Softw Syst Model
DOI 10.1007/s10270-009-0125-4

REGULAR PAPER

Modeling presentation layers of web applications for testing

Jeff Offutt · Ye Wu

Received: 1 August 2008 / Revised: 9 June 2009 / Accepted: 22 July 2009

Abstract Web software applications have become complex, sophisticated programs that are based on novel computing technologies. Their most essential characteristic is that they represent a different kind of software **deployment**—most of the software is never delivered to customers' computers, but remains on servers, allowing customers to run the software across the web. Although powerful, this deployment model brings 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 three aspects of web applications that are unique to this type of deployment: (1) an extremely loose form of coupling that features *distributed integration*, (2) the ability that users have to directly change the *potential flow* of execution, and (3) the dynamic creation of HTML forms. Taken together, these aspects allow the **potential** control flow to vary with each execution, thus the possible control flows cannot be determined statically, prohibiting several standard analysis techniques that are fundamental to many software engineering activities. This paper presents a new way to model web applications, based on software couplings that are new to web applications, dynamic flow of control, distributed integration, and partial dynamic web application development. This model is based on the notion of **atomic sections**, which allow analysis tools to build the analog of a control flow graph for web applications. The atomic section model has numerous applications in web applications; this paper applies the model to the problem of testing web applications.

Keywords Web applications · Web modeling · Test criteria

Communicated by Prof. James Bieman.

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

J. Offutt (✉)
Software Engineering, Volgenau School of Information Technology and Engineering, George Mason University, Fairfax, VA, 22030, USA
e-mail: offutt@gmu.edu

Y. Wu
Science Applications International Corporation, Bethesda, MD, USA
e-mail: ye.wu@saic.com

1 Introduction

Web applications are programs that are deployed in a novel way, allowing them to be accessed remotely across the Internet. 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 numerous back-end software components that are integrated in novel and interesting ways. Modeling, analyzing, evaluating, maintaining and testing these applications present many new challenges for software developers and researchers.

A previous paper explained why large web applications need to be highly reliable, very secure, continually maintainable, and constantly available [32]. Real world enterprise systems are currently implemented almost exclusively as web-based software and used extensively in day-to-day operations by consumers and businesses world wide. comScore, an agency that measures the digital world, reported that 772 million *unique* people were online worldwide in May 2007 [13]. A comScore study conducted in January 2008 reported that consumers spent $29.2 billion dollars online in the 2007 holiday season, up 19% from the previous year [13].

Published online: 09 August 2009

Web-based systems have many problems, and because real world enterprise systems have so many users, the problems can quickly affect millions of users and cost millions of dollars. Blumenstyk [8] reported that web application failures lead to huge losses in businesses; $150,000 per hour in media companies, $2.4 million per hour in credit card sales, and $6.5 million per hour in the financial services market. Pertet and Narasimhan [38] found that most failures did not occur during initial deployment, but during maintenance.

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 USD. More recently, several university web sites (including the authors' own) have been hacked into and thousands of social security numbers were stolen. In another instance at another university, hundreds of acceptance letters were fraudulently emailed to applicants who had previously 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 HTML content 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 Java Servlet, Active Server Page, or PHP component. 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, operating system, and even the 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 (HTML).

Web software is built with many different technologies, including scripting languages that run within the client's browsers (JavaScript, VBScript, Ajax), 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 sequential databases (SQL). These diverse technologies (and others) cooperate to implement web applications, resulting in a heterogeneous, multi-platform software environment where the software components exhibit new forms of coupling. These couplings are very loose, and feature message passing, communication across networks, and distributed integration of software components.

Whereas the high quality requirements of web software, multi-platform issues, concurrency, and issues arising from heterogeneous 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. As described below, some traditional models and analysis techniques are not sufficient with web applications. This paper presents a modeling and analysis technique, the *atomic section model* (ASM), that addresses the problem of distributed integration of web software applications that include dynamically created components. The issue is discussed and the problem is presented, then a solution using new modeling techniques called atomic sections and component expressions is developed. The ASM offers a fundamental way to model dynamic aspects of web software in a technologically independent way. The ASM can support a variety of different software engineering activities by allowing versions of traditional analysis techniques such as control flow, data flow and slicing to be applied. The paper also presents details of applying the ASM to integration testing, including proposed test criteria and results from a demonstration study.

2 Novel aspects of web applications

Most web applications use HTML to create user interfaces to the software. As such, this subsection introduces HTML and describes how software components interact to create web applications. Although some of this information may seem basic for readers who are experienced web application developers, it is important to clarify certain terms and concepts before proceeding.

2.1 Overview of HTML and web applications

An HTML document is accessed by a web browser, and then the content is *rendered* on the client's screen using the formatting instructions in the HTML. These instructions are in the form of tags that can have attributes. *Tags* are surrounded by angle brackets and usually come in *begin* and *end* pairs. For example, the bold tag, `<B>`**bold**`</B>`, tells the browser to render the enclosing text in bold face, and the 'A' tag, `<A>`**SoSyM**`</A>`, tells the browser the text is a hypertext link. Tags can contain attributes that parameterize the action.

For example, the 'A' tag needs to tell the browser where the link points to using the "href" attribute: <A href="http://www.sosym.org/">**SoSyM**</A>.

User interaction is accomplished using links (the 'A' tag) and form input. A form is contained within a form tag, <FORM> ··· </FORM>, and can contain various input elements, including text input boxes, checkboxes, radio boxes, selection boxes, and buttons. Because of the user interaction and because HTML "pages" are created dynamically, we simply call these *screens*. When a user submits a form on a screen, the browser generates a request to the server that contains all data the user entered into the form. The form tag uses the "action" attribute to specify the URL of the software component that will handle the request. The form tag can also use the "method" attribute to specify what type of HTTP request is generated. The two most common are "get" and "post," although others are available. A "get" request appends form parameters to the URL in the format "?name1=val1&name2=val2& …". The length is limited to 1024 bytes. A "post" request encodes form parameters into the body of the message and its length is unlimited.

HTML forms can also contain hidden form fields. A hidden form field is an input element that has the attribute type hidden (for example, <INPUT type="hidden" Name="Retry" Value="0">). 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 across multiple requests from the same user. Although called hidden, it is important to realize that the HTML source is stored on the client's computer and the hidden form field can be seen, used, and even modified.

A key element of web applications is the design of the **interactions** among the software components. Two key technologies that the J2EE platform uses are Java servlets and Java Server Pages (JSP). A *Java servlet* is a Java class that runs as a lightweight thread in a plug-in called a *servlet container*, and accepts HTTP requests from the web and creates responses. A *Java Server Page (JSP)* uses HTML templates that can include Java statements. JSPs are translated into Java servlet classes, then compiled and run as servlets.

A typical web application works as depicted in Fig. 1. A client first retrieves information from a server by requesting a particular URL (1), the server runs a software component such as a servlet (2), which generates HTML (3), which is then returned to the client as an HTML document (4). The client then sends a request with data to a server (5), which runs another software component (such as a Java server page, 6), which in turn accesses a database (7), and finally returns formatted data to the client in the form of HTML (8, 9, 10). Although not all interactions are through HTML pages, this is a useful model to start with and this paper assumes that clients and servers interact through HTML pages. The scenario shown can also be applied to other forms of interactions, such as software that uses XML.

This paper focuses on two interesting and new design challenges when deploying software across the web. One is remembering who the user is across multiple requests to the same server, and another is keeping data persistent across those multiple requests. In the example in Fig. 1, the user accesses different software components (the servlet and JSP) at different times. It is sometimes important for the server to recognize that different requests not only come from the same user, but are part of the same program execution. Recognizing a single program execution is trivial with most programs, because the entire execution belongs to a single process, or at most, multiple processes that are tightly connected. However, it is not obvious with web applications because HTTP is a **stateless** protocol. The different software components in the application execute independently, usually as separate execution threads. When a software component gets a request, it is syntactically independent of other requests and there is no obvious built-in mechanism to decide whether it is related to other requests.

The problem is solved by defining a *session* to be a sequence of related HTTP requests from the same user. First, "cookies" are deposited onto the client's computer. A *cookie* is a string that a web server creates and sends to a client. The client then sends the cookie to the server when generating requests. The server uses the cookie to decide if this is the same user across different requests, thus keeping track of the user session.

The other problem that stems from the stateless nature of the web is that of keeping data persistent. Web servers solve this problem by using new forms of data scoping. Web software components do not explicitly share the same memory space, so cannot share global variables or objects. However, the components run as threads inside the same process, which is controlled by the web server. Web servers create a special object, called the "session object," to store persistent data. Depending on their scope, these data can be accessed by (1) the same software component on different requests, (2) different software components on the same request, (3) all requests from the same session (as defined by the cookie), or (4) all applications running within the same web server.[1]

Web applications commonly divide the software components into several layers [3,44]. Although there are numerous design possibilities, a common division is based on the data. The *presentation layer* is responsible for displaying data to the user and interacting with the user. The *data content layer*

[1] Please note that this terminology is from the J2EE platform and differs slightly within .NET, although the concepts are the same. Also, certain details about the web server (including the servlet container and the session contexts) have been omitted for brevity. They are not necessary for the model described in this paper.

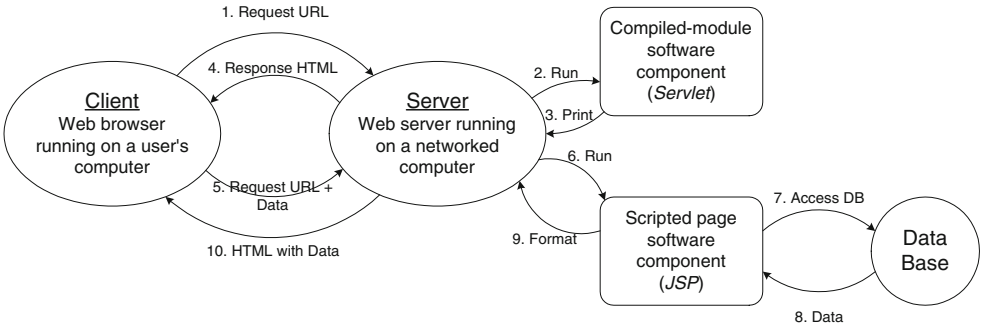

Fig. 1 Typical execution flow among web application components

(or *computation layer*) is responsible for most computation and data access. The *data representation layer* stores the data in-memory to be available to the data content layer. The *data storage layer* is responsible for permanent storage of data in files or databases, or for accessing non-local data.

2.2 Couplings among web application components

Most software engineering models and analysis techniques implicitly assume that programs are linked into one large executable. Thus, static analysis can be used to create control flow graphs, which define, in an abstract way, all possible executions of the program. However, web application software components are linked dynamically, using new types of couplings that are extensions of message passing, remote procedure calling, and sharing data stores. Because of the **independence** and **distributed integration** of web software components, the traditional model of a control flow graph **cannot** be created statically. Indeed, parts of the program structure of web applications can be created **dynamically**, thus the traditional control flow graph, if created, would be different for each user session.

Four interesting and potentially troubling issues are created by the unique structural aspects of web software.

1. Distributed integration and extremely loose coupling
2. The dynamic creation of HTML forms
3. The ability that users have to directly control the potential flow of execution
4. Web applications can dynamically integrate new software components during execution

This research addresses the first three of these issues, but not the fourth. The essential key element behind all of these issues is the way software components are connected. Traditional programs use logical decisions inside methods and calls among methods. These vary in web applications, and web applications use new types of connections, some of which are variations of message passing and remote procedure calling. The rest of this section analyzes how software components are connected, or *coupled*, and finishes with an enumeration of the most common types of couplings in web applications. These new couplings are used to create a new type of execution model in the rest of the paper.

The model presented in this paper is based largely on the way web software components are coupled. 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. The term *method* is used 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 logic 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 changes to the **structure** of the data, which in turn requires changes in the way B uses data items that are defined in A. Loose coupling is usually associated with data abstraction and information hiding.

A program exhibits *extremely loose coupling* 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 extremely loose coupling is used 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 programs will not interact correctly. Applications can even require extremely loose coupling, for example by using XML messages.

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 form or program, an ability that allows it to dynamically integrate with new software components. This could be used to temporarily store arbitrary data; for example such components are often used in Internet-based file sharing applications. This kind of distributed integration, usually combined 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.

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! A simple 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 ScholarOne, Inc's `manuscriptcentral.com`, 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. Web applications do not have this same property. When executing web applications, users can break the normal control flow without alerting the "program controller." The model of program controller that is still taught in basic programming and operating system classes does not apply to web applications because the flow of control is distributed across the client and one or more servers. Users can modify the expected control flow on the client by pressing the back or refresh buttons in the browser or by directly modifying the URL in the browser. These interactions introduce unanticipated 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, causing subsequent operations to malfunction.

This analysis leads to the following types of connections that web application software components use. This list cannot be complete, because new frameworks are regularly being developed with additional types of connections. This list is based on the J2EE framework, which is quite common and is the basis for many other frameworks.

1. *Static links* (HTML → HTML): Most of the early literature on web modeling and testing focused on link validation. Note that this does not address any software or dynamic issues. Static links use the `<A>` tag.
2. *Dynamic* `<A>` *links* (HTML → software): HTML links make a request to software components to execute some process, using an `<A>` tag. No form data is sent in the request, and the type of the HTTP request is always `get`. (This is a simple form of remote procedure calling.)
3. *Dynamic form links* (HTML → software): HTML forms send data to software components that process the data, using a `<FORM>` tag. The type of HTTP request can be either `get` or `post`, as specified in the `<method>` attribute of the `<FORM>` tag. The data that is submitted via forms impact the back-end processing, which is an important issue for dynamic techniques such as testing. (This is a form of message passing.)
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 analysis.
5. *State dependent, dynamically created GUIs* (software + state → HTML): HTML documents whose contents and forms are determined not just by inputs, but by part of the state on the server, such as the date or time, the user, database contents, or session information. The HTML documents can contain Javascript, which are parts of the web application program that execute on the client. They can also contain links, which determine the execution of the program. The Javascript and links may differ at different times, which is how different users see different programs.
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, forward, and refresh buttons, browser history, local caching, and URL rewriting. This type of transition is new to web software, very difficult to anticipate, and often leads to problems, as discussed later in the paper.
7. *Local software connections*: Connections among back-end software components on the web server, such as method calls.

8. *Off-site software connections*: Some web applications will access software components that are available at a remote site. They can be accessed by sending calls or messages to software on another server, using HTTP or some other network protocol. This type of connection, while powerful, is difficult to analyze because little is known about the off-site software. (This can use message passing or remote procedure calling.)
9. *Dynamic connections*: Both the J2EE platform and .NET allow new software 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 evaluate because the components are not available until **after** the software is deployed.

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 cannot accurately model web applications. 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. This paper introduces a new modeling technique that is able to describe structure, control and data flow of dynamic web applications.

3 Modeling web applications

This paper introduces a new model, the *atomic section model*, based on elemental pieces of software components. The ASM represents all possible user interface screens of web applications, similarly to how control flow graphs model all possible execution sequences for traditional programs. The ASM begins by describing elements of dynamically created web pages. The elements are combined to form regular expressions. Applications for the ASM are presented in Sect. 4.

The ASM is designed at two levels of abstraction, roughly corresponding to the unit and system levels for traditional software. First, a model is developed for individual software components, which are identified in individual HTTP requests to the server. This model is called the *component interaction model (CIM)*. Second, the results are combined to form a model of the entire web application, called an *application transition graph (ATG)*.

The ASM has several limitations. First, it assumes that all software that creates responses to the users is available in source form. This may not be true when external software components are used, but is not as serious a limitation as it might initially appear. A key point is that the analysis is based on the presentation layer (as described in Sect. 2) and access to other layers is usually not needed. Most reused components are in the data content or representation layers, and web application developers either write the presentation layer software in-house or modify existing software components, so the components of the application that create responses are usually available in source form. Specifically, all the decisions about the presentation must be available in source form, and all the data used to create links must be available. If this data is not available, the model will still be useful, but not complete. Second, this model does not address asynchronous client/server interactions such as implemented through Ajax.

A third limitation is based on uncomputable problems in the analysis. Part of the analysis (Sect. 3.3) depends on finding transitions among software components. Unfortunately, transitions can be based on variables, which could be read from configuration files or other external inputs. Thus finding all transitions is undecidable.

Three other restrictions are accepted to allow the ASM to be created statically. Fourth, problems with dynamic integration are not addressed. Web applications can integrate with new software components during execution; because this is a static analysis technique, this information cannot be available. Fifth, program state that is stored on the server in objects such as the session object is not modeled, and sixth, concurrency is not modeled. Concurrency is most important when multiple users are accessing the same web application at the same time, and potentially interacting through shared memory.

The remainder of this section describes the ASM. First, the basic building block of an atomic section is introduced, then atomic sections are combined into component expressions, which represent all HTML screens that can be created from one software component. Next, we analyze types of transitions that can occur among web software components and describe a method for representing those connections. These are then used to build a model of how server-side components interact, which finally is developed into a graph that models the user interaction portion of web applications.

3.1 Atomic sections

The atomic section was first introduced in our 2002 technical report [48]; the current paper includes more rules for combining atomic section, more types of transitions, and an expanded model for the entire web application. An *atomic section* (ATS) is a section of HTML (possibly including scripting language routines 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

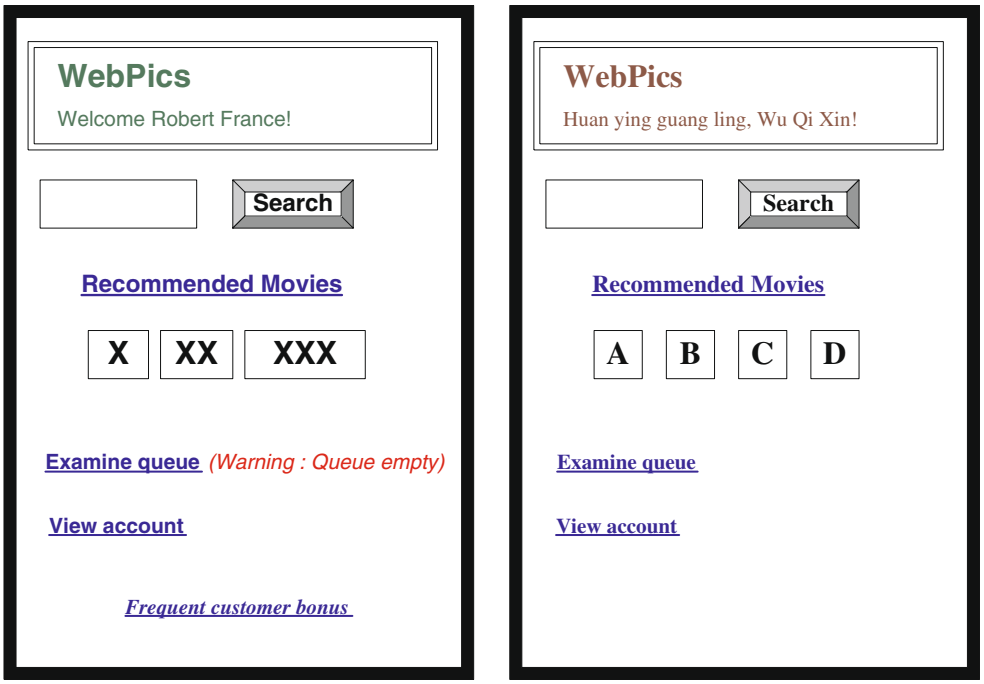

Fig. 2 Stylized web pages to illustrate atomic sections in dynamically created web pages

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 generates 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.

The definition of an ATS assumes that the user does not interrupt the transmission of data by using the "stop loading" button on the browser. Note that the stop loading button does not interrupt the execution of the program component on the server, but might cause part of the program's user interface to be lost.

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 (note that the atomic sections depend on the program, not the HTML, but we can assume the superficial analysis is correct for this example). The first atomic section consists of the header welcome message and the search box, which have been personalized to the user. The second consists of the list of recommended movies, which depends on customer data 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 available only in the screen on the left.

Now we turn to actual program source. Figure 3 shows a Java servlet from a software component P that produces six atomic sections ($p_1, p_2, \ldots p_6$). Note that only output statements are annotated as atomic sections. To be precise, only the outputs of those statements are atomic sections, not the Java statements, but we show the atomic sections with the surrounding statements 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 programs, but have many distinct differences. 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 Fig. 4 was constructed to illustrate this difference. It contains nine basic blocks (annotated on the right) but only three atomic sections (annotated on the left). The atomic sections reflect the relationship of this component with other components, while the first six basic blocks are relevant to the internal processing of this module. Thus, each atomic section corresponds to exactly one basic block, but some basic blocks will not correspond to any atomic sections.

The ASM can be applied to other technologies besides Java servlets. Figure 5 shows atomic sections in a Java Server Page (JSP). The complete component expression for the JSP example is $P \rightarrow p_1 \cdot (p_2 \cdot (p_3 \mid p_4)) \cdot p_5)^* \cdot p_6$.

Fig. 3 Servlet atomic section example

```
                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();
```

Fig. 4 Atomic sections vs. basic blocks

```
                String manufacture = req.getParameter ("manufacture");
                String productName = req.getParameter ("productname");
                String minPrice = req.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='selectProduct'>")                              BB7
                while (rs.next())
p2 =               out.println ("<INPUT type=checkbox name=prod>" +
                                rs.getString ("prod"));                          BB8
p3 =            out.println ("<INPUT type=submit><INPUT type=cancel");
                out.println ("</BODY></HTML>");                                  BB9
                out.close();
```

3.2 Component expressions

Server software components combine atomic sections together to form web pages that conform to rules, called component expressions. The combination is usually done dynamically and is affected by the control flow of the software component. Different users will get different complete HTML pages, and thus different user interfaces. In effect, they have access to different programs. Atomic sections can be combined in four ways: *sequence*, *selection*, *iteration*, and *aggregation.* Formally, p is a *component expression* of a server component c 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 component expressions, and p is composed of p_1 followed by p_2.

Fig. 5 JSP atomic section example

p1	=	`<html>`
		`<head><title>A JSP File Viewer</title></head>`
		`<body>`
		`<%@ page import="fileViewerBean" %>`
		`<%@ page import="java.util.Iterator" %>`
		`<jsp:useBean id="fv" scope="page" class="fileViewerBean"/>`
		`<hr>`
		`<%= fv.getDirectory() %>`
		`<table>`

		`<%`
		`fv.refreshList();`
		`while (fv.nextFile()) {`
		`%>`

p2	=	`<tr>`
		`<td><%= fv.getFileName() %></td>`
		`<td>`

		`<%`
		`if (!fv.getFileType()) {`
		`%>`

p3	=	`<%= fv.getFileSize() %>`
		`<% } else %>`
p4	=	`{ }`

p5	=	`</td>`
		`<td><%= fv.getFileTimeStamp() %></td>`
		`</tr>`

		`<% } %>`

p6	=	`</table>`
		`</body>`
		`</html>`

3. *Selection* ($p \rightarrow p_1 \mid p_2$): p_1 and p_2 are component expressions, and the server selects either p_1 or p_2, but not both.
4. *Iteration* ($p \rightarrow p_1^*$): p_1 is a component expression, and the server selects an arbitrary length sequence of p_1.
5. *Aggregation* ($p \rightarrow p_1\{p_2\}$): p_1 and p_2 are component expressions, p_2 is included as part of p_1 when p_1 is transmitted to the client. Examples are function calls in p_1 and file inclusion commands.

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

Atomic sections and component expressions define how HTML pages can be dynamically generated. Precisely, the component expression for a software component is a regular expression that represents **all** possible complete HTML pages that can be generated by the component. The complete component expression for the example in Fig. 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 unknown iterations 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 software components. To execute a complete run of a web application, the client and software components link the dynamically generated HTML pieces together. Web applications 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.

3.3 Modeling transitions among web software components

A challenging part of modeling web application execution is that execution flow goes through the client (and user) following the stateless nature of HTTP. Web applications combine components with HTML and links. 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 the software. The simplest example is when a user presses the back button in the browser, causing the application to return to a previous screen. This introduces a possibly unanticipated execution step into the control flow, without

notifying either the server or client software 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 ASM defines *dynamic interactions*.

The interactions among different software components can be classified into types of *transitions*. Different development frameworks offer different types of transitions and new types of transitions continue to be introduced. This paper presents five types that are derived from the J2EE framework. J2EE was one of the first, one of the most common, and many current frameworks are built on top of J2EE. The list below can be extended to incorporate other frameworks when needed. In the following, p and q are component expressions of atomic sections and c is a software component that generates HTML or c is an HTML file. Different arrows are used to distinguish the types of transitions.

1. *Simple Link Transition* ($p \longrightarrow c$): Invoking an <A> link in p causes a transition from the client to a component c on the server. If p has more than one <A> link and can thus invoke one of several static or dynamic pages, $c_1, c_2, \ldots, c_k$, then the destination is represented as $c_1 \mid c_2 \mid \ldots \mid c_k$.
2. *Form Link Transition* ($p \gg c$): Invoking a link in a <FORM> element in p causes a transition from the client to a component c on the server. Just as with simple link transitions, if p can invoke one of several static or dynamic pages, $c_1, c_2, \ldots, c_k$, then the destination is represented as $c_1 \mid c_2 \mid \ldots \mid c_k$.
3. *Component Expression Transition* ($c \multimap p$): The execution of software component c causes p to be produced and returned to the client. The software component c can normally produce several component expressions, which can be represented as $c \multimap p_1 \mid p_2 \mid \ldots \mid p_k$.
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 (including adding or modifying parameters, often called *URL rewriting*). 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 arrows are annotated with "back" to represent use of the back button, "forward" for the forward button, and "reload" for the refresh button.
5. *Redirect Transition* ($p \dashrightarrow q$): A redirect transition is a server-side software transition that is **not** under the control of the tester. Java servlets offer the "`res.sendRedirect`" command, which tells the client to re-generate the same request to a different URL. This redirection is invisible to the user. For example, if a user successfully logs in to an application, the login component may automatically redirect to another component.

It is sometimes necessary to annotate transitions with two types of information that are needed in the inter-component level of the model. The first indicates what type of HTTP request was used in the transition. These are most commonly `get` and `post`, as described in Sect. 2, although others are available. The second annotation is the data that is being transmitted, usually in the form of parameters. When written textually, the request type and parameters are written after the destination component in square brackets: $p \longrightarrow\!\!\gg c\ [type, (param1, param2, \ldots)]$. When the transitions are described, the parameter variable names are given, when actual transitions are written, parameter names and values are given as pairs ($param1 = value1$). If the parameter name is clear from context, just the value is given for brevity. When drawn in graph form, the information is placed onto the appropriate edge.

Calls to non-web specific software components in the data content layer (for example, normal Java classes) are not modeled separately. Likewise, include files (files that are explicitly included into another before compilation, such as "`include`" in JSP and "`#include`" in C) are considered to be part of the including file. That is, they are included as aggregation.

3.4 The component interaction model

Each web software component is modeled as a quadruple *Component Interaction Model*, $CIM = \langle S, A, CE, T \rangle$, where S is the start page, A is the set of atomic sections, CE is the component expression, and T is a set of transitions. These sets are fixed and static, based on the source of the presentation layer software. The start page may be the software component itself (if it is designed and available to be called directly from a browser with a URL) or a static HTML page that calls it. It is assumed that each software component has a unique start page.

The example in Fig. 6 is an HTML page that uses the Java servlet in Fig. 7 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 sends the email to the instructor (details about `sendMail` are omitted for brevity). The HTML file uses a "hidden" form field to keep track of how many login attempts the user has made (<INPUT

Fig. 6 HTML login page

```
<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="Password" Size="20">
       <INPUT Type="HIDDEN"   Name="Retry"    Value="0">
       <INPUT Type="SUBMIT"   Name="SUBMIT"   Value="SUBMIT">
       <INPUT Type="RESET"                    Value="RESET">
   </FORM>
   <A href="./syllabus.html">Class home page</A>
 </BODY>
</HTML>
```

Fig. 7 Atomic sections of servlet `gradeServlet`

	`ID = request.getParameter ("Id");`
	`passWord = request.getParameter ("Password");`
	`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))` `{`
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 if (retry < 3)`
	`{`

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

	`}`
	`else // (retry >= 3)`
	`{`

p5 =	`out.println`
	`("Wrong ID or wrong password, retry limit reached.");`
	`out.println`
	`("If you are in this class,`
	`please see your professor for help.");`

	`}`
p6 =	`out.println ("</BODY></HTML>");`
	`out.close();`

`Type="HIDDEN" Name="Retry" Value="0">` in Fig. 6).

The atomic sections for `gradeServlet` are marked in Fig. 7. `gradeServlet` uses three methods, `Validate()`, `CourseName()` and `CourseGrade()`. These methods are part of the data content layer of the web application, not the presentation layer, and do not directly affect the response page that `gradeServlet` produces. They generate data content, but do not effect the atomic sections, thus they are not necessary for the modeling. The start page, atomic sections,

component expressions, and transitions for `gradeServlet` are:

S = login.html

$A = \{p_1, p_2, p_3, p_4, p_5, p_6\}$

CE = gradeServlet = $p_1 \cdot ((p_2 \cdot p_3^*) \mid p_4 \mid p_5) \cdot p_6$

$T = \{login.html \ggg$ gradeServlet [*get*, (*Id*, *Password*, *Retry*)],
gradeServlet.$p_4 \longrightarrow$ sendMail [*get*, ()],
gradeServlet.$p_4 \ggg$ gradeServlet [*get*, (*Retry*)]$\}$

3.5 The web application transition graph

The web software CIM models a single software component by providing an abstract description of all the HTML pages that can be created and sent to the user. This can be thought of as being at the "unit level" for web software. Although the CIM represents very different information from the traditional control flow graph for methods, it models software at a similar level of abstraction.

A higher level of abstraction is needed to model the entire web application. The web *Application Transition Graph (ATG)* combines component interaction models to model an entire application. In an ATG, nodes are web software components and edges are links and other types of transitions among the nodes. Formally, a web application is modeled as a quadruple $ATG = \langle \Gamma, \Theta, \Sigma, \alpha \rangle$, where Γ is a finite set of web software components, Θ is a set of transitions among the web software components in Γ, Σ is the set of variables that define the state of the web application, and α is the set of start pages.

The first step in creating an ATG is to determine the set of web software components that make up the web application, $\Gamma = \{CIM_1, CIM_2, \ldots, CIM_n\}$. The web software components can be determined statically by hand, and in fact, are usually known. They can also be determined automatically by transitively following the transitions among web software components, starting from the start page. Unfortunately, this process may be unbounded. If one page contains a static link to an unrelated website, a transitive search of the transitions could wind up finding most of the world wide web! It may be possible to develop approximation techniques to bound this process, for now, however, we assume the components in the web application are given by the developer and leave the automatic determination as separate work (this particular problem has been addressed by Halfond and Orso [19].

The set of transitions is derived from the web software components, $\Theta = T_1 \cup T_2 \cup \cdots \cup T_n$, where T_i is the set of transitions in component CIM_i. The set of start pages α is created similarly, $\alpha = \{S_1, S_2, \ldots, S_n\}$, where S_i is the start page in component CIM_i. It is quite possible that not all component-level start pages should be considered to be web application start pages, and the developer may choose to omit some start pages from α.

Σ does not define the entire state of the web application, but just the state of the presentation layer. As such, the variables involved in Σ are derived from the software components and web pages in Γ. The type of HTTP request is always `get` in static links. It is either explicitly given in form links through the `method` attribute of the `<FORM>` tag, or defaults to `get` if not given. The parameters are explicitly defined in the HTML `<FORM>` elements and data stored in the session object are identified by the names of the objects referenced in software components. In our demonstration study, we have created Σ by hand, but the information is completely syntactic in nature and a simple tool could be built to discover this information by analyzing the HTML and program source.

As defined in Sect. 3.3, transitions are categorized as simple link transitions, form link transitions, component expression transitions, operational transitions, and redirect transitions. This model explicitly ignores method calls on the server, assuming they are tested by traditional modeling and testing techniques. Below is the ATG for the `gradeServlet` example:

Γ = {login.html, gradeServlet, sendMail, syllabus.html}

Θ = {login.html $\longrightarrow$ syllabus.html [get, ()], login.html $\ggg$ gradeServlet [*get*, (*Id*, *Password*, *Retry*)], gradeServlet.$p_4 \longrightarrow$ sendMail [*get*, ()], gradeServlet.$p_4 \ggg$ gradeServlet [*get*, (*Id*, *Password*, *Retry*)]}

Σ = {Id, Password, Retry}

α = {login.html}

Note that `login.html` is the only start component. Also note the simple transition to the class home page, `syllabus.html`. Although there may be numerous links on that page to other pages, they are **not** included as part of this web application.

Figure 8 shows the ATG for the `gradeServlet` example. The ATG has four components, the login HTML page, `gradeServlet`, `sendMail`, and `syllabus.html`. The `gradeServlet` component is drawn with its atomic sections, arranged in a graph representation of the component expression, to illustrate the two levels of this model. The dotted edges inside the `gradeServlet` component represent a graphic form of the component expression in `gradeServlet`.

Given the restrictions in Sect. 3, creating an automated tool to construct atomic sections, composite expressions, and the web ATG is fairly straightforward and highly scalable.

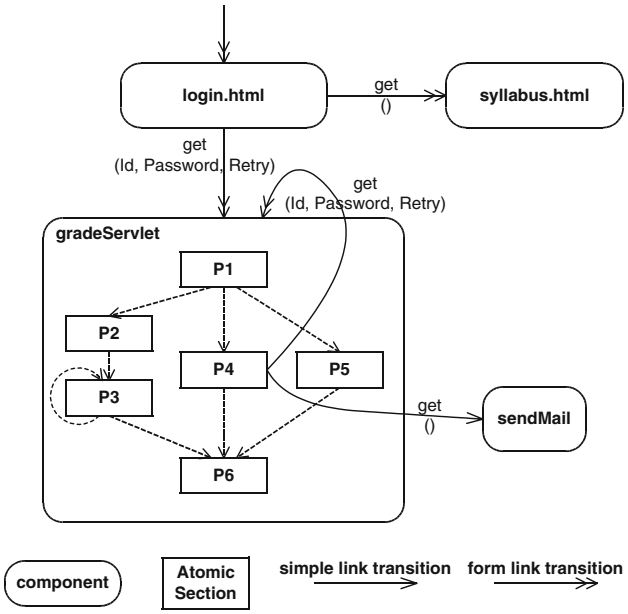

Fig. 8 Web application transition graph for `gradeServlet`

We have built a simple tool to construct the atomic sections and composite expressions for individual software components; it is almost the same as extracting control flow graphs from source, except that not all basic blocks are used. At the component interaction level, it is relatively straightforward to find statically defined connections as defined in Sect. 2.2 by parsing the components and locating the transitions based on keywords. This ability is affected by the third limitation in Sect. 3, that is, transitions whose values are based on variables cannot always be fully determined. Data that are transmitted are defined as HTML form elements, which are readily parsed from the atomic sections (with the same limitation).

As said previously, only the software components in the presentation layer need to be analyzed, not the entire web application. Each component is represented by one component expression and one node in the ATG, thus there is no obstacle to this model scaling to very large web applications.

4 Applying the atomic section model—integration testing

The ASM provides a tool to analyze web applications that is fundamental to web applications in the same sense that basic blocks and control flow graphs are to non-web applications. The ASM uses the parts of the software that users can view and interact with (presentation layer), but not the back-end software (data content and representation layers). This model can help address problems that flow analysis is traditionally used for, including maintenance activities, debugging, optimization and refactoring, and evaluation of design. This section provides details and a demonstration of one application of the model, integration testing, and leaves the rest for future work.

A *test case* for a web application is specified as a sequence of interactions between components on clients and servers; in other words, paths of transitions through the ATG. Graph coverage criteria [4,50] can be used to cover the graph model in Sect. 3.5, with details supplied by the atomic sections and component expressions. An important consideration with web applications is that of invalid transitions, which are not considered in traditional graph coverage criteria. Invalid transitions could be incorporated at two levels, either by adding them directly to the model or by extending the coverage criteria. For this paper, we choose to extend the coverage criteria, primarily to make it easier to keep track of the benefits of considering invalid transitions.

A sequence of transitions is represented as a *derivation*, which is a sequence of transitions that begins at a start page, and uses component expressions and transitions 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 Fig. 7 combines both the link transitions and component expressions:

login.html $\gg$ gradeServlet $\multimap p_1 \cdot p_2 \cdot p_3 \cdot p_6$

This derivation starts at the start page (login.html), then submits the form and moves to `gradeServlet`, and the component expression transition yields the title (p_1), header (p_2), one grade (p_3), and the ending HTML commands (p_6).

Several derivations can be made when a student enters an incorrect ID or password. Note that the ATG can contain loops, which traditionally causes difficulties for coverage criteria. This research handles loops in ATGs by using relatively new test and analysis techniques [4], touring and prime paths, which are defined later. Assume that a valid login for `gradeServlet` is ("demo," "demo"). Four derivations involving incorrect passwords are given below:

1. S $\gg$ gradeServlet [get, (demo, dumo, 0)] $\multimap p_1 \cdot p_4 \cdot p_6$
2. S $\gg$ gradeServlet [get, (demo, dumo, 0)] $\multimap p_1 \cdot p_4 \cdot p_6 \longrightarrow$ sendMail[2] ...
3. S $\gg$ gradeServlet [get, (demo, dumo, 0)] $\multimap p_1 \cdot p_4 \cdot p_6 \gg$ gradeServlet [get, (demo, demo, 1)] $\multimap p_1 \cdot p_2 \cdot p_3 \cdot p_6$
4. S $\gg$ gradeServlet [get, (demo, dumo, 0)] $\multimap p_1 \cdot p_4 \cdot p_6 \gg$ gradeServlet [get, (demo, xyzzy, 1)] $\multimap p_1 \cdot$

[2] The transition is actually from p_4, but this fact is not illustrated in the derivation for the convenience of notation.

$p_4 \cdot p_6 \rightsquigarrow$ back $\gg$ gradeServlet [get, (demo, dumo, 0)] $\multimap$ $p_1 \cdot p_4 \cdot p_6 \gg$ gradeServlet [get, (demo, gmu, 1)] $\multimap$ $p_1 \cdot p_4 \cdot p_6 \gg$ gradeServlet [get, (demo, demo, 2)] $\multimap$ $p_1 \cdot p_2 \cdot p_3 \cdot p_6$

In the first derivation, the user entered the incorrect password "dumo" on the login page. `gradeServlet` printed the error message (from ATS p_4) and the user quit. In the second derivation, the user entered the incorrect password "dumo," got the error message, then sent email to the professor. In the third derivation, the user got the correct password on the second try. In the fourth derivation, the user tried two passwords, then used the back button on the browser to avoid the error message in ATS p_5, finally getting four tries instead of the intended three. When the back button was used, the hidden form field was reset to 0, which illustrates a flaw of using hidden form fields to pass data.[3] 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.

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 ATG can be very large or infinite, so test criteria are used to choose a reasonable number of derivations. Suitable criteria are discussed in the next subsection.

4.1 Coverage criteria

Test coverage can be applied at both the intra-component level (using the CIM) and the inter-component level (using the ATG). Intra-component level testing roughly corresponds to traditional module testing and inter-component level testing roughly corresponds to traditional integration testing. 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, any graph coverage criterion can be used.

The empirical validation in Sect. 4.3 applies graph coverage techniques to the ATG and to the transitions in the CIM, including link, component expression, 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 [26,28,40] has focused exclusively on link transitions, with little attention paid to component expression transitions and none at all to operational transitions.

For a comparative evaluation, we apply four separate coverage criteria. The first is quite straightforward: *atomic section coverage* requires every atomic section to be covered at least once. The other three need more explanation and are defined in the following subsections. One is a variation of path coverage, and the other two are new to this paper.

4.1.1 Prime path coverage

Prime path coverage was developed as a way to cover complete paths in graphs, while avoiding problems with infinite loops. Covering all paths is infeasible because unbounded loops result in an infinite number of paths, and even bounded loops result in an impractical number of paths. Ammann and Offutt [4] invented a path-oriented criterion, prime path coverage, which essentially requires coverage of all paths, up to, but not including, loops.

The criterion is based on the notions of simple and prime paths. A subpath in a graph from node n_i to n_j is *simple* if no node appears more than once on the subpath, with the exception that the first and last nodes may be identical (that is, $n_i = n_j$). A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path. (In other words, prime paths are maximal length simple paths.)

A *test path* is a complete sequence of nodes in a graph from a start 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 practical because the number of paths can be infinite. Simply touring prime paths may not be reasonable because many paths cannot be executed unless loops are executed more than once. To solve this problem, 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, for example, executing a loop more than once. 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**.

The formal definition makes this idea sound more complicated than it is. Figure 9 illustrates touring. Nodes n_0 and n_f are the initial and final nodes. The prime subpath $[n_0, a, b, d\ n - f]$ can be toured with sidetrips by a test path that executes the loop from b to c an arbitrary number of times, such as by $[n_0, a, b, c, b, d, n_f]$. The prime paths on the graph in Fig. 9 are $[n_0, a, b, d, n_f]$, $[n_0, a, b, c, d, n_f]$, $[b, c, b]$, $[c, b, d]$, and $[c, b, d, n_f]$, so a set of test paths to satisfy prime coverage can be $\{[n_0, a, b, c, d, n_f], [n_0, a, b, c, b, d, n_f], [n_0, a, b, c, b, c, d, n_f]\}$. Although larger graphs can contain more prime paths, it is often possible to tour all of them with just a few test paths.

4.1.2 Invalid access coverage

The second criterion used is new to this research, but quite simple. The intent is to model the situation when users, either

[3] A simpler approach would certainly be to tell programmers "don't do that." However, the 30 year history of buffer overflow problems leads us to believe that programmers will continue to take shortcuts.

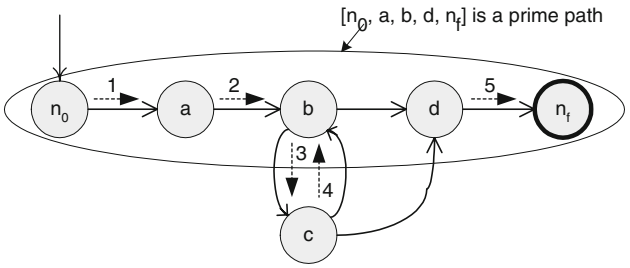

Fig. 9 Touring a graph with sidetrips

accidentally or purposefully, apply operational transitions by entering into the middle of a web application. This is called *invalid access* (IA), and each test is a sequence of one edge; a URL to a non-start page.

4.1.3 Invalid path coverage

The third criterion, *invalid path* (IP), extends the prime path criterion by adding operational transitions. Prime paths are extended by a two transition sequence: the first adds an operational transition (back button, forward button, refresh button, and URL rewriting), and if that is feasible, the second adds *every valid* transition out of the new node. Note that if the first transition is not possible or results in a software failure, the second transition cannot be taken.

The invalid path criterion has two variations. Because it has the potential to lead to a very large number of paths, it 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 ATG, **exactly one prime path** that ends in that node is extended.

4.2 Criteria examples

It may seem that these definitions mix the graph model and the test criteria. In fact, all of the nodes are included in the ATG, and it would be possible to define one test criterion that incorporated all three criteria by adding initial transitions to all nodes (for IA) and by explicitly adding all operational transitions to the model. We choose to separate them primarily for evaluation purposes. Previous research in testing web applications did not include operational transitions, so had nothing comparable with the invalid access and invalid path criteria. Separating the test criteria in this way allows the contribution of this research to be fairly evaluated. The prime path criterion subsumes edge coverage [50], edge-pair coverage [17,33,39], all-DU-Path coverage [16], and Chow's spanning tree coverage [12], so can be considered a very strenuous graph coverage criterion.

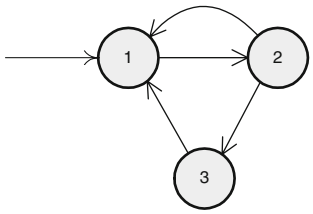

Fig. 10 Graph to illustrate coverage criteria

These criteria are illustrated on Fig. 10, which has three nodes, all of which are final nodes. Node 1 is the only start node. The graph in Fig. 10 has five prime paths:

```
1. [1 2 3 1]
2. [2 3 1 2]
3. [3 1 2 3]
4. [1 2 1]
5. [2 1 2]
```

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

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

The two non-start nodes have two invalid access tests to them, `[2]` and `[3]`. We assume that if there is no edge from one node to another, it represents an invalid transition. With this small example there are only two invalid transitions, `[1, 3]` and `[3, 2]`.[4] To compute the invalid paths, we start with the three 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 additional four paths containing an invalid transition to cover.

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

[4] In this idealized example, it does not matter whether these are back buttons or URL rewriting.

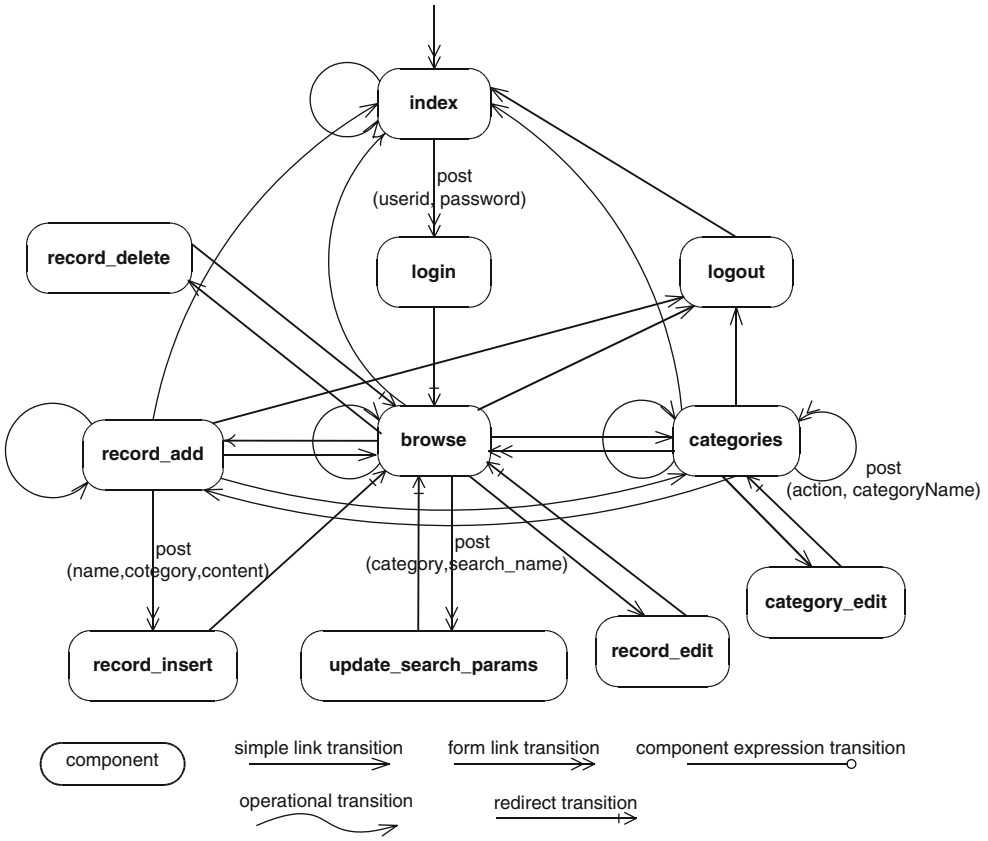

Fig. 11 STIS web application transition graph

4.3 A demonstration study on an example web application: STIS

To validate the model and test criteria, we applied them to a small but non-trivial web application. This is not a fully controlled experiment, but a first attempt to apply the new theoretical model, thus we call it a demonstration study. The Small Text Information System (STIS) helps users keep track of arbitrary textual information. STIS was implemented by a summer student working on an NSF-sponsored Research Experience for Undergraduate supplemental grant to support a previous project. *Text records*are stored and associated with *categories*. STIS stores all information in a database (mysql) and is comprised of 17 Java Server Pages and 5 Java bean classes. STIS requires users to log in. After being authenticated, users can search and view records ordered by categories, create categories and new records.

The ATG for STIS is shown in Fig. 11. Eleven of the seventeen web software 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 empirical example. Operational transitions are not explicitly shown on this graph. Also, the component expression 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 component expressions 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 occurrence 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 component expressions. When expanded, the expression 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 followed a 10 step process. Each step is annotated with whether it was accomplished automatically or by hand in this example. Proper tool support could automate the derivation of the ATG, but we did not have such a tool available for this study.

1. Determine atomic sections (automatically)
2. Derive the ATG (by hand)
3. Determine prime paths for the ATG (automatically)

Table 1 Component expressions for STIS components

Component	Component expression	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_srch_pars	$\multimap e$	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: *nav_bar*	$\multimap (p_1 \cdot (p_2 \mid e) \cdot p_3) \mid p_4$	5
record_search	$\multimap p_1 \cdot p_2{}^* \cdot p_3 \cdot (p_4 \mid e) \cdot p_5$	6

4. Determining 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 by hand)
10. Develop test inputs to complete ATS coverage (by hand)

4.3.1 Selecting test values

Tests for STIS are presented as test requirement derivations and the intra- and inter-component tests are merged. 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 example, 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 (this step could be automated with HTTPUnit or HTMLUnit). Consider the following test requirement to execute an operational transition:

index $\twoheadrightarrow$ login [post, (userid, password)] $\twoheadrightarrow$ browse $\longrightarrow$ logout $\rightsquigarrow$

record_add $\twoheadrightarrow$ record_insert [post, (name, category, content)]

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 redirect 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.

4.3.2 Analyzing test coverage

STIS contains 91 atomic sections, combined into the 17 CIMs summarized in Table 1. The atomic sections and component expressions 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, including 32 for prime path coverage 10 for invalid access, 50 for invalid path-1 and 64 for invalid path-2.

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 (infeasible), 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 and

Table 2 Test coverage and failure data

	Prime path	Invalid access	Invalid path		Total
			IP-1	IP-2	
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	6	27	14	47
Number of unique faults	0	6	19	10	35

Table 3 Types of failures found

	Failure category	PP	Inv. acc.	Invalid		Total
				P-1	P-2	
1.	Runtime failure	0	2	7	2	11
2.	Unhandled software exception	0	0	4	0	4
3.	Unexpected page content displayed w/o authentication	0	3	5	0	8
4.	Record added w/o authentication	0	1	2	2	5
5.	Search allowed w/o authentication	0	0	0	1	1
6.	Editing non-existent category	0	0	2	0	2
7.	Adding a record with empty fields; value is null	0	0	5	2	7
8.	Unexpected content displayed with authentication	0	0	2	5	7
9.	Irrelevant message	0	0	0	2	2
	Total number of failures	0	6	27	14	47

unique faults found for each group of tests is shown. The complete set of tests resulted in a total of 47 failures, representing 35 unique faults. All faults were naturally occurring and not known before testing began.

STIS has over 11,100 **full** invalid paths, so this example used the **each node** version of the invalid path criterion (IP). Recall that invalid paths are created by extending prime paths by adding a sequence of two transitions. Sometimes the first, operational, transition is feasible but the second, valid, is not, so for bookkeeping purposes they are referred to as IP-1 and IP-2 and split into two columns in the table. IP-1 tests contain only the first (operational) transition.

Of the IP-1 paths, 40 were infeasible, 23 produced correct output, and 27 produced incorrect output (failures). Of the 27 failing tests, 13 caused a runtime failure of STIS and 14 caused incorrect output. The second, valid, IP 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 redirect transitions; the next to last node contained a redirect transition (on the server), 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 3 breaks the failures that were found into nine categories. This is not meant to be a comprehensive list of web application failure categories, merely a categorization of the failures observed in this example. 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 less 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 users to see content that they were not authorized to see, and category 9 were messages that were irrelevant or invalid.

4.3.3 ATS *coverage*

ATS coverage could be considered to be similar to traditional node coverage at the unit testing level. The 156 tests from the ATG 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 some atomic sections were not covered. 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 original tests, the database never failed, so those atomic sections were not covered. This can be considered to be a question of controllability because databases typically do not fail as a result of test inputs, but through external events such as a network problem or hardware failure. 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. On retrospect, invalid inputs should probably be explicitly required.
3. Two atomic sections, $login.p_2$ and $login.HEAD$ (the first occurrence), 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 are no categories 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 the 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 category 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.

4.3.4 Analysis and discussion

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 edge and All DU-path coverage, among others). Other research papers have suggested testing static transitions, but not operational transitions. All of the failures in this study were found by using operational transitions and invalid form data, testing criteria for web applications that are new to this paper and directly based on the model.

The ATS coverage of the 156 ATG tests is worthy of detailed consideration. For most practical uses of test criteria, 87% coverage is considered reasonable and probably sufficient. Many of the 13 atomic sections that were not covered were particularly difficult to formulate tests for, particularly if tests are generated automatically. Incompletely entering forms is already a known technique [15,40], and one that can 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 and easy for a human tester to overlook. Yet, this test revealed a serious problem in the software. To summarize, the CIM/ATG 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 to reach full coverage 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 redisplays the data when operational transitions are used. Non-web applications **could not** have this failure.

Another example is when the software developers assume that the users will only access web applications through existing links. However, clever (or 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 (the subject of other papers [34,35]). Sometimes this can be used to discover files that the developers assume are hidden. Also, temporary or "debug" web software 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.

4.3.5 Limitations and threats to validity

The STIS demonstration study, though informative, has some limitations, including two threats to external validity. First, 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 anecdotal 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 and Travelocity).

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. However, a more robust method to generate values is needed. Other researchers are investigating this problem, both in the context of web applications [5,15] and for general GUI applications [31].

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. It is possible that the best automation we could hope for would be a tool that finds uncovered atomic sections, then presents them to a tester in a very clear way to guide the tester to create additional tests. The problem of difficult test requirements, of course, occurs with all strong test criteria.

5 Related work

Web applications tend to be based on gathering, processing, and transmitting data among heterogeneous hardware and software components, so data flow approaches [16,20,21,50] may be useful if adapted to web software. Most web software is object-oriented in nature, so inter-class [11,25,37,47] and intra-class [1] testing techniques can also help find some faults. These techniques would mostly be used for the data content and representation layers rather than the presentation layer, so would be complementary to this research.

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 [23]. 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 of programs deployed on the web.

The Web Modeling Language (WebML) [9,10] allows web sites to be conceptually described. The focus of WebML is primarily from the user's view and the data modeling. Modeling the distributed integration aspects of the software as presented here is complementary to the solutions proposed by WebML.

Some recent research has looked into testing software as statically determined, but none have addressed the problem of distributed integration. Kung et al. [26,28] 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 with only 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 model does not include the transitions of the ATG.

Liu et al. [29] developed a way to apply data flow analysis to web software components. They test definition-use pairs among client and server web pages and software components. The focus was on data interactions rather than control flow, and their model did not incorporate dynamically generated web pages or the operational transitions in this research.

Halfond and Orso [19] addressed the problem of finding the web application user interface screens. These screens are created dynamically, thus the problem of finding all screens is undecidable. Their algorithm uses static analysis to find more of the screens than previous techniques.

Ricca and Tonella [40] 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.

Di Lucca and Di Penta [14] 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. Timewise, this paper is the first published work addressing operational transitions, although it postdates our early technical report [48]. The technical report included the operational transitions, but had a much less complete model. Di Lucca and Di Penta's model focused on the browser capabilities without considering the server-side software or dynamically generated web pages.

Andrews et al. [5,6] introduced a system-level testing technique for web applications. The applications were modeled as hierarchical finite state machines with constraints. Tests were formed from subsequences of states in the FSM, and constraints were used to reduce the number of tests. The FSM model focuses on behavioral aspects of web applications, as opposed to the structural aspects the ASM represents.

Benedikt et al. [7] 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 provided 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 tool does not model atomic sections or the transitions in the ATG.

Elbaum et al. [15] 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" did not refer to the web application session object, but input data collected and remembered from previous user sessions. The user data were captured from HTML forms and included name-value pairs. Empirical 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, and can be complementary to the approach in this paper.

Sampath et al. [42,43] developed a novel method to reduce web application test suites. A subsequent paper by Sampath et al. [41] developed strategies to prioritize test suites, thereby improving the rate of fault detection. Alshahwan and Harman [2] introduced a technique that supports web application regression testing by repairing user session data when the software changes.

Lee and Offutt [27] describe a system that generates test cases using a form of mutation 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. This approach tests web software component interactions, whereas our current research is focused on the web application level.

Yang et al. [49] 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 [24] proposed an approach to formally describe 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.

In related research, we are investigating the use of a technique called *bypass testing* [34,35]. Some input data validation is done on the client, and web applications use HTML and Javascript to impose *constraints* on input data, for example, that a string must contain less than five characters. Bypass testing creates inputs that intentionally violate these validation rules and constraints, then submits the inputs directly to the web application without letting the web page validate them. We are also developing techniques to test web services, based on the XML messages that are used to exchange data among web services components [36].

Sprenkle et al. [45] looked at the problem of validating the results of testing web applications. They built a collection of comparators to look at the HTML responses from web applications. This research did not look at server side outputs from the web applications.

Ajax allows the browser on the client to send asynchronous requests to the web application on the server. Marchetto et al. [30] discussed some of the faults that this technique can cause and propose a preliminary state-based testing approach.

6 Conclusions and future work

This paper has presented two results. The primary result is a new theoretical model that captures dynamic aspects of the presentation layer of web applications. Second, test criteria based on the model were developed and defined. Results from a demonstration study of the model and tests on a fully worked out example 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 ASM 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 CIMs of 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. Analysis techniques such as control flow analysis, data flow analysis, and slicing can be applied to the graphs to support software engineering activities in testing and maintenance.

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). The ASM only depends on the properties of HTTP and HTML, thus is independent of software implementation 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.

The model was applied to the problem of testing web applications. The technique uses the model of the web application's behavior to define tests as sequences of user interactions. Previous research [26,28,40] 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 faults. Some of these faults would be fairly easy for developers to avoid with a better design; unfortunately, this is true of most software faults. It is possible that the ASM could be used as a design tool early in development.

6.1 Future work

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, but full automatic test data generation may be elusive. For the example in this paper, the ATG was generated by hand, but the transitions can easily be identified and the graph constructed automatically, as discussed in Sect. 3.5. The tests were also submitted by hand, but this can easily be automated by use of a tool like HTTPUnit, a framework for automating test execution for web applications [18,22,46].

An important problem for testing and analysis is that parts of web software applications can be integrated 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. We hope to address the problem of dynamic integration in the context of our model in the future.

Problems with state based behavior, multiple users, and concurrency have not explicitly been addressed. Anecdotal evidence indicates that many web applications suffer from failures of these types and they are very difficult to test for with human-based test generation. 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 address 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 validate the result that is sent to the client [45]. However, parts of the output are stored in 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.

Finally, this model is designed to be general and it captures fundamental aspects of the integration and execution of web applications that can be used for other analysis techniques. Control flow is explicitly in the model through the various transitions. Much, but not all, data flow information is partially captured in terms of the data that is transmitted through the transitions. Additional data items, in particular, those on the server, including in the shared session object, could be added into this model. With that information, the model can be extended to include a full data flow graph. This in turn gives the information needed for techniques like program slicing and change impact analysis.

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, Len Gallagher for help refining our notation and terminology, Yuan Yang for implementing STIS, and the anonymous reviewers for numerous very helpful suggestions.

References

1. Alexander, R.T., Offutt, J.: Criteria for testing polymorphic relationships. In: Proceedings of the 11th International Symposium on Software Reliability Engineering, October 2000, pp. 15–23, San Jose, CA. IEEE Computer Society Press, Silver Spring (2000)

2. 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), pp. 298–307, Lillehammer, Norway, April (2008)
3. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. 2nd edn. Prentice-Hall, Sun Microsystems Press, Englewood Cliffs (2003)
4. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008). ISBN 0-52188-038-1
5. Andrews, A., Offutt, J., Alexander, R.: Testing web applications by modeling with FSMs. Softw. Syst. Model. **4**(3), 326–345 (2005)
6. Andrews, A., Offutt, J., Dyreson, C., Mallery, C.J., Jerath, K., Alexander, R.: Scalability issues with using FSMWeb to test web applications. Inf. Softw. Technol. (accepted) (2009)
7. 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), pp. 654–668, Honolulu, HI, May (2002)
8. Blumenstyk, M.: Web application development—bridging the gap between QA and development. *StickyMinds.com*, 2006. http://www.stickyminds.com/s.asp?F=S3658_ART_2. Accessed November (2007)
9. Ceri, S., Fraternali, P., Bongio, A.: Web modeling language (WebML): a modeling language for designing web sites. In: Ninth World Wide Web Conference, Amsterdam, Netherlands, May (2000)
10. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, Los Altos, December (2002). Information available online at: http://webml.elet.polimi.it/webml/
11. Chen, M.-H., Kao, M.-H.: Testing object-oriented programs—an integrated approach. In: Proceedings of the 10th International Symposium on Software Reliability Engineering, pp. 73–83, Boca Raton, FL. IEEE Computer Society Press, Silver Spring, November (1999)
12. Chow, T.: Testing software designs modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
13. comscore, Inc. http://www.comscore.com. (2008)
14. Di Lucca, G.A., Di Penta, M.: Considering browser interaction in web application testing. In: 5th International Workshop on Web Site Evolution (WSE 2003), pp. 74–84, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press, Silver Spring (2003)
15. Elbaum, S., Rothermel, G., Karre, S., Fisher, M.: Leveraging user-session data to support web application testing. IEEE Trans. Softw. Eng. **31**(3), 187–202 (2005)
16. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Trans. Softw. Eng. **14**(10), 1483–1498 (1988)
17. Fujiwara, S., Bochman, G., Khendek, F., Amalou, M., Ghedasmi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991)
18. Gold, R.: Httpunit home. online: SourceForge, 2003. http://httpunit.sourceforge.net/. Accessed June 2005.
19. 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, pp. 145–154, Dubrovnik, Croatia (2007)
20. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: Symposium on Foundations of Software Engineering, pp. 154–163, New Orleans, LA. ACM SIGSOFT, December (1994)
21. Harrold, M.J., Soffa, M.L.: Selecting and using data for integration testing. IEEE Softw. **8**(2), 58–65 (1991)
22. Hatcher, E., Loughran, S.: Java Development with Ant. Manning Publications, Greenwich (2002)
23. Hower, R.: Web site test tools and site management tools. (2002) http://www.softwareqatest.com/qatweb1.html
24. Jia, X., Liu, H.: Rigorous and automatic testing of web applications. In: 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), pp. 280–285, Cambridge, MA, November (2002)
25. Jin, Z., Offutt, J.: Coupling-based criteria for integration testing. Wiley's Softw. Test. Verif. Reliab.**8**(3), 133–154 (1998)
26. 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), pp. 537–542, Taipei, Taiwan, October 2000. IEEE Computer Society Press, Silver Spring (2000)
27. 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, pp. 200–209, Hong Kong, China, November 2000. IEEE Computer Society Press, Silver Spring (2001)
28. 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, pp. 84–96, San Jose CA, October 2000. IEEE Computer Society Press, Silver Spring (2000)
29. Liu, C.H., Kung, D.C., Hsia, P., Hsu, C.T.: An object-based data flow testing approach for web applications. Int. J. Softw. Eng. Knowl. Eng. **11**(2), 157–179 (2001)
30. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008 Industry Track), pp. 121–130, Lillehammer, Norway, April (2008)
31. Memon, A.M., Soffa, M.L., Pollack, M.E.: Hierarchical GUI test case generation using automated planning. IEEE Trans. Softw. Eng. **27**(2), 144–155 (2001)
32. Offutt, J.: Quality attributes of web software applications. IEEE Softw. Special Issue Softw. Eng. Internet Softw. **19**(2), 25–32 (2002)
33. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Wiley's Softw. Test. Verif. Reliab. **13**(1), 25–53 (2003)
34. Offutt, J., Wang, Q., Ordille, J.J.: An industrial case study of bypass testing on web applications. In: 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008 Industry Track), pp. 465–474, Lillehammer, Norway, April (2008)
35. Offutt, J., Wu, Y., Du, X., Huang, H.: Bypass testing of web applications. In: 15th International Symposium on Software Reliability Engineering, pp. 187–197, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society Press, Silver Spring (2004)
36. Offutt, J., Xu, W.: Testing web services by XML perturbation. In: Proceedings of the 16th International Symposium on Software Reliability Engineering, Chicago, IL, November 2005. IEEE Computer Society Press, Silver Spring (2005)
37. Parrish, A., Zweben, S.H.: Analysis and refinement of software test data adequacy properties. IEEE Trans. Softw. Eng. **17**(6), 565–581 (1991)
38. Pertet, S., Narasimhan, P.: Causes of failure in web applications. Technical report, Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-05-109, December 2005. http://www.pdl.cmu.edu/Publications/
39. Pimont, S., Rault, J.C.: A software reliability assessment based on a structural behavioral analysis of programs. In: Proceedings of the Second International Conference on Software Engineering, pp. 486–491, San Francisco, CA, October (1976)
40. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: 23rd International Conference on Software Engineering (ICSE '01), pp. 25–34, Toronto, CA, May (2001)

41. Sampath, S., Bryce, R., Viswanath, G., Kandimalla, V., Koru, G.: Prioritizing user-session-based test cases for web application testing. In: 1st IEEE International Conference on Software Testing, Verification and Validation (ICST 2008 Industry Track), pp. 141–150, Lillehammer, Norway, April (2008)
42. Sampath, S., Sprenkle, S., Gibson, E., Pollock, L.: Web application testing with customized test requirements—an experimental comparison study. In: 17th International Symposium on Software Reliability Engineering (ISSRE'06), pp. 266–278. IEEE Computer Society Press, Silver Spring, November (2006)
43. Sampath, S., Sprenkle, S., Gibson, E., Pollock, L., Greenwald, A.S.: Applying concept analysis to user-session-based testing of web applications. IEEE Trans. Softw. Eng. **33**(10), 643–658 (2007)
44. Singh, I., Stearns, B., Johnson, M., Enterprise Team.: Designing Enterprise Applications with the J2EE Platform, 2nd edn. Sun Microsystems (2002)
45. Sprenkle, S., Pollock, L., Esquivel, H., Hazelwood, B., Ecott, S.: Automated oracle comparators for testing web applications. In: Proceedings of the 18th International Symposium on Software Reliability Engineering, pp. 253–262, Trollhatten, Sweden, November 2007. IEEE Computer Society Press, Silver Spring (2007)
46. Tappenden, A., Beatty, P., Miller, J., Geras, A., Smith, M.: Agile security testing of web-based systems via HTTPUnit. In: Proceedings of the Agile Development Conference (ADC '05), pp. 29–38, Denver, CO, July 2005. IEEE Computer Society (2005)
47. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93), pp. 302–310, Montreal, Quebec, Canada, September 1993. IEEE Computer Society Press, Silver Spring (1993)
48. Wu, Y., Offutt, J.: Modeling and testing web-based applications. Technical report ISE-TR-02-08, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2002. http://www.cs.gmu.edu/~tr_admin/2002.html
49. Yang, J.-T., Huang, J.-L., Wang, F.-J., Chu, W.C.: An object-oriented architecture supporting web application testing. In: Proceedings of the Twenty-Third Annual Computer Software and Applications Conference (COMPSAC '00), Phoenix, Arizona, October 2000. IEEE Computer Society Press, Silver Spring (2000)
50. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997)

Author biographies

Jeff Offutt is Professor of Software Engineering at George Mason University. He has published over 115 refereed research papers and is co-author of Introduction to Software Testing. He is editor-in-chief of Wiley's journal of Software Testing, Verification and Reliability; steering committee chair for the IEEE International Conference on Software Testing, Verification, and Validation; program chair for ICST 2009; and on the technical board of advisers for Certess, Inc. He received the PhD in Computer Science from the Georgia Institute of Technology. His current research interests include software testing, analysis and testing of web applications, object-oriented program analysis, module and integration testing, and software maintenance.

Ye Wu is a program manager of the Innovation Information Technology Division at Science Application International Corporation (SAIC). Dr. Wu is leading the quality assurance group that covers multiple projects for the National Cancer Institute, Center for Bioinformatics and Information Technology. Wu received a PhD degree in Computer Science from the State University of New York at Albany in 2000. His research interests include web-based software testing, component-based software testing and maintenance, and software architecture analysis.