2009 IEEE International Conference on Web Services

A Unified Test Framework for Continuous Integration Testing of SOA solutions

Hehui Liu, Zhongjie Li, Jun Zhu, Huafang Tan, Heyuan Huang
IBM China Research Lab
Beijing, China
{hehuiliu, lizhongj, zhujun, tanhuaf, huanghey}@cn.ibm.com

Abstract—**The quality of Service Oriented Architecture (SOA) solutions is becoming more and more important along with the increasing adoption of SOA. Continuous Integration Testing (CIT) is an effective technology to discover bugs as early as possible. However, the diversity of programming models used in an SOA solution and the distribution nature of an SOA solution pose new challenges for CIT. Existing testing frameworks more focus on the integration testing of applications developed by a single programming model. In this paper, a unified test framework is proposed to overcome these limitations and enable the CIT of SOA solutions across the whole development lifecycle.**

This framework is designed following the Model Driven Architecture (MDA). The information of an executable test case is separated into two layers: the behavior layer and the configuration layer. The behavior layer represents the test logic of a test case and is platform independent. The configuration layer contains the platform specific information and is configurable for different programming models. An extensible and pluggable test execution engine is specially designed to execute the integration test cases. A global test case identifier instrumentation approach is used to merge the distributed test case execution traces captured by ITCAM – an IBM integrated management tool. A verification approach supporting Boolean expression and back-end service interaction verification is proposed to verify the test execution result. Initial experiments have shown the effectiveness of this unified test framework.

Keywords-continuous integration testing; service oriented architecture

I. INTRODUCTION

It is becoming a common sense that the program bugs should be detected and fixed as early as possible in order to improve the software's quality and cut testing cost. This is why good unit testing is emphasized. Meanwhile, many critical bugs can only be discovered after different components of the software are integrated together. As performing the integration testing when all the components are completed violates the early detection principle, Continuous Integration Testing (CIT) is proposed to reveal the integration bugs earlier [1]. In CIT, the software is integrated continuously after some codes are developed or updated. The integration testing is done immediately to discover the integration bugs as quickly as possible [1]. The emerging of Service Oriented Architecture (SOA) is blurring the traditional fixed boundary of software development and integration, and hence adds more weight to the importance of CIT in SOA from the industry side.

In line with this technical trend, the objective of this paper is to build a unified test framework (UTF) to enable the CIT of SOA solutions across the whole development lifecycle. However, the diverse programming models and the distribution nature of an SOA solution hinder the application of CIT.

In SOA, the components of a system are usually implemented using different programming models, such as web service and EJB. Each model comes with its own mechanism for internal logic and external invocation. With the evolving of software technologies, newer ones are emerging. In this case, most traditional test frameworks, which more focus on the testing of the applications developed by a single programming model, could not be applied to the integration testing of SOA solutions [4, 7]. Besides, the components of an SOA system are usually deployed in different application servers or machines. It is difficult to verify the interactions between different components since the traces are distributed in different machines, especially when multiple test cases are executed concurrently.

In previous work, the authors have proposed a new simulation apparatus, called as surrogate, to simulate the behaviors of unavailable components in SOA environments [2]. This paper addresses the other aspects and completes the unified test framework. This framework adopts the Model Driven Architecture (MDA). An executable test case is separated into two layers: the behavior layer, which is programming model independent and the configuration layer, which is programming model dependent. From service interaction specification, test cases are generated. Then, a flexible and extensible test case execution engine is used to execute the integration test cases. A trace correlator based on global case identifier instrumentation is proposed to correlate the traces collecting by ITCAM [5] together. A verification approach is proposed to verify both the input/output data and the back-end operation interaction sequence. The initial experiments have shown the effectiveness of UTF.

The rest of this paper is organized as follows: section 2 explains the integration testing problems for SOA solutions with an example. Section 3 elaborates the unified test framework (UTF). Section 4 presents the UTF experimentation results in two real cases. Section 5

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.28

introduces the related works and section 6 concludes this paper with future work prediction.

II. INTEGRATION TESTING PROBLEMS FOR SOA SOLUTIONS

The software developed following SOA can bring many benefits, such as higher flexibility and lower development cost. At the same time, it also poses new challenges for the CIT. Let us use the system showed in figure 1 as an example to introduce the problems. This is an order process to handle an order request. Totally it has 6 components. For an input order request, firstly, the *Check Order Format* service is invoked to check its format. If the format is not permitted by this system, the order request is returned and not handled. For a valid input order request, *Process Order* service is used to dispose the order to be handled by other services. *Calculate Good Price* calculates the good's total price, and the *Calculate Shipping Price* service is invoked next to calculate the shipping price. After getting the price, the *Arrange Product* service is responsible to arrange the shipment of the product. Finally, if the price of the order is greater than 100$, a notification will be sent to the customer by *Prepare Notification*. This system is deployed in three machines. *Check Order Format* component is an existing web service, and is deployed in machine A. *Calculate Good Price* and *Calculate Shipping Price* component are two common web services deployed in machine C. The other components are developed only for this system, so they are exposed as SCA services, and deployed in machine B.

For this system, suppose we have an interaction scenario as is shown in figure 2. The pseudocode of this scenario's executable test case is showed in Figure 3. In traditional test framework, there are several problems or disadvantages to execute this test case.

Firstly, for the same input content, the order request, two different input objects, XML snippet (web service input object) and Service Data Object (SDO) [9] (SCA service input object) should be included in the executable test cases. Two different invocation protocols, web service invocation and SCA invocation, should also be supported.

Secondly, the output of *doCheckOrderFormat* (a method of the *Check Order Format* component) is an XML message, while the output of *doProcessOrder* (a method of the *Process Order* component) is an SDO. In order to verify the test results, two different verification methods should be used (one for the XML verification, the other for the SDO verification).

In summary, testers have to be concerned with different protocols and this means a lot of burden. However, in SOA, it is very common that multiple programming models are used to develop an application.

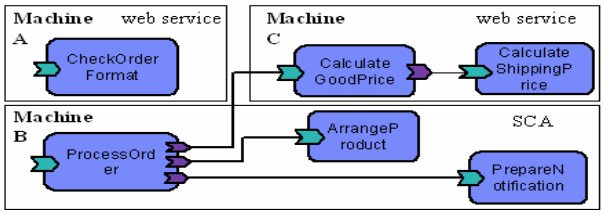

Figure 1. Order process example

Thirdly, if the implementation of the *Process Order* service is changed from SCA to web service, the executable test case of figure 3 should be rewritten. As such kind of changes can be frequent in SOA environments, test case maintenance is costly.

Fourthly, a bug may exist in the interactions between the *Process Order* and other services, such as *Calculate Good Price*. The bug may be a wrong service invocation, a wrong input sent, or a wrong output received. Unfortunately, such kind of back-end service interactions cannot be verified in traditional test frameworks, which only test a component or a cluster of components from black-box view - only external outputs are examined. There is no way to do back-end service interaction assertion even in a single machine, not to mention that in an SOA environment, the execution traces are distributed in different application servers or machines. They should be collected and correlated before any further processing.

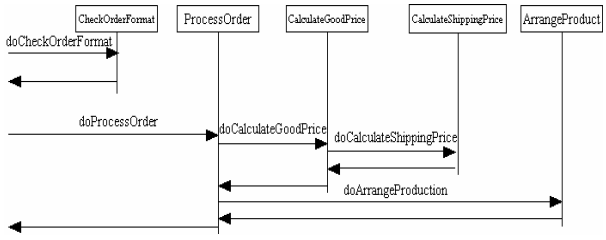

Figure 2. An interaction scenario

In fact, these four problems are mainly generated by the diverse programming models (problem 1, 2 and 3) and the distribution (problem 4) feature of SOA solutions. They should be solved in order to enable the CIT in SOA environments.

1: **get** *Check Order* Format Service
2: **construct** an XML snippet for the input order request
3: **invoke** *doCheckOrderFormat* by web service invocation
4: **assert** *true* for the returned result
5: **get** *Process Order* Service
6: **construct** a SDO for the input *order request*
7: **invoke** *doProcessOrder* by SCA invocation
8: **verify** the returned result

Figure 3. An executable test case example

III. A UNIFIED TEST FRAMEWORK FOR CIT OF SOA SOLUTIONS

A. Storyboard and Architecture

Figure 4 shows a storyboard of CIT with UTF. This storyboard includes 8 steps.

In step 1, the architect designs the architecture of an SOA solution and its sequence diagrams. Then in step 2, the integration test cases are generated semi-automatically with the input sequence diagrams. Therein, each test case describes the interactions between different components implemented following different programming models, including the input, output data and the interactions in the back-end services. The generated test cases are uploaded to a continuous integration testing server in step 3. This server is used to run the integration test cases continuously. In step 4, the surrogates for all designed components are generated automatically from the assembly graph and service description files and deployed to the CIT server. With the generated surrogates, the integration test cases could be executed automatically even when no components are ready in step 5. The test result is fed back to the architect. We call this case as pure simulation scenario. It is used to verify the configuration and design of services. In step 6, during development, the services are implemented or updated by different developers. The codes are committed to CVS. In step 7, the ongoing software is packaged with the unimplemented components replaced by surrogates and deployed to the CIT server. Finally, the same integration test cases are executed automatically to verify the updated software in step 8. When bugs are found, they are fed back to the developers immediately. Then this process is rerun from step 6. The case from step 6 to 8 is called as continuous integration testing scenario in this paper. It is used to verify the implementation and configuration bugs during development.

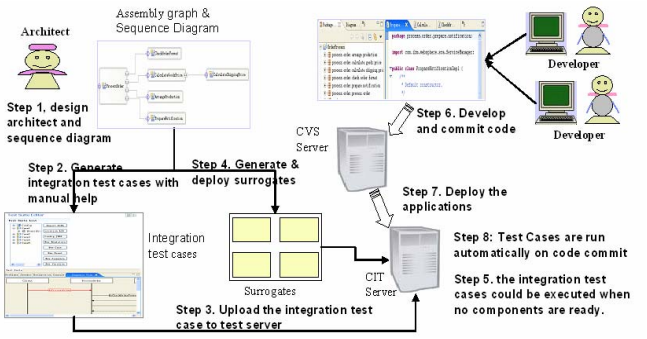

Figure 4. The storyboard of CIT with UTF

Following the storyboard, we design the architecture of UTF as figure 5.

The input of the framework is UML sequence diagrams. A sequence diagram should describe the interactions between the interfaces of different services. One example of this kind of sequence diagram is the SOMA component flow [3]. With the input sequence diagrams, the integration test cases are generated by the integration test case generator with manual help. The software under test is deployed in the SOA environment, and the integration test case is executed by a test case execution engine. During execution, for unavailable components, the surrogate engine [2] is used to simulate their behaviors.

During test execution, the ITCAM [5], an integrated application management tool, is applied to capture and collect the traces. Then the traces are inputted into the trace correlator. The traces of the same test case are correlated together. As the execution traces of different programming models have different formats, the trace transformation is applied to transform the correlated traces into a common trace representation. Finally, the verification is performed on the integration test case and execution trace. The test result is fed back to the test case execution engine. In this architecture, we omit the surrogate generator [2]. Because it is an isolated plug-in and will not be introduced in this paper. The following sections will introduce the details of each part in the architecture.

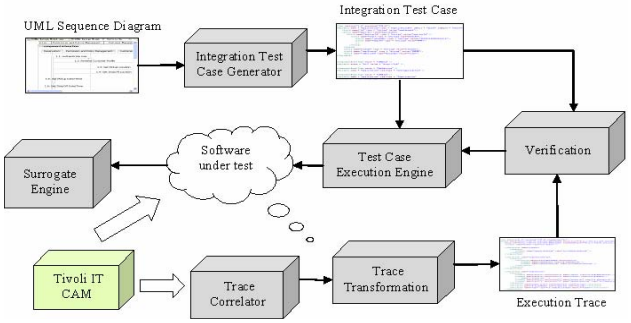

Figure 5. The architecture of UTF

B. Two-layer executable test case model

In traditional test case representation, all information of an executable test case is included in a single case model. In fact, an executable test case is composed of two kinds of information: the behavior logic and technical specific information. The behavior logic is independent of the programming model, and is the core of a test case. It could be reusable for different platforms and programming models. The technical specific information is a wrap of the behavior logic. It changes with the specific programming model.

Following these considerations, we could use a two-layer model to represent the executable test case. The first layer is a platform independent test case model representing the behavior logic (including the data logic); the second layer is a test case configuration model representing the specific technical information (including the data type). The specific technical information could be separated into a configuration file. Then for the specific service, we only need to set its technical specific information.

Figure 6 shows the test case behavior logic model. In this model, the *test logic* and *event* together represent the invocation sequence of a test case. In current UTF implementation, the *sequence*, *while* and *concurrent* test logic are considered to be supported and the *sequence* test logic has been implemented. For an *event* (representing an

invocation of a service), only the service, interface and operation name are contained in this model. The specific programming model information, such as the namespace of a web service, is put into the configuration model. There are two event types: two-way (default value) and out. The two-way means that the next event could be executed until the output for the current invocation is received. The out means that this is an asynchronous invocation, and there is no response message. The *assertion* attached with an *event* is the assertion expression for the expected output data. It can be written as JavaScript. If no assertion is attached, the defined output is used as the expected output to check the real output.

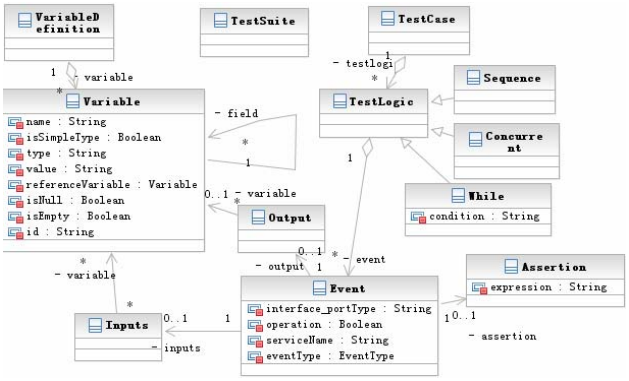

Figure 6. The test case behavior logic model

The input/output data could be defined as *variable* and a variable could be reused by other variables via reference. If a variable is a complex type, its attributes are included in this variable by its field attributes. Currently, the simple data types of java programming language are considered as simple types and supported in UTF. Other types are considered as complex types. The *isNull* attribute of the variable indicates whether this variable is a null pointer. The *isEmpty* attribute indicates whether this variable is a container, for example a list. If *isEmpty* is true, it indicates that this variable is a container, which is not null, but no item is contained in this variable. The technical specific information of the variable, such as the namespace of an SDO, is also divided into the configuration model.

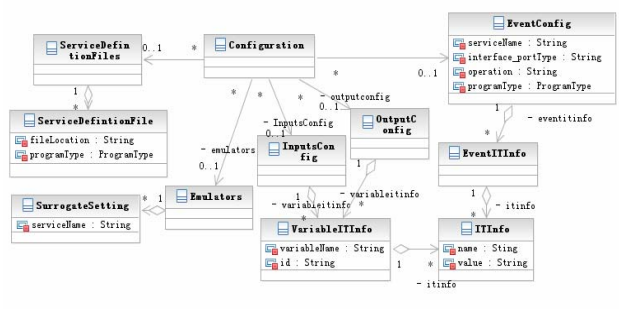

Figure 7. The test case configuration model

Figure 7 shows the configuration model. The *serviceDefinitionFile* indicates the file location of the services and data structure definitions. By the files, the test case editor could load the technical specific information of the services. The *emulator* specifies which components should be surrogates even when the real components have been implemented. The *eventITInfo* specifies the technical specific information of a service, such as the namespace, composite name of a SCA service. The technical specific information is different for different programming model. For web service, the soap address is required, while for SCA, the composite name is required. In order to handle this case, the *ITInformation* class is used to represent all the technical specific information of a service.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite:TestSuite ...>
  <variables>
   <variable id="0" name="apples" isSimpleType=" false" type="Order">
    <field name="goodName" type="java.lang.String" value="apple"/>
    <field name="unitPrice" type="double" value= "2.0" />
    <field name="quantity" type="double" value="45" />
   </variable>
  </variables>
<testcase id="1" description="put a valid order request" >
  <sequence>
   <event serviceName="CheckOrderFormat" interface_portType="CheckOrderFormat" operation="doCheckOrderFormat">
    <inputs>
     <input id="1" name="requestOrder" refVariable="//@variables/@variable0"/>
    </inputs>
    <output name="result" type="boolean" value="true"/>
    <script expressionValue=""/>
   </event>
   <event serviceName="ProcessOrder" interface_portType="ProcessOrder" operation="doProcessOrder">
    <inputs>
     <input id="2" refVariable="//@variables/@variable0"/>
    </inputs>
    <output name="result" type="java.lang.String" value="success"/>
    <assertion expression= "assert.assertEquals(result, &quot;success&quot;);"/>
    ...
   </event>
  </sequence>
 </testcase>
</testsuite:TestSuite>
```

Figure 8. The behavior logic file of figure 3

Figure 8 and 9 respectively give a behavior logic and configuration example for the executable test case of figure 3. In figure 8, the reference to a variable is indicated by the path to the variable and its id. For example, the //@variables/@variable0 means that the variable is under the node variables, and its id is 0. From figure 8 and 9, it could be seen that by this presentation, the test logic of an executable test case is separated with the technical specific information, and when the technical specific information is changed, only the configuration is required to be modified.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration ...>
 <ServiceDefinitionFiles>
  <ServiceDefinitionFile filelocation= ... programType=
"webService"/>
  ...
 </ServiceDefinitionFiles>
 <eventConfig serviceName="CheckOrderFormat"
interface_portType="CheckOrderFormat" operation=
"doCheckOrderFormat" programType="webService">
  <eventITInfo>
   <ITInfo name="soapAddress" value="http://.../
CheckOrderFormat"/>
   ...
  </eventITInfo>
  <inputs>
   <variableITInfo variableId="1" name="requestOrder"
>
    <ITInfo name="nameSpace" value="http://Order
ProcessLib"/>
    ...
   </variableITInfo>
  </inputs>
 </eventConfig>
</configuration>
```

Figure 9. The configuration file of figure 3

IV. TEST CASE GENERATION BASED ON SEQUENCE DIAGRAMS

The sequence diagram specifies the interactions between different components, and is fit for being used to generate the integration test cases. While, in general, it only gives a high level overview of the system behavior, and is not sufficient to generate an executable test case. In this paper, the sequence diagram is only used to generate the platform independent test case in UTF, and the execution required information is generated by leveraging the service description file such as the WSDL of web service, EJB description file of EJB. This information is separated in the configuration file of the test case. For simplification, in this paper, we do not generate the data because there is not always enough information (such as the OCL constraint of the message) is provided by the sequence diagram.

In platform independent test case generation, firstly, the input sequence model is transformed into an intermediary model, which is called as service interaction meta-model (SIM) in this paper. SIM in fact is a tree structure. In the tree, every node represents an operation of an interface or a condition control node, such as sequence, choice node. In the SIM, by applying the tree travel algorithm, the interaction sequences are generated. Each interaction sequence is considered as the behavior logic of an integration test case initially.

Based on the interaction sequences and configuration files, an integration test case editor is also developed to input the test data and modify the interaction sequences. In the test case editor, the platform independent test case model is merged with its configuration information to form a memory test case model, and then the editor is done to the memory model. When save action happens, the memory test case model is separated into platform independent test case and test case configuration model.

V. A FLEXIBLE AND EXTENSIBLE TEST CASE EXECUTION ENGINE

Based on the platform independent test case model and test case configuration model, we design a flexible and extensional execution engine to execute the test cases. The architecture is showed in figure 10. In this execution engine, with the input test case behavior logic and configuration file, the test schedule is responsible for the behavior logic explanation. The test input data is constructed by the test input constructor. Then the interfaces of the software under test are invoked to execute by the test invoker.

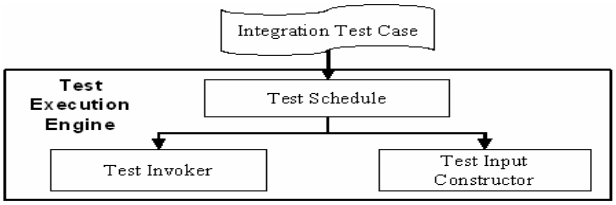

Figure 10. The test execution engine

The test invoker and test input constructor both should consider the specific programming model information. So they are designed as two extensible plug-in. For a specific programming model, two specific plug-in should be implemented by extending from these two plug-in. In current UTF, we have implemented the plugin for three program types: web service, EJB and the SCA developed by IBM WebSphere Integration Developer (WID) [11].

VI. TRACE CORRELATOR BASED ON TEST CASE IDENTIFIER INSTRUMENTATION

In order to capture the test case execution traces distributed in different application servers or machines, ITCAM [5] is used in UTF to collect the execution traces. However, ITCAM could only capture and correlate the traces for a single request (invocation). In a test case, it is usual that multiple invocations (multiple requests) are executed and the traces coming from multiple invocations need to be correlated together. So, in this paper, a trace correlation

method based on the test case identifier instrumentation is proposed to correlate the traces collecting by ITCAM together.

Figure 11 shows how our approach works at the high level. When the requests are triggered during test case execution, the test controller at testing client side will intercept the requests and attach an identifier which identifies the correlation between the test case and the request to the request message. When the request goes into the software under test, the mapping between request and the identifier is recorded, the execution path of request is tracked and correlated by ITCAM (the ITCAM agent is deployed in the same machine with the application server, and the ITCAM server is deployed in a central server) [5]. Thus according to the linkages of case identifier and requests, the execution traces of a same test case could be correlated by the trace correlator.

To add identifier to the requests during test case execution, the test controller should work as an API probe or a proxy which intercepts the requests to the system under test (SUT) and adds the identifier to the request. In current UTF implementation, the API probe is integrated as a part of the test execution engine, and the test case id with time stamp is used as the case identifier.

To record the mapping between the case identifier and requests, instrumentation in the system under testing is added. It is located at the middleware layer, such as application server for J2EE applications. So there are no changes involved at application code level.

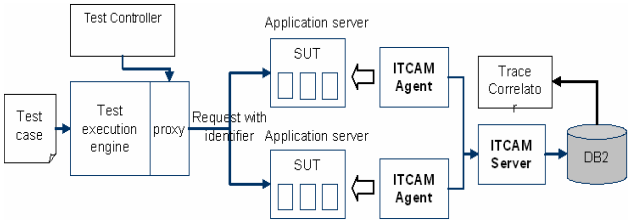

Figure 11. Execution traces correlation

VII. TEST RESULT VERIFICATION

The test result verification includes the verification of the interaction sequences and the input/output data. Firstly, the interaction sequence will be verified to check whether the correct operation sequence is executed. Then, the input/output data specified in the test case file will be verified with that captured in the execution trace.

Firstly, the execution traces for a specified programming model is transformed into a common trace. Based on the test case and the common trace, each operation is fetched to be verified. If the operation in the trace does not match that in the test case, an error message is created and the verification is stopped. For the correct invocation, two data verification approaches are applied to verify the data. One is to verify the real output with expected output by constructing the object for the data. For a complex data type, the comparison is decomposed to compare each attribute of the data type. If one of the attributes is wrong, an error message is generated. For the *event* with *assertion* written by JavaScript, the data is constructed as a JavaScript object and the Rhino [12] is leveraged to execute the expression script and return the executed result.

VIII. EXPERIMENTS

Experiments have been done on a HR management system and a meeting room management application.

A. HR management case

In this experiment, we want to verify whether the bugs could be found by UTF even in the case that no codes is implemented, and what defect types could be found by UTF during the whole development lifecycle.

1) Overview: This case is a HR management system. Figure 12 shows the architecture of the system. Therein, the common service is implemented by one company. It is exported as web service, and invoked by payroll management and recruiting management component. The payroll management, recruiting management component and the other data layer components are implemented by another company and the SCA was adopted as the programming model. Between the SCA components, the SCA invocation method is applied, while the web service invocation method is applied to consume the web service. Overall, there are 17 components in the business logic layer including 58 interfaces, 22 components in the data layer including 22 interfaces that should be developed, each interface contains several operations. This system was developed using WID 6.0.2 and deployed to the IBM WebSphere Process Server (WPS) 6.0.2.

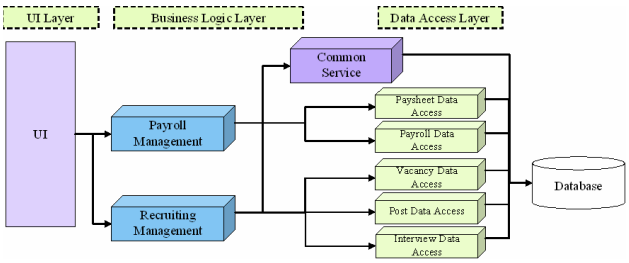

Figure 12. The architecture of HR management system

2) Experiment result: In this experiment, two scenarios: pure simulation and continuous integration testing introduced in section 3.1 are performed. Some exciting results have been observed as is shown in table 1. Therein, we classify the bugs found in integration testing into five types.

1. **Incorrect method call** means that a redundant service invocation is implemented in the code or a missing service invocation, or a wrong service is invoked.
2. **Incorrect parameter passing** means that the mismatched inputs (the data type is correct but the content is wrong) are passed into the invoked service or the output returning from the invoked service violates the expected output.

3. **Configuration problem** is mainly related with the configuration problem of the service, such as missing service export (an interface is designed to be exported as a service, but in implementation, it is not), wrong service reference (a component references a service that is different with the design document), missing service reference (a service should be referenced in the assembly graph of the application but not).

4. **Interface mismatch** means that the invoked service mismatches the actual, e.g. the invoked operation does not exist in the service definition.

5. **Function mismatch** means that the required function for the invoked service does not exist or the invoked service provides more function (or duplicated services are defined in multiple components).

In the experiment, total 22 bugs are found. Therein, pure simulation finds 7 bugs. These 7 bugs are mainly the configuration problem and function mismatch. In traditional testing, these problems are found only after the components have been finished, which greatly adds difficulty to problem determination and increases fixing cost. Half of the 22 bugs are related with the incorrect method call and interface mismatch. They also could only be found after the components have been finished and integrated in traditional fixing boundary testing method. While by applying UTF, these bugs are discovered immediately after the codes are finished, greatly reducing the time to bug fixing. More importantly, during the whole development process, only one group of test cases are needed to be designed, and in case that some services' implementation technology is changed, only the configuration information need to be modified.

TABLE I. THE EXPERIMENT RESULT OF HR CASE

Incorrect Method call	Redundant invocation	
	Lost invocation	2
	Wrong invocation	2
Incorrect parameter passing	Wrong input	
	Wrong output	1
Configuration problem	Miss service export	3
	Service reference error	4
	Miss service reference	2
Interface mismatch		7
Function mismatch	Redundant function (service)	1
	Lost function (service)	

B. *Meeting Room Management Case*

In this case, we want to verify whether the UTF could still find bugs effectively in the case that a very rigid unit testing and code review has been applied to the unit code, and what types of bugs could be found.

1) Overview: The meeting room management application is a sub system of an enterprise business management ERP system. This system is developed by EJB. Totally 4 components are included the meeting room management application. Figure 13 shows the architecture. In this system, an external service, common data access service, is invoked by the personal notes pad and meeting room components. Totally 24 interfaces in the business logic layer and 20 interfaces in the data access layer are developed. The application is developed by the IBM Rational Software Developer 6.0.2 and deployed to the WebSphere Applocation Server (WAS) 6.0.2. In the experiment, 5 students are chosen to implement the application and 2 students are chosen to do code review after the code has been implemented. For each piece of unit code, the all-code-line coverage criterion should be satisfied for the unit testing before the code is submitted to the CVS. Once the code is submitted to the CVS, the integration test cases are executed to verify the application.

2) The experiment result: During the integration testing, total 7 bugs are found, and their distributions are showed in table 2. Table 3 shows the found bugs in code review, unit test and CIT. It could be seen that even the rigid code review and unit testing are done to the unit codes, in integration testing, some bugs are still found immediately after the code is committed. Most of the bugs are related with wrong output of the invoked services. There are no incorrect method call or configuration bugs. As the service interaction in this application is relatively simple, by rigid code review and unit test, nearly all the configuration bugs have been removed before integration testing. However, even very rigid test activities are performed before the integration testing, the semantic related bugs still cannot be avoided, for example, the output format mismatches with what expects. In this case, CIT could find the bugs immediately after the code is submitted to the CVS, and identify the failed interfaces. The CIT continues to show its powerful capability to find the bugs earlier before the system test.

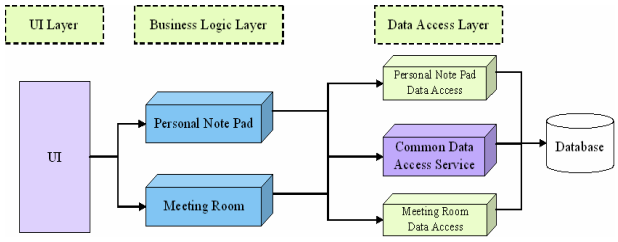

Figure 13. The architecture of meeting room management application

TABLE II. THE BUGS FOUND IN INTEGRATION TEST

Incorrect parameter passing	Wrong input	1
	Wong output	6

TABLE III. THE BUGS FOUND IN DIFFERENT TEST ACTIVITIES

	Code Review	**Unit Test**	**CIT**
Bug number	10	22	7

IX. RELATED WORKS

The CIT has advanced the integration testing to an earlier time during the whole development lifecycle and makes it become more important in bug discovery. However, in academic research, few works are done in the area of integration testing [4, 7, 8]. Most integration testing works mainly focus on the integration test case generation [8], only few works take the test execution into consideration [4, 7]. In

[4], the sequence diagram is applied to generate the integration test cases and stubs. Then the integration testing could be done immediately. However, this work could only be applied to test the pure and stand-alone java program. In [7], an integration test framework is proposed for the object oriented program. In this method, based on the sequence and class diagram, the method interaction order and condition constraints are generated, which are called as coordination contract. Then these coordination contracts are embedded into the implemented java codes by the coordination development environment (CDE). The test data and test driver are designed or generated for the software under test and the embed coordination contract is applied to verify the behavior of software under test. This method also could only be applied to the pure and stand-alone java program and definitely is not suitable for the SOA solution.

Along with the wide adoption of SOA, some works have been started to focus on special testing problems introduced by SOA solutions. In 2004 [6], a Service Integration Test Tool (SITT) is proposed to test SOA solutions. This work focuses on how to collect the execution traces from the distributed services. In our work, the trace collection is in fact done by ITCAM. Further, SITT only considers the web service program type, and the test result verification should be done manually. When different test cases are executed concurrently, no approach is proposed on how to merge the execution traces.

The OMG MDA is an approach to producing code from abstract, human-elaborated specifications. One benefit of MDA is to support platform independent design that is reusable across implementation technologies. In recent years, largely driven by application integration, many SOA researchers and developments have embraced the MDA paradigm to enable the consistent and persistent modeling, design, implementation and management of solutions that integrate existing legacy systems as well as introduce new services. For example: Service Component Architecture (SCA), Business Process Modeling Notation (BPMN) [10] and Business Process Execution Language (BPEL). The introduction of UTF is a natural action in such a trend. It allows generating platform independent test cases from the SOA models, and then attaching platform specific configuration when the implementation technologies are determined. In this way, the integration testing of SOA solutions can be greatly accelerated.

X. CONCLUSION AND FUTURE WORK

CIT is an effective technology to discover the bugs continuously. In this paper, a unified test framework (UTF) is proposed to support the CIT under the SOA environment across the whole development lifecycle. In this framework, a test case is separated into two layers, behavior logic layer and test case configuration layer including the technology specific information. The interaction sequence of the integration test case is generated from the sequence diagram, and a flexible and extensible execution engine is proposed to execute the integration test case. Our previous work, surrogate [2], is applied to simulate the behavior of unavailable components. Base on the execution trace captured by ITCAM [5], the traces coming from a same test case is correlated together by a global case identifier. Based on the object comparison and expression verification, the execution trace is verified with the expected result. The initial experiments have shown the effectiveness of UTF. In the experiments, we have tried to identify the bug types existing in the integration testing of SOA solutions. By this initial classification, we hope to trigger more research about the integration bug types causing by SOA solutions.

As an initial integration test framework, we haven't had chance to apply it in the CIT for large SOA solutions, although the relative small case experiments have shown its potential effectiveness. However, we believe that UTF also works for large SOA solutions. Of course, more technical research topics are still required to improve UTF, such as test data generation from sequence flows, test case execution ordering during CIT, automated test case selection based on change analysis of the submitted code.

ACKNOWLEDGMENT

We thank for Xiong Xiong, Lichuan Shi and Bo Xiao for helping to implement the unified test framework.

REFERENCES

[1] Fowler M., Continuous integration, (2006). http:// martinfowler.com/articles/continuousIntegration.html

[2] He Yuan Huang, He Hui Liu, Zhong Jie Li, Jun Zhu. Surrogate: A Simulation Apparatus for Continuous Integration Testing in Service Oriented Architecture, 2008 IEEE International Conference on Services Computing (SCC 2008), July 8-11, 2008, Honolulu, Hawaii, USA.

[3] Liang-Jie Zhang, Nianjun Zhou, Yi-Min Chee, Ahamed Jalaldeen, Karthikeyan Ponnalagu, Renuka R. Sindhgatta, Ali Arsanjani, Fausto Bernardini . SOMA-ME: A platform for the model-driven design of SOA solutions. 47(3) 397-413. IBM Systems Journal, 2008.

[4] Falk Fraikin, Thomas Leonhardt. SeDiTeC - Testing Based on Sequence Diagrams. In the proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), pages. 261-269. Washington, DC, USA, September, 2002.

[5] Tivoli. Software. IBM Tivoli Composite Application Manager for J2EE.

[6] Schahram Dustdar, Stephan Haslinger. Testing of Service Oriented Architectures – A practical approach. In the proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, pages. 97-109. Erfurt, Germany, September, 2004.

[7] Tom Maibaum, Zhe (Jessie) Li. A Test Framework for Integration Testing of Object-Oriented Programs. In the proceedings of the 2007 conference of the center for advanced studies on Collaborative Research, pages, 252-255. Richmond Hill, Ontario, Canada, 2007.

[8] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger. UML-Based Integration Testing. In the proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00), pages. 60-70. Portland, Oregon, United States, 2000.

[9] Open SOA Collaboration, Service Data Object Specifications, available at http://www.osoa.org/ display/Main/Service+Data +Objects+Specifications.

[10] BPMN. http://www.bpmn.org/.

[11] IBM Software. http://www-01.ibm.com/software/ integration/wid.

[12] Mozilla.org. http://www.mozilla.org/rhino/.