An Evaluation of the Effectiveness of the Atomic Section Model

Sunitha Thummala and Jeff Offutt

Software Engineering
George Mason University, Fairfax VA, USA
{sthumma3, offutt}@gmu.edu

Abstract. Society increasingly depends on web applications for business, work, and pleasure. As the use of web applications continues to increase, the number of failures, some minor and some major, continues to grow. A significant problem is that we still have relatively weak abilities to test web applications, and often rely on informal, ad-hoc, and ineffective techniques. A key driver to this problem is that web applications use novel technologies, including control structures that are not available in traditional software, and new state handling techniques, including new variable scopes. Traditional testing techniques do not adequately model or test these novel technologies. The atomic section model (ASM), which was introduced in a previous paper, models these novel technologies to support design, analysis, and testing. This paper presents an empirical study to evaluate the effectiveness of the ASM to design tests. The model was implemented into a tool, WASP, which extracts the ASM from the implementation and supports various test criteria. We studied ten web applications, totaling 156 components and 11,829 lines of code. Using WASP, we generated 207 tests, which revealed 31 faults. Seventeen of those faults exposed internal information about the application and server.

Keywords: Web applications; test criteria; model based testing; atomic section modeling

1 Introduction

A web application is a program that is deployed on a server and accessible through the Internet. Web applications form an important part of our daily lives as we use them for business, e-commerce, and even paying bills. Most businesses use web applications to interact with customers and business partners. A major benefit of web applications is that they can be accessed anytime from anywhere. A previous paper explained why large web applications need to be reliable, secure, maintainable, usable, and available [1]. A 2013 comScore study reported that consumers spent \$42.3 billion dollars online in 2012 during the holidays [2].

Unfortunately, web applications continue to have many failures. The London Olympics website in 2011 crashed due to the sudden increase in customers

after it announced 2.3 million tickets were available for purchase [3]. According to a Forbes report in 2013, Amazon.com lost \$66,240 per minute of downtime [4]. The US health care web application in 2013 had numerous failures that led to significant costs, both monetary and political [5]. Target also suffered a massive credit card breach in 2013. All of these examples highlight the fact that we still do not always build reliable and secure web applications.

Web applications use many different technologies, including interpretive languages (Perl), scripted page modules (JSPs and ASPs), compiled module languages (servlets and ASPs), programming language extensions (JavaBeans and EJBs), general-purpose programming languages (Java and C#), sequential databases (SQL), and data representation languages (XML). As will be described in section 2, web application software uses new forms of control couplings, including forward, redirect, and user-controlled buttons on browsers. They also use new state handling and variable scoping mechanisms, including objects stored in a session and in a context.

As described in detail in a previous paper [6], traditional models and analysis techniques are not sufficient to capture all of the nuances of web applications. This causes model-based testing techniques that rely on traditional models such as statecharts and collaboration diagrams to be inadequate to test web apps. The same paper introduced the atomic section model (ASM), which has explicit mechanisms to model the novel control connections and data handling mechanisms in web applications.

ASM addresses the following issues in web applications:

1. Distributed integration
2. Dynamic creation of HTML forms
3. Ability of users to directly control the potential flow of execution

This paper presents an empirical evaluation of using the ASM to test existing web applications. Tests were designed from the requirements as a traditional testing method, and compared with tests designed from the ASM.

The rest of the paper is organized as follows. Section 5 presents prior research related to this paper. Section 2 summarizes the atomic section model, and section 3 describes a tool that we developed to partially automate test generation using the ASM. Section 4 then presents an empirical evaluation of using the ASM to generate tests on ten web applications, and finally section 6 concludes the paper.

2 Modeling Web Applications

This study is based on prior work by Offutt and Wu [6]. Web applications are deployed on a server and users run them across the internet through a browser. Thus, web application UIs are in HTML, using static links and form inputs. For conciseness, we often shorten “web application” to simply *web app*.

Web apps are composed from independent, and often distributed, components built with diverse technologies such as Java servlets, JSPs, ASPs, and

many others. Thus, key to a web app are the *interactions* among the software components. Users run web apps through browsers, which generate HTTP *requests* to web servers, which then activate the requested software components, which in turn interact with other components, and eventually return *responses* to the users.

Deploying software across the web brings two essential new design challenges. One is remembering who the user is across multiple requests to the same server, and another is keeping data persistent across those multiple requests. HTTP is a *stateless* protocol, so each request is independent and run as a separate execution thread. To solve the first challenge, web apps define a *session* to be a sequence of related HTTP requests from the same user. Session states are maintained in special state variables on the server that are indexed by *cookies* that are stored on the users' computers and submitted to the server with requests.

For the second challenge, web app technologies use new forms of data scoping. Web software components cannot share global variables or objects. But since components run as threads within a process, the server creates a special object called the "session object" to store persistent data. Session data are commonly scoped to be available to (1) the same software component on different requests, (2) different software components on the same request, (3) all requests from the same session, or (4) all applications running within the same web server¹.

These powerful new technologies are, in a sense, new language features. Although they are built on top of traditional languages (such as Java and C#), programmers use them as if they are basic parts of the language. Not surprisingly, traditional modeling languages do not fully model all aspects of these technologies. The remainder of this section summarizes the atomic section model from Offutt and Wu [6], and discusses how the model is used to design tests.

2.1 The Atomic Section Model

An *atomic section* (ATS) is a section of HTML that has the property that if part of a section is sent to a client, then the entire section is (the "all-or-nothing property"). Atomic sections are combined to form regular expressions called *component expressions* (CE). Let c be a component consisting of n atomic sections $p_1, p_2, \dots, p_n$. CEs combine ATSs in one of four ways:

1. *Sequence* ($CE \rightarrow p_i \bullet p_j$): The component expression is p_i followed by p_j .
2. *Selection* ($CE \rightarrow p_i \mid p_j$): The component expression is either p_i or p_j , but not both. A typical example could be a code block that is sent from an *if-else* construct. Only the HTML section in the block that gets executed is sent to the client.
3. *Iteration* ($CE \rightarrow p_i^*$): The component expression contains zero or more occurrences of p_i in sequence, usually because the HTML is in a loop.

¹ This terminology is from the J2EE framework. Other frameworks use slightly different terminology and some have slightly different scopes.

4. *Aggregation* ($CE \rightarrow p_i\{p_j\}$): p_j is included as part of p_i when p_i is sent to the client. These come from function calls in p_i and from file *include* commands.

A complete component expression models all possible web pages that the component can generate, much as a control flow graph models all possible execution paths in a method.

Transitions. Web apps are inherently component-based, and connect the components in novel ways. The ASM identifies five kinds of transitions. In the following definitions, p and q are component expressions and c is a component that generates HTML. The arrows are used in pictorial representations to distinguish the different types of transitions.

1. *Simple Link Transition* ($p \longrightarrow c$): The anchor tag $\langle a \rangle$ causes a link transition from the client to a component c on the server. Note that p may have more than one simple link transition, and so more than one possible destination. Anchor tags are the only source for simple link transitions.
2. *Form Link Transition* ($p \twoheadrightarrow c$): Submitting forms in p causes a transition to component c on the server.
3. *Component Expression Transition* ($c \multimap p$): A component c executes and causes a component expression p to be returned to the client. The same component can produce more than one component expression.
4. *Operational Transition* ($p \rightsquigarrow q$): The user can create new transitions out of the software's control by pressing the refresh button, the back button, the forward button, accessing the history menu, or directly modifying the URL in the browser (URL rewriting). This amounts to random control flow jumps in the software, which is not possible in traditional and desktop software.
5. *Redirect Transition* ($p \rightarrowtail q$): A redirect transition is a server-side software transition that is **not** under control of the tester. Redirection sometimes goes through the browser, and sometimes remains on the server, but is normally not visible to the user. Form data from the request is usually sent to the new component, but there is no return (hence this differs from a traditional method call).

Component Interaction Model. A component expression describes all web pages that can be produced by a web software component. Component expressions are composed with transitions to model an entire web app. A *Component Interaction Model* (CIM) defines the start page of a web application, a set of atomic sections, component expressions, and a set of transitions to model the possible flows of control through a web app. When taken together with the set of variables that define the state of a web app, the CIM is called a web application transition graph (ATG).

ASM currently does not model data flow. An interesting future study would be to investigate whether adding data flow criteria to the ASM could help detect more faults.

2.2 Test Criteria

Abstract model-based tests are designed from the ATG using three coverage criteria. Using coverage criteria to design tests has several advantages, including fewer tests that reveal more faults, traceability, and direct guidance for when to stop testing.

1. **Prime path coverage:** A *simple path* is a path in a graph such that no nodes except the first and the last appear more than once. A *prime path* is a simple path that does not appear as a proper sub-path of any other simple path [7]. Prime path coverage requires all prime paths in the graph to be executed.
2. **Invalid Access Coverage:** Each non-start node in an ATG is accessed directly, as if a tester jumps into the middle of a program. Users might do this accidentally, may save a URL to a page purposefully to avoid going through the lengthy application logic, or as a malicious attack. Note that this is not possible in traditional software.
3. **Invalid Path Coverage:** Invalid paths are created by first traversing a prime path, then adding an operational transition (IP1). Each invalid path is then extended, when possible, by adding transitions to all nodes accessible from the end of the invalid path (IP2).

The original ASM paper by Offutt and Wu [6] used one experiment subject and very little automation. This paper presents a significantly stronger empirical evaluation using ten experiment subjects and a complete tool that constructs ASMs automatically from the source code.

Fig. 1. Web Atomic Section Project Tool

3 The Web Atomic Section Project (WASP)

We developed the Web Atomic Section Project (WASP) to automate creation of the atomic section model. WASP was built as a web app. Users start by uploading source files. WASP then calculates the atomic sections and builds the web application graph. A screenshot of WASP is shown in figure 1.

Figure 1 shows the atomic section calculation for a simple JSP, *for1.jsp*. The expression shows sequence by concatenating the components rather than using the $\bullet$ symbol. p_2 is the HTML produced within the *for* loop body ("*br*)The counter value is: (% count %)").

Figure 2 shows a sample web application graph constructed by WASP. *Login* is the start component, and has a form link transition to *gradeServlet* and a simple link transition to *syllabus*. The servlet *gradeServlet* has a form link transition to itself and a simple link transition to *sendMail*.

Fig. 2. Application Graph Construction

The major components of WASP are the atomic section calculator and the application transition graph constructor. The atomic section calculator was built using ANTLR (ANother Tool for Language Recognition) [8]. ANTLR is used to construct compilers, recognizers, etc. It uses an input grammar for a language and develops the output source code based on semantic actions written by the user. Since servlets are special-purpose Java classes, they can be translated by ANTLR with the Java grammar. JSPs consist of HTML mixed with Java code

and so are required to be pre-processed. A JSP is pre-processed by converting into its corresponding servlet by using the Jasper library of Tomcat.

The second major component, the application transition graph constructor parses Java servlets, JSPs and HTML files (for simple links, form links, forward transitions, redirect transitions) and uses the open source tool Graphviz [9] to construct the web application transition graph. WASP has some limitations. First, it is a static analysis tool, so dynamic references in the source code cannot be processed. Second, JSPs containing tag libraries are not processed. WASP extracts atomic sections and creates the application transition graph, but does not directly generate test values.

4 Empirical Evaluation

For this study, we used WASP to generate atomic sections and web application graphs for ten web applications. Our goal was to evaluate the usefulness of the ASM for testing. Specific research questions were:

1. RQ1: Can the ASM be computed with sufficient speed and usability to be effectively used by engineers?
2. RQ2: Can the ASM help find faults in web applications?
3. RQ3: What kinds of faults does ASM help identify?

4.1 Experimental Design

For experimental subjects, we used ten web applications that were built with the J2EE framework (servlets and JSPs) by graduate students in a graduate component-based software engineering course.

Two web apps (subjects one and two) were developed by the first author; the first web app is used to help students find internships, and the second allows customers to buy books online. Four web apps (subjects three, four, five and six) let users fill out surveys after visiting a campus and also see survey results from other visitors. Two web apps (subjects seven and eight) let users create events, create other users, assign them to a group, and make events available to everybody in a group. Two web apps (subjects nine and ten) store information about employees and provide access to employers.

All the web apps had been tested lightly by the developers. The programmers had considerable freedom in their designs, thus the programs written to the same requirements were quite different. This was intentional—we wanted to investigate the ability to find naturally occurring faults, but not trivial faults.

We generated test sets according to four different criteria. First, as a control, we generated requirements-based tests to represent the kinds of tests that are typically used in industry. We gave the requirements documents to a professional software tester with eight years of experience system testing, who generated tests for each subject. The professional tester was free to make any assumptions based on the requirements document. The tests were based purely on the

requirements document and the tester allocated eight hours to design the tests. The requirements document did not mention things such as URL manipulation and browser controls, so the tester did not design tests for such actions. Then the first author ran the tests and recorded failures by hand.

Next we generated tests for the three test criteria in section 2. We used WASP to generate the CEs and ATGs, then the graph coverage web application [7] associated with Ammann and Offutt’s book to generate the prime paths and test paths [10]. The graph coverage web app generated 73 prime path test requirements. The remaining 205 tests (41 invalid access and 164 invalid path) were generated by hand based on the ATG generated by WASP. We then created values by hand to satisfy all the 278 test paths. No tool is available to automatically generate test inputs to tour test paths. The manual work involved in setting up the system environment and database, deploying the executable web archive files, generating test case inputs, running tests, evaluating test outputs, and calculating coverage took around 60 hours over a period of ten days.

4.2 Experimental Results

The subject programs and experimental results are summarized in table 1. The second, third, and fourth columns summarize statistics about the web apps—the number of software components, the number of lines of code², and the number of atomic sections created by WASP. The software components were Java servlets, JSPs, and plain Java classes. The final eight columns give data from the tests. For each of our four test sets, the table gives the number of tests and the number of tests that failed. Recall that the invalid path tests are extended versions of the prime path tests. The bottom row gives the sum of each column.

Table 1. Experimental results of applying four test sets to ten web applications.

Sub- ject	Comp- onents	LOC	Atomic Sections	Tests							
				Requirements		Prime Path		Invalid Access		Invalid Path	
				Tests	Failures	Tests	Failures	Tests	Failures	Tests	Failures
1	15	825	22	20	1	9	0	4	1	24	3
2	14	927	12	20	0	4	0	6	2	12	2
3	14	1485	8	20	1	9	1	5	2	20	1
4	18	1327	15	20	0	9	0	5	2	25	2
5	14	986	9	19	1	10	1	2	1	7	1
6	20	1473	19	12	0	3	0	2	0	7	1
7	9	782	7	13	1	2	0	2	0	7	1
8	9	688	8	9	0	2	0	2	0	11	1
9	19	1758	30	20	1	16	0	7	2	30	1
10	24	1578	45	16	0	9	0	6	3	21	3
Total	156	11,829	175	169	5	73	2	41	13	164	16

² Counted by loc-calculator (<https://code.google.com/p/loc-calculator/>)

The requirements-based tests found a total of five failures with 169 tests. Thus, around 3% of the tests detected failures. The failures produced by the requirements-based tests are documented in table 2.

Table 2. Summary of failures found by requirements-based tests.

Failure Type	Failure Description	Number found of this type
1	Did not display expected messages for invalid input	3
2	Allowed user to check out when total was \$0	1
3	List of events not displayed, threw exception	1
	Total	5

We generated a total of 278 tests from the ASM (prime paths, invalid access, and invalid path). The prime path tests found two failures, the invalid access tests found 13, and the invalid path tests found 16.

Four of the five failures the requirements-based tests found were also found by the invalid access and invalid path tests. The failure not found by ASM in table 2 is of failure type 3. This exception occurs only when a file that needs to be updated is locked by another process (for example, when the file is open in an editor).

Thus, around 16% of the ASM tests found faults, a number that most testers would consider to indicate very efficient testing. These 31 failures are summarized in table 3.

Table 3. Types of failures found by ASM tests.

Failure Type	Failure Description	Number of failures found by ASM tests
1	Duplicate records added	2
2	Unhandled software exception	7
3	Irrelevant error message	4
4	Runtime exception	11
5	Empty student record displayed due to URL rewriting	1
6	Displayed incorrect information	3
7	Displayed blank page	1
8	Allowed to check out when total was \$0	1
9	Doubled the number of items in cart when page refreshed	1
	Total	31

Finally, we analyzed each failure to determine the underlying fault. Each failure was found to represent a distinct fault. The faults are summarized in table 4.

Table 4. Types of faults found by ASM tests.

Fault Type	Fault Description	Number of faults found by ASM tests
1	Allowing double submit form action	2
2	Not checking for null object	11
3	Resource URL not mapped correctly	3
4	Did not check for invalid input values such as empty or incorrect syntax	8
5	Did not have code to check for error scenario	2
6	Did not check if value being inserted is already present in the DB (duplicate item)	2
7	Did not check if value is greater than zero for enabling submit button	1
8	doGet() method not implemented	1
9	Search function for when only one parameter was given was incorrect	1
Total		31

4.3 Atomic Section Coverage

Atomic section coverage requires each atomic section to be covered at least once [6]. After the above study was complete, we measured the ATS coverage of the tests.

The ASM tests (prime paths, invalid access, and invalid path) covered 170 of the 175 atomic sections (ATS). We hand-designed tests to cover two of the remaining ATSs, but the other three were in unreachable code. The additional two tests reached atomic sections that handled unexpected behavior and abnormal database conditions, and each resulted in an additional failure. The first of the three unreachable ATSs was in a JSP component. One JSP checks for a null value, and if the value is not null, passes control to another JSP. The second JSP also checks for a null value. Redundant checks such as this are usually considered to be sound engineering practice. The other two were in code that printed messages if a variable had a particular value. However, that variable was not given a value in the back-end code. These probably represent faults in the program—either the message is not needed, or the back-end code should set a value for the variable.

This is additional evidence of the strength of the atomic section model. Simply covering atomic sections helped reveal faults that were not revealed by previous tests.

4.4 Discussion

It is telling to note that the requirements-based tests found only five faults, and four of those were also found by the ASM tests. Four sets of tests were derived from the ASM. The prime path criterion is the most traditional, and is defined

on arbitrary graphs, yet it only found two faults. Most of the faults were found by tests that were only generated as a result of the atomic section model, and would be unlikely to be generated if the ASM were not used.

Of the 31 failures found by the ASM tests, 17 revealed information about the components or server where the web app was deployed. These failures are not only distressing to users, but also potential security vulnerabilities. IP coverage of ASM requires an operational transition (back button, forward button, refresh button or URL rewriting) to be added to each prime path. We classified the failures found by ASM into failures that could potentially be discovered by a user using normal interaction sequences (such using the back button, the forward button, the refresh button, or by entering invalid data in forms) versus failures produced using IA or IP-URL rewriting. Of the 31 failures, 13 could have been found by users using normal interaction sequences, and 18 could only be found using IA or IP-URL rewriting.

Our study started with three research questions. We were able to successfully use WASP to generate CEs and ATGs, and then successfully design and create tests. The CEs and ATGs were calculated by WASP for each experiment subject in less than a minute, and the users were easily able to turn those into tests. Thus, the answer to RQ1 is clearly yes. Among the failures found using this study, the majority of the failures were found with the ASM tests, but not the requirements-based test. Thus, we conclude the answer to RQ2 is also yes. We explore RQ3, the kinds of faults identified, in the following subsection.

4.5 Example Failures

We already mentioned several failures that could be security vulnerabilities. Another common problem we found was web apps that assumed all requests would be HTTP POST requests. The invalid access tests called components directly with GET requests, and some components that did not expect to be called directly had runtime exceptions.

Several invalid path tests caused session variables to be accessed incorrectly, thus causing inconsistencies in the program state. This is a common problem in commercial web apps that we see regularly, so the ability to identify such flaws during pre-release testing can be quite helpful.

We also found several instances of the “double submit form problem,” where if a user clicks on a back or refresh button after a form is submitted, the form is submitted again. This sometimes can lead to two orders being placed. The many commercial web apps that warn users to “not click the back button” indicate that this kind of fault is common, often known, but seldom corrected. These faults are usually found after the software is completed, and not corrected because changing the design would be too difficult. They might be easier to correct if found earlier during testing.

One fault in the bookstore application was revealed by an invalid access test, causing a blank page to be returned to the user. When given an invalid ID through URL rewriting, the bookstore application also crashed with a generic “application error” message, as opposed to correctly telling the user that the ID

was invalid. This was a case where the programmer assumed the component would only be accessed through normal channels, and the ID would always be valid.

One of the additional tests added to cover the remaining ATS discovered an interesting fault. The code was:

```
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher ("/banner");
if (dispatcher != null)
    dispatcher.include (request, response);
```

The atomic section model places HTML generated by the statement inside the *if* block into one atomic section. The *if*-block does not have an *else* clause, but a blank atomic section is created in the component expression as a selection. Thus, a test was created to take the (null) *else* clause by making the banner servlet unavailable. This test resulted in an HTTP 505 internal server error, revealing a flaw in the program.

An invalid access test for the internship app revealed a null pointer exception because a session variable was accessed without being set. Another resulted in the message "You are applying for internship id: *null*." Again, the id variable had not been given a value prior to being used.

4.6 Statistical Analysis of Experiment Results

Table 5. Number of faults found by ASM tests in each test subject

Experiment Subject	1	2	3	4	5	6	7	8	9	10	Total
No. of faults	4	4	4	4	3	1	1	1	3	6	31

Hypothesis Testing: The null hypothesis for this experiment states that the ASM is not effective in revealing faults (number of faults that can be found is zero). The alternate hypothesis states that ASM is effective in finding faults (number of faults that can be found is greater than zero).

$H_0: \mu \leq 0;$
 $H_A: \mu > 0$ (research hypothesis);
 Sample size $N = 10$
 Sample mean $X = 3.1$
 Standard deviation $S = 1.66$
 $DF = 9$

Referring to the experiment results found in table 5, the test will be conducted at 0.05 level of significance ($\alpha = 0.05$). Assuming a normal distribution

of faults found using ASM over all programs, we use a T distribution with nine degrees of freedom. Based on our null and alternate hypothesis, this will be a one tailed test. The critical value for α of 0.05 and 9 degrees of freedom is (t_α) 1.83. The test statistic

$$t = \frac{\bar{X} - \mu}{S / \sqrt{N}}$$

The test statistic is 5.96, which is greater than 1.83 (t_α), and so we reject the null hypothesis. Thus, our alternate hypothesis that ASM is effective in finding faults holds.

Confidence Interval Analysis: Our experiment used ten experiment subjects. The number of faults found in each test subject is shown in table 5. Given the small random sample size of 10 experiment subjects, assuming a normal distribution of the average number of faults found using ASM over all the programs, we use a T distribution with nine Degrees of Freedom (DF).

$$\begin{aligned} CI &= \bar{X} \pm T_{0.05} \times S / \sqrt{N} \\ CI &= 3.1 \pm 1.83 \times 1.66 / \sqrt{10} \\ CI &= 3.1 \pm 0.96 \\ CI &= [2.14, 4.06] \end{aligned}$$

Assuming a 90% confidence interval, the estimate of the average mean is in the range between 2.14 and 4.06. We are 90% confident that the average number of faults found by using ASM can be between 2 and 4 (by rounding to the nearest whole number).

4.7 Threats to Validity

Although the results strongly indicate that the ASM is an effective way to design tests, the study has some limitations. As usual with most software engineering studies, there is no way to show that the selected subjects are representative. This is true both for the web apps and the human programmers. In particular, the programmers were graduate students who may make different mistakes than experienced programmers. Because of the amount of work involved, we could only get one professional tester to participate in the requirements based tests. If more testers had participated, the results could have been different. A threat to internal validity is that manual analysis was used for various activities, including running tests, calculating coverage, and calculating the atomic sections. Finally, we cannot be sure that the tools used worked perfectly.

5 Related Work

Research into testing web applications can be split into static techniques, which analyze HTML and other static aspects of websites, and dynamic techniques, which test web application software. Dynamic techniques can be further divided into external techniques, which use only the URLs to access the program,

and internal techniques, which can access the source code or the server. Our research analyzes the source code to design tests, thus is dynamic and internal.

Most static research has focused on client-side validation and static server-side validation of links. An extensive listing of existing web test support tools is on a web site maintained by Hower [11]. The list includes link checking tools, HTML validators, capture/playback tools, security test tools, and load and performance stress tools. These are all static validation and measurement tools, none of which support functional testing of programs deployed on the web, as our research does.

Kung et al. [12, 13] developed a model to represent web sites as a graph, and provided preliminary definitions for developing tests based on the graph in terms of web page traversals. Their model includes static link transitions and focuses on the client side with only limited use of the server software. They did not model application transitions, as the ASM does.

Halfond and Orso [14] studied the problem of finding web app user interface screens. These screens are created dynamically, thus the problem of finding all screens is undecidable. They used static analysis to find more of the screens than previous techniques. Instead of focusing on individual screens, the ASM technique is internal and identifies atomic sections that represent all possible screens.

The closest research to this paper was by Di Lucca and Di Penta [15]. They proposed testing sequences through a web application that incorporate some of the operational transitions in this paper, specifically focusing on the back and forward button transitions. Di Lucca and Di Penta’s model focused on some browser capabilities, but not server-side software or web pages generated by the software.

Andrews et al. [16, 17] modeled web applications as hierarchical finite state machines, and generated tests as paths through the FSMs. This approach was pure system level and focused on behavioral aspects of web applications, as opposed to the structural aspects the atomic section model represents. Benedikt, Freire, and Godefroid [18] presented VeriWeb, a dynamic navigation testing tool for web applications. VeriWeb explores sequences of links in web applications by exploring “action sequences” [18]. Values are provided by the tester. VeriWeb only follows explicit HTML links, as opposed to the many transitions the atomic section model uses.

Finding input values is a difficult problem that we do not address. Elbaum, Karre, and Rothermel [19] proposed the use of “user session data,” which are values supplied by previous users in log files. Alshahwan and Harman [20] introduced a technique that supports web application regression testing by repairing user session data when the software changes. Lee and Offutt [21] used a form of mutation analysis to generate test values from previously existing XML messages. Jia and Liu [22] also generated test values using XML.

In related research, we have developed *bypass testing* [23, 24] to send invalid data to web applications, bypassing some of the input validation. This is a stress

testing approach that does not explore the kinds of interaction paths that ASM explores.

6 Conclusions and Future Work

This paper makes two contributions. First, the atomic section model has been implemented in a tool that analyzes web app source files, extracts atomic sections and computes component expressions, then identifies transitions and constructs web application transition graphs. Second, the ATSS and ATGs were used to design three different types of tests, and compared with requirements-based tests, which are typical of those used for system testing, on the basis of how many naturally occurring faults they detected. Because the ASM test generation process is incomplete and imperfect, we did not compare cost.

We generated a total of 278 ASM tests (73 PP, 41 IA, and 164 IP), which revealed 31 failures. The two tests that were added to cover additional atomic sections revealed two more failures, and the analysis to cover the remaining atomic sections uncovered two more. This contrasts with the 169 requirements-based tests, which only revealed five failures. Thus, we conclude that the ASM tests are much more effective at detecting failures than the requirements-based tests. This should not be surprising; the atomic section model captures elements of the design and implementation that are not captured in the requirements, and some of those elements are commonly misused and misunderstood by software engineers.

In the future, we hope to extend WASP to handle more technologies such as GWT, AJAX, JSF, etc. The atomic sections are less visible in some of these technologies, so computing the component expressions could be quite different. We also hope to include JavaScript in our analysis, including the concurrency aspects. Finally, we would like to explore other uses of the ASM. It captures fundamental aspects of the integration and execution of web applications, and could be used to support design modeling, program slicing, and change impact analysis.

Bibliography

- [1] Offutt, J.: Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software* **19**(2) (March/April 2002) 25–32
- [2] comScore: 2012 u.s. online holiday spending grows 14 percent vs. year ago to 42.3 billion dollars. Online (January 2013) URL: <http://www.comscore.com/>, last access: June 2014.
- [3] LogiGear: The seven most stunning website failures of 2011. Online (2012) URL: <http://www.logigear.com/magazine/issue/past-articles/the-seven-most-stunning-website-failures-of-2011/>, last access: June 2014.
- [4] Clay, K.: Amazon.com goes down, loses \$66,240 per minute. Online (2013) URL: <http://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/>, last access: March 2014.
- [5] Ford, P.: The obamacare website didn't have to fail. How to do better next time. *Bloomberg Businessweek Technology* (October 2013) <http://www.graphviz.org/>, last access: June 2014.
- [6] Offutt, J., Wu, Y.: Modeling presentation layers of web applications for testing. *Software and Systems Modeling, Springer* **9**(2) (April 2010) 257–280
- [7] Ammann, P., Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK (2008) ISBN 978-0-521-88038-1.
- [8] Parr, T.: Antlr parser generator. Online (2003) <http://wwwantlr.org/>, last access: June 2014.
- [9] Software, O.S.: Graphviz. Online <http://www.graphviz.org/>, last access: March 2014.
- [10] Ammann, P., Offutt, J., Xu, W., Li, N.: Graph coverage web applications. Online (2008) <http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage>, last access: March 2014.
- [11] Hower, R.: Web site test tools and site management tools (2002) www.softwareqatest.com/qatweb1.html.
- [12] Kung, D., Liu, C.H., Hsia, P.: An object-oriented Web test model for testing Web applications. In: *24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, Taipei, Taiwan, IEEE Computer Society Press (October 2000) 537–542
- [13] Liu, C.H., Kung, D., Hsia, P., Hsu, C.T.: Structural testing of Web applications. In: *Proceedings of the 11th International Symposium on Software Reliability Engineering*, San Jose CA, IEEE Computer Society Press (October 2000) 84–96
- [14] Halfond, W.G.J., Orso, A.: Improving test case generation for web applications using automated interface discovery. In: *Proceedings of the Foundations of Software Engineering*, Dubrovnik, Croatia (2007) 145–154

- [15] Lucca, G.A.D., Penta, M.D.: Considering browser interaction in web application testing. In: 5th International Workshop on Web Site Evolution (WSE 2003), Amsterdam, The Netherlands, IEEE Computer Society Press (September 2003) 74–84
- [16] Andrews, A., Offutt, J., Alexander, R.: Testing Web applications by modeling with FSMs. *Software and Systems Modeling, Springer* **4**(3) (August 2005) 326–345
- [17] Andrews, A., Offutt, J., Dyreson, C., Mallery, C.J., Jerath, K., Alexander, R.: Scalability issues with using FSMWeb to test web applications. *Information and Software Technology, Elsevier* **52**(1) (January 2010) 52–66
- [18] Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically testing dynamic Web sites. In: Proceedings of 11th International World Wide Web Conference (WWW'2002) – Alternate Paper Tracks (WE-3 Web Testing and Maintenance), Honolulu, HI (May 2002) 654–668
- [19] Elbaum, S., Rothermel, G., Karre, S., Fisher, M.: Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering* **31**(3) (March 2005) 187–202
- [20] Alshahwan, N., Harman, M.: Automated repair of session data to improve web application regression testing. In: 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008 Industry Track), Lillehammer, Norway (April 2008) 298–307
- [21] Lee, S.C., Offutt, J.: Generating test cases for XML-based Web component interactions using mutation analysis. In: Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong China, IEEE Computer Society Press (November 2001) 200–209
- [22] Jia, X., Liu, H.: Rigorous and automatic testing of Web applications. In: 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), Cambridge, MA (November 2002) 280–285
- [23] Offutt, J., Wu, Y., Du, X., Huang, H.: Bypass testing of web applications. In: 15th International Symposium on Software Reliability Engineering, Saint-Malo, Bretagne, France, IEEE Computer Society Press (November 2004) 187–197
- [24] Offutt, J., Papadimitriou, V., Praphamontriping, U.: A case study on bypass testing of web applications. *Springer's Empirical Software Engineering* (July 2012) 1–36 Published online, DOI: 10.1007/s10664-012-9216-x.