A Novel Approach to Regression Test Selection for J2EE Applications

[1]Sheng Huang, [2]Zhong Jie Li, [2,3]Jun Zhu, [1,*]Yanghua Xiao, [1]Wei Wang
[1]Fudan University, China {shhuang, shawyh, weiwang1}@fudan.edu.cn
[2]IBM China Research Lab lizhongj@cn.ibm.com
[3]Peking University, China zhujun07@sei.pku.com.cn

***Abstract*—Selective regression testing involves retesting of software systems with a subset of the test suite to verify that modifications have not adversely impacted existing functions. The J2EE platform has come to dominate the commercial Java application market. Unlike standalone Java applications, J2EE applications also use configuration files to control the presentation, service, data access, and persistence layers. Traditional regression test selection approaches for standalone Java applications generate regression test suites by only deriving test cases that traverse the changed parts of pure Java classes and thus are generally insufficient for the regression testing scenarios of J2EE applications. This paper proposes an end-to-end regression test selection solution for J2EE applications by providing two unique features—hybrid test-case tracing and unified change identification—that are not addressed by existing approaches. An empirical study is presented to show that this approach can ensure change coverage, and reduce regression test cost for J2EE applications efficiently and effectively.**

Keywords-component; Regression Testing, Tracing, J2EE

I. Introduction

"*Regression testing* is the process of validating modified software in order to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by modifications" [1], which is the key activity for software maintenance. Thus, regression testing is essential to software quality throughout its lifecycle. In a typical scenario, a program P has been tested by a test suite T and then released. During maintenance, developers update P to P' by adding new functions or fixing defects. Thus, regression testing over P' is required before it can be released.

Although in academia a variety of regression test selection (RTS) [1–9] techniques have been proposed to select a cost-minimized subset test suite T' out of T to verify the modified program, there are, to our knowledge, no RTS tools for Java 2 Platform Enterprise Edition (J2EE) applications in the market place as of this writing. We have surveyed current commercial Java applications and found that most of them are J2EE applications. Besides pure Java, other programming languages such as JavaScript (JS) and JavaServer Pages (JSP) are popularly used in J2EE applications. In addition, various frameworks (Struts [19], Spring [20], iBATIS [21], short for SSI, composing a representative three tier J2EE structure, are addressed with end-to-end solution by this paper) are applied to make J2EE applications (When we talked about J2EE applications in the rest of this paper, we mean the J2EE applications based on Struts, Spring, iBATIS, or other similar frameworks) more scalable, maintainable, and structured. As a consequence, unlike pure Java applications, the logic of J2EE applications is controlled by the combination of code written in different programming languages and framework configuration files. In addition to modifications to code (Java, JS, and JSP), configuration changes also make P' behave differently. When we use configuration or configuration files in the rest of this paper, we mean J2EE framework configuration or J2EE framework configuration files.

The characteristics described above create big challenges for existing RTS approaches to be applied on J2EE applications directly because existing approaches handle only pure Java code while ignoring JS, JSP, and configuration files that make significant contributions to J2EE application logic. Thus, existing approaches don't satisfy the safety [7] property that characterizes a good RTS technique. A safe RTS technique selects, under certain assumptions [5], all test cases (T') in the test suite T that may behave differently from P to P'. In this paper, we present a new RTS approach that can handle these challenging characteristics of J2EE applications. Our approach is safe, under the same assumptions, by building the end-to-end behavior of T for P, and identifying all the possible changes that could infer the change of T's behavior in P'. We also implemented the proposed approach by a tool named Optimized Regression Test Selection (ORTS for short). Our empirical study shows that ORTS can ensure change coverage, reduce the regression testing effort efficiently, and scale to industry-size regression testing scenarios under resource and time constraints. To the best of our knowledge, ORTS is the first tool that targets regression test selection for J2EE applications by offering the following two unique features:

Hybrid test execution tracing. Our approach captures end-to-end run-time execution traces at appropriate granularities to construct the call graph specific to J2EE applications by using customized instrumentation and call graph building techniques.

Unified change identification. In our approach, a unified change framework is defined to model any changes by

[*] Correspondence author: Yanghua Xiao (shawyh@fudan.edu.cn). The paper was supported by the National Natural Science Foundation of China under grants No.61003001; Key Program of National Natural Science Foundation of China under grants No. 61033010; Specialized Research Fund for the Doctoral Program of Higher Education No.20100071120032；National Science and Technology Major Project of the Ministry of Science and Technology of China under grants No. 2010ZX01042-003-004.

limited categories of atomic changes with the same granularities as tracing. Besides the modifications to Java, JS, and JSP, the changes made in the framework configuration files are also considered. This feature gives ORTS the ability to identify a complete set of change points of J2EE applications in a unified way.

The remaining sections are organized as follows. Section 2 presents the approach details. An empirical study that shows the effectiveness and efficiency of the approach is reported in section 3. Section 4 analyzes the related work and is followed by a summary and future work in section 5.

II. Proposed Approach

Generally speaking, an end-to-end regression test selection tool consists of three phases: phase 1 involves the test-case tracing of *P*, where the linkage between test cases and run time execution traces is established; phase 2 consists of comparison between the code of *P* and *P′* and identification of change points; in phase 3, the test cases covering the changes are selected to be rerun.

Before moving to the details, we'll briefly introduce the programming model of J2EE applications and explain with an example application why existing approaches are insufficient for J2EE applications.

A. Overview of J2EE Applications

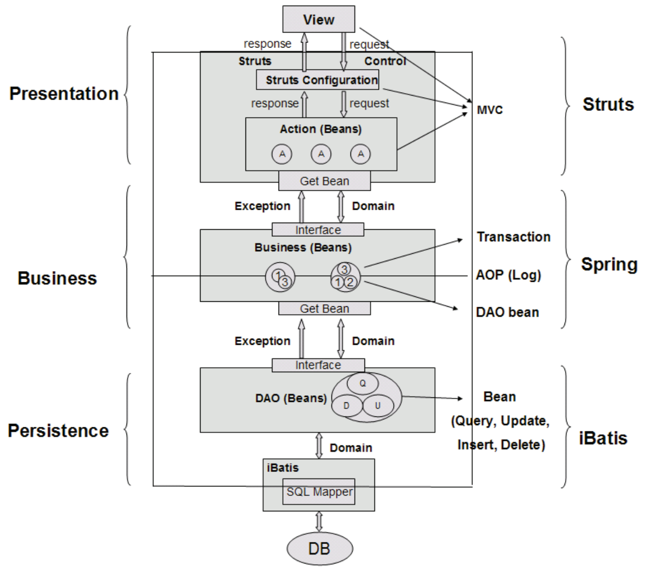


Figure 1. The typical programming model based on common J2EE application frameworks

J2EE is a widely-used platform, differs from the Java2 Standard Edition Platform (J2SE) in that it adds libraries that provide functionality to deploy fault-tolerant, distributed, multi-tier Java software based largely on modular components running on an application server. J2EE applications usually have a three-tier structure controlled by a set of mature frameworks at the three tiers, depicted in Figure 1. In this paper, we target to address the end-to-end regression selection solution of the J2EE applications composed by three most widely used frameworks: Struts [19] (MVC), Spring [20] (Dependency Injection) and iBATIS [21] (OR mapping, from Object to Relational DB schema). All these frameworks configuration files are packaged together with other code to encapsulate J2EE applications in EAR/WAR/RAR formatted by XML files. Generally speaking, the configuration files work together with code to control the logic of the application for better scalability, explicit and clear logical flow, convenient object relation mapping, etc. Figure 1 visualizes the control flow by these three most common used frameworks, which are representatives of all J2EE applications that exist today.

Figure 2 shows the code snippets of a BookStore application as the example, which provides the following functions: login, book CRUD operations (create, retrieve, update, delete). There are totally five test cases for this sample application, T1, T2, T3, T4, T5, corresponding to Login, Create, Retrieve, Update, Delete, respectively. Figure 2 shows some code fragments.

At the Presentation tier, Struts implements MVC in the following way: 1) the user submits a Form through a JSP page, 2) a Controller class of Struts passes the Form to a specific Action class as guided by a configuration file named struts-config.xml, 3) the Action class executes and returns a result string (e.g. "success", "fail"), 4) the Controller directs the control flow to a next JSP page as configured in the same configuration file. In the BookStore example, login.jsp defines a LoginForm which is submitted to the server for user login. The struts-config.xml file has a <action …> element that relates this request to a Java class LoginAction, and also a <forward …> element that relates the "success" login result to the next JSP page list.jsp, where ordinary users can view existing books, bookstore administrators can update, delete existing book items or create new books.

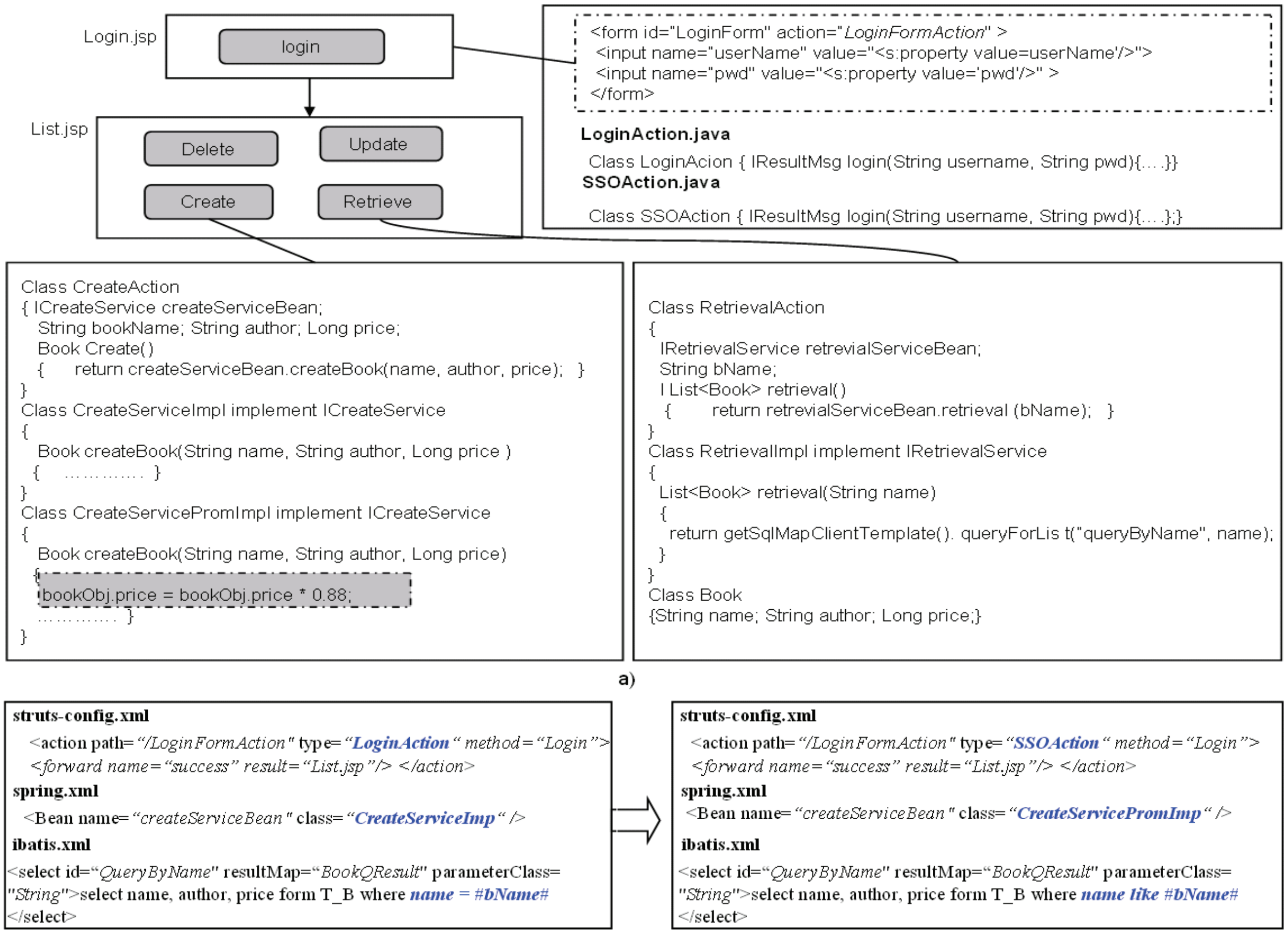
At the Business tier, Spring provides a container for managing the lifecycle of object-oriented code: creating objects, calling initialization methods, and configuring objects by wiring them together. Objects created by the container are also called Managed Objects or Beans. The container is configured by loading XML files containing bean definitions which provide the information required to create the beans (e.g. the Java class that should be instantiated). In the BookStore example, the CreateAction class handling the "create book request" has a createService bean. There can be multiple implementations for the same bean interface. Which implementation to use can be configured in an XML configuration file spring.xml. As is shown in Figure 2b, the initial configuration links the createServicebean to CreateServiceImp class.

At the Persistence tier, iBATIS provides a persistence framework that automates the OR mapping and decouples the mapping from the application logic by packaging the SQL statements in XML configuration files. The benefit is a significant reduction in the amount of code that a developer needs to access a relational database using lower level APIs like JDBC and ODBC. In the BookStore example, the "retrieve book request" is served by the RetrievalAction class, which has a retrievalServicebean. This bean defines a retrievalBooksByName() method (among other possible searching methods), which has its SQL query externalized in an XML configuration file ibatis.xml.

getSqlMapClientTemplate is an iBatis library class that links Java code to query in ibatis.xml by a String ID. Here the ID reference to "QueryByName" will trigger the SQL statement "select name, author, price form T_B where name = #bName#" in initial version of iBATIS configuration file as showed in Figure 2b (The left side is the initial version of all the configuration files and the right is the updated version.).

Now suppose the BookStore application will be integrated into a Software As a Service (SaaS) platform, which provides a single sign-on (SSO) common service. To switch to use the SSO service, there's no need to revise the LoginAction class. We only need to change the <action …> element in the configuration file by replacing LoginAction with SSOAction as is shown in Figure 2b. Also as part of the migration, we decide to launch a promotion campaign when selected books will be sold at an 88% discount. To support this, we don't need to modify the original CreateServiceImpl class. Suppose there is already a class CreateServicePromImpl there, we only need to make changes in the Spring configuration file. We also decide to improve the book query by supporting a fuzzy search. This can be easily done by modifying the SQL statement in the iBatis configuration file.

Figure 2b shows the changes to the configuration files for this BookStore application. Figure 3 shows the call graphs for all the test cases impacted by the changes. Notice that there are no code changes here, thus traditional RTS techniques will select zero test cases to rerun, which can result in undetected regression defects in the Login, Create, and Retrieve functions released to production. The novel approach proposed in this paper could help avoid such risks through three phases. In phase 1, we construct the run-time call graph for each test case of the old version application. The call graph represents the end-to-end control flow of a J2EE application, i.e., the complete invocation chain from an http request is received till an http response is returned. We name it J2CG (J2EE Call Graph) in this paper. The detailed implementation to build the J2CG will be described in Section 2.B. In phase 2, once the new version application is ready for testing, ORTS compares it with the old version and identifies all the changes. Here, we proposed a unified change model to decompose the code edit into a set of interdependent atomic changes. Section 2.C will give the



```
<form id="LoginForm" action="LoginFormAction" >
 <input name="userName" value="<s:property value=userName'/>">
 <input name="pwd" value="<s:property value='pwd'/>" >
</form>
LoginAction.java
 Class LoginAcion { IResultMsg login(String username, String pwd){….}}
SSOAction.java
 Class SSOAction { IResultMsg login(String username, String pwd){….};}
```

```
Class CreateAction
{ ICreateService createServiceBean;
   String bookName; String author; Long price;
   Book Create()
   {     return createServiceBean.createBook(name, author, price);  }
}
Class CreateServiceImpl implement ICreateService
{
   Book createBook(String name, String author, Long price )
  {    …………. }
}
Class CreateServicePromImpl implement ICreateService
{
   Book createBook(String name, String author, Long price)
   {
   bookObj.price = bookObj.price * 0.88;
   …………. }
}
```

```
Class RetrievalAction
{
  IRetrievalService retrevialServiceBean;
  String bName;
  I List<Book> retrieval()
   {      return retrevialServiceBean.retrieval (bName);  }
}
Class RetrievalImpl implement IRetrievalService
{
  List<Book> retrieval(String name)
  {
   return getSqlMapClientTemplate(). queryForLis t("queryByName", name);
  }
}
Class Book
{String name; String author; Long price;}
```

```
struts-config.xml
  <action path="/LoginFormAction" type="LoginAction" method="Login">
  <forward name="success" result="List.jsp"/> </action>
spring.xml
  <Bean name="createServiceBean" class="CreateServiceImp" />
ibatis.xml
<select id="QueryByName" resultMap="BookQResult" parameterClass=
"String">select name, author, price form T_B where name = #bName#
</select>
```

```
struts-config.xml
  <action path="/LoginFormAction" type="SSOAction" method="Login">
  <forward name="success" result="List.jsp"/> </action>
spring.xml
  <Bean name="createServiceBean" class="CreateServicePromImp" />
ibatis.xml
<select id="QueryByName" resultMap="BookQResult" parameterClass=
"String">select name, author, price form T_B where name like #bName#
</select>
```

Figure 2. Sample application snippets and configuration files

details. In phase 3, we map the changes to the J2CG of each test case, and then identify those test cases whose J2CG has a changed node or edge. By considering additional changes of framework configuration files, T1, T2, and T3 will be selected to verify the configuration change as Figure 3 shows. In contrast, code analysis could only find test cases impacted by code changes, and ignore those impacted by configuration file changes like T1, T2, T3 in this case.

SQL statement. The nodes representing such functions will be named *virtual node* in this paper. For an edge, if it represents a direct, explicit function invocation, it's named *node call*; if it represents an indirect, implicit function invocation or a web page transition guided by the configuration file, it's named *virtual call*; if it represents a function invocation using Java reflection or dynamic dispatch [12], it's named *look up call*.

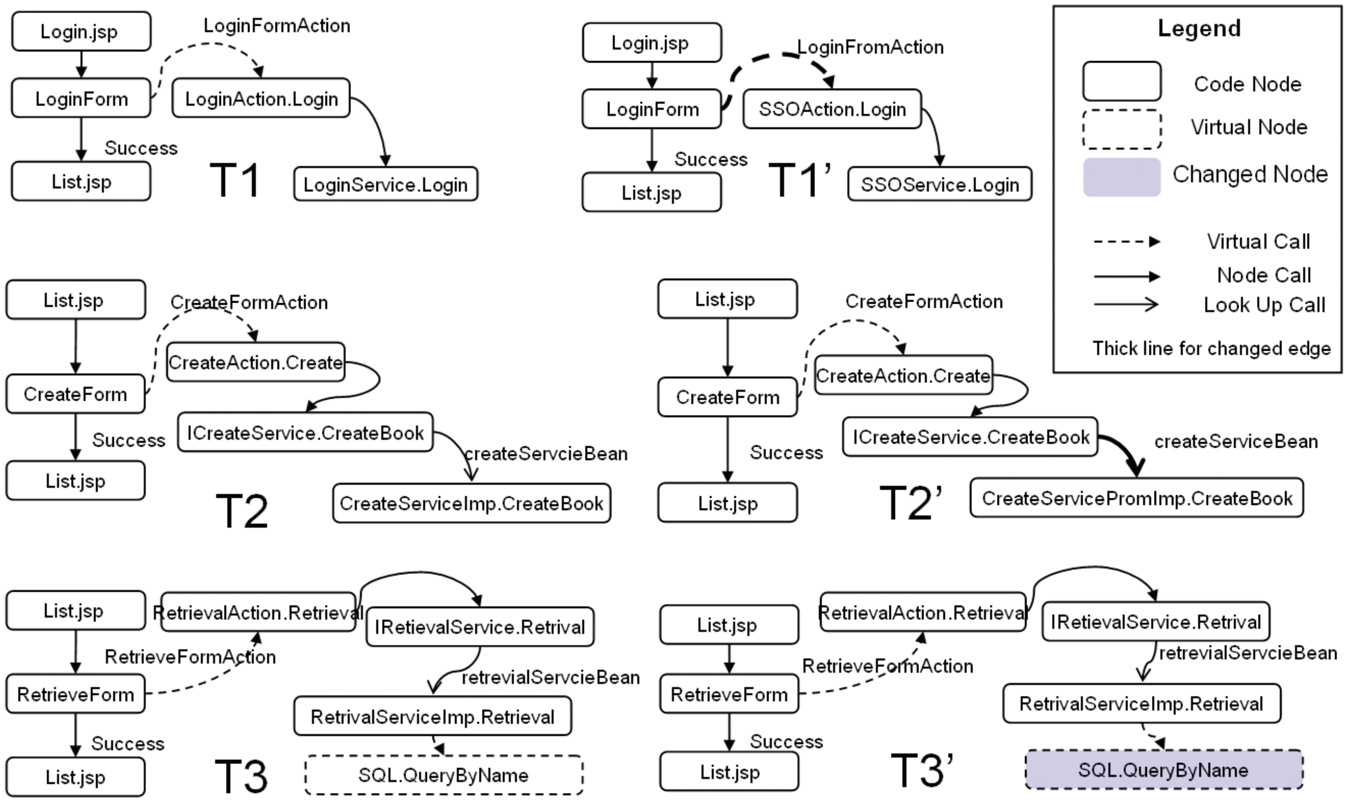


Figure 3. J2CG for selected test cases of the sample application

B. *Hybrid Test-Case Tracing*

In order to ensure the safety property of RTS, a tool needs to build a complete end-to-end call graph in phase 1. If any part is missing in a call graph, the change to this part will not be identified as having impact on the call graph and the related test case. A simple call graph is suitable for representing the flow within a general Java application as used in [5], but cannot accommodate some characteristics of a J2EE application, such as virtual calls, reflection method lookup calls represented in configuration files, etc. We define an extended call graph to represent the run-time behavior of J2EE applications—J2EE Call Graph (J2CG).

The J2CG is made up of nodes and edges, where nodes represent execution of functions, and edges represent function invocations or transitions from one function to another. Table 1 lists the types of functions relevant in J2EE applications. The first three categories are easy to understand. The nodes representing these types of functions will be named *code node* in this paper. The last one denotes functions defined in framework configuration files, e.g. an

Reflection [1] allows programmatic access to information about the fields, methods and constructors of loaded classes and use of the reflected fields, methods, and constructors to operate on their underlying counterparts on objects. Notice that reflection is one assumption of regression bias [13] by Rothermel, and impact of reflection change is not considered in their RTS solution to Java applications [1, 7]. But in current popular J2EE programming model, reflection is widely used in J2EE frameworks to implement external logical control to the J2EE applications by regarding configuration files. That's the reason to analyze the configuration files update for safe regression test selection for J2EE applications.

Formally, we use J2CG(P, t_i) to model the call graph of test case t_i for application P. It consists of Nodes(P, t_i) and Edges (P, t_i). Each node is uniquely identified by a global name that is constructed as explained later.

Table 1. Types of functions

Category	Functions
Java	Java method invocation
JSP	Script functions

	JSP loading event Form submit
JS	Script functions JS method invocations
Configuration files	Execution of a fragment, e.g. SQL statement

A J2CG graph is constructed from the test case execution trace based on code instrumentation. Instrumentation is the addition of source or byte-codes to methods for the purpose of gathering data to be utilized by tools. Since the changes are purely additive, these tools do not modify application state or behavior. To support different programming languages and control logic introduced by J2EE frameworks, ORTS uses a hybrid instrumentation approach. A customized Ajax[22]-like instrumentation over JSP, JS and configuration files is used to collect JSP loading events, JS method call, and Form submissions.

J2EE applications use JSP and JS to program user interfaces. ORTS instruments JSP and JS code through their Form and JavaScript method, respectively. A JSP file contains multiple forms, each of which is associated with an Action class as specified in the framework configuration file. When a form is submitted, its Action class is executed and the service logic is triggered. Thus, the J2CG needs to capture the dynamic information about which specific Action class execution is triggered by which Form to make test cases distinguishable. There are two places of instrumentation to capture the virtual call relationship. The first place is the instrumentation to the body of a Form by adding two query parameters, *FormPar* and *PathPar* to the *action* attribute as the following code fragment shows. Then these values will be encapsulated in the HttpRequest[23] and transferred to a Servlet[24] class also instrumented into the application package.

*<form id="LoginForm" action="LoginFormAction**? Form-Par=LoginForm&PathPar=LoginFormAction**" >*

The second place is to add the following code to the Action class (popularly used in frameworks implementing MVC pattern reading the Form and Path transferred in from HttpRequest.

Form=HttpRequest.getParameter("FormPar")
Path= HttpRequest.getParameterByName("PathPar")

Here we use $Form^{Path}$ to represent the Form Submission event triggered in UI (User Interface). Once the action receives one $Form^{Path}$, an edge from $Form^{Path}$ to the method of action class is added to represent the virtual call from UI to Action class.

Notice that the virtual calls specified by configuration files (Struts here for JSP actions) are in fact embodied by a series of external calls in the external library classes of J2EE frameworks. The external library classes implementing a framework will not be instrumented for two reasons: 1) too much instrumentation will cause too high, unacceptable runtime overhead; 2) usually the framework library class has no change during the lifecycle of a J2EE application, and even if it changes, it should be backward-compatible and have no impact to the user applications.

The instrumentation to JS happens at two places as well. Firstly, the following Ajax JS code fragment will be instrumented to each JS method. During the execution of the JS method, a global *JS Method Name* will be transferred to the server side asynchronously to build the call graph of the presentation part. The JSP loading events are also instrumented with the same approach as the JS Method to complement the UI call graph.

xmlHttp = new XMLHttpRequest();
xmlHttp.open("GET","ProfilerURL?"+JSMethod Name,true);
xmlHttp.send(null);

Secondly, a Servlet class (Say UI profiler in ORTS, used to collect the runtime Ajax call back to build the call graph for Script functions, JS method invocations and JSP loading events) is instrumented into the application package to receive the Ajax call back by mapping to the URL "*ProfilerURL?*".

AspectJ [18] is used to instrument Java code, combined with some other customized codes as Java profiler of ORTS, to capturing the class variables, class names and method names as the globally-qualified method identifier and build the call graph for Java code. For example, the call to method createBookBean by the class variable ″createService″ in the ″Create″ method of the ″CreateAction″ class will be named by ″createService:ICreateService.CreateBook″. For all the Java method invocations that belong to lookup call, ORTS will add another call edge to the actual method invoked, e.g. ″createService:CreateServiceImpl.CreateBook″ in this example. An edge from ″createServiceBean:ICreate-Service.CreateBook″ to ″createServiceBean:CreateService-Impl.CreateBook″ will be added in the final J2CG as is shown in Figure 3. Notice that, for simplicity the variable name is not added in the node identifier. This method of representing dynamic dispatch also helps to identify the correlation between J2CG to virtual method lookup changes in Java code and reflection method lookup by configuration files, to be described in section 2.C.

With such hybrid instrumentation, the ORTS profiling collector could link the call graph built by the Java Profiler and the UI Profiler together. The dashed arrows in the J2CG are virtual calls from Form to Action class, where the source node belongs to UI part (JSP, Form, JS method) and the target node belongs to Java code part.

After that the ORTS profiling collector will do a further calibration to add some virtual nodes that come from the configuration files and are also called during the runtime execution. Notice that the value of the String variable *"queryByName"* passed into the static method *"getSqlMapClientTemplate().queryForList"* is instrumented (per the semantic of iBATIS configuration rules), where *"queryByName"* indicates another call to the virtual node *queryByName* in the iBATIS configuration file according to its internal reflection semantic. Thus we need to add one edge of virtual call from *RetriveServiceImp.Retrieve* to *queryByName.*

Once the above calibration is done, J2CG for each JSP page is ready. The last step is to compose the page flow. Similar to the virtual call from UI to Servlet /Action Class, instrumentation over the page forward call site and page code (JSP/Servlet) can build this linkage. Now we have got the complete J2CG for one test case t_i. Figure 3 shows some examples of J2CG of T1, T2, T3 for P.

C. Unified Change Identification

This section gives the formal unified change identification method. Our method is to decompose all kinds of changes during software update into Unified Changes, which extend Chianti [10] by formally modeling changes to things other than Java code in J2EE applications. Chianti selects tests to indicate to the functionality that has been affected by a program edit by mapping the test cases tracing to a set of atomic changes. Our method complements Chianti by modeling the changes to JSP, JS and framework configuration files into another 9 fine-grained levels of atomic changes Notice that only the fine-grained atomic changes have the same granularity to the J2CG are considered, like only CM, DM, LC are mapped to the call graph of test cases in Chianti[10]. Table 2 shows all the atomic changes defined here.

Table 2. Atomic changes

Java	DM	Delete a Java Method
	CM	Change body of a Java Method
	LC	Change virtual method lookup
JS	DS	Delete body of script method/fragment
	CS	Change body of script method/fragment
JSP	DP	Delete a JSP page
	CP	Change the loading event of a JSP page
	DF	Delete a Form
	CF	Change body of a Form
Configuration files	VC	Change virtual call
	VLC	Change virtual method lookup by reflection type change
	VNC	Change virtual node call by virtual node delete or body update

DM, CM, LC in Java. DM represents the atomic change that a Java Method is deleted. CM represents that the body of a Java Method is changed. Notice that in real implementation, the comments and format changes will not be counted in. LC represents changes in dynamic dispatch behavior that may be caused by various kinds of code changes. LC is defined as a pair of nodes <V:X.m(), V:Y.m()> indicating that the lookup call from V:X.m() at the call site of variable V to V:Y.m() is changed to V:Y'.m() as various kinds of code changes (e.g., by the addition of methods, by the addition or deletion of inheritance relations, or by changes to the access control modifiers of methods. Y' and Y have inheritance relationship here.) Please refer to [11] for the detailed implementation of LC.

DS, CS in JS. DS represents the atomic change that a JS Method is deleted. CS represents that the body of a JS Method is changed.

DP, CP, DF, CF in JSP. DP represents the atomic change that a JSP page is deleted. CP represents that a code fragment in JSP that will be executed to paint the page is changed. DF represents the atomic change that a Form in the JSP is deleted. CF represents that the body of a Form is changed.

VC, VLC, VNC in Configuration Files. VC is defined as a pair <n, X.m()>, indicating that the virtual call specified by configuration files from n to X.m has changed. Notice that here the call site variable instantiated by class X is in the external library of the framework code, thus the call site class variable is not considered and is not recorded in the J2CG. This kind of atomic changes can be derived by pair-wise syntactically and semantically comparison over the configuration files of P and P'. To exemplify this atomic change, consider the changes of struts_config.xml from P to P'. The virtual call $LoginForm^{LoginFormAction}$ →*Login-Action.Login* in P will be changed to $LoginForm^{LoginFormAction}$ →*SSOAction.Login* in P' while there is no other code changes making it happen as T1' in Figure 3 shows. Then we could mark up < $LoginForm^{LoginFormAction}$, *LoginActi-on.Login* > as a VC. T1 will be selected due to this VC change point.

VLC is defined as a pair <V:X.m(), V:Y.m()>, indicating that the virtual call from V:X.m() by the call site of variable V to V:Y.m() is changed due to the reflection instantiation of bean controlled by the configuration files because of instantiation of V will be changed from Y to Y'. For example the bean *CreateServicebean* in the Action class will be instantiated by the Type *CreateServiceImp* in P. In contrast, it will be instantiated by the Type *CreateServicePromImp* in P' as the changes in the spring.xml according to the semantic of Spring Framework. That will lead the J2CG of T2 changes to T2' in P' as Figure 3 shows. Thus, <createServicebean.ICreateService.Create-Book, createServicebean. CreateServicePromImpl.Create-Book()> should be marked as VLC. This change will make T2 be selected to rerun.

VNC is defined as a pair <n, VN>, indicating that the virtual call from a code node n to a virtual node VN is changed because VN is deleted or the body of the VN is changed in the configuration files of P'. Consider the VN *QueryByName* in P, which is the SQL logic to query the book information by exact author name. In order to support fuzzy search the SQL logic is changed to *"select name, author, price form T_b where **name like #bName#**"* in P' as T3' showed in Figure 3. Thus <*RetrivalService.retrieval* , *QueryByName*> should be marked as a change of VNC. This change leads to rerunning T3.

D. Selection

Once the J2CG graphs of P is obtained and the unified changes are also derived, ORTS could select the change impacted test cases as a regression test suite. We will use the equations below to more formally define how we find regression test suites for J2EE applications. Associated with P is a set of tests T={t_1, … , t_n}. We use the bold text of an atomic change type to denote the universal set of program changes of this type.

$RTS(T)=$

$\{t_i \mid t_i \square T, (Nodes(P, t_i) \cap (\mathbf{CM} \square \mathbf{DM})) \neq \emptyset\} \square$

$\{t_i \mid t_i \square T, (Nodes(P, t_i) \cap (\mathbf{DS} \square \mathbf{CS})) \neq \emptyset\} \square$

$\{t_i \mid t_i \square T, (Nodes(P, t_i) \cap (\mathbf{DP} \square \mathbf{CP} \square \mathbf{DF} \square \mathbf{CF})) \neq \emptyset\} \square$

$\{t_i \mid t_i \in T,\ V{:}X.m,\ V{:}Y.m \in Nodes(P, t_i),\ V{:}X.m \rightarrow V{:}Y.m \in Edges(P,t_i), <V{:}X.m, V{:}Y.m> \in \mathbf{LC}\} \square$

$\{t_i \mid t_i \in T,\ n,\ X.m \in Nodes(P, t_i),\ n \rightarrow X.m \in Edges(P,t_i),\ <n, X.m> \in \mathbf{VC}\} \square$

$\{t_i \mid t_i \in T,\ V{:}X.m,\ V{:}Y.m \in Nodes(P,\ t_i),\ V{:}X.m \rightarrow V{:}Y.m \in Edges(P,t_i),\ <V{:}X.m,\ V{:}Y.m> \in \boldsymbol{VLC}\}$□

$\{t_i \mid t_i \in T,\ n,\ vn \in Nodes(P,\ t_i),\ n \rightarrow vn \in Edges(P,t_i),\ <n,vn> \in \boldsymbol{VNC}\}$

Here a lookup call relationship for VLC and LC is represented as *V:X.m*→*V:Y.m*, indicating the run time type of variable V dispatch the call to *V:X.m* triggering another lookup call from *V:X.m* to *V:Y.m*. Likewise, the virtual call specified by the configuration files is represented as *n*→*X.m*, indicating the virtual call from the UI node *n* ($Form^{Path}$ in this paper) to Action class method *X.m*. The virtual call from a code node *n* to a virtual node of configuration files is represented as *n*→*vn*, where *vn* indicates the virtual node. By applying the selection rules on VC, VLC and VNC, T1, T2, T3 will be selected out respectively and combined as the regression test suite *RTS(T)*. As [10], we also implicitly make the usual assumptions that program execution is deterministic and that the library code used and the execution environment (e.g., JVM, DB) itself remain unchanged.

III. EMPIRICAL EVALUATION

To assess the validity and effectiveness of the proposed approach, we implement the approach in this paper as a tool [17], named ORTS, and conduct experiments on three J2EE projects by focusing on three questions, each related with one of the three assessment factors: scalability, reduction, and effectiveness:

Question 1: Can ORTS scale to this J2EE application?

Question 2: How does ORTS benefit testers by reducing the test suite?

Question 3: How does ORTS perform, especially compared with traditional strategies?

The independent variable in this study is the particular RTS technique used. Five techniques are selected to do the comparison, including our approach:

1) *Rerun All*—All the test cases are rerun for each iteration.

2) *Manual Selection*—The testers select test cases to rerun based on their experience and understanding of the iterations and changes.

3) *General RTS*—Only Java code changes are considered in regression test selection, as supported by most existing RTS techniques for Java applications, [10] is used here

4) *Enhanced RTS*—Extend *General RTS* by considering the JSP and JS atomic changes.

5) *ORTS Selection*—As is proposed in this paper.

A. Experiment Setup

As the objects of our study, we used several releases of each of three J2EE applications produced in three projects, varied by different maturity levels. All of the test cases are user acceptance tests. Project A (with 48 test cases) and project B (with 181 test cases) are both real-world cases of IBM customers in China market while project C is an IBM internal project with 600 test cases.

We followed project A for one month of four iterations and project B for two months of three iterations. The initial version of project A was released to the customer and was in the maintenance phase; each week a new build is produced for bug fixes. In contrast, project B was under urgent development before the first official release. The first iteration took seven days, the second iteration took 14 days to add new features, and the third iteration took 35 days to update a critical component. For project C, since users had evaluated its stability over a long period of use, the main update was to fix bugs or perform minor function changes. The basic information of these projects under evaluation can be found in Table 3.

Table 3. Experiment projects information

Proj ect	#File	KLOC	#Code Node	#Virtual Node	#V	#TC
A	162	38	1,227	36	5	48
B	472	185	2,705	96	4	181
C	1,642	1,960	14,220	58	2	600

Table 3 shows the project information including file number, KLOC, number of Code Node, number of Virtual Node, number of versions (#V), and number of test cases (TC). The projects vary by size: 38 KLOC, 185 KLOC, and 1960 KLOC for projects A, B, and C, respectively. We implemented ORTS and the other RTS techniques in Java on same x386 platform with MS Windows® 2003 Server operating system and 2 GB memory. We then collected the packages updated by the project owners, and executed RTS on the same platform.

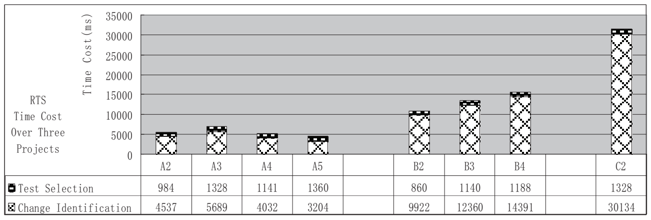
We do not compare the test execution time saving like some other RTS papers [1] for the following two reasons. (1) All the test are GUI manually executed test cases and hard for automation per to the experience of target's test leaders. We think it is not objective to statistic to the time of manual test execution as it varies for tester's skill. (2) As the reduction of RTS has linear relationship with the number of test cases, we think it should be enough to use number reduction to depict the benefit of RTS.

B. Scalability

Table 4. Change and bug statistics

Subjec t	ITV (days)	#Code change	#Java Bug	#PR Bug	#CFA change	#CFA Bug
A2	7	7	2	1	2	2
A3	7	17	1	1	2	2
A4	7	3	1	0	1	1
A5	7	2	0	0	1	0
B2	7	38	10	5	8	3
B3	14	89	29	9	20	5
B4	35	437	43	12	86	0
C2	14	74	3	0	2	2

For the first part, Question 1 concerns the change identification capability for J2EE applications – how completely can all kinds of changes be identified? Only by revealing all changes with the tendency of behavior change of test and mapping them to the behavior of test in P, the RTS process could be safe [5]. To address this part, we used ORTS to run the change identification process automatically. Section 2 has introduced the design that considers all kinds of changes and could guarantee the completeness of change identification. In case studies, to test and verify the ORTS implementation, we further worked with the developers to double-confirm the correctness of the changes identified. In



	A2	A3	A4	A5		B2	B3	B4		C2
▬ Test Selection	984	1328	1141	1360		860	1140	1188		1328
⊠ Change Identification	4537	5689	4032	3204		9922	12360	14391		30134

Figure 5. RTS time consumption data

the evaluation process, after ORTS generates the change report, the developers will check manually whether changes in the report are really logic changes marked by leveraging the code repository CVS tool like Eclipse CVS plug-in to compare change report regarding files of *P* and *P'* one by one with their knowledge of *P* and *P'*.

As Table 4 shows, all the code related atomic changes (short for code changes in table 4, including Java code changes, JSP and JS code changes), and configuration file atomic changes (CFA Change) identified by ORTS are logic changes confirmed by developers. The column "Subject" lists project iterations, e.g. A2 stands for project A 2nd build for the 1st iteration. The column "ITV" stands for iteration interval. The column "#Java Bug" indicates the number of bugs resulting from Java code changes while the column "#PR(Presentation) Bug" indicates the additional bugs resulted from presentation (JSP, JS) regarding changes. It is straightforward to get such kind of information because the developers will help us to point out which changes do cause bugs when they fixed relevant bugs. From the two columns we could find that ORTS could complement existing Java-Oriented regression test selection approach by revealing additional bugs resulting from PR changes. Likewise, the column "CFA Bug" indicates the number of additional bugs caused only by configuration file changes. Notice that although B4 has the most configuration file changes, the "CFA Bug" is zero. The reason is that the bug-revealing test cases covering these changes also cover code changes and thus the related bugs are counted in code change categories. This evidence shows that ORTS could be scaled to find all the changes with tendency to reveal bug in J2EE application. In addition, compared with exhausted developers, ORTS only needs a couple of seconds to do this job. In fact, in the manual selection method for the IBM internal project C, the testers will try to identify the changes manually and map them to impacted test cases. Regarding IBM's benchmark data, the testers needed to spend one person day to select out 20 test cases to be rerun from the total test suite. For this case, ORTS saves 5.5 person day, as our previous work [17] shows. And in other real cases, the testers may not have enough time or professional skills to identify all these changes. It is also a big challenge for them to map the changes to impacted test cases due to lack of the exact linkage between test case and code. That's why ORTS will outperform manual selection in most cases as we will describe in later sections.

The second part of Question 1 is about the overall RTS time consumption. As Orso proposed in [9], it's important that a RTS technique is scalable to large applications to be practical. ORTS aims at a linear time consumption complexity in terms of project size. As Figure 5 shows, ORTS finished the change identification from 4 seconds to 32 seconds as the project size increases from 38 KLOC to 1,960 KLOC. The test selection time is steady around 1~2 seconds. Although we do not have a formal proof, this result is promising and shows that our technique is linear in complexity, and it can scale to even larger J2EE application and complete regression test selection efficiently.

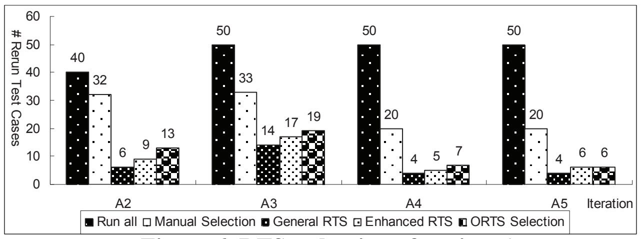


Figure 6. RTS reduction of project A

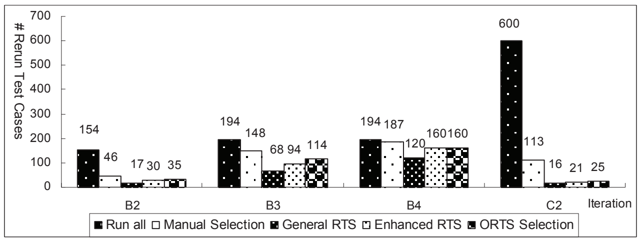


Figure 7. RTS reduction of project B and C

C. Reduction

Question 2 asks how much ORTS can benefit testers by reducing the test suite. This is a primary mission of RTS.

As Figure 6 and Figure 7 show, *General RTS, Enhanced RTS* and *ORTS Selection* offer considerable savings in rerun test cases compared with *Manual Selection* under most conditions.

The *ORTS Selection* may select more test cases than the *General RTS* and *Enhanced RTS* because additional test cases impacted only by the CFA changes are selected out in *ORTS Selection*. However, the additional test cases have the extra ability of revealing more defects, as shown in Figure 8 and Figure 9.

General RTS has the best reduction by only considering the Java atomic changes, but it has the highest risk of missing bugs caused by JS, JSP and CFA Changes, as shown in Figure 8 and Figure 9. For B4, *ORTS Selection* and *Enhanced RTS* select the same rerun suite because the test cases affected by CFA configuration changes are also selected by looking at changes in Java, JS and JSP code. From Figure 7, we find that for B4, the test case reduction effect is not significant (ORTS selects 160 out of the 194 total). The reason is that this iteration lasted for one month when a large number of design changes were made and impacted most of the test cases.

From this investigation, we can see that the application maturity impacts the rerun test suite reduction rate. Here we use *RSRR* as follows: *RSRR = (#Rerun All – #RTS Approach)/(#Rerun All)* to depict the RTS Suite Reduction Rate(*RSRR*). We find that the average *RSRR* for the three projects increases as the maturity increases, 42% for low maturity project B, 77% for middle maturity project A, and

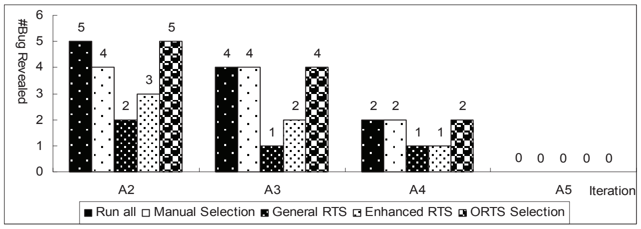


Figure 8. RTS effectiveness of project A

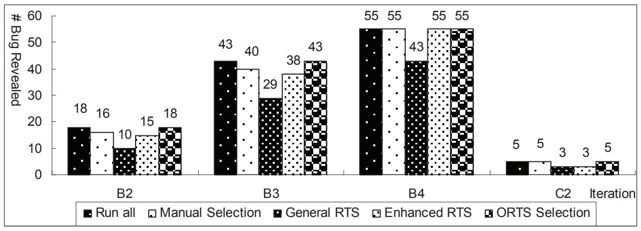


Figure 9. RTS effectiveness of projects B and C

96% for high maturity project C. Although more studies are required to confirm the observation, based on the existing experiments we can conclude that ORTS can deliver considerable regression test suite size savings during the regression testing, especially for more mature projects.

D. *Effectiveness*

Question 3 concerns the effectiveness of ORTS in defects discovery capability. Although *Rerun All* always offers the best defect discovery capability, its high cost makes it unacceptable. To make comparison, we use the defect number found by *Rerun All* as the total regression defect number during an iteration.

As the experiment results in Figure 8 and Figure 9 show, *ORTS Selection* always achieves the same safety as *Rerun All* and outperforms all the other three techniques because it can select additional bug revealing test cases correlated by CFA changes, JS and JSP changes.

Compared with *Enhanced RTS*, *General RTS* has the risk of missing bugs introduced by JS and JSP changes. It missed 1 and 1 bugs for A2 and A3, respectively and missed 5, 9, and 12 bugs for B2, B3, and B4, respectively, and missed 2 bugs for C2. Notice tha*t General RTS* and *Enhanced RTS* revealed the same number of bugs in C2 because the JSP and JS changes didn't introduce any additional bugs.

Compared with ORTS Selection, *Enhanced RTS* has the risk of missing bug revealing test cases due to configuration file changes. It missed 2, 2, and 1 bugs for A2, A3, and A4, respectively, and missed 3 and 5 bugs for B2 and B3, respectively, and missed 2 bugs for C2.

These results clearly show the limitation of only considering Java code changes for J2EE applications and the value added by considering other types of changes.

IV. Related Work

Software change is frequent and risky. Regression testing is necessary, but also costly. The retest-all strategy is straightforward and safe, but suffers from consuming a lot of time and resources [7]. The selective regression testing strategy aims at finding a subset of test cases to rerun.

Many techniques, such as program dependence graph [2], path analysis [3], dataflow [4], and graph walk [5], have been proposed for regression test selection. All are source-code based. They can be applied effectively to regression testing at the unit or component level, but they are generally too expensive when applied to large programs [6]. In addition, they are unsafe to be applied on J2EE applications because they do not take into consideration the change impact of J2EE framework configuration files. Safety [7] is important because it ensures that the selected test suite *T'* reveals all of the regression defects in the revised program *P'*. The method proposed by Harrold et al. [1] is a safe one for pure Java applications. However, it does not take into account the changes caused by J2EE framework configuration file changes. Existing code-based regression tools for large Java applications, such as DEJAVOO [9], provide a safe approach to scaling regression testing for large Java applications by pair wise control flow traversal approach, but they lack support for popular programming languages (e.g., JS and JSP) and J2EE framework configuration changes. Ren et al. [10,12] introduce a change impact system, which models the Java changes in atomic changes and derives the impacted test cases as regression test suite by mapping the atomic changes to call graphs of test cases. Our approach outperforms it by modeling the changes of JSP, JS and other popular used frameworks in atomic changes and deriving the regression test suite by mapping them to the J2CG representing the run time behavior of test cases in J2EE applications. With such extensions, we provide a safe and scaling regression test selection approach to J2EE applications.

Although [8] seems a work that addresses configuration-aware regression testing, it uses "configuration" in a different meaning - it's user configuration of the software via software provided settings such as Tool->Option. Our work addresses the problem of selecting affected test cases considering the changes of framework configurations in a new software version. [8] addresses the problem of generating test cases for different user configurations or selecting (sampling) test cases with higher defect revealing capability from the potential infinite test cases due to the large user configuration space. In terms of their usage in configuration-aware regression testing for evolving J2EE software systems, the two works are orthogonal and can be used sequentially: work in [8] is used first to select configurations and the related test cases, and then our work is used to further select those test cases affected by the application changes.

Other works in recent years focus on the problem of regression test selection in difference views. [14] concerns the regression testing for Web Service and describes an approach and a tool to allow users to run a test suite against a service to discover if functional and non-functional expectations are maintained. [15] discusses the compatibility and regression testing of COTS-Component-Based software if the COTS components are updated or changed to another one. [16] proposes a new regression test selection technique for AspectJ programs. At the core of this technique is a new control-flow representation for AspectJ software which captures precisely the semantic intricacies of aspect-related interactions. Based on this representation, they develop a novel graph comparison algorithm for test selection. None of them could be applied to solve the regression test selection problem faced in J2EE applications.

V. SUMMARY and Future Work

This paper proposes a RTS approach with systematic support to the end-to-end J2EE programming stack—not only Java, JSP, and JavaScript, but also widely used frameworks such as Struts, Spring, and iBATIS that leverage *configuration files* to embed control flow and programming logic. The novelty of this work lies in two unique features—hybrid test-case tracing and unified change identification—that can overcome the limitations of existing approaches for J2EE applications. During the hybrid test-case tracing, a weaver utilizes AspectJ [18] to instrument Java code and also uses a customized Ajax-like instrumentation over JSP and JS to collect such things as JSP loading events, JS method invocations, configuring fragment execution, etc., during runtime tracing. From the collected trace data, a hybrid call graph named J2EE Call Graph (J2CG) is constructed to represent the end to end function invocations for each test case execution. In the unified change identification phase, 12 types of changes are extracted by analyzing the builds of P and P'. Besides the code changes as supported by the generally safe RTS techniques, configuration changes are also detected. The preliminary use of the tool in real-world customer cases shows that this approach can help ensure effective change-point coverage and reduce the regression testing effort efficiently, making RTS scenarios possible for J2EE applications. We have done an extensive investigation over frameworks popularly used in J2EE applications and come to the confidence that most of the logic controlled by them is similar to SSI frameworks. Although not thorough, they're sufficient in most cases, and extensible to accommodate other cases. By the time we submitted this paper, we have already applied the approach to support other frameworks like WebWork [25], Hibernate[26].

In future, we plan to experiment the approach on more J2EE applications with more kinds of frameworks to further verify its generality and refine our strategy for better regression test selection effectiveness and regression test-case management. Another interesting aspect of software change is database schema change, which is not considered in this paper. To address this kind of change, we need to build the linkage between test cases and data access using test case profiling, as well as identify database schema changes during change identification. This is another future work direction.

REFERENCES

[1] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A. and Gujarathi, A. 2001. Regression test selection for Java software. Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (Tampa Bay, FL, 2001). ACM, New York, NY, 312-326.

[2] Bates, S. and Horwitz, S. Incremental program testing using program dependence graphs. 1993. Proc. 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (Charleston, SC, 1993). ACM, New York, NY, 384-396.

[3] Benedusi, P., Cimitile, A., and De Carlini, U. 1988. Post-maintenance testing based on path change analysis. Proc. Conf. on Software Maintenance (Scottsdale, AZ, 1988). IEEE Computer Society, Washington, DC, 352-361.

[4] Harrold, M. J. and Soffa, M. L. 1989. An incremental data flow testing tool. Proc. 6th Intl. Conf. on Testing Computer Software (Washington, DC, May 1989). ACM, New York, NY.

[5] Rothermel, G. and Harrold, M. J. 1993. A Safe, Efficient Regression Test Selection Technique, TOSEM 6(2) Apr 1997, 173 – 210.

[6] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. 2001. An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Meth. 10, 2 (April 2001), 184-208.

[7] Rothermel, G. and Harrold, M. J. 1996. Analyzing Regression Test Selection Techniques. IEEE Trans. Softw. Eng. 22, 8, (Aug. 1996), 529-551.

[8] Qu, X., Cohen, M. B., and Rothermel, G. 2008 Configuration-aware regression testing: an empirical study of sampling and prioritization. Proc. of the International Symposium on Software Testing and Analysis (Seattle, WA, 2008). ACM, New York, NY, 75-86.

[9] Orso, A., Shi, N., and Harrold, M. J. 2004. Scaling regression testing to large software systems. Proc. of the 12th International ACM SIGSOFT Symposium on the Foundations of Software Engineering (Newport Beach, CA, 2004). ACM, New York, NY, 241-251.

[10] Ren, X., Shah, Tip, F., Ryder, B. G., and Chesley, O. Chianti: a tool for change impact analysis of Java programs. Proc. 19th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (Vancouver, BC, Canada, 2004). ACM, New York, NY, 432-448.

[11] Ryder, B. G., and Tip, F. Change impact for object oriented programs. Proc. of the 2001 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis and Software Testing (Snowbird, Utah, United States, 2001). ACM, New York, NY, 46-53

[12] Ren, X., Ryder, B. G., Stoerzer, M., Tip, F.. Chianti: a change impact analysis tool for java programs. Proc. 29th Intl. Conf. on Software Engineering (Minneapolis, MN, May, 2007), ACM, New York, NY, 664-665.

[13] Bible, J. and Rothermel, G. A unifying framework supporting the analysis and development of safe regression test selection techniques. Technical Report 99-60-11, Oregon State University, Dec. 1999.

[14] Penta, M., Bruno, M., Esposito,G., Mazza, V. and Canfora, G. Web Service Regression Testing. In book Test and Analysis of Web Services, 2007, Part II, 205-234. Springer Press.

[15] Mariani, L., Papagiannakis, S., Mauro Pezz, M.. Compatibility and Regression Testing of COTS-Component-Based Software. Proc. 29th Intl. Conf. on Software Engineering (Minneapolis, MN, May, 2007), ACM, New York, NY, 85-95.

[16] Xu, G., Rountev , A.. Regression Test Selection for AspectJ Software. Proc. 29th Intl. Conf. on Software Engineering (Minneapolis, MN, May, 2007), ACM, New York, NY, 65-74.

[17] Huang, S., Zhu, J., Ni, Y. ORTS: a tool for optimized regression testing selection. OOPSLA Companion 2009: 803-804

[18] Eclipse AspectJ. http://www.eclipse.org/aspectj/.

[19] Apache Struts project. http://struts.apache.org/.

[20] Spring. http://www.springsource.org/.

[21] The iBATIS Date Mapper. http://ibatis.apache.org/.

[22] http://en.wikipedia.org/wiki/Ajax_(programming)

[23] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[24] http://en.wikipedia.org/wiki/Servlet

[25] www.opensymphony.com/webwork/

[26] www.hibernate.org/