University of Southern California

CSCI-599: Testing and Analysis of Web Applications

Class Project Report

on

Guided Testing of Web Applications

Team Members: 2

Neeraj Kumar

Srikanth Gandupalli

Project Objective:

Develop a tool that improves code coverage obtained while testing an application feature, suggesting modifications in the test suite, and automating the whole process.

Problem Description:

Failure of a test suite in satisfying a variety of constraints/conditions may lead to lower code coverages, which may be detrimental in revealing all faults. There has to be a tool which can identify such constraints/conditions in the code, satisfy such constraints by appropriate selection of input test data and produce further test cases which will improve coverage of such conditions. This report discusses about how we exploit the information available to us from common application resources or other tools to achieve our objectives.

Keywords:

Following are a few sources of information/tools which can be exploited to reveal important information about uncovered conditions/branches in the code:

1. Tomcat Access Logs.
2. Cobertura Coverage reports.
3. Other tools, like WAM analysis reports.
4. Control-flow graphs of bytecode.
5. Source code analysis.

Methodology:

The tool developed consists of three major techniques of finding unused code elements within the scope of initial test suite:

1. Finding unused variables using WAM analysis.
2. Finding unused variables using CFGs of bytecode.
3. Finding unclicked links on a web page.

1. Finding unused variables using WAM analysis:

The tomcat access logs capture all the application interactions that happen as a result of running the initial test suite. However, there is an important information in these logs about servlets which were hit, and the html form variables and possibly their values which participated in these interactions. On the other side, a user can generate WAM analysis report containing a superset of static information on components, interfaces, parameters and their values existing in these servlets. To focus only on the servlets which are specific to testing an application feature, we fetch the superset of static attributes for only those servlets and subtract from them the attributes that have already participated in the interactions. This gives a most probable list of attributes within participating servlets that

can be called in future test cases. Hence, we generate corresponding urls with query strings, values to which are fed by a random value generator, randomly picking up values from the user defined boundary test data. These set of urls are then triggered using wget.

2. Finding unused variables using CFGs of bytecode:

The cobertura coverage reports can be parsed to find line-by-line the uncovered/partially covered elements within the source code. We store the servlet names and corresponding line numbers which have a zero hit or branching with less than 100% coverage. We, then, iterate over these caught servlets and generate bytecode CFGs for every method inside them. Every CFG is then traversed in a DFS fashion to locate any uncovered/partially-covered instruction and logs a match in another list. The nodes consequent to this logged instruction node are not traversed by the algorithm, indicating they are shadows to what happened wrong with the already logged parent node. This generates the list containing a subset of source code lines in servlets. The participating servlets are also parsed in entirety to look for any getParam or getParameter calls made on http request objects, giving a superset of user-submitted variables. Each of these variables is then looked up for its presence in the subset of source code lines captured earlier, and the intersection gives us the potential variables which were never triggered by the test suite. The variables are used in conjunction with their servlet names to trigger new urls using wget.

3. Finding unclicked links on a web page:

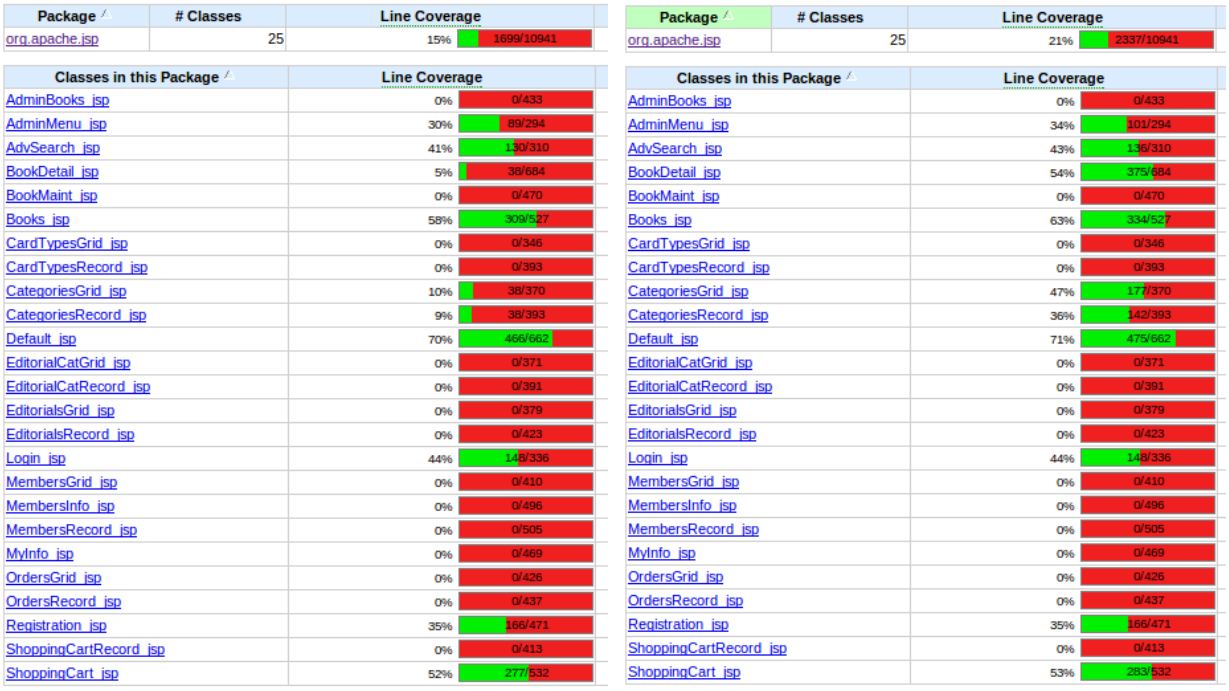
A manually written test suite may fail to capture all the hyperlinks found in a web page consisting of hundreds of hyperlinks. However, we capture the list of clicked links and servlets traversed from the tomcat access logs. The list of servlets are called again dynamically using http client connection to generate the html source code which is parsed to find out a superset of links existing on all these pages. Subtracting the list of clicked links from the list of all links gives the list of unclicked links, which are added to the test suite and called using wget.

Experiment & Evaluation:

The tool was used on a sample bookstore application that allows users to search and order books online from their respective accounts. Five different test suites, consisting of wget calls to urls, were selected and were fed to this tool. Below is the table and snapshots of the results obtained and percentage increase in coverage values for these set of 5 test suites randomly choosen:

TestSuite#	Number of servlets	Initial Coverage	New Coverage	% Increase in Coverage
TestSuite1	25/25	43%	48%	11.6 %
TestSuite2	10/25	15%	21%	40.0 %
TestSuite3	8/8	46 %	52 %	13 %
TestSuite4	17/25	29 %	32 %	10.3 %
TestSuite5	10/25	17 %	21 %	23.5 %

Snapshot: (Example)



Package	# Classes	Line Coverage
org.apache.jsp	25	15% 1699/10941

Classes in this Package	Line Coverage
AdminBooks_jsp	0% 0/433
AdminMenu_jsp	30% 89/294
AdvSearch_jsp	41% 130/310
BookDetail_jsp	5% 38/684
BookMaint_jsp	0% 0/470
Books_jsp	58% 309/527
CardTypesGrid_jsp	0% 0/346
CardTypesRecord_jsp	0% 0/393
CategoriesGrid_jsp	10% 38/370
CategoriesRecord_jsp	9% 38/393
Default_jsp	70% 466/662
EditorialCatGrid_jsp	0% 0/371
EditorialCatRecord_jsp	0% 0/391
EditorialsGrid_jsp	0% 0/379
EditorialsRecord_jsp	0% 0/423
Login_jsp	44% 148/336
MembersGrid_jsp	0% 0/410
MembersInfo_jsp	0% 0/496
MembersRecord_jsp	0% 0/505
MyInfo_jsp	0% 0/469
OrdersGrid_jsp	0% 0/426
OrdersRecord_jsp	0% 0/437
Registration_jsp	35% 166/471
ShoppingCartRecord_jsp	0% 0/413
ShoppingCart_jsp	52% 277/532

Package	# Classes	Line Coverage
org.apache.jsp	25	21% 2337/10941

Classes in this Package	Line Coverage
AdminBooks_jsp	0% 0/433
AdminMenu_jsp	34% 101/294
AdvSearch_jsp	43% 136/310
BookDetail_jsp	54% 375/684
BookMaint_jsp	0% 0/470
Books_jsp	63% 334/527
CardTypesGrid_jsp	0% 0/346
CardTypesRecord_jsp	0% 0/393
CategoriesGrid_jsp	47% 177/370
CategoriesRecord_jsp	36% 142/393
Default_jsp	71% 475/662
EditorialCatGrid_jsp	0% 0/371
EditorialCatRecord_jsp	0% 0/391
EditorialsGrid_jsp	0% 0/379
EditorialsRecord_jsp	0% 0/423
Login_jsp	44% 148/336
MembersGrid_jsp	0% 0/410
MembersInfo_jsp	0% 0/496
MembersRecord_jsp	0% 0/505
MyInfo_jsp	0% 0/469
OrdersGrid_jsp	0% 0/426
OrdersRecord_jsp	0% 0/437
Registration_jsp	35% 166/471
ShoppingCartRecord_jsp	0% 0/413
ShoppingCart_jsp	53% 283/532

How to configure?

1. Setup build.xml to call automate-coverage-project.sh before process starts.
2. Run wam to create an analysis report.
3. Make all the necessary application-specific configurations in automate-coverage-project.sh file:
a) location of logs,
b) wam analysis report,
c) test suites,
d) base url,
e) source code and compiled code,
f) coverage reports, etc.
4. Also change the output directory paths wherever required.
4. Execute the automate-project.sh script that calls build.xml to instrument code and run automate-coverage-project.sh

Conclusion:

Benefits:
1. Application-independent: No knowledge of application is required to integrate the tool.
2. Coverage increase of more than 10 % over all test suites, as experimented. Limitations: Output test suite is only in the form of urls.
Future Work: Generation of input test data from tomcat access logs, that can be fed to the urls constructed using parameters from WAM or CFGs.