Open Source Integration Testing

Pierre Seguin
University of Ottawa
S.I.T.E, 800 King Edward Avenue
Ottawa, ON, Canada, K1N6N5
1-613-562-5800x2182
seguin_pierre@yahoo.ca

Liam Peyton
University of Ottawa
S.I.T.E, 800 King Edward Avenue
Ottawa, ON, Canada, K1N6N5
1-613-562-5800x2122
lpeyton @site.uottawa.ca

Bernard Stepien
University of Ottawa
S.I.T.E, 800 King Edward Avenue
Ottawa, ON, Canada, K1N6N5
1-613-562-5800x2122
bernard@site.uottawa.ca

ABSTRACT
Large heterogeneous software systems that integrate open-source components require a framework for integration testing beyond what current open source unit testing tools can provide. We present a test agent architecture for integration testing based on TTCN-3 and HttpUnit. TTCN-3 is an open standard test specification and implementation language developed by the European Telecommunications Standards Institute. We report our experiences with using TTCN-3 and discuss the current state of the F/OSS stack for TTCN-3.

Keywords
F/OSS, TTCN-3, service-oriented architecture, integration testing

1. INTRODUCTION
In enterprise environments, large software systems are deployed onto distributed multi-processor, multi-server environments. The architecture of such systems is complex, heterogeneous and involves inter-process communication over a variety of protocols. Increasingly, the underlying framework is a service-oriented architecture in which core components are packaged as reusable services shared between different systems and applications. It is a challenge to test and maintain such software systems in an efficient and reliable fashion in the face of change. And this is true, whether the open source components used in such systems are simple libraries (like log4j), individual web services, or a major platform like a Linux OS or Tomcat application server.

Testing tools designed to test individual components of a system independently are widely used in the free/open source software (F/OSS) community. For example, JUnit, HtmlUnit, HttpUnit and ServletUnit, are widely used for Java-based web applications. OpenSTA is another popular example. However, large software systems cannot rely solely on unit testing. A framework for integration testing is needed which can:

- Manage and encapsulate the complexity of large software systems at differing levels of abstraction
- Coordinate and manage test scenarios that cut across a component-based architecture
- Flexibly and efficiently deal with component interactions that occur under high volume multi-user scenarios

In this paper, we present a test agent architecture for integration testing using TTCN-3[3] based on its support for test agent coordination, templates, set-based pattern matching, and test specification at different levels of abstraction. TTCN-3 also has a flexible adapter framework that allows one to leverage and coordinate open source unit testing tools (HttpUnit in our case). We report our experiences using TTCN-3 and discuss the current state of the F/OSS stack for TTCN-3.

2. TEST AGENT ARCHITECTURE
The purpose of a test agent architecture is to mirror the architecture of the system being tested in order to integrate and coordinate test components that can run tests and monitor behavior. A large heterogeneous software system can be decomposed into individual web applications and the components they use. In particular, we look at the testing of large heterogeneous software systems based on a service oriented architecture in which the components used by the web applications are web services.

In ”black-box” unit testing, each web application and web service is unit tested individually as a separate "black box" in which only the inputs and outputs of the black box are tested. Black box unit testing does not address the possible interactions between web services especially under complex multi-user scenarios. It is also problematic to maintain black box unit tests, as different web services are upgraded or replaced.

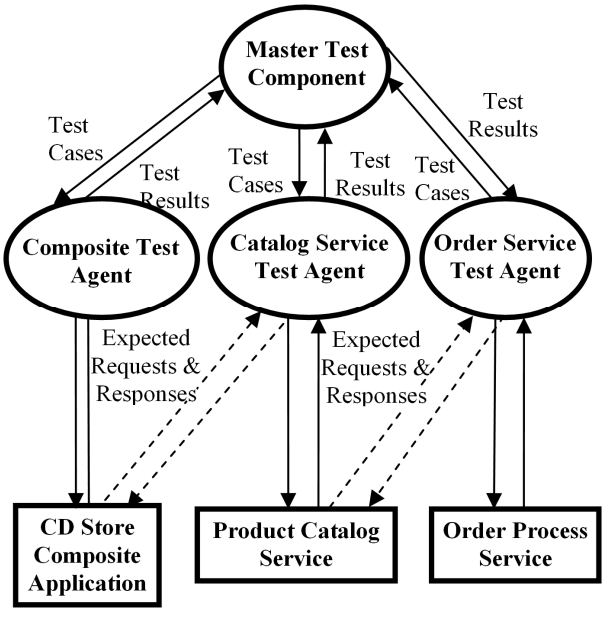

Figure 1 - Test Agent Architecture

Figure 1 shows a “grey box” test agent architecture in which tests are written that cut across the components of the system

and tests their interactions with each other. We refer to this as "grey box" testing because we do not treat the overall system as a black box, rather we treat it as a "grey" box in which we are aware of all of its applications and web services and can monitor and test the interactions between these components. Each service test agent tests its web service by intercepting the requests from the web application, and validating them before passing the requests on to the web service. The test agent also intercepts the responses from the service and verifies that they are the expected responses before returning them to the web application (which can be done in real time by using proxies or a posteriori using log files from the related components). The master test component is able to correlate precisely where faults are occurring and it 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.

3. ABSTRACTION WITH TTCN-3

TTCN-3 is an open standard test specification and implementation language developed by the European Telecommunications Standards Institute (ETSI). A common approach to designing a large heterogeneous software system is to decompose the system and describe its behavior at different levels of abstraction. Similarly, a test specification language, like TTCN-3 allows one to specify and reuse test cases 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 [6]. This enables test specifications and implementation to be reused across different levels of test activities [5] and different component projects.

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 at different levels of abstraction. 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 a template that is a combination of test data and matching rules. Thus, the tree and templates represent a separation of concerns between behavior and conditions governing behavior.

A test case consists of a sequence of requests and responses encoded as a system behavior tree and 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.

Testing using TTCN-3 consists of four essential steps:

- First create data types for the different messages being exchanged in the application.
- Second, create templates for both outgoing and incoming messages content with potential complex matching rules for the incoming messages.
- Third, use the defined templates to specify a choreography of messages and set appropriate verdicts for specific message sequences.
- Fourth, write an adapter to handle communication and coding and decoding of messages and their representation at the abstract level

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. Therefore, the first three steps mentioned above are implementation independent and can be reused. The fourth step is done to connect the abstract test suite to a concrete implementation. Our concrete implementation used Java and HttpUnit for low level communication with the System under Test (SUT) or Component under Test (CUT). Figure 2 shows our 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 communicates with HTTPUnit to send messages to the SUT or CUT. HTTP requests and responses are in the form of text that needs to be decoded to obtain the relevant information for a test.

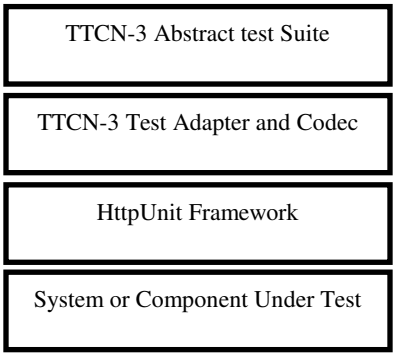

Figure 2 - Separation of Concerns

Currently our codecs are 'hard coded' for each expected request/response. A more complete discussion on how we approach codecs and adapters is presented in [6] where a thorough study on using TTCN-3 for the unit testing of web-applications and web-services is done. In summary, a traditional programming language (Java) combined with the open source unit testing tool for http testing, httpUnit, are used to implement the adapters and codecs. httpUnit is flexible enough to handle many types of http requests/responses. There are two important types of test agents in our implementation. The ones that emulate a user interacting with the SUT and the ones that stand between the web-application and its underlying services. When emulating a user agent httpUnit acts as a browser with the ability to handle JavaScript, basic http authentication and cookies. When acting as the service proxy httpUnit allows the messages

to be parsed from text and be forwarded to their proper recipients.

However there are drawbacks to the current approach. Each time a new message type is needed to be coded/decoded or a message structure is changed the codec must be created/edited by hand. In a future iteration we will be implementing a generic codec for the web-service request/response handling by relying on the WSDL associated with services. In this way the tedious hard coding of request/responses will be shortened by automatic generation of templates and their associated codecs.

4. COMPONENT INTERACTIONS AND MULTI-USER SCENARIOS

There are some significant implementation challenges associated with this test agent architecture, especially if the application test agent is simulating many users making multiple simultaneous requests. TTCN-3's powerful set-based pattern matching mechanisms combined with the concept of parallel test components can help address these.

Two important examples of implementation challenges are:

- Caching: Previous responses from an underlying service may be cached so that identical requests to the web 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 the particular user request of the web application.

4.1 Caching

Web 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 for a user's session. It is important that the caching mechanism should be well documented by the designers of the web application since caching will be based on assumptions of how the service 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.

To handle caching in TTCN-3, the service test agent must check if the cached event occurs, and if it does, set the verdict to fail. This requires a TTCN-3 implementation to represent a caching mechanism and detect a non-event. Our approach is to store received messages into a set of cached messages ("var cachedRequests" below) and verify that a subsequent message does not belong to that set. The function "isnotCached" below checks if the current request "matches" any of the requests cached so far.

```
function isNotCached(RequestType theRequest)
runs on ServiceAgentType return  boolean {
    var integer i;
    for(i:=0; i < nbRequests; i:=i+1) {
      if(match(theRequest, cachedRequests[i])) {
        return false;
        }
    }
    setverdict(fail)
}
```

In TTCN-3 the cache checking is considerably simplified using the TTCN-3 matching mechanism potentially saving considerable coding and debugging effort.

4.2 Correlation Gap

The correlation gap is a temporal ordering problem. A web application may place its requests to the service in a different order than what was received from the users. Similarly, services may return responses in a different order from the order in which it receives requests. Figure 3 shows an interaction diagram of two users (simulated by the application test agent) interacting with a web application. Request 1 is submitted first by User1 however Request2 from User2 is fulfilled first by the web 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. Ideally there would be unique IDs associated with requests associating them with particular users. However, when services are not under control of the development team this will often not be the case. Therefore, in the general case of web applications, simple end to end tracking does not work.

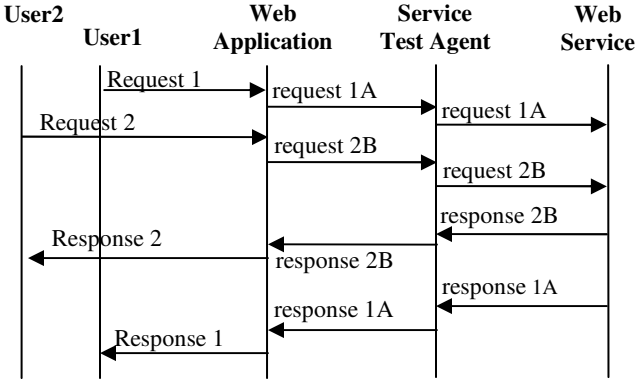

Figure 3 – Correlation Gap for Multi-user Requests

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 kinds of checking:

- It checks if such a message was expected for a specific test campaign, if yes, then forwards it to the service.
- It enforces the expected response from the service and if successful forwards the service response to the web application.

In TTCN-3, to handle the correlation gap, the master test component tells the service test agent what requests to expect

but not in which order. This is handled in a template represented as a set of messages. Using the powerful 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.

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 expected for a given test case have actually been received by the test service agent.

The first consideration is actually addressed in our architecture since the expected service requests are represented as a set.

The second consideration consists in updating a set of received messages as the messages arrive at the service. 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.

While the update of the received messages could be implemented easily in any programming language, the verification of completeness of the received set of messages is specified in a very concise and expressive way in TTCN-3 using the *match* operator as follows:

```
if(match(expectedRequests, receivedRequests)) {
   setverdict(pass);
}
else {
   setverdict(fail);
};
```

5. FUTURE DIRECTIONS

In this paper we have demonstrated how TTCN-3 used in a test agent architecture can be a valuable tool for integration testing large software systems. However more work is needed to support this within the open source community.

First, an open source integration framework based on TTCN-3 would make adaptation of integration testing more widely available. The TRex and iTTCP projects are partially working towards this goal within the context of the Eclipse framework but generic open source TTCN-3 codecs and adapter layers specifically for open source components is needed as well.

Second, the packaging of open source components as web services intended to fit within a service oriented architecture would both facilitate their inclusion in large scale software systems, from both a development and integration testing perspective. Service oriented architectures are becoming the norm for large software systems. They allow heterogeneous components to be more easily integrated with each other in the scalable distributed environments of today's enterprises. For open source components to be available for integration into these systems it is important to supply web service interfaces to make them more attractive as alternatives to COTS components. For example, packaging a component such as JFreeReport as a web-service would make it more appealing to use in these systems and enable standardized test frameworks.

Third, the integration testing framework described in this paper is not limited to web services. It can be applied to integration testing of other types of components and protocols (like RMI, JDBS, JMS etc.) by supplying the proper codecs and adapters to the TTCN-3 stack. In addition, the test agent architecture, can be adapted to do passive integration testing of components based on processing and analysis of log files (generated using a utility like Log4J) as a validation of system behavior.

A fourth avenue of research is in the data collection and representation aspects of TTCN-3 test verdicts. Currently, when TTCN-3 tests are run the information is viewable in a simple table format within their respective development environments (Telelogic TAU, TestingTech, Danet and OpenTTCN). However, this information can be improved in multiple ways in the open source tools:

- A more intuitive visualization of test results.
- A standardized data model to persistently store test results.
- An analysis tool which can report on and data mine past test results.

The final avenue of research we suggest is integration with model-based approaches for describing large software systems. Other research has attempted to address testing of large scale software systems by using model-based testing where test scripts are generated from models. This was done in the AGEDIS case studies [2] where HTTPUnit and HTMLUnit scripts were generated from UML models. In [1] User Requirements Notation (URN), an ITU standard for requirements modeling in telecommunications was used to test web applications. And in [7] evaluations done with JML-JUnit used JUnit scripts generated from JML models of Java classes. A similar approach could be taken to generate TTCN-3 specifications from models.

6. REFERENCES

[1] D. Amyot, J-F Roy, M. Weiss, UCM-Driven Testing of Web Applications. SDL Forum 2005

[2] Craggs I., Sardis M., and Heuillard T. AGEDIS Case Studies: Model-based Testing in Industry. Proc. 1st European Conf. on Model Driven Softw. Eng. (Nuremberg, Germany, Dec. 2003), imbus AG, 106—117

[3] ETSI ES 201 873-1, "The Testing and Test Control Notation version 3, Part1: TTCN-3 Core notation, V2.1.1", June 2005

[4] iTTCP TTCN-3 Compiler, http://ittcp.wiki.sourceforge.net, last retrieved: 2007/8/15

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

[6] B. Stepien, L.Peyton, P.Xiong, "Framework Testing of Web Applications using TTCN-3", to appear in International Journal on Software Tools for Technology Transfer, Springer-Verlag,Berlin,Germany

[7] R.P.Tan, S.H. Edwards, Experiences Evaluating the Effectiveness of JML-JUnit Testing, ACM SIGSOFT Software Engineering Notes, September 2004 Volume 29 Number 5

[8] TRex – the TTCN-3 Refactoring and Metrics Tool, http://www.trex.informatik.uni-goettingen.de/, last retrieved: 2007/8/15