Integration Testing of Composite Applications

Liam Peyton
University of Ottawa, S.I.T.E, 800
King Edward Avenue
Ottawa, ON, Canada, K1N6N5
lpeyton@site.uottawa.ca

Bernard Stepien
University of Ottawa, S.I.T.E,
800 King Edward Avenue,
Ottawa, ON, Canada,K1N6N5
bernard@site.uottawa.ca

Pierre Seguin
University of Ottawa, S.I.T.E, 800
King Edward Avenue
Ottawa, ON, Canada, K1N6N5
seguin_pierre@yahoo.ca

Abstract

A service-oriented architecture enables composite applications that support business processes to be defined and built dynamically from loosely coupled and interoperable web services. The testing and debugging of such applications presents special challenges in terms of localizing faults within the architecture, as well as addressing distributed, multi-user interactions. In this paper we define an integration test framework for composite applications based on defining test cases of expected behavior for composite applications and the web services used. The test framework is implemented in TTCN-3 using a test agent architecture that supports coordinated "grey-box" testing of application behavior and web service interaction. The essential issues that must be addressed in implementing such a framework are identified, and we illustrate how TTCN-3 support for templates, abstraction levels, set operations, and pattern matching allow one to address these issues efficiently and effectively.

1. Introduction

A service-oriented architecture enables composite applications that support business processes to be defined and built dynamically from loosely coupled and interoperable web services. The testing and debugging of such applications presents special challenges. A defect observed at the level of user interaction with the application could be

- a fault or quality of service issue (performance, security, scalability, etc.) in the application or process logic

- a fault or quality of service issue 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.

A systematic and comprehensive test framework is needed in order to successfully deploy and upgrade applications in such a complex environment. The test framework must be able to simulate the full complexity of large volume, multi-user scenarios, and be able to localize faults and quality of service issues to individual services within the service oriented architecture while correlating them to user interactions and requests originating from within the composite application. Existing approaches to testing have focused on unit testing of individual services, or on verifying the correctness of the choreography and orchestration of web services used by a composite application. Neither of these approaches can adequately address quality of service issues that arise when the full system is in use under load. Nor can they adequately correlate behavior of individual services to overall system behavior.

In this paper we define an integration test framework for composite applications that addresses these issues based on defining test cases of expected behavior for composite applications and the web services used. The test framework is implemented in TTCN-3 using a test agent architecture that supports coordinated "grey-box" testing of application behavior and web service interaction. The essential issues that must be addressed in implementing such a framework are identified, and we illustrate how TTCN-3 support for templates, abstraction levels, set operations, and pattern matching allow one to address these issues efficiently and effectively.

2. Background

TTCN-3 [5] 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 [7]. In particular, it allows testers to start working in parallel to developers from the same system requirements and specifications.

A composite application is a piece of software that composes functionality drawn from services within a service oriented architecture to perform operations or tasks on behalf of a user, often in the context of a well defined business process. The OASIS organization [6] has developed frameworks and 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 [11]. Composite applications support dynamic run-time configuration and collaboration of services. This is problematic for traditional testing approaches which assume a static set of components have been pre-compiled into a single monolithic application.

The complexity of composite applications dictates that a systematic test framework [8] reflective of a service oriented architecture is needed rather than a patchwork of tools and test scripts. Several approaches have focused on supporting formal verification of services against defined protocols. In [12] formal verification of web services using TTCN-3 was presented, while [3] leveraged the UML 2.0 protocol state machine to define the expected protocol for web service conformance. In [10], ebXML dynamic collaboration protocols from OASIS are extended with temporal logic and timing constraints and showed how a distributed framework of test agents [2] could be used for dynamically verifying completeness and consistency of service invocations in compliance with the protocol.

The approach we describe here differs from formal verification because we focus on defining test cases that link expected outcomes at the application level to the intermediate results returned by individual services. However, the test agent architecture we employ is similar to the test agent architectures that have been used in formal verification. One might imagine employing a mix of both test-case based verification and formal verification. In this paper, we identify the special challenges specific to test case driven verification in a service oriented architecture, and highlight how special language features in TTCN-3 can be used to address them.

Model-based testing, in which test cases and test scripts are generated from models is also relevant. This was done in the AGEDIS case studies [4] 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 generate TTCN-3 test scripts. And in [9] evaluations done with JML-JUnit used JUnit scripts generated from JML models of Java classes. Similar approaches could potentially be used to generate test cases (or formal characterizations of protocols) within a test agent framework, but we have not addressed this in this paper. Our focus has been on issues that have to be addressed in implementation whether that implementation is manually created or automatically generated.

3. Test Agent Architecture for Composite Applications

The purpose of a test agent architecture is to exploit the architecture of the system being tested in order to integrate test components that can run tests and monitor behavior. Typically, each component of the system being tested is paired with a test agent specific to that component. A master test component can be used to coordinate the activities of all test agents.

3.1 Composite Application

Figure 1 gives a simple example of a composite application that is composed of services available in a service oriented architecture.
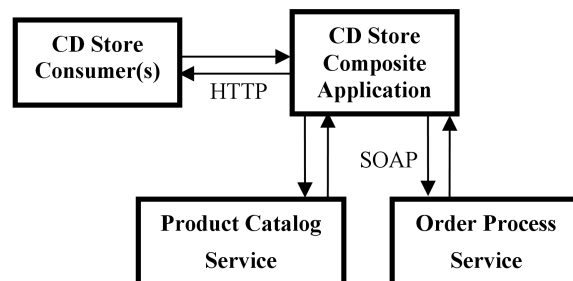

Figure 1. Composite application

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 a

Product Catalog service that provides information on available products, and an Order Process service that will create and process orders for fulfillment. The composite application communicates with these services using SOAP requests. For this example, consumer requests to the CD Store composite application are shown as HTTP requests on the assumption that consumers are using a simple browser interface. In general, though, the format of requests handled by the composite application will depend on the application and type of consumers targeted. For example, the composite application could itself be a service that supported SOAP requests from other applications.

3.2 Black Box Test Agent Architecture

The simplest approach to testing a composite application is to simulate the behavior of a consumer interacting with the composite application as a black box. Figure 2 below 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.
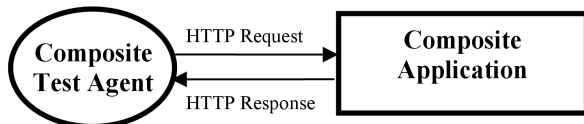

Figure 2 - Application test with consumer emulation

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 [13].

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 test agent 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 test agent instead of the real service. A master test component (MTC) coordinates the overall test. The MTC sends test cases for the composite test agent to run against the composite application and it sends a corresponding test case, if necessary, to the service test agent. The service test agent test case 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. At the end of a test campaign, the MTC correlates the test verdicts it receives from each of the underlying test agents.
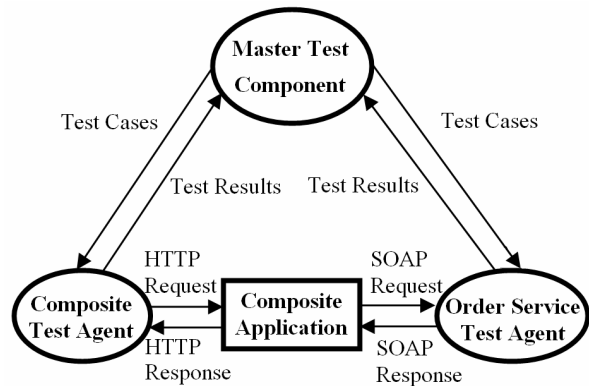

Figure 3 - Application test with service emulation

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 web application processing as long as each service test agent is accurately emulating its service. The problem is that 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.

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.
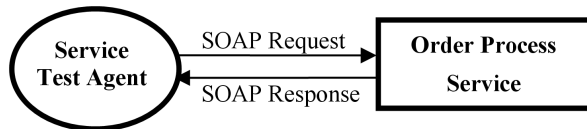
Figure 4 - Service Test with Application Emulation

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.

3.3 Grey Box Test Agent Architecture

In the black box test agent architecture, the composite application and each service used is unit tested as a separate "black box" in which only the inputs and outputs of the black box are tested (in figure 3, the requests made by the composite application to each service are treated as outputs). 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 is the possible interaction between web services as the composite application choreographs and orchestrates its use of the web services.

Figure 5 shows a grey box test agent architecture in which the application test from Figure 3 is combined with the service tests from Figure 4 into 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 we do not treat it as a black box, rather we treat it as a "grey" box in which we are aware of all of its components and can monitor and test the interactions between these components. Each service test agent emulates its service by forwarding the request from the composite application on to the service itself. In doing so, however, it both validates that it is an expected request from the composite application and verifies that the response from the service is the expected response. 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. Careful design of the service test agents should also make it possible for them to be implemented in such a way that they are completely reusable by any composite application.
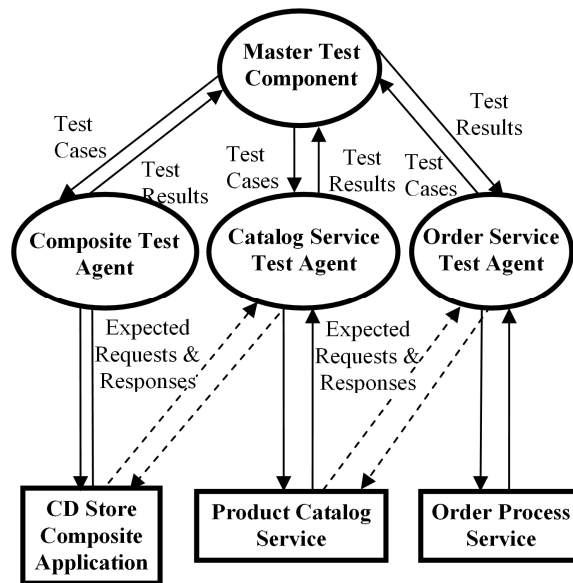

Figure 5 - Grey Box Test Agent Architecture

However, 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. Two important challenges are:

- 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 the particular user request made to the composite application.

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 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, the test can pass, and if it does occur, the test should fail. This requires three mechanisms (which we demonstrate in section 4):

- 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.

The correlation gap is a temporal ordering problem. The composite application may place its requests to the service in a different order from what was received from the users. Similarly, services may return responses in a different order from the order in which it receives requests. Figure 6 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. 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 composite applications, simple end to end tracking does not work.

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, it forwards it to the service.

- It enforces the expected response from the service and if successful forwards the service response to the web application.
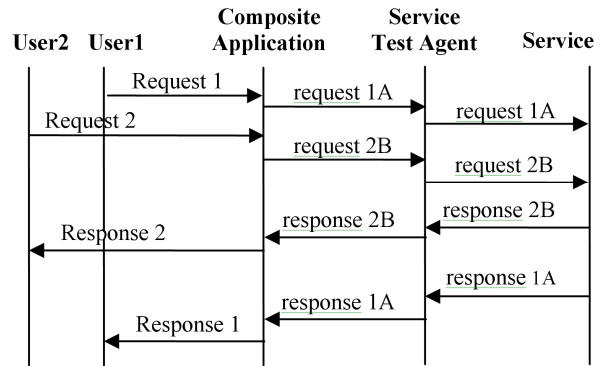

Figure 6 – Correlation Gap for Multi-user Requests

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 matches the set of actually received messages.

4. Implementation Considerations

The grey box test agent architecture in figure 5 was implemented using the TTCN-3 test specification and implementation language and applied to a basic CD Store composite application.

4.1 Basic concepts of TTCN-3

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.

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.

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. Our implementation uses the Sun Java programming language for our adaptation layer and more specifically the htmlUnit libraries that support low level message parsing.

4.2 Test Case Definitions

Individual test cases that define expected behavior in the test agent architecture shown in figure 5 can be coded directly at an abstract level into TTCN-3 code. The example in Figure 7 shows how we create and activate a service test agent and two composite test agents simulating users performing different behaviors that are coded in separate functions and thus can be re-used for various tests. The test case is defined to run on the master test component (*runs on MTCType* below). The three test agent processes are created and started. The *OrderService* test agent is passed the expected order requests that it will validate, and each user test agent is passed the user requests and responses it will simulate and validate (*User_1_behavior()*, *User_2_behavior* below).

```
testcase CompositeAppTesting1() runs on
MTCType
            system SystemComponentType {
  var ServiceAgentType theOrderServiceTest;
  var CompositeAgentType theUserTest [2];

  theUserTest [0] :=
            CompositeAgentType.create;
  theUserTest [1] :=
            CompositeAgentType.create;
  theOrderServiceTest:=
                ServiceAgentType.create;

  theOrderServiceTest.start(
      serviceEventsTest(
            theExpectedOrderRequests));

  theUserTest[0].start(User_1_behavior());
  theUserTest[1].start(User_2_behavior());

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

  servCoordPort.send("end test");

  all component.done;
}
```

Figure 7 – Activation of test agents

The structure of the test case shown in figure 7 remains the same for different tests. The only difference is in the use of the appropriate test behavior functions for the different components. The test case is built in two steps:

- First, build each individual user behavior that depicts their interaction with the composite application only.

- Second, build individual 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 test behavior of the users (*User_1_behavior()*, *User_2_behavior()* above) and expected responses is relatively straight forward to implement for the composite test agent processing. However, the service test agent processing is more complex. First for a given test campaign, it must check both that correct requests have been sent and that the corresponding expected response has been received. Due to the interleaving of requests among several users, the behavior of the service test agent can not be a simple sequence of requests and response as in the user behavior. It is more a recursive machine that receives one request at a time, relays it to the service and expects and enforces a response. Since the order of service request is unpredictable, the only solution is to use a choice construct that in TTCN-3 is called an alternative. Each alternate behavior is composed of a pair of request and corresponding response or even several alternative

responses to the same request, thus forming a behavior tree. These alternatives can be structured efficiently using the TTCN-3 altstep construct which is a powerful alternatives composition construct. The different alternatives can be defined as a pool of potential behaviors described individually in an altstep which is a type of function. Then, the appropriate service test behavior for a given test campaign can be assembled in a TTCN-3 alt construct where the different alternatives are a selection of the individual altsteps previously defined.

Figure 8 shows the behavior of the service test agent in relation to a request, B2, and its corresponding response and shows how request B2 can be encapsulated in an altstep named B2_behavior. Once this behavior tree has been processed a recursive invocation of the *serviceEventsTest()* function will ensure that the next request from the composite application is handled.

```
altstep B2_behavior(RequestsType
expectedRequests)
                       runs on SOAComponentType
{
   var ServiceRequestWrapperType incomingMsg;
   timer theServiceTimer;
   …

   [] soaWebPort.receive(request_2B) -> value
                              incomingMsg {
      servicePort.send(
                  incomingMsg.theRequest);
      theServiceTimer.start(5.0);
      alt {
        [] servicePort.receive(response_2B) {
            theServiceTimer.stop;
            soaWebPort.send(response_2B)
                                   to …
        }
        [] servicePort.receive {
           setverdict(fail); stop
        }
        [] theServiceTimer.timeout {
           setverdict(fail); stop
        }
      }
      serviceEventsTest(expectedRequests)
   }
}
```

Figure 8 – Service test agent behavior for an example request.

The behavior subtree shown in figure 8 describes one alternative that consists in receiving and matching message request B2 and then after relaying the incoming message to the service it will wait for a response where three possible behaviors could occur:

- Receive the correct response response_2B
- Receive an incorrect response (catch all receive) and set the test verdict to fail.

- Encounter a timeout and set the test verdict to fail.

Once individual behavior trees have been defined for each type of service request message, preferably in a separate module, they can be assembled into an expected overall behavior for a given test campaign as shown in figure 9 by selecting the appropriate ones.

```
function serviceEventsTest(RequestType
     expectedRequests) runs on
SOAComponentType {
   timer theCompositeApplicationTimer;
   alt {
      [] A1_behavior(expectedRequests) {}
      …
      [] B2_behavior(expectedRequests) {}
      [] soaWebPort.receive {setverdict(fail)}
      [] endTestBehavior(expectedRequests)
   }
}
```

Figure 9 – Abstract service test agent behavior for all expected requests

In the function shown in figure 9 there are two more alternatives in addition to all the predicted request/response pairs. One is a catch all receive construct for messages that are not on the expected messages list. This means that an incoming message could not be matched against any of the alsteps assembled for a given test campaign. This is a way to catch an incorrect request from the composite application and thus set the test verdict to fail. Finally there is an additional altstep that catches the coordination message from the MTC that signals the end of a test campaign. This is where potential missing messages are determined. A missing message would indicate that the composite application has produced a response to a user without consulting a service either because it produced an incorrect caching operation or it has some fault in its logic. This is explained further in the correlation discussion section below.

4.3 Caching

To handle caching, 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* in figure 10) and verify that a subsequent message does not belong to that set. Due to the principle of separation of concerns, in TTCN-3 this can be implemented in a concise way. First the entire code that handles the receiving of a message and its decoding is relegated to the adaptation layer and thus not visible at the abstract layer. Here we merely specify that whatever message received at the adaptation layer must match our

specification. For example, in the altstep B2_behavior presented in figure 8 we should specify that if we receive a particular message, say request_2B, we should first check the cache using a user defined function and, if satisfied, we should update the cache. Otherwise we should set the verdict to fail and stop the execution of the test case. This has been implemented in two steps via two separate functions. The first one, *cacheChecking()* can be concisely inserted in any of the behavior trees and basically handles the setting of the test verdict. The second one, *isNotCached()* handles the lower level cache lookup.

In figure 10 we show the definition of a component type where variables are declared for the cache mechanism:

```
type component ServiceAgentType {
        integer nbRequests = 0;
        var RequestsType cachedRequests := {};
        …
}
```

Figure 10 – Declaring cache variables for service test agent

Then, in figure 11 we present a modified version of the *B2_behavior()* altstep which includes the cache verification. Note that this approach naturally allows us to separate the messages that are subject to caching or not. For a non-cached message, one just needs to omit the invocation to the *cacheChecking()* function.

```
altstep B2_behavior(RequestsType
        expectedRequests)
                runs on SOAComponentType {
  …
  alt {
    [] soaWebPort.receive(request_2B) ->
                        Value incomingMsg {
        cacheChecking(incomingMsg.theRequest);
        …
```

Figure 11 – Incorporating caching into service agent test

The *cacheChecking()* function shown in figure 12 is trivial. It merely invokes another function *isNotCached()*, shown in figure 13, that performs a lookup of the cache. If this lookup returns false we merely update the cache with this incoming message and if it is positive we set the verdict to fail.

While so far, the cache update could have been achieved in any conventional programming language, in TTCN-3 the cache checking is considerably simplified using the TTCN-3 matching mechanism. It relegates the processing of the comparison of two complex messages to the tool, thus potentially saving considerable coding and debugging effort as shown in figure 12.

```
function cacheChecking(ServiceRequestType
    theRequest) runs on SOAComponentType
                        return  boolean {
  if(isNotCached(theRequest)) {
      updateCache(theRequest);
      return true;
  }
  else {
      log("has received a cached message:_"
            & theRequest);
      setverdict(fail);
      stop
  }
}
```

Figure 12 – Checking the cache

```
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; }
    }
    return true
}
```

Figure 13 – Checking if a message is cached or not

Finally, it is to be noted that in the special case where all messages could be subject to caching, the set of received messages that is dynamically updated as messages arrive at the service test agent really can fulfill two functionalities simultaneously; one throughout the test to enable the cache checking mechanism by verifying that a newly arrived message is not already in the cache, the other at the very end of the test when we compare the expected versus the received messages for completeness checking. Consequently for this special case, it may be more appropriate to use one common variable for both types of checking.

4.4 Correlation Gap

To handle the correlation gap, the master test component must tell 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 that are expected for a given test case have actually been received by the test service agent.

The first consideration is actually addressed naturally by the content of the function *serviceEventsTest()* shown in figure 9. 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 in the generic receive statement where a verdict of fail can be set as discussed before.

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.

The verification of completeness of the received set of messages is specified in a very concise and expressive way in TTCN-3 through the altstep *endTestBehavior()* using the *match* operator as shown in figure 14 This is in fact similar to the use of the matching mechanism explained in section 4.3 for the cache look up but with the difference that here the individual messages comparison is performed at the set comparison level.

```
altstep endTestBehavior(RequestType
                expectedRequests) runs on … {

    [] serviceCoordPort.receive("end of test"){
        if(match(expectedRequests,
                        receivedRequests)) {
            setverdict(pass);
        }
        else {
            setverdict(fail);
        };
    }
}
```

Figure 14 – Correlation gap handling

5. Results

This framework has successfully been implemented and used for functional testing of composite applications and is able to localize faults within the service oriented architecture and correlate them to user interactions under multi-user load. The performance of the framework has been evaluated compared to traditional approaches using JUnit, HttpUnit and OpenSTA. It requires more effort and sophistication to set up the unit testing tools, but it results in a more comprehensive framework that can be used to localize and detect faults down to the level of individual web services correlated to user interactions. The framework also complements more formal methods of verifying service orchestration and choreography. The framework is able to verify that complex multi-user scenarios under load do not result in unexpected side effects to the logic verified under the assumption of a simple single user scenario.

Although it has not been shown in the examples used in this paper, quality of service issues related to response times experienced by the user (performance and scalability) can be addressed by including a measure of response time and throughput at each point in the test agent architecture as part of the definition of test cases and expected results.

6. Future Work

In our current approach, the test agents are dynamically integrated into the architecture of the composite application by a redirection or proxy of HTTP and SOAP requests and responses to test agents that process and analyze them before forwarding them on. This requires that the test agent perform its processing in real time with a likely significant impact on performance, scalability and timing. This can affect the processing of the system under test and compromise the integrity of the testing approach. Simple stated, the composite application may behave differently under test than it does when not under test. An alternative is to simply log a history of all requests and responses that occur at each service as the composite application is tested using a composite test agent. Once the test has completed, each test agent can perform its actions by processing their respective log files after the system has finished executing. There are tools available which can perform such message logging with minimal and more predictable impact on system performance. Further, the processing logic and test case definition that we have defined will stay largely the same. It should suffice to build an adaptor plug-in that parses the request and response messages from a log file, as opposed to parsing them from a socket connection. We are currently exploring this approach.

Security and resilience in the face of inappropriate use is another aspect of testing that this framework is relevant to. In this paper, we have focused on test cases to simulate and validate the system under normal usage. Just as important are test cases that look for security vulnerabilities and simulate users with malicious intent. As with normal usage test cases, it is critical to be able to localize any fault or quality of service issues to individual services and correlate them to user interactions under load intensive multi-user scenarios.

7. Conclusions

The complexities of composite applications in a service oriented architecture necessitate the construction of a test agent architecture that closely mirrors the underlying architecture of the composite application. Given the independence of underlying services from the composite applications that use them, it is essential that grey box integration testing approaches supplement black box unit testing approaches. Testing must stress the overall system under the actual combination (orchestration and choreography) of web service calls that the system must support, correlating the expected requests, responses and quality of service of the composite application with the expected requests, responses and quality of service of the underlying services.

We have demonstrated the basic principles of how this can be achieved using the TTCN-3 test specification and implementation language. It is able to leverage existing unit test tools into a more coordinated framework. It complements formal verification approaches well by ensuring that the expected results are still obtained under high volume multi-user scenarios.

There is still future work to be done in order to ensure the approach does not introduce an unacceptable overhead on system performance as well as to address test scenarios related to security and users with malicious intent.

8. References

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

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

[3] 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,

[4] 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

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

[6] OASIS: Web Services Composite Application Framework, http://www.oasis-open.org/committees/ws-caf/ Accessed 05/2007

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

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

[9] 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

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

[11] W3C Working Group, "Web Services Architecture", Note 11 February 2004, http://www.w3.org/TR/ws-arch, last retrieved: Oct 28, 2006.

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

[13] 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.