test case and therefore makes errors more easier to locate.

5.4.2 Effort to Debug Test Framework

If a problem is not found with the test scripts themselves then the error may be with the underlying test framework. It is important to know how much effort it would take to debug the framework itself. With the manual approach, both the TTCN-3 test scripts and the adapter/codecs are custom built for each test script, therefore there is little distinction between debugging the framework and debugging the individual test scripts when using this method. They are one and the same.

For test scripts written with generic adapter/codecs, the underlying framework consists of the generic adapter/codecs. It is unlikely that a bug in the test case would originate from the adapter/codecs. Since they were developed earlier and reused many times bugs would be found and fixed once and not occur again. However, when a bug is found in the generic adapter/codec it must be addressed by someone familiar with the internals of the adapter/codec. This would, of course, involve the tester being familiar with the GPL the adapter/codec was written in, which is not always the case (see section 5.3.1). The generic adapter/codecs are more complex than the custom ones used in the regular approach and therefore more complex to debug.

For test scripts written with the model based approach, the underlying framework consists of the generic adapter/codecs and the generator. The method associated with

generic adapter/codecs also applies to the model based approach. Additionally, since the TTCN-3 test scripts are generated then a bug in the TTCN-3 code could also indicate a bug in the generator. Bugs in the TTCN-3 code must therefore be addressed by a developer familiar with the generator. It is much more likely that the bug can be traced back to a fault in the model the tester has created and, in general, the tester would concentrate only on the model of the test script written in the modeling language. The generator is complex and would be relatively difficult to debug compared to the regular approach.

5.4.3 Error Rate of Test Scripts

The error rates in each of the three methods are different. For the manual approach since both the TTCN-3 test scripts and the adapter/codecs are customized for each test script, errors in both the TTCN-3 and adapter/codec are likely to occur. In the second method, with generic adapter/codecs, errors in the adapter/codec are rarer since the adapter/codec has been written and tested in advance. In the model based approach, the adapter/codecs have the same error rate as with the second approach. Additionally, the error rates for the TTCN-3 code is lower since it is generated based on the model.

Chapter 6. Conclusions

6.1. Summary of Contributions

In section 1.2, we presented the key contributions of our research. We discuss the impact of these contributions in this section.

Contribution 1: Grey-box test agent architecture for integration testing of composite applications.

The grey-box test agent architecture we proposed in chapter 3 and its implementation outlined in chapter 4 was successfully implemented. The first attempt was as a hard coded TTCN-3 approach which addressed the main integration issues outlined in chapter 2. The basic grey-box test agent architecture was able to address the problems of testing individual components through the use of custom adapter/codecs which could communicate with the SUT on a message by message basis, Testing component interaction was also successfully done by relaying messages between components to our test agents. This allowed us to locate where faults occur within the composite application. Finally, we were able to test multi-user scenarios by supporting multiple concurrent test agents. All of these were implemented under the constraints of the caching and correlation gap issues. However, the grey-box architecture was cumbersome to express tests in and repetitive. This is what motivated our model based approach and generic adapter/codecs.

Contribution 2: Evaluation of TTCN-3 as a standards based foundation for our architecture.

TTCN-3 was successfully leveraged as the base for our grey box architecture. The facilities of TTCN-3 for separation of concerns between test case definition and communication with the system under test were successfully used to create a generalized reusable infrastructure to support the 'grey box' integration testing approach proposed in this thesis. More specifically, the following requirements (section 3.1) for an integration testing framework were addressed:

- testing individual component interfaces
- testing component interaction
- testing multi user scenarios

The testing of individual component interfaces is possible by creating test cases that interface with a single component. The testing of component interaction was successfully implemented by our service test agents. We were able to localize faults within the composite application and testing of multi user scenarios are supported by the multi threaded aspect of TTCN-3 which allows multiple concurrent test agents to be active at the same time.

The problems associated with the 'grey box' testing architecture (namely caching and the correlation gap) were also successfully addressed by using TTCN-3's extensive set matching facilities.

Contribution 3: A set of generic adaptor/codes for testing web applications (HTML, web services).

A set of generic adapter/codecs were created for the grey box integration testing composite applications. This includes adapter/codecs for interfacing with HTML over HTTP and SOAP based services. In the case of HTML based interfaces, we integrated the functionality of HTTPUnit into our adapter/codec in order to maintain user session state. In the case of our service adapter/codec TCPMon was leveraged to place test agents between consumer and service. XPath was leveraged in a novel way to support the validation of messages in both the HTML and SOAP based adapter/codecs. In our experience, hard-coded adapter/codecs can vary between 20 and 400 lines of code for each message exchanged with the SUT. With our generic adapter/codecs this has been cut drastically. In the case of responses from composite applications and requests/responses to services it has been cut to one line of code for each element in a message that must be verified and in the case of requests to a composite application to one line which represents the URL requested. The generic adapter/codecs also alleviate the problems of managing and debugging the inevitably large set of custom adapters/codecs that must be created for a large composite application with the hard coded approach. This has of course been at the cost of flexibility in our message matching mechanism. One solution to this is to have a library of generic adapter/codecs that can be extended to support different matching mechanisms and test goals.

Contribution 4: A declarative model and notation for defining SOA test frameworks and scripts.

An extensible model definition for expressing integration test suites was defined which is used to express the system architecture of composite applications, test cases to

be run against the system under test and test campaigns to test for different objectives (validity, performance, security, etc..) The model was defined in XML. This was successful for greatly reducing the complexity of specifying and maintaining a test campaign for composite applications. When hard-coded in TTCN-3 it took 5 times the number of lines of code to fully specify a test campaign as with our notation (>500 lines of TTCN-3 versus 105 lines of XML) in addition to not having to write custom adapter/codecs by using the generic ones which saved us an estimated 1000 lines of code. The other great value of our notation is that test cases are specified in an intuitive way that allows their definition to be expressed in a single set of nested steps whereas with the manual approach the behaviour of a test case is spread over multiple behaviour sections corresponding with each component in the composite application. Of course, XML is tedious for humans to interact with, but if suitable supporting tools and interfaces could be provided this would be a much better approach to integration testing.

Contribution 5: A generator of executable or run-time TTCN-3 test frameworks from declarative test models.

Based on a combination of our hard-coded TTCN-3 approach and our declarative model a generator of our model which generates executable TTCN-3 was successfully implemented.

6.2. Future Work

6.2.1 Caching

Caching requests from services is an important aspect of composite applications. There are many strategies used by composite application developers for caching responses from services. For example some strategies may be:

- Cache a response from a service once its request is made once for each user session
- Cache a response from a service once its request is made once
- Cache a response from a service for a pre determined length of time
- Etc.

Our hard coded framework has the flexibility for the test developer to express all of these caching mechanisms and supports the matching mechanisms to identify the caching events once they have been expressed by the test developer. Our model based framework can also be used with test cases that take caching into account, however, it does not address each of these varied caching issues separately in systematic way. For example, a test case can be created which have multiple requests that would result in caching behaviour where the caching mechanism is expressed as a request to the underlying service on the first request and in subsequent requests no request to the underlying service is denoted. However, it would be a substantial improvement for our model based testing framework to support these varied caching mechanisms in a more systematic way. This would include two aspects; a comprehensive study on the categorization of caching strategies and extending our model based framework to support

the direct specification of how the caching of particular requests to services are handled. This way the caching mechanism could be expressed in the message definition section instead of in the test case section of the model.

6.2.2 Model Language

As stated in section 6.1 our model based definition is expressed in XML. XML is tedious for humans to interact with, but if suitable supporting tools and/or graphical interfaces could be provided this would be a much more accessible approach to integration testing. The Unified Modeling Language (UML) would be a particularly good candidate as a base for such tools because of its generality and flexibility in describing software systems. The main entities that may easily be modeled are components, messages, and test cases. Components in our model are most analogous to the class entity in UML since they contain a set of messages and since messages are most analogous to class methods. Test cases also have a suitable analogy in UML; Sequence diagrams can model test cases by expressing component interaction and specifically which messages are being exchanged between components during a test case and in what order. These diagrams would have to be enhanced with application specific information but in our opinion the preceding entities could quite easily be written in UML and a relatively simple translator could generate the XML model described in this thesis. The test campaign section of our model may be more difficult to express with known graphical tools and may still have to be expressed with the use of text based configuration assisted by the above stated UML entities.

References

[Amyot03] D. Amyot, X. He, Y, He, D. W. Cho, Generating Scenarios from Use Case Map Specifications, QSIC'03

[Amyot05] D. Amyot, J.-F. Roy, and M. Weiss, UCM-Driven Testing of Web Applications, to appear in: 12th SDL Forum (SDL'05), Grimstad, Norway, June 2005.

[Bai06] X.Bai, G. Dai, D. Xu, W. Tsai, A Multi-Agent Based Framework for Collaborative Testing on Web Services, The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, pp. 205-210, 2006.

[Bertolino06] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, Architecting Systems with Trustworthy Components, number 3938 in LNCS. Springer-Verlag, 2006

[Binder99] R. V. Binder, Testing Object-Oriented Systems - Models, Patterns, and Tools, Addison-Wesley, 1999.

[Buhr98] R.J.A. Buhr, Use Case Maps as Architectural Entities for Complex Systems, Transactions on Software Engineering, pp. 1131-1155, August 1998.

[Canfora06] G. Canfora and M. D. Penta. Testing services and service-centric systems: Challenges and opportunities. IT Professional, 2006, 8(2):10–17.

[ETSI08] ETSI ES 201 873-1, "The Testing and Test Control Notation version 3, Part1: TTCN-3 Core notation, V3.4.1", March 2008

[FITNESS08] FitNess Project, http://www.fitnesse.org/, last retrieved: Nov 15, 2008

[Fowler07] M. Fowler, Mocks Aren't Stubs, http://martinfowler.com/articles/mocksArentStubs.html, last retrieved: Nov 15, 2008

[Hamill04] P. Hamill., Unit Test Frameworks, O'Reilly, November 2004.

[HTTPUnit08] HTTPUnit Project, http://httpunit.sourceforge.net/, last retrieved: Sep 18, 2008.

[JMeter08] JMeter Project, http://jakarta.apache.org/jmeter/, last retrieved: Sep 18, 2008

[JUNIT08] JUnit Project http://sourceforge.net/projects/junit/, last retrieved: Sep 18, 2008

[Lucas00] H.C. Lucas Jr., Information Technology for Management, Seventh Ed., Irwin/McGraw-Hill, Boston, 2000

[Mackinnon00] T. Mackinnon, S. Freeman, and P. Craig. Endo-Testing: Unit Testing with Mock Objects. Proc. of Extreme Programming and Flexible Processes in Software Engineering (XP2000), 2000.

[MSC04] ITU-T. Recommendation Z.100 (04/04), Message Sequence Chart (MSC). International Telecommunication Union, Geneva

[Mulvihill03] Mulvihill, B., Generation of TTCN Test Cases from Use Case Map Scenarios. Graduate student report, School of Information Technology and Engineering, University of Ottawa, Canada. Supervisor: Daniel Amyot, 2003

[NUNIT08] NUnit Project http://www.nunit.org/index.php, last retrieved: Sep 18, 2008.

[Oasis07] OASIS: Web Services Composite Application Framework, http://www.oasis-open.org/committees/ws-caf/, last retrieved: Sep 18, 2008

[OMG08] http://www.uml.org/, last retrieved: Sep 18, 2008

[OpenSTA08] http://www.opensta.org/, last retrieved: Sep 18, 2008

[Papazoglou03] M. P. Papazoglou, Service-Oriented Computing: Concepts, Characteristics and Directions, 4th International Conference on Web Information Systems Engineering (WISE'03) , Rome, Italy, 2003.

[Peyton08] L. Peyton, B. Zhan, B. Stepien., A Case Study in Integrated Quality Assurance for Performance Management Systems, MSVVEIS08, pp. 129-138, 2008.

[Probert99] RL Probert, AW Williams, Fast Functional Test Generation Using an SDL Model, Proceedings of the 12th International Conference on Testing Communicating Systems, IWTCS'99, Klummer Publishers, Hungary, 1999.

[Probert04] R. L. Probert, Pulei Xiong, Bernard Stepien, Life-cycle E-Commerce Testing with OO-TTCN-3, FORTE'04 Workshops proceedings, September 2004

[Rankin02] Rankin C., The Software Testing Automation framework, IBM Systems Journal, Software Testing and Verification, Vol. 41, No.1, 2002

[SDL96] ITU-T. Recommendation Z.100, Specification and Description Language, revised 1996

[SOAPUI08] SoapUI Project, http://www.soapui.org/, last retrieved: Sep18, 2008

[Stepien08] B. Stepien, L.Peyton, P.Xiong, Framework Testing of Web Applications using TTCN-3, International Journal on Software Tools for Technology Transfer (STTT), Volume 10, Number 4, August 2008

[TCPMon08] TCPMon Project, https://tcpmon.dev.java.net/, last retrieved: Sep 18, 2008

[Tsai06] W.T. Tsai, Q. Huang, B. Xiao, Y. Chen, Verification Framework for Dynamic Collaborative Services in Service-Oriented Architecture, Sixth International Conference on Quality Software (QSIC'06), pp. 313-320, 2006.

[Troschütz07] Web Service Test Framework with TTCN-3, Master's Thesis, University of Gottigen, ISSN: 1612-6793, 2007

[W3C01] W3C Working Group, Web Services Description Language (WSDL) 1.1, Note 15 March 2001, http://www.w3.org/TR/wsdl, last retrieved: Sep 18, 2008

[W3C04] W3C Working Group, Web Services Architecture, Note 11 February 2004, http://www.w3.org/TR/ws-arch, last retrieved: Sep 18, 2008

[W3C07] W3C Working Group, SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), Recommendation 27 April 2007 http://www.w3.org/TR/soap12-part1/, last retrieved: Sep 18, 2008

[Wong97] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A Study of Effective Regression Testing in Practice. Proceedings of the Eighth International Symposium on Software Reliability Engineering, pp 230-238, November 1997.

[Xiong05] P.Xiong, R. L. Probert, B. Stepien , An Efficient Formal Testing Approach for Web Services with TTCN-3, SoftCom 2005, September 2005

Appendix A – CD Store Test Suite Model

In this section an example test suite for the CD store is given. It demonstrates how caching, the correlation gap, multi-user scenarios, and test facets are represented in our model. It consists of three components; the CD Store web application which is accessed through a browser and serves HTML over HTTP and two web services, the product catalog service and the order process service which process SOAP requests. Each of the components can accept a set of messages. The messages are used in conjunction with each other to define the behaviour of test cases and finally test cases are run sequentially within a session defined in a facet. There may be many concurrent sessions during a particular test.

```
<testSuite>
   <!—A componenent within the composite application. In this case a web server serving
   HTML formatted web pages -->
   <component name = "cdStore" type = "HTMLWebService" url =
   "http://localhost:8080/eStore/">
      <!—The complete set of messages that the component can accept during all of the
      test cases -->
      <messages>
         <!—Each message has a request and a response, in this case the request is a url
         and the valid response is a set of XPath expressions that will be matched
         against the actual response -->
         <message name = "getIndex" url="index.jsp">
            <xpath value="/html/body//table/tr[td='Ideal CD Store']"/>
            <xpath value="/html/body//table/tr/td[a='Main Page']"/>
            <xpath value="/html/body//table/tr/td[a='Category List']"/>
            <xpath value="/html/body//table/tr/td[a='Shopping Cart']"/>
         </message>
         <message name = "getCategoryList" url="Store?action=showCategoryList">
            <xpath value="/html/body//table/tr/td[a='Blues']"/>
            <xpath value="/html/body//table/tr/td[a='Classical']"/>
            <xpath value="/html/body//table/tr/td[a='Jazz']"/>
            <xpath value="/html/body//table/tr/td[a='Opera']"/>
            <xpath value="/html/body//table/tr/td[a='Pop']"/>
            <xpath value="/html/body//table/tr/td[a='Rock']"/>
         </message>
         <message name = "getCdListBlues"
         url="Store?action=showCdList&amp;categoryid=BLUES">
            <xpath
            value="/html/body//table/tr/td/a[@href='Store?action=showCdDetail&amp;product
            id=ITEM-1']"/>
            <xpath
            value="/html/body//table/tr/td/a[@href='Store?action=showCdDetail&amp;product
            id=ITEM-2']"/>
         </message>
         <message name = "viewCDBlues1"
         url="Store?action=showCdDetail&amp;productid=ITEM-1">
            <xpath
            value="/html/body//table/tr/td/a[@href='http://localhost:8080/eStore/Store?ac
            tion=addItemToShoppingCart&amp;productid=ITEM-1']"/>
```

```
      </message>
      <message name = "addCdToCartBlues1"
      url="Store?action=addItemToShoppingCart&amp;productid=ITEM-1">
         <xpath value="/html/body//table/tr[td='Martin Scorsese Presents the
         Blues']"/>
         <xpath value="/html/body//table/tr[td='1']"/>
         <xpath value="/html/body//table/tr[td='18.98']"/>
      </message>
      <message name = "removeCdFromCartBlues1"
      url="Store?action=removeItemFromShoppingCart&amp;productid=ITEM-1">
         <xpath value="/html/body//table/tr[td='The shopping cart is empty.']"/>
      </message>
   </messages>
</component>
<!-A web service component which is communicated with using SOAP calls -->
<component name = "ProductCatalogService" type = "SOAPWebService" url =
"http://localhost:8080/eStore/services/ProductCatalogService">
   <!-The set of messages accepted by this component for the tests -->
   <messages>
      <!-With web-service components requests and responses are validated based on the
      supplied XPath expressions -->
      <message name = "getCategoryList" >
         <request>
            <xpath value = "//getCategoryList"/>
         </request>
         <response>
            <xpath value = "//getCategoryListReturn[catId='BLUES']"/>
         </response>
      </message>
      <message name = "getCDListBlues">
         <request>
            <xpath value = "//getProductList[catId='BLUES']"/>
         </request>
         <response>
            <xpath value = "//getProductListReturn[catId='ITEM-2']"/>
         </response>
      </message>
      <message name = "getCDBlues1">
         <request>
            <xpath value = "//getProductDetail[productId='ITEM-1']"/>
         </request>
         <response>
            <xpath value = "//getProductDetailReturn[prodName='Martin Scorsese
            Presents the Blues']"/>
         </response>
      </message>
   </messages>
</component>

<component name = "OrderProcessService" type = "SOAPWebService" url =
"http://localhost:8080/eStore/services/OrderProcessService">
   <messages>
      <message name = "getCategoryList" >
         <request>
            <xpath value = "//getCategoryList"/>
         </request>
         <response>
            <xpath value = "//getCategoryListReturn[catId='Blues']"/>
         </response>
      </message>
      <message name = "getCDListBlues">
         <request>
            <xpath value = "//getProductList[catId='BLUES']"/>
         </request>
         <response>
            <xpath value = "//getProductListReturn[catId='ITEM-1']"/>
         </response>
      </message>
      <message name = "getCDBlues1">
         <request>
            <xpath value = "//getProductDetail[productId='ITEM-1']"/>
         </request>
         <response>
            <xpath value = "//getProductDetailReturn[prodName='Martin Scorsese
            Presents the Blues']"/>
         </response>
      </message>
   </messages>
</component>
```

```
   <!-Test cases are defined as a set of messages that a single entity is exchanging with
   the SUT. The script writer only defines the interaction between components and the
   generator will calculate which messages are expected at which services -->
   <test-cases>
      <test-case name = "customerWorkflowTest">
         <!-This step does not involve any calls to other components -->
         <step component = "cdStore" message = "getIndex"/>
         <!-This step in the test case is associated with an expected request at an other
         component hence the sub step defined within -->
         <step component = "cdStore" message = "getCategoryList">
            <step component = "ProductCatalogService" message = "getCategoryList"/>
         </step>
         <step component = "cdStore" message = "getCdListBlues">
            <step component = "ProductCatalogService" message = "getCDListBluest"/>
         </step>
         <step component = "cdStore" message = "viewCDBlues1">
            <step component = "ProductCatalogService" message = "getCDBlues1"/>
         </step>
         <step component = "cdStore" message = "addCdToCartBlues1"></step>
         <step component = "cdStore" message = "removeCdFromCartBlues1"></step>
      </test-case>
         <!-This test case shows how we handle caching. Here we expect the category list
         will already be cached at the cd store component, concequently there is not call
         made to the underlying product catalog service as with the preceding test case -
         ->
         <test-case name = "categoryListCacheTest">
            <step component = "cdStore" message = "getCategoryList">
            <step component = "ProductCatalogService" message = "getCategoryList"/>
         </step>
               <step component = "cdStore" message = "getCategoryList">
         </step>
                </test-case>
   </test-cases>

   <!- Each facet defines a different test goal (validation, performance, security, etc.)
   -->
   <facets>
      <!-The validation facet is concerned with validating that request/responses match
      what was expected -->
      <validation>
         <!-In this test two concurrent user sessions are defined -->
         <sessions>
            <!-Each session can be iterated any number of times -->
            <session iterations = "1">
               <test-case name = "customerWorkflowTest"/>
               <!-This test case verifies that caching of the category list was properly
               implemented -->
               <test-case name = "categoryListCacheTest"/>
            </session>
            <!-The second user session -->
            <session iterations = "1">
                <test-case name = "categoryListCacheTest"/>
            </session>
         </sessions>
      </validation>
   </facets>
</testSuite>
```

Appendix B - Framework Definitions

In this section we list the entities referred to in Chapter 4 but not defined. The first listing shows the base TTCN-3 types used in our framework and the second listing shows referenced functions that were not defined.

```
module WebSOATypes {
  type record urlType {
     charstring  url
  }
  type record xPathType {
     charstring xPathValue
  }
  type set of xPathType xPathListType
  with {
     encode "XPATH"
  }
  type record xPathRequestType {
     urlType url,
     xPathListType xPathList
  }
  with {
     encode "XPATH"
  }
  type port webPortType message {
     out xPathRequestType;
     in xPathListType;
  }
  type component ptcWebType {
     port webPortType webPort;
     timer localTimer := 6.0;
  }
  type record serviceLocationType {
```

```
        charstring server,
        charstring remotePortNum,
        charstring localPortNum
    } with {
        encode "service"
    }
    type record requestToService {
        xPathListType xpaths
    }
    type record responseFromService {
        xPathListType xpaths
    }
    type port servicePortType message {
        out serviceLocationType;
        in  requestToService, responseFromService;
    }
    type component ptcServiceComponentType {
        port servicePortType servicePort;
        timer localTimer := 6.0;
    }
    type component mtcType {}
}
```

Figure 24 - Base Types

```
function User_1_behavior() runs on ptcWebType {
  var integer i := 0;
  for(i:=0; i < 1; i:=i+1) {
    webPort.send(getIndexRequest);
    localTimer.start;

    alt {
      [] webPort.receive(getIndexXPaths) {
        log("Web client 0 has received getIndex");
        localTimer.stop;
        setverdict(pass);
      }
      [] webPort.receive {
        log("Web client 0 has receive wrong response.");
```

```
      setverdict(fail);
   }
   [] localTimer.timeout {
      log("Web client 0 has timed out.");
      setverdict(fail);
   }
}
webPort.send(getCategoryListRequest);
localTimer.start;

alt {
   [] webPort.receive(getCategoryListXPaths) {
      log("Web client 0 has received getCategoryList");
      localTimer.stop;
      setverdict(pass);
   }
   [] webPort.receive {
      log("Web client 0 has receive wrong response.");
      setverdict(fail);
   }
   [] localTimer.timeout {
      log("Web client 0 has timed out.");
      setverdict(fail);
   }
}
webPort.send(getCdListBluesRequest);
localTimer.start;

alt {
   [] webPort.receive(getCdListBluesXPaths) {
      log("Web client 0 has received getCdListBlues");
      localTimer.stop;
      setverdict(pass);
   }
   [] webPort.receive {
      log("Web client 0 has receive wrong response.");
      setverdict(fail);
   }
   [] localTimer.timeout {
      log("Web client 0 has timed out.");
      setverdict(fail);
   }
```

```
}
webPort.send(viewCDBlues1Request);
localTimer.start;

alt {
   [] webPort.receive(viewCDBlues1XPaths) {
      log("Web client 0 has received viewCDBlues1");
      localTimer.stop;
      setverdict(pass);
   }
   [] webPort.receive {
      log("Web client 0 has receive wrong response.");
      setverdict(fail);
   }
   [] localTimer.timeout {
      log("Web client 0 has timed out.");
      setverdict(fail);
   }
}
webPort.send(addCdToCartBlues1Request);
localTimer.start;

alt {
   [] webPort.receive(addCdToCartBlues1XPaths) {
      log("Web client 0 has received addCdToCartBlues1");
      localTimer.stop;
      setverdict(pass);
   }
   [] webPort.receive {
      log("Web client 0 has receive wrong response.");
      setverdict(fail);
   }
   [] localTimer.timeout {
      log("Web client 0 has timed out.");
      setverdict(fail);
   }
}
webPort.send(removeCdFromCartBlues1Request);
localTimer.start;

alt {
   [] webPort.receive(removeCdFromCartBlues1XPaths) {
```

```
            log("Web client 0 has received removeCdFromCartBlues1");
            localTimer.stop;
            setverdict(pass);
         }
         [] webPort.receive {
            log("Web client 0 has receive wrong response.");
            setverdict(fail);
         }
         [] localTimer.timeout {
            log("Web client 0 has timed out.");
            setverdict(fail);
         }
      }
   }
}
```

Figure 25 - User_1_Behaviour