An Automated Test Code Generation Method for Web Applications using Activity Oriented Approach

David A. Turner[†], Moonju Park[‡], Jaehwan Kim[‡], and Jinseok Chae[‡]
[†]Department of Computer Science
California State University San Bernardino
dturner@csusb.edu
[‡]Department of Computer Science and Engineering
University of Incheon, Korea
{mpark,jhkim,jschae}@incheon.ac.kr

***Abstract*—Automated tests are important for Web applications as they grow more complex day by day. Web application testing frameworks have emerged to help satisfy this need. However, used without a model that is designed for system evolution and realization, maintaining test code becomes cumbersome and inefficient. This paper describes an activity oriented approach to engineer automated tests for web applications with reference to a web application developed for grant funding agencies. In this approach, the developer defines a domain model to represent application state, and uses a test activity graph comprised of self-validating user interactions to verify application correctness. We have implemented a test code generator called iTester using activity-oriented approach.**

***Index Terms*—Activity Oriented approach, Automated test, Web application**

I. Introduction

The ubiquity of web browsers has drastically increased the popularity of web applications. Hence web applications have become common as well as complex. As web applications become more complex, inevitably testing web applications have also become complex and the code cumbersome to maintain. To ease the difficulty of web application testing, a plethora of automated tools and testing frameworks are now available for different aspects of testing, such as unit testing, functional testing, and load and performance testing. HttpUnit is one such open-source, request/response-based testing framework that is typically used with the JUnit test framework. It emulates the user interaction and browser behavior of web applications, and also facilitates examination of responses using Java code. It is fairly easy to write tests that quickly verify the functioning of a web site using HttpUnit.

Our activity oriented approach is one possible approach to test web applications; it is a black-box test based on the user interactions with the web application. As web applications become more sophisticated, the functionalities of web pages have become more intricate, convoluted and loaded with links, buttons, multiple forms and so on. Manual testing of such web applications, though unavoidable, is grueling and often not dependable. Hence it is inevitable to develop automated tests that can expose failures and deviations from intended behavior. The user interactions may be as simple as clicking a button or as complicated as filling several forms to accomplish a task. Such likely user interactions are identified, analyzed, and defined to build an activity oriented testing model. This test model can be applied to functional testing and load testing. It can also be used for data building (populating the application with data) for the purpose of manual testing and intermediate client evaluations.

A number of other researchers have recently published results on various web application testing methodologies. [1] proposed a model to analyze Web applications and test strategy using UML on static web pages. [2] studied test case generation for XML-based Web applications to test reliability in data interaction between system components. Kung *et al.* [3] describe a methodology that uses an object-oriented web test model to support application testing. Lucca *et al.* [4] describe an object-oriented testing methodology that is based on modeling pages, their subcomponents and relationships. Tian *et al.* [5] describe a hierarchical testing approach that utilizes both white box and black box testing strategies, and employees automated test generation using Markov chains. Sampath *et al.* [6] describe the application of concept analysis to user-session data captured by an application that is in service to compute a reduced set of tests with good coverage properties. Qi *et al.* [7] report on an agent-based approach to testing. Karam *et al.* [8] describe a testing methodology based on a work-flow model for testing applications built around a model-view-controller architecture. Research on automatic test case generation based on use cases can be found in [9], [10]. The use case driven approach is a top-down approach that builds test cases from high-level scenarios based on use cases.

In general, our approach is distinguished from other approaches by hiding details of page interactions behind a simple domain model view of the application state, and expressing interdependencies through an activities graph. Our intent is to provide an effective methodology that application developers can comprehend and adopt quickly and easily. The rest of this paper is organized as follows. Section II describes the activity-oriented approach, and Section III describes the test process. Section IV explains how the test code generation program iTester is implemented. Finally, Section V concludes our work.

II. Definition of Test Acitivities

We define a test activity as a self-validating user interaction with the system. A test activity serves two purposes: it tests a given functionality and it constructs a test fixture for other test activities. An example of a test activity is for the admin user to login and verify that the admin home page is returned. We refer to this activity as the *admin-login* activity. Our naming convention for test activities is to identify the user (doer of the activity), followed by the action (in the form of a verb), and optionally followed by the object being acted on. For another example, *applicant-submit-application* is the test activity in which an applicant submits an application and then verifies that the application exists within the system. In the remainder of this paper, wherever we use the word *activity*, we mean test activity.

Test activities may have dependencies with other test activities. Dependency between test activities is defined as follows: test activity B depends on test activity A if the construction of an application state for running B requires the running of the activity contained in test activity A. (We define *application state* to be the aggregate of all data within the application at a given moment.) The test activity graph in Figure 1 shows an example of dependencies among activities in the Solicitation Management System (SMS), which is a web application developed for grant funding agencies.[1].

An independent test activity does not depend on application state in order to carry out a test. In the SMS system, the admin user is part of the initial state of the application, and can not be deleted. Thus, the *admin-login* activity is an example of an independent activity, because its execution does not depend on the application state.

A dependent test activity, on the other hand, depends on application state. For example, a test activity in which the applicant submits an application depends on the existence of a program (solicitation) that is open for applications. The *applicant-submit-application* activity is a dependent activity, because its execution depends on the existence of a solicitation and an applicant. In general, dependent activities require suitable application states in which to execute. To run the *applicant-submits-application* test activity, we need to have first run the activity contained in the *officer-create-solicitation* test activity.

A test activity may involve one or more HTTP requests and one or more web pages. The *admin-login* activity sends three HTTP requests to the application server: a request to fetch the page with the login form, a request to submit the login form, and a request to fetch the admin home page. The *applicant-submit-application* activity sends four HTTP requests: a request to retrieve the page with the initial application form, a request to submit the initial form, a request to submit the second form, a request to submit the third form.

In the context of testing, an application state required to execute a given test is referred to as a *test fixture* for the test.

[1]Development of SMS was funded by the Office of Technology Transfer and Commercialization at California State University San Bernardino.

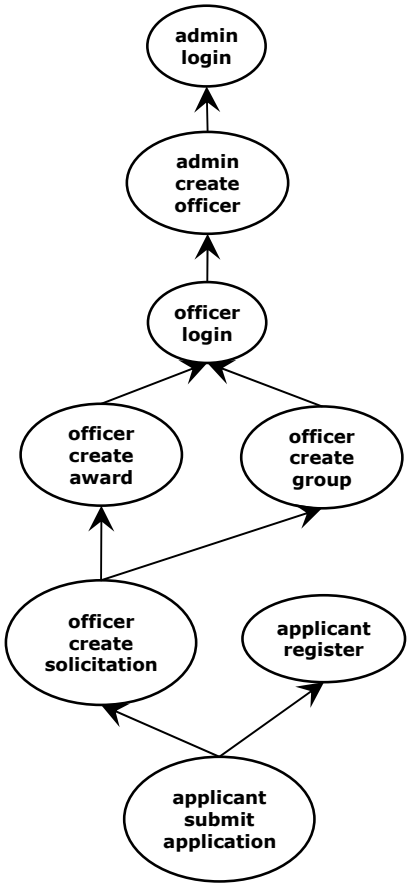

Fig. 1. Test activity graph for the applicant-application submit activity

In the activities-oriented approach to testing, we transform an application state into a test fixture for a given test activity by executing the test activities on which the given test activity depends. For example, to create a test fixture for the *applicant-submit-application* test activity, we execute the *officer-create-solicitation* and *applicant-register* test activities. Because test activities validate themselves, they can be used as test cases; because test activities transform the application state, they can be used to construct fixtures for other tests.

III. Activity Oriented Test Process

The activity oriented approach follows a four step process to test web applications. These steps are explained in the following.

Step 1: Identifying Activities

Because it is infeasible to test every possible interaction with the system, the developer must decide which activities provide sufficient coverage of system functionality to insure the absence of faults. For each of these selected activities, the developer identifies those activities that are needed to construct their test fixtures, which may require the identification of

additional activities. The activities within the resulting set are classified as independent or dependent. Analysis of our SMS system revealed three independent activities: *admin-login*, *applicant-register* and *evaluator-register*.

Step 2: Developing a Test Activity Graph

After identifying the activities to be executed, the developer creates a test activity graph to represent the activities and their dependencies. A test activity graph is basically a work flow graph with arrows reversed to show dependencies rather than sequencing constraints. Thus, a test activity graph $TAG = \{A, D\}$ is a directed acyclic graph where the vertex set A is comprised of the identified activities, and the directed edge set D is comprised of dependencies between activities. There exists an outgoing edge in the graph from activity i to j if i is dependent on j. Conversely, there exists an incoming edge to activity i from another activity k, if k is dependent on i. Figure 1 illustrates the sub-graph of the SMS test activity graph that shows the *applicant-submit-application* activity and its dependencies.

The test activity graph may be simplified by omitting redundant edges. An edge (i, j) is *redundant* if there exists a path from activity i to j other than the trivial path comprised of the edge (i, j). As an illustration, the *officer-create-solicitation activity* depends on *officer-login*, *officer-create-award* and *officer-create-group*. However, since *officer-create-award* and *officer-create-group* are also dependent on *officer-login*, the dependency edge from *officer-create-solicitation* to *officer-login* is redundant, and is therefore omitted. The simplified graph upholds the dependency information as well as reduces the number of edges in the graph. The simplified graph is desirable since the test algorithm, as explained later, will traverse fewer edges, and is therefore more efficient. It is important to note that each activity in the graph represents a test case by itself, because each activity tests some functionality of the target system.

Step 3: Developing Activity Oriented Test Model

The test model is derived from the test activity graph. This model is built using two primary classes: `DomainObject` and `Activity`. The `DomainObject` class is used to define subclasses that represent the objects within the application that are understood by the user, such as officer, applicant, application, etc. The test code uses these domain objects to represent application states and build test fixtures.

Domain objects have `id` attributes and `createRandom` methods. The `id` attribute is used by the application to identify particular instances of domain objects. The `createRandom` method is a static method that subclasses override; it is called in order to obtain an instance of the domain object whose member variables are set to random, unique values (when possible).

Domain objects that represent user roles have the capacity to interact with the application. For each defined activity in the system, there will be a corresponding method in a user class to carry out the interaction with the target application.

```
test()
{
    if state != UNTESTED then return

    for each dependent activity ACT
    {
        if ACT.state == UNTESTED
        then ACT.test()

        if ACT.state != PASSED
        then state := UNREACHABLE
    }

    if state == UNREACHABLE
    then return
    else execute()

    if exception occurs during execute()
    then state := FAILED
    else state := PASSED
}
```

Fig. 2. Activity Test Algorithm

The `Activity` class is used to define subclasses that represent the activities that operate on the domain objects. `Activity` classes encapsulate the dependencies represented in the test activity graph. For each activity that appears in the test activity graph, we define a subclass of `Activity`. Each of these subclasses must implement the abstract method *execute*, which carries out the activity by invoking appropriate methods on one or more domain objects. For this reason, an `Activity` must contain a reference to at least one user object in order to interact with the web application.

Step 4: The Activity Test Algorithm

After building the test model, the final step is to write code that instantiates all needed instances of domain objects and activities, and which properly captures dependencies as expressed in the test activity graph. The `test` method of a given activity iterates through the activities on which the given activity depends, invoking their `test` methods prior to calling the *execute* method of the activity. In this way, the test fixture is created prior to calling the activity's *execute* method.

A free activity is an activity on which no other activity depends. After the domain objects and activities are created, the `test` method is executed on all free activities. It is sufficient to invoke the `test` method on only the free activities, because the test function recursively invokes itself for all dependent activities. Thus, the test function will be invoked at least once for all activities in the system. When viewed from the perspective of a work flow graph, the activity test algorithm

essentially produces a topological sort of activities, i.e. a list of activities that satisfies sequencing constraints.

The result of calling *test* on all free activities is a list of activities and their statuses of either PASSED, FAILED, or UNREACHABLE. When the status of an activity is reported as UNREACHABLE it means that a test fixture could not be created for the activity. This result is characteristic of black box testing because the test code does not have access to the internal components of the system. This is contrary to white-box unit testing in which test fixtures can be created that are independent of failures in other parts of the system.

The worst-case time complexity of the activity test algorithm is $\Theta(n^2)$, where n equals the number of activities. To see that the running time is in $O(n^2)$, note that the `test` method of an activity will be called once for each activity that depends on it, which corresponds to the edges coming into the activity in the test activity graph. Since the maximum number of edges in a directed graph is n2, and since the time complexity of an activity's *execute* method is $O(1)$, the algorithm's running time is in $O(n^2)$. To see that the running time is in $\Omega(n^2)$, consider the case where $n/2$ activities are independent, $n/2$ activities are free, and each free activity is dependent on all independent activities. In this case, there are $n/2{*}n/2$ dependencies, hence $n^2/4$ invocations of the `test` method.

IV. Automated Generation of Test Code

While the activity oriented approach makes it easy to analyze a Web application, generation of test code for body of $execute()$ also needs to be automated. To this end, we can utilize a particular pattern in Web application test code. For example, when a Web page has a form that users should fill in, we can test the page by: 1. searching for HTML input tags, 2. filling in with random data, 3. sending data to the server, and 4. checking the results.

We developed a software called *iTester* to automate the code writing using such a pattern on Windows 2003 server with JUnit 4.1 and HttpUnit 1.6.2. iTester gets URLs as its inputs, then analyzes and generates test code for the Web appication. Users can modify or add their own code to the generated test code. The Web page is parsed using *ElementIterator*, which is provided through the J2SE standard API. Sequentially examining the elements returned by the *ElementIterator*, if we find an HTML attribute following a pattern we pre-defined, we can obtain the value of the attribute, such as type, name, value, and maximun length. Using the obtained data, iTester generates XML test codes according the attribute type.

Due to the space limitation, other details of iTester and the test results are not presented here. Experiments on users with various skill levels show that iTester can help making test codes for all skill levels, and significantly reduce the time required to make test codes.

V. Conclusions

In this paper, we presented an approach to black-box testing of web applications that is based on domain objects, test activities, a test activity graph, and an activity test algorithm. In this approach, the tester develops a test model that includes domain objects and activities. The domain objects represent the objects in the application as seen by the user, and represent application state. Some domain objects represent users, such as admin, officer, applicant, etc., which interact with the application. Other domain objects represent non-user domain objects, such as application, award, etc. The test activities are self-verifying user interactions with the application. As such, they are both test cases and the transformations needed to create test fixtures for other test activities. Test activities also encapsulate dependencies, which the activity test algorithm uses to execute the test activities according to a topological sort that respects sequencing constraints. The activity test algorithm is theoretically efficient for realistic web applications.

The activities-oriented approach adds another layer of structure on top of testing frameworks, such as HttpUnit, that provide lower-level functionality. The architecture enables developers to organize code in a consistent manner, and to execute tests more quickly by removing redundancy when building test fixtures.

ACKNOWLEDGEMENT

This work was supported by the Brain Korea 21 project in 2008.

References

[1] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," in *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*, Toronto, Ontario, Canada, May 2001, pp. 25–34.

[2] S. C. Lee and J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutaion Analysis," in *Proceedings of the 12th International Symposium on Software Reliability Engineering*, 2001, pp. 200–209.

[3] D. C. Kung, C.-H. Liu, and P. Hsia, "An Object-Oriented Web Test Model for Testing Web Applications," in *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, Los Alamitos, CA, 2000, p. 111.

[4] G. A. D. Lucca, A. R. Fasolino, F. Faralli, and U. de Carlini, "Testing Web Applications," in *Proceedings of the 18th International Conference on Software Maintenance, ICSM 2002*, Montreal, Quebec, Canada, October 2002, pp. 310–319.

[5] J. Tian, L. Ma, Z. Li, and A. G. Koru, "A Hierarchical Strategy for Testing Web-Based Applications and Ensuring Their Reliability," in *Proceedings of the 27th International Computer Software and Applications Conference*, Los Alamitos, CA, 2003, p. 702.

[6] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A Scalable Approach to User-Session based Testing of Web Applications through Concept Analysis," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, Los Alamitos, CA, 2004, pp. 132–141.

[7] Y. Qi, D. C. Kung, and W. E. Wong, "An Agent-Based Testing Approach for Web Applications," in *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Edinburgh, Scotland, UK, July 2005, pp. 45–50.

[8] M. Karam, W. Keirouz, and R. Hage, "An Abstract Model for Testing MVC and Workflow Based Web Applications," in *Proceedings of AICT-ICIW'06*, Los Alamitos, CA, 2006, p. 206.

[9] J. J. Gutiérrez, M. J. Escalona, M. Mejías, and J. Torres, "An Approach to Generate Test Cases from Use Cases," in *Proceedings of 6th International Conference on Web Engineering*, Palo Alto, CA, 2006, pp. 113–114.

[10] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.