Modeling and Testing of Dynamic Aspects of Web Applications

Ye Wu, Jeff Offutt and Xiaochen
(wuye,ofut,xdu)@ise.gmu.edu

Information and Software Engineering Department
4400 University Dr.
George Mason University
Fairfax, VA, 22030 USA

Abstract–Web software applications have become complex, sophisticated programs that are based on novel computing technologies. Although powerful, these technologies bring new challenges to developers and testers. Checking static HTML links is no longer sufficient; Web applications must be evaluated as complex software products. This paper focuses on two unique aspects of Web applications, an extremely loose form of coupling that features dynamic integration and the ability that users have to directly change the potential flow of execution. Taken together, these allow the **potential** control flow to vary with each execution, and means the possible control flows cannot be determined statically. Thus we cannot perform standard analysis techniques that are fundamental to many software engineering activities. This paper presents a new theoretical model of new couplings and the dynamic flow of control of Web applications. This model is based on atomic sections, which allow tools to build the equivalent of a control flow graph for Web applications. The atomic section model is used to propose new test criteria for Web applications, and results are shown from a case study on a moderate-size application.

Index Terms–Testing strategies, software engineering for Internet projects, analysis, object-oriented programming

[†]This work was sponsored in part by the National Science Foundation under grant number CCR-0097056 and CCR-0097056 (supplemental).

1 Introduction

The World Wide Web gives software developers a new way to deploy software. The first programs deployed on the Web were simple applications that accepted form data from HTML pages and processed or stored the data. Modern Web applications are sophisticated, interactive programs with complex GUIs and large amounts of back-end software that is integrated in novel and interesting ways. Analyzing, evaluating, maintaining and testing these applications present many new challenges for software developers and researchers.

In a previous paper, Offutt explained why large Web applications need to be highly reliable, very secure, continually maintainable, and constantly available [23]. Problems with Web applications can affect hundreds of thousands of people and can cost millions of dollars. For example, a glitch during an unscheduled maintenance at Amazon.com in 1998 put the site offline for several hours, with an estimated cost as high as $400,000 US. More recently, in early 2003 a public University's Web site was hacked into and thousands of social security numbers were released to the public. In another instance at another University, hundreds of acceptance letters were emailed to applicants who had previously (and correctly) been denied. Even more than the monetary costs, the relationship between customers and organizations can be seriously damaged by such problems; the users do not care why the problem happened, but they will find another site to do business with.

Industry has been responding to these needs by developing new technologies and programming models that impact the way software is designed, built, tested and maintained. While solving some important problems, these technologies also introduce other problems that need the attention of software researchers. This research project is investigating these problems.

This paper defines a *Web page* to be information that can be viewed in a single browser window. A Web page may be stored as a static HTML file, or it may be dynamically

generated by software such as a JSP or Java Servlet. A *Web site* is a collection of Web pages and associated software elements that are related semantically by content and syntactically through links and other control mechanisms. A *static Web page* is unvarying and the same to all users, and is usually stored as an HTML file on the server. A *dynamic Web page* is created by a program on demand, and its contents and structure may be determined by previous inputs from the user, the state on the Web server, and other inputs such as the location of the user, the user's browser or operating system, and time of day. A *Web application* is a software program that is deployed across the Web. Users access Web applications using HTTP requests and the user interface typically executes within a browser on the user's computer.

Web software is built with many different technologies, including scripting languages that run within HTML on the client (JavaScript, VBScript), interpretive languages that run on the server (Perl), compiled module languages on the server (Servlets, ASPs), scripted page modules on the server (JSPs, ASPs), general purpose programming languages (Java, C#), programming language extensions (JavaBeans, EJBs), data manipulation languages (XML) and databases. These diverse technologies (and others) cooperate to implement Web applications, resulting in a heterogenous, multi-platform software environment where the software components are loosely and dynamically coupled.

Whereas the high quality requirements of Web software, multi-platform issues, concurrency, and issues arising from heterogenous languages are problems that have been addressed in other types of software, some of the issues involving coupling and integration are new to Web software. This paper presents a novel analysis and modeling technique that addresses the problem of dynamic integration of Web software applications. The issue is discussed and the problem is presented, then a solution using atomic and composite sections is developed. This model offers a fundamental way to model dynamic aspects of Web software in a technologically independent way. The model can support a variety of different software

engineering activities by allowing analysis techniques such as control flow, data flow and slicing to be applied. The paper also presents specific integration testing criteria and results from an empirical study.

1.1 Problems in Testing Web Applications

The literature on testing Web applications is still scarce and there is no widespread agreement on how to categorize the technical problems. An important factor that influences Web applications is how the different pieces are connected. We make an initial attempt to categorize Web application testing in terms of the following connections.

1. *Static links* (HTML → HTML): Most of the early literature on Web testing focused on link validation. Note that this does not address any software or dynamic issues.

2. *Dynamic links* (HTML → software): HTML links call software components to execute some process. This kind of coupling is difficult to test because of the networked nature of the software.

3. *Dynamic form links* (HTML → software): HTML forms send data to software components that process the data. One issue with testing dynamic links is that data must be found or created for the forms.

4. *Dynamically created HTML* (software → HTML): Web software typically responds to the user with HTML documents. The contents of the HTML documents often depend on inputs, which complicates testing.

5. *State specific GUIs* (software + state → HTML): HTML documents whose contents and form are determined not just by inputs, but by part of the state on the server, such as the data or time, the user, or session information.

6. *Operational transitions* (user): Transitions that the user introduces into the system outside of the control of the HTML or software. Operational transitions include use of the back button, the forward button, and URL rewriting.

7. *Software connections*: This includes connections among back-end software components, such as method calls.

8. *Off-site software connections*: Some Web applications will access software components that are available at a remote site. This type of connection, while powerful, is difficult to test because little is known about the off-site software.

9. *Dynamic connections*: Both the J2EE platform and .NET allows new Web components to be installed dynamically during execution, and the Web application can detect and use the new components. Web services in J2EE uses Java reflection to accomplish this. This type of connection is especially difficult to test because the tester cannot be sure how the components will behave before deployment and execution.

2 Dynamic Integration of Web Software

The term software coupling has been in use since at least the 1970s, with general acceptance that "less" coupling is better. However, it has been difficult to define or quantify "less" or "more" coupling. We offer the following as working definitions that are useful in building our model, without claiming that they are universally applicable. We use the term *method* generically to refer to methods, procedures, subprograms and functions. A program exhibits *tight coupling* if dependencies among the methods are encoded in the logic of the methods. That is, if A and B are tightly coupled, and A calls B, a change in A might require the logical structure of B to be changed.

A program exhibits *loose coupling* if dependencies are encoded in the structure and flows of data among the methods. This typically occurs when data is defined in the callers

and used in the callees, or one method calls two different methods, one that defines a data object and the other that uses it. One ramification is that if A and B are loosely coupled, and A calls B, a change in A might result in the **structure** of the data changing, which in turn requires changes in the way B uses data items that are defined in A. Loose coupling is normally seen when data abstraction and information hiding is employed.

A program exhibits *extremely loose coupling (ELC)* if dependencies among the methods are encoded entirely in the **contents** of the data being transmitted, not in the structure. For example, extremely loose coupling is achieved when XML messages are exchanged and when HTTP requests are made. If A and B are extremely loosely coupled, and A sends data to B, a change in A might change the contents of the data that B uses, but not the structure of the data. Thus a change in A would have minimal, and perhaps no effect on B.

Although some programmers have used ELC in other types of software, Web software actively encourages extremely loose coupling. Indeed the multi-platform (usually multi-tier) design of Web applications makes it difficult to use loose or tight coupling. For example, data that is passed from an HTML page on the client to a servlet on the server is transmitted in HTTP requests. The data formatting must satisfy a strict definition of structure or the two programmers will not get the interaction right. Applications can even require ELC, for example by using XML messages. This is common among *Web services*, which are Web applications that communicate with other Web applications without user interaction.

Extremely loose coupling allows non-obvious engineering practices such as software modules that *dynamically* integrate with other software modules that use the same data structure. A very simple example is that of a Java servlet that can accept and process form data from any arbitrary HTML page, an ability that allows it to dynamically integrate with new software components. A simplified demo of this capability that is used as a classroom example is available at: `http://www.ise.gmu.edu/~ofut/formhandler/`. The HTML page is a simple form whose contents are sent to a generic Java servlet that can process inputs from

any HTML form. This kind of dynamic integration, usually coupled with more advanced technologies like enterprise Java beans, is sometimes used by Web software applications to look for and use an appropriate handler or service during execution.

An important problem for testing and analysis is that parts of Web software applications can be created dynamically. Server-side JSPs and servlets create HTML pages that contain data, responses to users, and user interfaces. These pages contain JavaScripts, hyperlinks, and content that is created dynamically. Server-side components such as Enterprise Java Beans (EJBs) can be inserted into the system at any time, and existing (even currently running) Web software applications have the ability to recognize and begin using them immediately.

These abilities mean that parts of Web software applications are *generated dynamically*. Another way to say this is that different users will see different programs at different times! One common example is that of news sources such as `washingtonpost.com`, which shows different content based on the time of day and user's location as determined by the IP address. Another is `amazon.com`, which makes different features available to users depending on whether they have logged in, have an active Amazon cookie on their computer, or where they are located as determined by their IP address. Still another example is IEEE's `manuscriptcentral.com` (built by ScholarOne, Inc.), which offers different programs to users depending on their login information.

Web software applications also allow unusual changes in the control of execution of the application. In traditional programs, the control flow is fully managed by the program, so the user can only affect it with inputs. Control flow graphs can be derived statically based on the structures in the programming language. Control flow graphs for traditional programs can describe all the possible sequence of statements that users can execute.

Web applications do not have this same property. When executing Web applications, users can break the normal control flow without alerting the program controller. This can

be done by pressing the back or refresh buttons in the browser or by directly modifying the URL in the browser. These interactions introduce arbitrary changes in the execution flow, creating control paths in the software that are impossible to represent with traditional techniques such as control flow graphs. Users can also directly affect data in unpredictable ways, for example, by modifying values of hidden form fields. Furthermore, changes in the client-side configuration may affect the behavior of Web applications. For example, users can turn off cookies, which can cause subsequent operations to malfunction.

Although these dynamic properties of Web applications are powerful and offer advantages to the developers, they introduce new problems to software engineers. Specifically, traditional analysis structures such as control flow graphs, call graphs, data flow graphs, and data dependency graphs can no longer accurately represent the program. That is, the program's *possible* flow of control cannot be known statically. These analysis structures are needed for data flow analysis and slicing, techniques that are used in many activities, including design, testing, and maintenance. New techniques are needed to model our program to support these activities.

3 Modeling Web Applications

This paper introduces a new model based on elemental pieces of software components for describing Web applications that can be used to model all possible execution flows, just as control flow graphs can be used for traditional programs. This model begins by describing elements of dynamically created Web pages. The analysis model then uses regular expression notation to combine the elements. Criteria for developing tests are presented in Section 4.

3.1 Analysis Model

A key element of Web applications is the collection of **interactions** among the software components. A typical Web application works as depicted in Figure 1. A client first retrieves

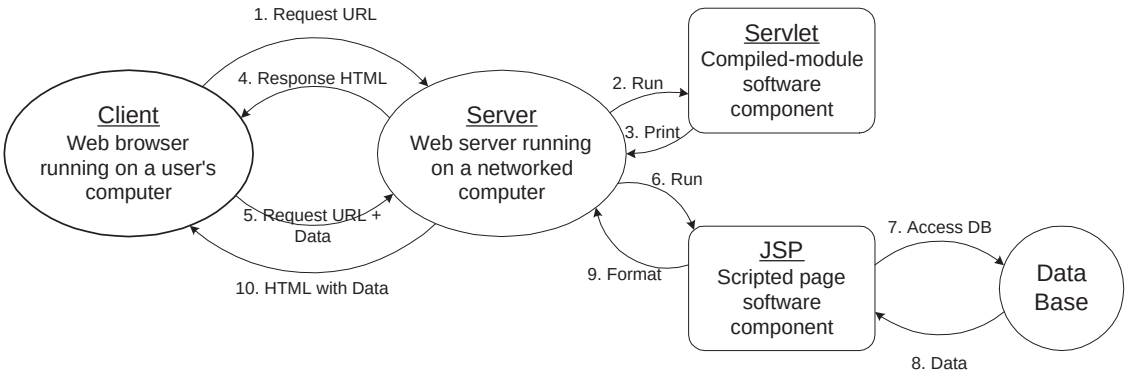

Figure 1: Typical execution flow among Web application components.

information from a server by requesting a particular URL, the server runs a servlet, which prints HTML, which is then returned to the client as an HTML document. The client then sends a request with data to a server, which runs a Java server page, which in turn accesses a database, and finally returns formatted data to the client in the form of HTML. Although not all interactions are done through HTML pages, this is a useful model to start with and this paper assumes that clients and servers interact through HTML pages. In the future, this model can be applied to other forms of interactions, such as software that uses XML.

Without loss of generality, we assume that each Web application uses a unique *start page* S. If there is more than one start page $(S_1, S_2, ..., S_k)$, then we assume a unique start page can be created that has links to each actual start page.

Clients and servers interact with each other through requests and HTML messages. Many Web software applications generate HTML pages dynamically by assembling pieces from separate files and program statements. This process needs to be formalized before we can model interactions among Web applications. Thus we first identify the basic elements that can be generated, then define a set of operations that can be used to generate new compound elements.

3.2 Atomic Sections

An *atomic section* (AtS) is a section of HTML (possibly including scripting language code such as JavaScript) that has the property that if part of the section is sent to a client, the entire section is. This is called an "all-or-nothing property" and atomic sections are analogous to basic blocks in traditional programs (although the focus is on data presentation, not execution, and many executable statements are ignored). The simplest AtS is a complete static HTML file. Dynamically generated HTML pages are typically comprised of several atomic sections from a server program that prints HTML. An AtS may be constant (pure HTML), it may be an HTML section that has a **static structure** with content variables, or it may be empty. A *content variable* is a program variable that provides data to the HTML page but not structure.

Figure 2 illustrates a pair of stylized dynamically created Web pages for a fictional Web application called `WebPics`. They appear to be made up of at least four atomic sections. The first consists of the header welcome message and the search box, which have been personalized to the user. The second consises of the list of recommended movies, which depends on data on the customer drawn from the data base. The third consists of the short menu, which again is customized to the user. Finally, the fourth ATS contains an extra interface point to the program, and is only available to the customer on the left.

Figure 3 shows a Java servlet from a server component P that produces six atomic sections (p_1 .. p_6). Note that only output statements are annotated as atomic sections. To be precise, only the output of those statements are atomic sections, not the actual Java code, but we show the atomic sections with the surrounding code for clarity. Section $p5$ is an example of an empty AtS; even though it may not be in the original program, it must be included as an alternative to $p2$.

Atomic sections can be thought of as being analogous to basic blocks in traditional

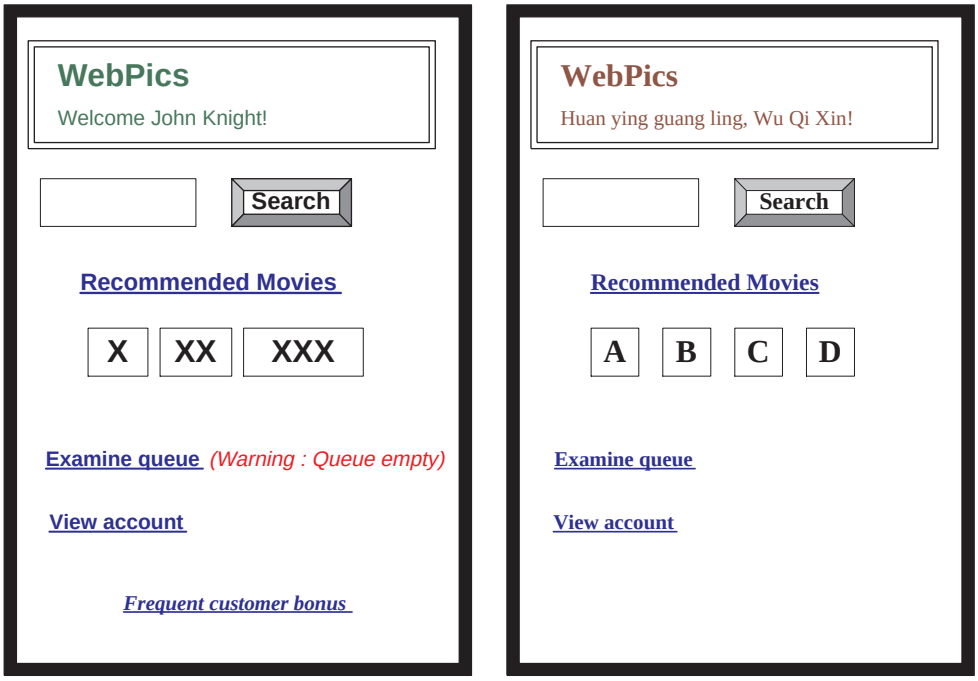

Figure 2: Stylized Web pages to illustrate atomic sections in dynamically created Web pages.

programs, but have many distinct differences. First, Web applications are extremely loosely coupled and have frequent dynamic interactions through HTTP requests. Atomic section analysis can ignore most of the internal processing of the software and focus on the HTML responses. The example in Figure 4 contains nine basic blocks but only three atomic sections. The atomic sections reflect the relationship of this component with other components, while the first six basic blocks are only relevent to the internal processing of this module.

3.3 Composite Sections

Atomic sections are combined together to form more complex units, called composite sections. The composition is usually done dynamically and the actual composition is affected by the control flow of the server component. Different users will get different complete HTML pages, and thus different user interfaces. In effect, they have access to different programs. Possible compositions are *sequence*, *selection*, *iteration*, and *aggregation*. Formally, p is a

	`PrintWriter out = response.getWriter();`

p1 =	`out.println ("<HTML>");`
	`out.println ("<HEAD><TITLE>" + title + "</TITLE></HEAD>");`
	`out.println ("<BODY>");`

	`if (isUser) {`
p2 =	`out.println ("<CENTER>Welcome!</CENTER>");`
	`for (int i=0; i<myVector.size(); i++)` `if (myVector.elementAt(i).size > 10)`
p3 =	`out.println ("<P><B>" + myVector.elementAt(i) + "</B></P>");`
	`else`
p4 =	`out.println ("<P>" + myVector.elementAt(i) + "</P>");`
	`}` *else*
p5 =	`{ }`
p6 =	`out.println("</BODY></HTML>");`
	`out.close();`

Figure 3: Servlet atomic section example.

composite section of a server program P in the following situations.

1. *Basis*: p is an atomic section.

2. *Sequence*: $(p \rightarrow p_1 \cdot p_2)$: p_1 and p_2 are composite sections, and p is composed of p_1 followed by p_2.

3. *Selection* $(p \rightarrow p_1 \mid p_2)$: p_1 and p_2 are composite sections, and the server selects either p_1 or p_2, but not both.

4. *Iteration* $(p \rightarrow p_1^*)$: p_1 is a composite section, and the server selects repeated copies of p_1.

5. *Aggregation* $(p \rightarrow p_1\{p_2\})$: p_1 and p_2 are composite sections, p_2 is included as part of p_1 when p_1 is transmitted to the client. For example, a function call in p_1 or a file inclusion command will include p_2 in p_1.

	String manufacture = request.getParameter ("manufacture"); String productName = request.getParameter ("productname"); String minPrice =request.getParameter ("minPrice")	BB1
	if (productname != null)	
	queryCriterion = "Where productname='"+productname+"'";	BB2
	if (manufacture != null)	
	if (queryCriterion == null)	
	queryCriterion = "Where manufacture='"+manufacture+"'";	BB3
	else	
	queryCriterion = queryCriterion + " and manufacture='" + manufacture + "'";	BB4
	if (minPrice != null)	
	if (queryCriterion == null)	
	queryCriterion = "Where price >"+minPrice;	BB5
	else	
	queryCriterion = queryCriterion + " and price >>"+minPrice;	BB6
	ResultSet rs=dbConnection.executQuery ("select * from db " + queryCriterion); PrintWriter out = response.getWriter();	
p1 =	out.println ("<HTML>") out.println ("<BODY>") out.println ("<FORM action='http://ise.gmu.edu/servlet/selectProduct'>")	BB7
	while (rs.next())	
p2 =	out.println ("<INPUT type=checkbox name=product>"+ rs.getString ("product"));	BB8
p3 =	out.println ("<INPUT type=submit><INPUT type=cancel");	BB9
	out.println ("</BODY></HTML>");	
	out.close();	

Figure 4: Atomic Section vs. Basic Block

These elementary operations can be extended as needed using typical BNF notation. For example, p^+ can be used to represent one or more composite sections concatenated together and p^n to represent exactly n composite sections. Other expressions can be used as needed. It is often necessary to denote atomic and composite sections by the program unit that generates them. We use the "dot" operator, so $S.p_1$ indicates the composite section p_1 is produced by program component S.

Atomic and composite sections define how HTML pages can be dynamically generated. Given a server component, a *composition rule* is a regular expression that represents all possible complete HTML pages that can be generated by the component. The composition rule for the example in Figure 3 is $P \rightarrow p_1 \cdot (p_2 \cdot (p_3 \mid p_4)^* \mid p_5) \cdot p_6$. Note that it might be possible to replace the unbounded iteration in P with "myVector.size()." However, this value cannot be computed statically, so we choose to model an unknown iteration as unbounded. That is, a goal of this research is to produce a static analysis technique.

The above representation can be used to model the internal structure of individual server components. To execute a complete transaction in a Web application, the client and server components link the dynamically generated HTML pieces together. Interactions among client and server components are generally more complex than those of traditional applications. Most traditional applications have deterministic function invocations; even the uncertainty caused by polymorphism is limited by, for example, the polymorphic call set of Alexander and Offutt [1]. Web applications, on the other hand, have function invocations whose binding cannot be known or even limited statically. The functions that can potentially be invoked are not necessarily known until execution time.

3.4 Modeling Dynamic Interaction

Web applications use HTML and action links to combine components. When HTML pages are generated dynamically, these links may rely on dynamic information, which means the contents are not known until execution time. Furthermore, users can modify the execution flow of Web applications, taking some of the control away from server and client components. The simplest example is when a user hits the back button in the browser, causing the application to return to a previous screen. This changes the control flow without notifying either the server or client components, and can introduce data anomalies if the data on the screen does not match the current state. To fully model the behavior of dynamic Web

applications, the ATS analysis model defines *dynamic interactions*.

The interactions among different server components can be classified into four types of *transitions*. In the following, p and q are composite sections and s is a servlet or other software component that generates HTML.

1. *Link Transition* ($p \longrightarrow q$ and $p \longrightarrow\!\!\gg q$): Invoking a link in p causes a transition to q from the client to the server. If p can invoke one of several static or dynamic pages, $q_1, q_2, \ldots, q_k$, then the destination is represented as $q_1 \mid q_2 \mid \ldots \mid q_k$. Link transitions are divided into two types. A *simple* link transition, $p \longrightarrow q$, is an HTML link defined in an <A> tag, and a *form* link transition, $p \longrightarrow\!\!\gg q$, is defined in a <FORM> tag.

2. *Composite Transition* ($s \multimap p$): The execution of s causes p to be produced and returned to the client. The servlet s will normally be able to produce several composite Web pages, which can be represented as $s = p_1 \mid p_2 \mid \ldots \mid p_k$.

3. *Operational Transition* ($p \rightsquigarrow q$): The user can create new transitions out of the software's control by pressing the back button, the refresh button, or directly modifying the URL in the browser (including adding or modifying parameters). Operational transitions also model transitions caused by system configurations, for example, the browser may load a Web page from the cache instead of the server. The notation "previous S" represents use of the back button and "reload S" represents use of the refresh button.

4. *Forward Transition* ($p \longrightarrow\!\!\!\!\!+\; q$): A forward transition is a server-side software transition that is not under the control of the tester. For example, if a user successfully logs in to an application, the login component may automatically forward to another component.

3.5 A Model of a Web Application

Web applications are modeled at two levels. Atomic and composite sections are used to model individual components (intra-component) and then the dynamic interactions are used to create a graph model of the inter-component relations.

3.5.1 The Intra-component Level

A Web component is modeled as a quintuple $WC = (S,\ C,\ T,\ ATS,\ CS)$, where S is the start page, C is a set of composition rules for each server component, T is a set of transition rules, ATS is the set of atomic sections, and CS is the set of composite sections.

The example in Figure 5 is an HTML page that uses the Java servlet in Figure 6 to provide online grade queries to students. A student must access the main page first to enter an id and password. Then a servlet validates the id and password; if successful, the servlet retrieves the grade information and sends it back to the student. If unsuccessful, an error message is returned to the student asking the student to either retry or send an email to the instructor for further assistance. This small application includes a static HTML file, a query servlet, and another servlet that processes the email to the instructor (not shown). The HTML file uses a "hidden" form field to keep track of how many login attempts the user has made. A hidden form field is an `INPUT` tag that has the attribute type `HIDDEN` (`<INPUT Type="HIDDEN" Name="RETRY" Value="0">` in Figure 5). Web browsers do not render hidden form fields on the user's screen, but the data that is stored in the field is submitted to the server. Hidden form fields are sometimes used to keep data persistent from one request from the same user to the other. Although called hidden, it should be noted that the HTML source is stored on the user's computer and the hidden form field can be seen, used, and modified.

The atomic sections for `GradeServlet` are shown in Figure 6. `GradeServlet` uses three methods, `Validate()`, `CourseName()` and `CourseGrade()` (which are omitted for brevity).

```
<HTML>
 <HEAD>
    <TITLE>Grade Query Page</TITLE>
 </HEAD>
 <BODY>
    <FORM Method="GET" Action="GradeServlet">
       Please input your ID and password:
       <INPUT Type="TEXT"     Name="ID"     Size="10">
       <INPUT Type="PASSWORD" Name="PASSWD" Size="20">
       <INPUT Type="HIDDEN"   Name="RETRY"  Value="0">
       <INPUT Type="SUBMIT"   Name="SUBMIT" Value="SUBMIT">
       <INPUT Type="RESET"                  Value="RESET">
   </FORM>
 </BODY>
</HTML>
```

Figure 5: Simple HTML login page.

The start page, composition rules, and transition rules for `GradeServlet` are:

$S = \{\text{index.html}\}$
$C = \{\text{GradeServlet} = p_1 \cdot ((p_2 \cdot p_3^*) \mid p_4) \cdot p_5\ \}$
$T = \{S \longrightarrow \text{GradeServlet}, \text{GradeServlet}.p_4 \longrightarrow \text{SendEmail} \mid \text{GradeServlet}\}$

It should be noted that when link transitions are generated dynamically, the composite sections that are targeted **cannot** be known statically.

3.5.2 The Inter-component Level

The software is modeled by combining ATS and composite sections into a *Web Application Graph (WAG)* in which nodes are Web components and edges are links and other types of transitions among the nodes. Formally a Web application is modeled as a quintuple $WAG = (C,\ L,\ T,\ s,\ f)$, where C is a finite set of Web components, L is a set of link transitions, T is the transition relation (a subset of $C \times L \times C$), $s \in C$ is the initial component, and $f \in C$ is the final component.

The proliferation of technologies means that there are many types of components, including HTML pages, Java Servlets, JSPs, PHPs, ASPs, Java Beans, CGI files, and Java

	`ID       = request.getParameter ("ID");`
	`passWord = request.getParameter ("PASSWD");`
	`retry    = request.getParameter ("RETRY");`
	`PrintWriter out = response.getWriter();`

p1 =	`out.println ("<HTML>");`
	`out.println ("<HEAD><TITLE>" + title + "</TITLE></HEAD>");`
	`out.println ("<BODY>");`

	`if (Validate (ID, passWord) && retry < 3)` `{`
p2 =	`   out.println ("<B> Grade Report </B>");`
	`   for (int I=0; I < numberOfCourse; I++)`
p3 =	`      out.println ("<P><B>" + CourseName(I) + "</B>" + CourseGrade(I) + "</P>");`

	`}`
	`else`
	`{`

p4 =	`   retry++;`
	`   out.println ("Wrong ID or wrong password");`
	`   out.println ("<FORM Method=\"GET\" Action=\"GradeServlet\">);`
	`   out.println ("<INPUT Type=\"TEXT\" Name=\"ID\" Size=\"10\">");`
	`   out.println ("<INPUT Type=\"PASSWORD\" Name=\"PASSWD\" Width=20>");`
	`   out.println ("<INPUT Type=\"HIDDEN\" Name=\"RETRY\" Value=" + (retry) + ">");`
	`   out.println ("<INPUT Type=\"SUBMIT\" Name=\"SUBMIT\" Value=\"SUBMIT\">");`
	`   out.println ("<A HREF=\"sendMail\">Send Mail to the Instructor</A>");`
	`   out.println ("<INPUT Type=\"RESET\" Value=\"RESET\"></FORM>");`

	`}`
p5 =	`out.println ("</BODY></HTML>");`
	`out.close();`

Figure 6: Atomic sections of servlet GradeServlet.

classes. An advantage of this model is that the information being captured is independent of the technology used to create the components (although extracting the information certainly depends on the technology). A second advantage is that the model does not depend on whether the software components are on the same computer, two computers, or multiple computers. One restriction to note is that this model requires access to the source.

As defined in Section 3.4, transitions are categorized as link transitions (simple and form), composite transitions, operational transitions, and forward transitions. This model explicitly ignores method calls, assuming they are tested by traditional modeling and testing

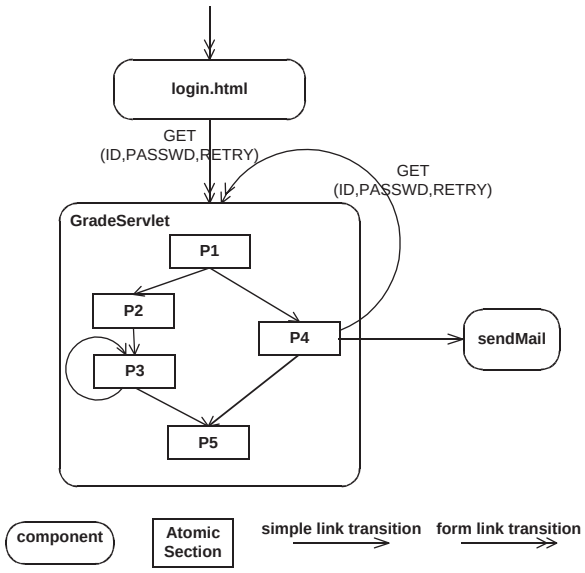

Figure 7: Web application graph for GradeServlet.

techniques. Figure 7 shows the WAG for the GradeServlet example. The WAG has only three components, the login HTML page, GradeServlet, and sendMail. The GradeServlet component is drawn with its atomic sections, arranged in a graph representation of the composite section rules, to illustrate the two levels of this model.

Inter-component links are also annotated with two types of information. The first indicates what type of HTTP request was used in the transition. These are most commonly `get` and `post`, although others are available. Calls to non-web specific software components (for example, normal Java classes) are not modeled separately, but considered as part of the Web component. Likewise, JSP include files are considered to be part of the including JSP file. The second annotation is the data that is being transmitted, usually in the form of parameters.

The WAG may also include a current state representation to model execution information. During execution, each component has a $CurrentState$, which is a set of name-value pairs for the component's static variables, application scope attributes, session scope at-

tributes, and session objects. These are the values that can be shared among two or more components in a single Web application. Variables that cannot be shared (local scope variables) are not used in this model.

4 Generating Web Application Tests

A test case for a Web application is specified as a sequence of interactions between components on clients and servers. Graph coverage criteria [2, 28] can be used to cover the graph model in Section 3.5, with details supplied by the atomic and composite sections. An important consideration with Web applications is that of invalid transitions, which are not considered in traditional graph coverage criteria. Thus it is necessary to extend the criteria to cover invalid transitions.

A sequence of interactions is represented as a *derivation*, which is a sequence of transitions that begins at the start page S, and uses composition and transition rules to reach a desired page. A *subsequence* of a derivation is the sequence of transitions between two intermediate Web pages. A derivation for a normal grade query from the GradeServlet example in Figure 6 combines both the link transitions and atomic sections:

$$\text{S} \longrightarrow\!\!\!\!\gg \text{GradeServlet} \longrightarrow\!\!\!\!\circ\ p_1 \cdot p_2 \cdot p_3 \cdot p_5$$

This derivation starts at the start page S (the login HTML page), then submits the form and moves to GradeServlet, and the composite transition yields the title (p_1), header (p_2), one grade (p_3), and the ending HTML commands (p_5).

Several derivations can be made when a student enters an incorrect ID or password. Note that the WAG can contain loops, which traditionally cause difficulties with coverage criteria. We handle loops in WAGs by using the relatively new criterion prime path coverage [3]. Derivations for incorrect ID or passwords are:

1. $\text{S} \longrightarrow\!\!\!\!\gg \text{GradeServlet} \longrightarrow\!\!\!\!\circ\ p_1 \cdot p_4 \cdot p_5$

2. S $\longrightarrow\!\!\gg$ GradeServlet $\multimap$ $p_1 \cdot p_4 \cdot p_5 \longrightarrow$ SendMail ...

3. S $\longrightarrow\!\!\gg$ GradeServlet $\multimap$ $p_1 \cdot p_4 \cdot p_5 \rightsquigarrow$ previous S $\longrightarrow\!\!\gg$
   GradeServlet $\multimap$ $p_1 \cdot p_2 \cdot p_3 \cdot p_5$

4. S $\longrightarrow\!\!\gg$ GradeServlet $\multimap$ $p_1 \cdot p_4 \cdot p_5 \rightsquigarrow$ previous S $\longrightarrow\!\!\gg$
   GradeServlet $\multimap$ $p_1 \cdot p_4 \cdot p_5 \rightsquigarrow$ previous S $\longrightarrow\!\!\gg$
   GradeServlet $\multimap$ $p_1 \cdot p_4 \cdot p_5 \rightsquigarrow$ previous S $\longrightarrow\!\!\gg$
   GradeServlet $\multimap$ $p_1 \cdot p_2 \cdot p_3 \cdot p_5$

Note that in the fourth derivation above the user tried three wrong usernames and passwords, as indicated by the inclusion of AtS p_4. The variable `RETRY` was then equal to three and further access should be denied. However, the user used the browser's back button to reload the previous page, as indicated by the transition `previous S`. This meant the value of the hidden form field for `RETRY` was reset to zero and the servlet was not able to recognize that the retry limit was reached. Traditional analysis techniques would not be able to model this interaction, and thus would be unlikely to help the tester find a test to find that fault in the software[1].

Each derivation represents a specification for a test case and the *CurrentState* information for each component is then used to create executable test cases. For complex applications, the number of possible derivations on the WAG can be very large or infinite, so test criteria are used to choose derivations.

4.1 Coverage Criteria

Test coverage can be applied at both the intra-component level (using the composite sections) and the inter-component level (using the WAG). As is usually the case, the tests at the two levels could be independent or could be merged into one set of tests. And as noted previously,

[1] As an aside, we have found a similar fault with the National Science Foundation's FastLane system. Specifically, login information is stored in hidden fields in the Web pages, and use of the back button can put the system into unstable states.

any graph coverage criterion can be used.

The empirical validation in Section 5 applies graph coverage techniques to the WAG and to the transitions in the composite sections, including link, composite, and operational transitions. Coverage criteria are divided into several levels to evaluate the effectiveness of the unique aspects of the model. Past research on Web application testing [19, 21, 25] has focused exclusively on link transitions, with little attention paid to composite transitions and none at all to operational transitions.

For a comparative evaluation, we apply three separate coverage criteria. The first is the prime criterion, defined by Ammann and Offutt to explicitly and quantitatively allow loops to be covered [2, 3]. A subpath in a graph from node n_i to n_j is *prime* if no node appears more than once on the subpath, with the exception that the first and last nodes may be identical. A *test path* is a complete sequence of nodes in a graph from a start node to an end node. A test path p is said to *tour* a subpath q if and only if q is a subpath of p. Touring paths in graphs with loops is not very practical because there are an infinite number of loops. The notion of touring with a "sidetrip" allows a test path to tour q while at the same time adding a few nodes "in the middle" of q, that is, executing a loop more than one time. Formally, A test path p is said to *tour a subpath q with sidetrips* if and only if every edge in q is also in p in the same order.

Figure 8 illustrates touring. Nodes S_0 and S_f are the initial and final nodes. The prime subpath $[a,\ b,\ d]$ can be toured by sidetrips by a test path that executes the loop from b to c an arbitrary number of times, such as by $[S_0,\ a,\ b,\ c,\ b,\ d,\ S_f]$. The set of prime subpaths on the graph in Figure 8 includes $[c,\ d]$, $[b,\ c,\ b]$, and $[c,\ b,\ c]$ (among others), so a set of test paths to satisfy prime coverage can be $\{\ [S_0,\ a,\ b,\ c,\ d,\ S_f]$, $[S_0,\ a,\ b,\ c,\ b,\ d,\ S_f]$, $[S_0,\ a,\ b,\ c,\ b,\ c,\ d,\ S_f]\ \}$.

The second criterion used is new to this paper, but quite simple. The intent is to model the situation when users, either accidentally or purposefully, apply operational transitions

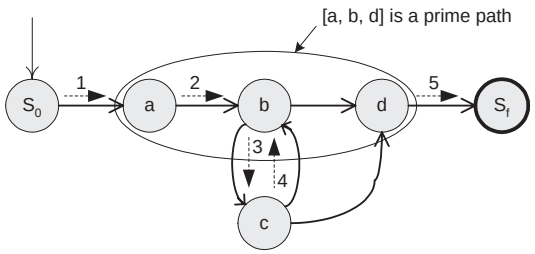

Figure 8: Touring a graph with sidetrips.

by entering into the middle of a Web application. This is called *invalid access* (IA), and each test is exactly one sequence long; a URL to a non-start page.

The third criterion, *invalid path* (IP), extends the first criterion by adding operational transitions. Prime paths are extended by two transitions: the first adds one operational transition, and if that is feasible, the second adds every valid transition out of the new node. Sometimes the first transition is feasible but the second is not, so it is necessary to distinguish between the two. In these cases, they are referred to as IP-1 and IP-2. If applied to all prime paths, the invalid path criterion has the potential to lead to a very large number of paths. So the invalid path criterion is defined at two levels. *All invalid paths* requires that **every** prime path be extended with all possible operational transitions. *Each node invalid path* requires that for each node in the graph, exactly one prime path that ends in that node is extended.

Previous research in testing Web applications did not include operational transitions, so had nothing comparable with the invalid access and invalid path criteria. Moreover, the prime criterion subsumes edge coverage and edge-pair coverage, so can be considered a very strenuous graph coverage test. These criteria are illustrated on Figure 9, which has three nodes, all of which are considered to be final nodes. Node 1 is the only start node.

There are a total of 12 prime paths on the graph in Figure 9; four paths of length 1, five paths of length 2, and three of length 3.

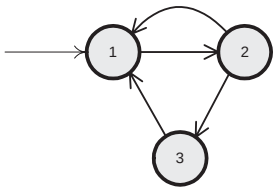

Figure 9: Graph to illustrate coverage criteria.

```
[1, 2] [2, 1] [2, 3] [3, 1]
[1, 2, 1] [1, 2, 3] [2, 1, 2] [2, 3, 1] [3, 1, 2]
[1, 2, 3, 1] [3, 1, 2, 3] [2, 3, 1, 2]
```

The prime paths can be toured with the following two test paths:

```
[1, 2, 3, 1, 2, 3]
[1, 2, 1, 2]
```

There are two invalid access tests to the two non-start nodes, `[2]` and `[3]`. With this small example there are only two invalid transitions, `[1, 3]` and `[3, 2]`. To compute the invalid paths, we start with the eight prime paths that end in nodes 1 and 3, add another (invalid) transition to each, and then the transition `[3, 1]` to the paths that end with 3 and the two transitions `[2, 1]` and `[2, 3]` to the paths that end with 2. This results in the following 11 paths to cover.

```
[2, 1, 3, 1]
[2, 3, 2, 1]        [2, 3, 2, 3]
[3, 1, 3, 1]
[1, 2, 1, 3, 1]
[1, 2, 3, 2, 1]     [1, 2, 3, 2, 3]
[2, 3, 1, 3, 1]
```

```
[1, 2, 3, 1, 3, 1]
[3, 1, 2, 3, 2, 1] [3, 1, 2, 3, 2, 3]
```

These can be toured with the following test paths:

```
[1, 2, 1, 2, 3, 2, 1]
[1, 2, 1, 2, 3, 2, 3]
[1, 2, 3, 1, 3, 1]
[1, 2, 1, 3, 1]
[1, 2, 3, 1, 2, 3, 2, 1]
[1, 2, 3, 1, 2, 3, 2, 3]
```

5 Empirical Evaluation

To validate the model and test criteria, we applied them to a small but non-trivial Web application. The Small Text Information System (STIS) helps users keep track of arbitrary textual information[2]. Text records are stored and associated with categories. STIS stores all information in a database (currently mysql) and is comprised of 17 Java Server Pages and 5 Java bean classes. STIS requires users to log in. After being successfully authenticated, users can search and view records ordered by categories, create categories and new records.

The Web application graph model of STIS is shown in Figure 10. Eleven of the seventeen Web components are shown. Four Java server pages (HEAD, FOOT, BAR and SEARCH) are statically included inside the other components so are not shown. Two others are used only with administrative access and we chose to omit them from the graph and the experiment. In theory, operational transitions exist between every pair of components, but they are not

[2]STIS was implemented by a summer student working on an NSF-sponsored Research Experience for Undergraduate supplemental grant. A demo version can be viewed online at `http://www.ise.gmu.edu:8080/ofut/jsp/stis/`, with userid "demo" and password "demo."

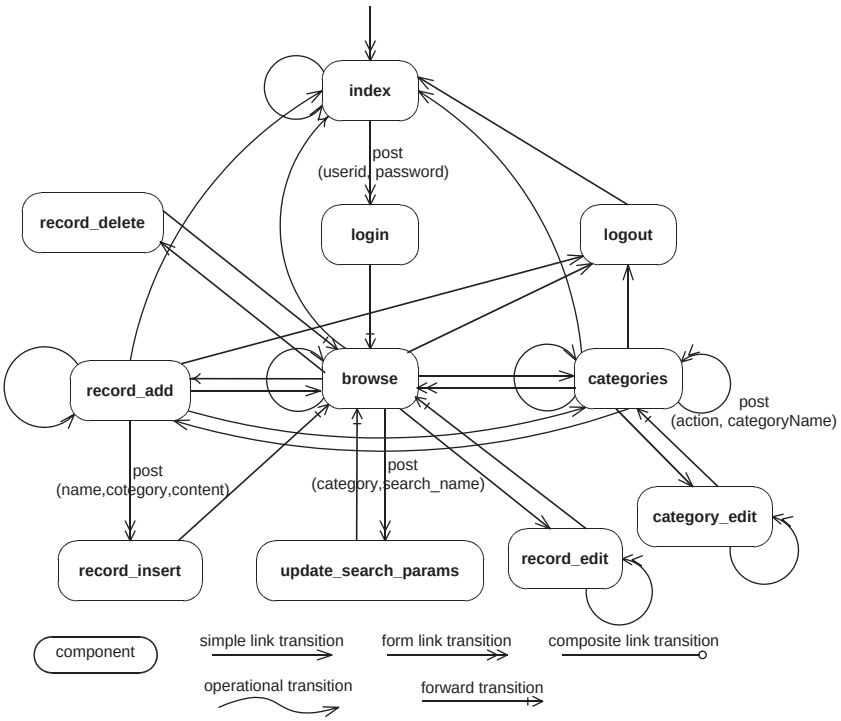

Figure 10: STIS Web application graph model.

shown on this graph to reduce eyestrain. Also, the composite transitions are internal to the nodes, and are not shown in this graph.

Atomic section analysis is performed by an automated tool that parses Java servlets, Java Server Pages, and other Java classes. The composition section rules for the 17 JSP components in STIS are shown in Table 1. The number of atomic sections for each component is given in the third column. This count includes empty sections, and each occurence of an included JSP (HEAD, FOOT, BAR, and SEARCH) is counted as one atomic section. The instances of the included JSPs are used as abbreviations for their composite sections. When expanded, the rule for *browse* is:

$$browse \multimap p_1 \cdot ((p_2 \cdot (p_3 \mid e) \cdot p_4) \mid p_5) \cdot p_6 \cdot {p_7}^* \cdot p_8 \cdot (p_{10} \cdot {p_{11}}^* \mid p_9) \cdot p_{12} \cdot {p_{13}}^* \cdot p_{14} \cdot$$
$$((p_{15} \cdot (p_{16} \mid e) \cdot p_{17}) \mid p_{18}) \cdot (p_{19} \mid e) \cdot p_{20}$$

The empirical evaluation proceeded according to the following steps. Each step is anno-

Table 1: Composition rules for STIS components.

Component	Composition Rule	ATS
index	$\multimap HEAD \cdot p_1 \cdot FOOT$	3
login	$\multimap p_1 \cdot ((HEAD \cdot p_2) \mid (HEAD \cdot p_3) \mid (p_4 \cdot BAR \cdot p_5))$	8
browse	$\multimap HEAD \cdot p_1 \cdot SEARCH \cdot p_2 \cdot (p_3 \mid p_4 \cdot {p_5}^*) \cdot p_6 \cdot SEARCH \cdot FOOT$	10
record_add	$\multimap HEAD \cdot p_1 \cdot {p_2}^* \cdot p_3 \cdot FOOT$	5
categories	$\multimap HEAD \cdot p_1 \cdot (p_2 \mid (((p_3 \mid e) \mid e) \cdot p_4 \cdot ((p_5 \cdot {p_6}^* \cdot p_7) \mid p_8) \cdot p_9)) \cdot FOOT$	13
record_insert	$\multimap HEAD \cdot p_1 \cdot (e \mid (p_2 \cdot {p_3}^* \cdot p_4)) \cdot FOOT$	7
update_search_params	$\multimap e$	1
logout	$\multimap p_1 \cdot ((p_2 \cdot HEAD \cdot p_3) \mid p_4)$	5
record_edit	$\multimap HEAD \cdot p_1 \cdot (e \mid (p_2 \cdot (p_3 \mid e) \cdot p_4 \cdot (p_5 \cdot (p_6 \mid e) \cdot p_7)^* \cdot p_8) \cdot FOOT$	13
record_delete	$\multimap HEAD$	1
category_edit	$\multimap HEAD \cdot p_1 \cdot (p_2 \mid (p_3 \mid e) \cdot p_4) \cdot p_5 \cdot FOOT$	8
register	$\multimap HEAD \cdot p_1 \cdot (p_2 \mid p_3) \cdot FOOT$	5
register_save	$\multimap p_1 \cdot ((p_2 \cdot HEAD \cdot p_3) \mid (HEAD \cdot p_4 \cdot FOOT))$	7
HEAD: *page_header*	$\multimap p_1 \cdot BAR$	2
FOOT: *page_footer*	$\multimap BAR \cdot (p_1 \mid e) \cdot p_2$	4
BAR: *navigation_bar*	$\multimap (p_1 \cdot (p_2 \mid e) \cdot p_3) \mid p_4$	5
SEARCH: *record_search*	$\multimap p_1 \cdot {p_2}^* \cdot p_3 \cdot (p_4 \mid e) \cdot p_5$	6

tated with whether it was accomplished automatically or by hand in this experiment. Proper tool support should be able to automate every step but the last.

1. Determine atomic sections (automatically)
2. Derive the web application graph (by hand)
3. Determine prime paths for the WAG (automatically)
4. Determing test paths to tour the prime paths (automatically)
5. Determine invalid access transitions (automatically)
6. Extend prime paths to create invalid paths (automatically)
7. Choose input values for forms (by hand)
8. Run tests and record results (by hand)

9. Determine AtS coverage (automatically, with instrumentation)

10. Develop tests to complete AtS coverage (by hand)

5.1 Test Value Selection

Testing Web applications requires more than just covering transitions; input values also must be supplied. In Web applications, most inputs are accepted through HTML forms. For this experiment, strings were generated by hand and for the most part arbitrarily. The exception was with userids and passwords, which must match a pair in the database for the test to proceed past the login.

The test requirements were refined into actual tests by adding input values. They were run through a Web browser by entering values into the HTML forms by hand. Consider the following test requirement to execute an operational transition:

index $\twoheadrightarrow$ login $\twoheadrightarrow$ browse $\longrightarrow$ logout $\rightsquigarrow$ record_add $\twoheadrightarrow$ record_insert

The first transition is a form link transition and values for the form are `userid = "demo"` and `password = "demo"`. Since the login was successful, a forward transition is taken to browse. The test logs out with a simple link transition, and then accesses the record_add component with an operational transition. From there, the test requires values for the form, which are chosen arbitrarily as `name="X"`, `category="A"`, and `content="xxx"`, and takes a form link transition to record_insert. This test resulted in a failure, because the record was added after the logout.

5.2 Test Coverage Analysis

Tests for STIS are presented as test specification derivations and the intra- and inter-component tests are merged. A parsing tool was used to generate the atomic sections and to generate the composite sections; the WAG was generated by hand, as were the test specifications. Values for form parameters were selected by hand, mostly arbitrarily except for

userid and passwords. STIS contains 91 atomic sections, combined into the 17 composite sections listed in Table 1. The atomic and composite sections were used as a method for test evaluation; tests were measured by calculating the number of atomic sections that were reached. A total of 156 tests were generated.

The complete set of tests resulted in a total of 56 failures. All failures were naturally occurring and not known before testing began. Table 2 provides data on the tests and the test results. For each group of tests, Table 2 shows the number of test requirements, the number of test requirements that could not be satisfied, and the actual number of tests created. The number of atomic sections covered are shown for each group of tests (the total ATS coverage is cumulative across the test sets, not a sum). Finally, the number of failures found for each group is shown.

STIS has over 11,100 **full** invalid paths, so this experiment used the **each node** version of the invalid path criterion. Data for the invalid path tests are also split into two columns to separate the IP-1 and IP-2 tests. Recall that invalid tests were created by extending prime path tests by adding a sequence of two transitions. IP-1 tests are those that contain only the first (operational) transition. 40 of those paths were infeasible, 20 produced correct output, and 30 produced incorrect output. Of the 30 failing tests, 13 caused a runtime failure of STIS and 17 caused incorrect output. The second transitions (extending IP-1 tests to IP-2) could not be executed on the 40 infeasible paths or the 13 that resulted in runtime failure, so were infeasible. Of the 210 IP-2 tests, 92 were created by extending IP-1 tests that were infeasible and 18 from IP-1 tests that had a runtime failure. Of the remaining 100 IP-2 tests, the last transitions for 31 were unavailable because the incorrect output from the operational transition resulted in the links for the transition not being present. An additional 5 were infeasible because of forward transitions. The next to last node contained a forward transition, which went to another node, thus the last transition in the IP-2 test was unavailable. So there were 92+18+31+5 = 146 infeasible IP-2 tests.

Table 2: Test coverage and failure data.

	Prime Path	**Invalid Access**	**Invalid Path IP-1**	**Invalid Path IP-2**	**Total**
Number of test requirements	32	10	90	210	342
Infeasible test requirements	0	0	40	146	186
Number of tests	32	10	50	64	156
ATS coverage (91 total)	75	44	79	79	79
Total number of failures	0	7	30	19	56

Table 3 lists the number of failures found by each group of tests. The failures are divided into nine categories. This is not meant to be a comprehensive list of Web application failure categories, merely a categorization of the failures observed when testing STIS. In the first two categories, a component of the software on the server failed and no valid page was returned. These included the 13 failures that prohibited invalid path tests from being completed. Categories 3, 4 and 5 allowed users to access STIS without being properly authenticated through the login. Categories 6 through 9 are not as severe. Category 6 allowed users to edit categories that were not in the database, category 7 allowed empty records to be added, category 8 allowed the user to see the wrong content and category 9 failures were messages that were irrelevant or invalid.

5.3 ATS Coverage

The 156 tests from the WAG covered 79 of 91 atomic sections (86.8%). While this seems like good coverage, at least on the surface, it is important to understand why coverage was not complete. Specific questions are: why did the tests not cover the remaining 13 atomic sections, how difficult would it be to achieve 100% ATS coverage by hand, and how many more failures would be detected if 100% ATS coverage was achieved? Accordingly, we analyzed the remaining atomic sections and generated tests by hand to cover them.

1. Five atomic sections display information when the database fails. During the origi-

Table 3: Types of failures found.

Failure Category	Prime Path	Invalid Access	Invalid Path-1	Invalid Path-2	Total
1. Runtime failure	0	2	8	2	12
2. Unhandled software exception	0	0	5	0	5
3. Unexpected page content displayed w/o authentication	0	4	6	0	10
4. Record added w/o authentication	0	1	2	2	5
5. Search allowed w/o authentication	0	0	0	2	2
6. Editing non-existent category	0	0	2	0	2
7. Adding a record with empty fields; value is null	0	0	5	4	9
8. Unexpected content displayed with authentication	0	0	2	6	8
9. Irrelevant message	0	0	0	3	3
Total number of failures	0	7	30	19	56

nal tests, the database never failed, so those atomic sections were not covered. This can be considered to be a question of controllability because it is difficult to force a database failure through testing. The specific atomic sections from Table 1 that were not covered are $login.HEAD$ (the second occurrence), $login.p_3$, $logout.p_2$, $logout.p_3$ and $logout.HEAD$.

2. Three atomic sections, $login.p_4$, $login.BAR$, and $login.p_5$ from Table 1, are displayed when the user uses an invalid userid or password. All of the original tests used correct user ids and passwords.

3. Two atomic sections, $login.HEAD$ (the first occurrence) and $login.p_2$, are displayed when the user does not completely fill out the login form, leaving missing values.

4. One atomic section, $BAR.p_2$, displays the link for the STIS administrator to add new user accounts. We did not test STIS using the administrator account, so this atomic section was not covered.

5. One atomic section, $categories.p_8$, is displayed when there is no category in the database. During our testing, the database was already initialized with categories, so this AtS was not covered.

6. One atomic section, $record\_edit.p_3$, displays an error message when there is something wrong when editing a record. No inputs were selected to enforce this message, so this AtS was not covered.

Based on the above analysis, we generated seven additional test cases. Data base failure was simulated for two tests by closing the connection before running tests. A third test case was created that uses an invalid user id and password. One test case was generated by leaving the password field blank. A fifth was created to log in to the administrator's account.

A sixth was created by deleting all categories. A final test changed the name of a record to a name that already existed in the database.

The sixth test found an additional failure, a database integrity problem. Removing the cateogry names cause the remaining records to have null categories. That is, the software did not delete records in a category or change the records' categories to "none" when category names are deleted. These additional seven tests gave 100% ATS coverage, and found one more failure.

5.4 Analysis and Discussion

One point to note is that the number of tests in Table 2 is slightly overstated. The experimental process of this paper resulted in 342 test requirements and 156 tests. If applied in actual practice, a test engineer would directly create the invalid path-2 tests and skip the prime path and invalid path-1 tests. The test and results listed under the prime path and invalid path-1 columns would be invalid path-2 tests that did not complete. If infeasible tests could be detected automatically, which is possible, only 74 tests would be needed for STIS.

The fact that the prime path tests did not find any failures is particularly noteworthy, because those are the most "traditional" tests, and prime path coverage is a very strong graph-based criterion (it subsumes node and edge coverage, among others). Other research papers have suggested testing static transitions, but this is the first research that tests operational transitions. All of the failures in this study were found by using operational transitions and invalid form data, testing for web applications that is new to this paper.

The ATS coverage of the 156 WAG tests is worthy of detailed consideration. For most practical use of test criteria, 87% coverage is considered reasonable and probably sufficient. Many of the 13 atomic sections that were not covered are particularly difficult to formulate tests for, particularly when generating tests automatically. Incompletely entering forms

is already a known technique [9, 25], and one that can easily be incorporated into ATS testing. Simulating a database failure is a testing step that should be fairly obvious to a human tester, but somewhat complicated to achieve with purely automated testing. It is also fairly obvious to generate invalid logins and to enter new records incorrectly, but these require rather specific inputs that are more likely to be generated by a human tester than an automated tool. The "no category" failure is particularly devious. Putting the system in a state where there are no categories, yet records still exist for categories that used to be present, is difficult to do automatically. Yet, this test revealed a serious problem in the software. To summarize, the automated testing achieved good coverage and found lots of failures, yet not quite full coverage and not quite all failures. Whether it is worth the extra manual analysis and test generation is not just an engineering decision, economics and marketing factors must also be considered.

Because the Web is a relatively new way to deploy software, it is instructive to look closely at some of the new types of failures that occur. An example of an operational transition error is that of editing a deleted data record in STIS. If the user enters a record R, then deletes R, it should be removed from the database. In fact, it is correctly removed. However, an operational transition (`back` button) can be used to return to a screen where R was shown! If the user then tries to edit R, STIS creates a page with the record label but an erroneous message in the field ("`User not found`"). This exact error was also found in a commercial Web application deployed at the authors' university for faculty use. This behavior is only possible because the client caches data locally, and then redraws it when operational transitions are used. Non-web applications could not exhibit this type of behavior.

Another example is when the software developers assume that the users will only access Web applications through existing links. However, clever (or even careless) users can access a Web application by entering URLs directly in the browser's URL window or by bookmarking

and reusing URLs. Sometimes this can be used to bypass security to directly access parts of the Web application that should be protected. Sometimes this can be used to discover files that the developers assume are hidden. For example, the second author was putting quizzes in a directory before class, and found that some students were finding the quizzes early by guessing the file names. Also, temporary or "debug" Web components are occasionally stored in a directory with the rest of a Web application. If a user discovers and uses these components, either accidentally or intentionally, the application may be put into an invalid state.

5.5 Limitations and Threats to Validity

The STIS case study, though informative, has some limitations, including two threats to external validity. STIS is only one application and it is not certain that results on STIS will apply to other Web applications. Concern about this threat should be ameliorated somewhat by the anectodal examples of failures in commercial Web applications mentioned in the paper. Second, STIS was built by an inexperienced student and all the failures found were naturally occurring. It is possible that a more experienced developer would not make so many mistakes of this type (although we have found several faults of this type in deployed programs, including the NSF's FastLane).

Another issue is with input values. Input values to the HTML forms were generated by hand, and composed mostly of arbitrary strings. This is a cumbersome process. The subject application, STIS, does not actually do much with the input data except pass it through to a database, which avoided some problems in this study. However, a more robust method to generate values is needed. Other researchers are investigating this problem, both in the context of Web applications [4, 9] and for general GUI applications [22].

Finally, the test process in this paper used a mix of automation and manual work. With proper tool development, nearly all of the steps can be automated. The most glaring

exception, of course, is the final step to ensure 100% AtS coverage. This requires careful semantic analysis of the uncovered atomic sections and the software, and further work to create the tests.

6 Related Work

Web applications tend to be based on gathering, processing, and transmitting data among heterogeneous hardware and software components so data flow approaches may be useful if adapted to Web software [11, 13, 14, 28]. Most Web software is object-oriented in nature, so inter-class [8, 18, 24, 26] and intra-class [1] testing techniques can also help find some faults.

Most research in testing Web applications 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 [16]. 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 or black box testing. This project addressed problem 1 in Section 1.1.

The Web Modeling Language (WebML) [6, 7] allows Web sites to be conceptually described. The focus of WebML is primarily from the user's view and the data modeling. Modeling the dynamic integration aspects of the software as presented here is complementary to the solutions proposed by WebML.

More recent research has looked into testing software as statically determined, but none have addressed the problem of dynamic integration. Kung et al. [19, 21] have developed a model to represent Web sites as a graph, and provide 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 without limited use of the server software. They define *intra-object* testing, where test paths are selected for the variables that have def-use chains within the object, *inter-object* testing, where test paths are selected for variables that

have def-use chains across objects, and *inter-client* testing, where tests are derived from a reachability graph that is related to the data interactions among clients. This research addressed problems 1 and 7 in Section 1.1.

Ricca and Tonella [25] proposed an analysis model and corresponding testing strategies for *static* Web page analysis. As Web technologies have developed, more and more Web applications are being built on dynamic content, and therefore strategies are needed to model these dynamic behaviors, thus this paper goes further by investigating more and different execution paths. Ricca and Tonella addressed problems 1, 2, and 3 in Section 1.1.

Benedikt, Freire and Godefroid [5] presented VeriWeb, a dynamic navigation testing tool for Web applications. VeriWeb explores sequences of links in Web applications by nondeterministically exploring "action sequences", starting from a given URL. Excessively long sequences of links are limited by pruning paths in a derivative form of prime path coverage. VeriWeb creates data for form fields by choosing from a set of name-value pairs that are initialized by the tester. VeriWeb's testing is based on graphs where nodes are Web pages and edges are explicit HTML links, and the size of the graphs is controlled by a pruning process. This research addressed problems 1 and 2 in Section 1.1.

Elbaum, Karre and Rothermel [9] proposed a method to use what they called "user session data" to generate test cases for Web applications. Their use of the term "user session data" was nonstandard for Web application developers. Instead of looking at the data kept in J2EE servlet session, their definition of user session data was input data collected and remembered from previous user sessions. The user data was captured from HTML forms and included name-value pairs. Experimental results from comparing their method with existing methods show that user session data can help produce effective test suites with very little expense. The user session data approach addresses the testing of Web applications whose inputs are entered via forms, problem 3 in Section 1.1.

Lee and Offutt [20] describe a system that generates test cases using a form of muta-

tion analysis. It focuses on validating the reliability of data interactions among Web-based software system components. Specifically, it considers XML based component interactions, since one of the primary vehicles for transmitting data among components is now XML [10]. This approach tests Web software component interactions, whereas our current research is focused on the Web application level. This research addressed problem 7 in Section 1.1.

Yang et al. [27] focused on the so called "three-tier" model, and developed a tool that helps to manage a test process across the three tiers. Their tool supports six subsystems to help track documents, monitor the processes, monitor tests, monitor failures and support test measurement, but does not directly address test generation. Their tool does not help generate tests or provide criteria for which tests should be designed.

Jia and Liu [17] proposes an approach for formally describing tests for Web applications using XML. A prototype tool, WebTest, based on this approach was also developed. Their XML approach could be combined with the test criteria proposed in this paper by expressing the tests in XML.

7 Conclusions and Future Work

This paper has presented three results. First, a new theoretical model that captures dynamic aspects of Web applications was introduced. Second, test criteria based on the model were developed and defined. Third, results from applying the model and tests on a case study were presented. The tests caused a number of failures that were related to transitions that are not modeled by other analysis and testing techniques.

The AtS model is based on identifying atomic elements of dynamically created Web pages that have static structure and dynamic data contents. These elements are combined to create composite Web pages using sequence, selection, aggregation, and regular expressions. This modeling technique solves a significant problem with analyzing Web software applications – that of statically representing the dynamic integration. This in turn allows

analysis techniques such as control flow analysis, data flow analysis, and slicing to be used to support software engineering activities in testing and maintenance.

The AtS model only depends on the properties of HTTP and HTML, thus is independent of software implementation technology. An application to testing Web applications was also introduced. The testing uses the model of the Web application's behavior to define tests as sequences of user interactions. Previous research [19, 21, 25] developed tests that addressed issues similar to the static link prime tests, the other tests are new to this paper and were found to be the most effective at finding failures.

A number of open issues still remain with the model and the test criteria. The problem of finding values to fill forms must be addressed. Solutions from other research efforts can probably be incorporated into the test criteria described in this paper. In our study, we generated the WAG by hand, but relatively straightforward algorithms can identify the various transitions and construct the graph automatically. The tests were also submitted by hand, but this can be automated by use of a tool like httpUnit, which allows web applications to be accessed by bypassing the browser [12, 15].

Problems with session data, multiple users, and concurrency have not explicitly been addressed. The Web transition model can be used to support maintenance and regression testing by helping the developer determine which software components are affected by changes. We expect that adding data flow information to the model and understanding the types of couplings among Web software components will help those problems. We also plan to address ancillary problems such as automatically deriving test cases.

Another issue is that of output validation. This is particularly difficult with Web applications because of the low observability. The simplest form of output validation is simply to check the result that is sent to the client. However, other parts of the output are stored as state on the server, including files, data bases, and in-memory data stores such as session data and beans. Some parts of this state are hard to view and may be used by the same

client in the same session, the same client in another session, or other users. Tracking these kinds of outputs is very difficult and we are not aware of any research that has addressed this problem as yet.

8 Acknowledgments

We would like to thank Paul Ammann for help identifying a key problem in this research, Anneliese Andrews and Roger Alexander for helpful discussions on atomic sections, and Yuan Yang for implementing STIS.

References

[1] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose CA, October 2000. IEEE Computer Society Press.

[2] Paul Ammann and Jeff Offutt. *Coverage Criteria for Software Testing*. In preparation, 2004.

[3] Paul Ammann, Jeff Offutt, and Liu Ling. Touring prime paths. *In preparation*, 2004.

[4] Anneliese Andrews, Jeff Offutt, and Roger Alexander. Testing Web applications. *Software and Systems Modeling*, 2004. To appear.

[5] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic Web sites. In *Proceedings of 11th International World Wide Web Conference (WW W'2002)*, Honolulu, HI, May 2002.

[6] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (WebML): A modeling language for designing Web sites. In *WWW9 Conference*, Amsterdam, Netherlands, May 2000.

[7] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, December 2002. Information available online at: http://webml.elet.polimi.it/webml/book.html.

[8] M. H. Chen and M. H. Kao. Testing object-oriented programs - An integrated approach. In *Proceedings of the Tenth International Symposium on Software Reliability Engineering*, pages 73–83, Boca Raton, FL, November 1999. IEEE Computer Society Press.

[9] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Portland, Oregon, May 2003. IEEE Computer Society Press.

[10] S. Feldman. Electronic marketplaces. *IEEE Internet Computing*, pages 93–95, 2000.

[11] P. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[12] Russell Gold. Httpunit home. online: SourceForge, 2003. http://httpunit.sourceforge.net/, last access February 2004.

[13] M. J. Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[14] M. J. Harrold and M. L. Soffa. Selecting data flow integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[15] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning Publications, August 2002.

[16] Rick Hower. Web site test tools and site management tools, 2002. www.softwareqatest.com/qatweb1.html, last access February 2004.

[17] Xiaoping Jia and Hongming Liu. Rigorous and automatic testing of Web applications. In *6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 280–285, Cambridge, MA, November 2002.

[18] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.

[19] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *Proc. of IEEE 24th Annual International Computer Software and Application Conference (COMPSAC 2000)*, Taipei, Taiwan, October 2000.

[20] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[21] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structure testing of Web applications. In *Proceedings of the 11th Annual International Symposium on Software Reliability Engineering*, pages 84–96, San Jose CA, October 2000.

[22] A. M. Memon, M. L. Soffa, and M. E. Pollack. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[23] Jeff Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.

[24] A. Parrish and S. H. Zweben. Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering*, 17(6):565–581, June 1991.

[25] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 25–34, Toronto, CA, May 2001.

[26] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *IEEE International Conference on Software Maintenance*, Montreal, Quebec, Canada, September 1993.

[27] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, and William C. Chu. An object-oriented architecture supporting Web application testing. In *Proceedings of IEEE 23th Annual International Computer Software and Application Conference (COMPSAC 2000)*, Phoenix, Arizona, October 2000.

[28] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.