Automated Driver Generation for Analysis of Web Applications

Oksana Tkachuk and Sreeranga Rajan

Fujitsu Laboratories of America,
Sunnyvale, CA USA
{oksana,sree.rajan}@us.fujitsu.com

Abstract. With web applications in high demand, one cannot underestimate the importance of their quality assurance process. Web applications are open event-driven systems that take sequences of user events and produce changes in the user interface or the underlying application. Web applications are difficult to test because the set of possible sequences of user inputs allowed by the interface of a web application can be very large. Software model checking techniques can be effective for validating such applications but they only work for closed systems. In this paper, we present an approach for closing web applications with a driver that contains two parts: (1) the application-specific Page Transition Graph (PTG), which encodes the application's possible pages, user and server events, their corresponding event-handlers, and user data and (2) the application-independent PTG-based driver, which generates test sequences that can be executed with analysis tools such as Java PathFinder (JPF). The first part can be automatically extracted from the implementation of a web application and the second part is written once and reused across multiple web applications belonging to the same framework. We implemented our approach in a driver generator that automatically extracts PTG models from implementation of JSP-based web applications, checks the extracted PTGs for navigation inconsistencies, and enables JPF analysis. We evaluated our approach on ten open-source and industrial web applications and present the detected errors.

1 Introduction

Web applications are open event-driven systems that take sequences of user events (e.g., button clicks in a browser window) and produce changes in the user interface (e.g., transition to a different page) or the underlying application (e.g., the shopping cart becomes empty). Web applications are difficult to test because the set of possible user event sequences allowed by the interface of a web application can be very large.

Current approaches to testing web applications often rely on constraining the analysis based on a formal model (e.g., UML in [13], WebML in [3], state machines in [1,7,9,14]). Some approaches rely on user specifications (e.g., [3,7]) or models extracted using run-time (e.g., [8,11,13]) or static analysis (e.g., [9,10]).

All approaches have their strengths and weaknesses. User specifications can capture the behavior that may be difficult to extract automatically but require manual work. Run-time crawling techniques are designed to visit web pages automatically. However, without user guidance (e.g., specification of user input data), crawlers may not be able to visit all possible pages of the application and the extracted models may miss some behavior. Static analysis techniques work well for specific domains and extracting specific features. For example, the approach in [10] extracts control flow graphs for web applications written in PLT Scheme and checks their navigation properties with a specialized model checker.

At Fujitsu, we have successfully used Java PathFinder (JPF) [2] to check business logic properties of web applications [12]. Our previous approach relied on user specifications describing (1) sequences of user events using regular expressions and (2) mappings from user events to the underlying application's event handlers that process them. Using these specifications, we automatically generated drivers that set up the event-handling code of the application under test, generated and ran test sequences with JPF. However, using this approach, the interface of the web application (e.g., its pages, buttons, and links) is abstracted away, since JPF cannot handle non-Java components and the generated drivers test the event-handling code of the application directly. In addition, the necessity to specify use case scenarios hindered the usability of the approach within the Fujitsu's testing teams.

To address the above limitations, we needed (1) to encode possible use case scenarios and event-handler registration information in a model, with an option to represent this model in Java, so JPF could handle it and (2) to increase usability by extracting this model automatically. In this paper, we present an approach to closing web applications with a driver that contains two parts: (1) the application-independent Page Transition Graph (PTG), which encodes the application's possible pages, user and server events, their corresponding event-handlers, and user data and (2) the application-independent PTG-based driver, which generates test sequences that can be executed with analysis tools such as JPF. By splitting the driver into two parts, we enable automation: (1) using framework-specific knowledge, the PTG can be automatically extracted from implementation of web applications and (2) the PTG-based driver is written once and reused across multiple web applications belonging to the same framework.

We implemented our driver generator and evaluated it on ten open-source and industrial JSP-based web applications. For each application, the driver generator (1) extracts and visualizes its PTG, (2) checks the extracted PTG for navigation errors and (3) generates and executes test sequences with JPF. To the best of our knowledge, this is the first driver generator that automatically extracts both navigation and event-handling aspects of the web applications and enables checking web applications with JPF.

This paper is organized as follows. The next section describes the PTG model and its usage for analysis of general web applications. Section 3 describes automated PTG extraction for JSP-based web applications, using Struts[1] as an

[1] `http://struts.apache.org/`

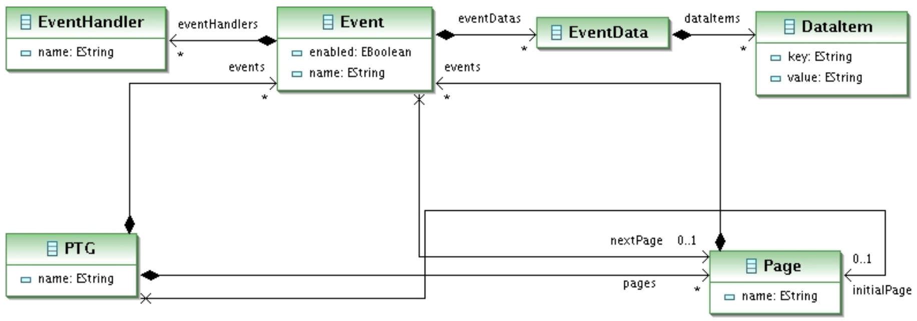

Fig. 1. PTG Class Diagram

example framework. Section 4 describes implementation of the PTG extractor. Section 5 presents experiments for ten open-source and industrial examples, including a discussion on the detected errors and limitations of the current approach. Section 6 describes the related work and Section 7 concludes.

2 Approach

In this section, we describe the PTG model, which is general enough to be applicable to various types of event-driven applications (e.g, GUIs). The goal of the PTG is to encode the navigation information and enable the registration of the application event handlers and generation of sequences of user events to drive the event-handling code of the web application under test. The PTG model is a graph with nodes corresponding to possible pages of the web application and transitions corresponding to possible user and server events, along with information about the event-handlers, event data, and possible pre-conditions for those events that require it. Equivalently, the PTG can be represented by a finite state machine.

Figure 1 shows a UML class diagram of the main elements of the PTG model designed based on our previous experience with model checking J2EE and GUI applications using JPF. A populated `PTG` encodes application-specific information about the application's set of available pages. Some pages can be marked as initial. Each `Page` contains a set of events attached to it. Each `Event` represents a possible transition to another page. In this paper, we concentrate on user events such as clicks on links and buttons and server events such as automatic redirection from one page to another. Some events may not be always enabled (e.g., buttons and links may become disabled depending on some condition). To reflect this possibility, `Event` declares a field `enabled`. Button click events are usually handled by the registered event handlers, therefore, events may have a set of `EventHandler`s associated with them. Some events may require user data, when filling out a form or writing into a text box. To reflect this possibility, an event may have a set of `EventData`, filled with `DataItem`s, attached to it. The `EventData` is designed to hold keys and values, which correspond to the names

```
public class EnvDriver {
public static HttpServletRequest req;
public static HttpServletResponse res;
public static AbstractPTG ptg;
public static void main(String[] args){
 init(); //get populated ptg
 Page initPage=ptg.getInitPage();
 processPage(initPage);
}
public void processPage(Page p){
 List eList=p.getEvents();
 int nEvents=eList.size();
 //nondeterministic choice over events
 int eIndex=Verify.random(nEvents-1);
 Event e=eList.get(eIndex);
 if(e.isEnabled())
  processEvent(e);
}
```

```
public void processEvent(Event e){
 List dList=e.getEventData();
 int nData=dList.size();
 //nondeterministic choice over data
 int dIndex=Verify.random(nData-1);
 EventData ed=dList.get(dIndex);
 req=new HttpServletRequestImpl();
 processEventData(ed, req);
 processEventHandling(e, req);
}
public Page processEventHandling(
 Event e,  HttpServletRequest req){
 for(EventHandler h:e.getHandlers()){
  res=h.perform(req, res);
  nextPage=e.getNextPage(res);
  processPage(nextPage);
 }
}
```

Fig. 2. PTG-Based JPF Driver (excerpts)

of the text fields (e.g., `userid` and `password` on a login form and the user values entered in those fields).

2.1 PTG Analysis

The PTG model offers two levels of information: (1) navigation and (2) navigation with event-handling. The navigation level, containing information about possible page transitions, can be used to perform analysis with respect to navigation properties. This analysis does not require the business tier of the application under test to be part of the model, thus, it can be scalable and fast.

One can check various reachability properties (e.g., Page A is reachable from page B). However, such checks require selecting specific page names from a list of application pages. To simplify this step even further, we designed several checks that look for possible violations without user specifications. In Section 5, we present our experience with the following types of possible errors:

– Unreachable pages: pages unreachable from the initial page
– Ghost pages: referenced pages that do not exist
– Undefined transitions: transitions whose references do not match any of the available definitions (e.g., due to a typo)

2.2 PTG-Based Analysis

To check the business tier of the application under test, we developed a reusable application-independent driver that, given a populated application-specific PTG, traverses the PTG and generates and executes sequences of user events. Figure 2 shows a driver, tuned to J2EE event-handling APIs. The driver implements the following methods: `processPage()` calls `processEvent()` for each enabled event available on that page; `processEvent()` creates an `HttpServletRequest` object, populates it with event data, using `processEventData()`, and executes the event handling code; `processEventHandling()` calls the event handling method of the event handler and returns a next page depending on the result. The algorithm recursively calls `processPage()` on the next page, thus traversing the PTG using Depth First Search (DFS). This algorithm can be augmented with conditions to vary the flavor of traversal (e.g., a bound can be used to limit the length of each generated sequence).

The implementation of the driver, along with the populated PTG, implementation of the web application under test, and database stubs (developed in our previous work [12]) are fed to JPF. One can use JPF's listener framework to implement listeners that support property specification based on temporal logic (e.g. "the cart becomes empty after checkout") or, in the absence of specifications, check for run-time errors or coverage. In section 5, we report the number of generated test sequences and coverage in terms of PTG nodes and transitions traversed by the driver.

One key feature of the driver is its ability to encode a *nondeterministic* choice over a set of events attached to a page or a set of data available for population of `HttpServletRequest`. To encode the nondeterministic choice, we use JPF's modeling primitive `Verify.random(n)`, which forces JPF to systematically explore all possible values from a set of integers from `0` to `n`. Another feature of the driver is its extensible APIs. One can override certain methods to customize the driver according to the desired traversal algorithm (e.g., keeping track of visited transitions to avoid loops), the underlying analysis framework (e.g., to execute with the Java JVM by changing nondeterministic choices to for-loops), and the framework used to encode the event-handling APIs (e.g., Struts, as presented in the next section).

3 PTG Extraction

Driver generation in general is a hard problem. By concentrating on specific frameworks, we can achieve a high degree of automation. In this section, we present an automated PTG extraction technique for Java-based web applications that encode their page transitions in JSP and XML files. As an example of such a domain, we use Struts-based applications. The choice of the Struts framework was motivated by a request from Fujitsu's development teams that use Struts to speed up the development time. In this section, we describe the Struts framework, give a small example and present the PTG extraction technique.

3.1 Struts Framework

Struts[2] is an open-source framework based on the Model-View-Controller (MVC) design pattern. The view portion is commonly represented by JSP files, which combine static information (e.g., HTML, XML) with dynamic information (e.g., Java as part of JSP scriptlet).

[2] There are two editions of Struts: Struts1 and Struts2. In this section, we describe Struts1; Struts2 is handled similarly.

The controller is represented by the Struts servlet controller, which intercepts incoming user requests and sends them to appropriate event-handlers, according to action mappings information specified in the XML descriptor file usually called `struts-config.xml`. In Struts, the request handling classes are subclassed from the `Action` class in the `org.apache.struts.action` package. Their event-handling method is called `execute()`. Actions encapsulate calls to business logic classes, interpret the outcome, and dispatch control to the appropriate view component to create the response. Form population is supported by the `ActionForm` class, which makes it easy to store and validate user data.

3.2 Example

To demonstrate our technique, we use a small registration example that allows users to register and login to their accounts, followed by a logout. This is a simplified version of the original application[3]. To simplify presentation, we removed several pages and events from the original application. In Section 5, we present experiments for both versions of this example.

The registration example encodes its page transitions using two XML files (`web.xml` and `struts-config.xml`) and six JSP pages: `index.jsp` (the initial page, marked in the `web.xml` configuration file), `welcome.jsp`, `userlogin.jsp`, `loginsuccess.jsp`, `userRegister.jsp`, and `registerSuccess.jsp`. The event-handling part of the example contains four `Action` classes and two `ActionForms`.

The PTG extraction approach has two major steps: (1) parsing JSP/XML/-Java files, mining relevant information from them, and storing the information in the convenient form, commonly referred to as the Abstract Syntax Tree (AST) and (2) building PTG based on the previously mined information. Next, we describe these steps.

3.3 Extracting PTG-Related Data

The parsing step mines information from (1) JSP files, (2) XML configuration files, and (3) class files that encode `Actions` and `ActionForms`.

Analyzing JSP. The first step parses all JSP files of the application. Each JSP page corresponds to a `Page` node in the AST. Each JSP file is scanned for information about possible user and server events, encoded statically in JSP. Figure 3 (left) shows examples of such encodings on the pages of the registration example: `index.jsp` redirects to `welcome`, the `welcome.jsp` page contains links back to itself, to `userRegister.jsp`, and `userlogin.jsp`, the `userlogin.jsp` page contains a reference to the `/userlogin` action, and `loginsuccess.jsp` contains a redirect to another page.

To find references to possible user and server events, the parsers need to know the types of encoding to track. Figure 4 shows examples of such encodings: link and form JSP/HTML tags and attributes, redirect tags and scriptlet keywords to

[3] `http://www.roseindia.net/struts/hibernate-spring/project.zip`

```
//from index.jsp
<logic:redirect forward=welcome
//from welcome.jsp
<html:link page="/Welcome.do">
  <font color="white">Home</font></html:link>
<html:link page="/pages/user/userRegister.jsp">
  <font color="white">Register</font></html:link>
<html:link page="/pages/user/userlogin.jsp">
   <font color="white">Login</font></html:link>
//from userlogin.jsp
<html:form action="/userlogin" method="post">
//from loginsuccess.jsp
if(userid==null)
  response.sendRedirect("../userlogin.jsp)
```

```
//from struts-config.xml
<action mappings>
<action path="/Welcome"
  forward="/pages/Welcome.jsp"/>
</action>
<action path="/userlogin"
  name="UserLoginForm"
  type="UserLoginAction">
  <forward name="success"
      path="loginsuccess.jsp"/>
  <forward name="failure"
      path="userlogin.jsp"/>
</action>
</action-mappings>
```

Fig. 3. Example JSP tags and XML definitions (excerpts)

```
//links
LINK_TAG = <html:link;<c:url;<s:url
LINK_TAG_ATTRIB = href
//forms (actions)
FORM_TAG = <html:form;<s:form
```

```
//redirects
REDIRECT_TAG = <logic:redirect;<jsp:forward
REDIRECT_SCRIPTLET = response.sendRedirect
//includes
INCLUDE_TAG = <jsp:include;<jsp:param
INCLUDE_SCRIPTLET = include
```

Fig. 4. JSP Parser Configuration File

track redirect and inclusion relationships (i.e., one JSP page can include another and display forms and links available on the included page). These encodings allow parsers to find references to possible user and server events and store them as part of the AST `Events`. Each AST `Event` stores information about its `path`, which can be a reference to the next JSP page or an action, defined in the XML configuration files. When the `path` refers to a URL or a file not related to PTG (e.g., an image), the parsers filter out such events based on the naming conventions.

Analyzing XML. XML configuration files contain various definitions needed at deployment time. For example, `web.xml` contains information about the naming conventions and initial pages, whereas `struts-config.xml` contains action definitions. As an example, we describe the definition of the `/userlogin` action, shown in Figure 3 (right), referenced on the `userlogin.jsp` page: (1) this is the form submission event, taking `UserLoginForm`, (2) it is handled by the event-handler of type `UserLoginAction`, (3) the outcome of this event depends on whether the event handling code returns "success" or "failure". In case of "success", the next page is `loginsuccess.jsp`; in case of "failure", the next page is `userlogin.jsp`. Note that in this particular case, the `path` of each outcome references a JSP page. In general, it may reference another definition, available in the given or another XML configuration file.

The XML parsers parse and store all of the XML information as part of the AST `Definitions`. Definitions describing form submission events require additional data, used to populate `ActionForm` objects. This information can be mined from several sources, including `ActionForm` classes themselves.

Analyzing Java Classes. This step finds and loads all `ActionForm` classes of the application. For each application form, e.g., `UserLoginForm`, it loads its class file

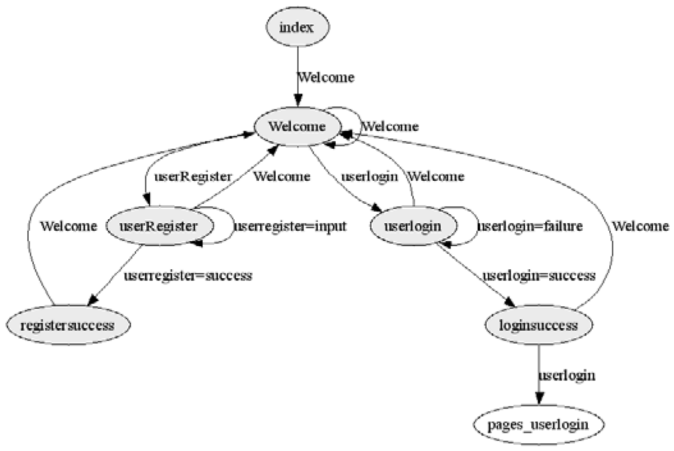

Fig. 5. Example PTG (Visualization)

and, using reflection APIs, finds all of its fields, e.g., `userid` and `password`. Using naming conventions, each field can be set through a field-specific setter method. For example, `setUserid(String)` sets the `userid` field and `setPassword(String)` sets the `password` field. Thus, after discovering field names, we are able to generate code that sets these fields to values that represent user values entered into the corresponding text fields. To generate user values, this step relies on values specified in a file (e.g., based on symbolic execution that supports strings [4]). In the absence of specifications, this steps generates default values (e.g., common corner cases such as empty, non-empty strings).

3.4 PTG Construction

After the parsing step, the PTG construction step takes the AST, containing the information about all JSP pages, their possible user and server events, and available XML definitions, and populates the PTG. For each AST `Page`, the algorithm creates a PTG `Page`. Then, for each reference to the user or server event available on each AST page, the algorithm tries to resolve the reference, based on the encoded `path` of the event. If the `path` references a JSP page, then a lookup mechanism is invoked to find that page. If the page is not found, a new ghost page is created and set as a destination page of the event. If the `path` does not reference a JSP page, then it is looked up under the definitions. If the definition is not found, the next page of the event is set to the special `Undefined` ghost page, otherwise, the definition is used to resolve the possible destination pages of the event. Each definition updates the event according to its domain-specific encoding, e.g., the `Action` definitions from `struts-config.xml` generate `EventHandlers` according to the `type` information and calculate destination pages according to the information encoded in `forwards`. This part of the algorithm is recursive, since a definition may reference other definitions. After the event construction is finished, the event is added to its source page.

By construction, our approach extracts an overapproximation of the page transitions, with respect to static JSP and XML encodings specified to the tool. Currently, the tool treats all transitions as *possible* and does not keep track of conditions under which some transitions may become enabled or disabled. We note that the model of `Event` includes the `enabled` field and `isEnabled()` method, used by the PTG-based driver. It is currently the limitation of the JSP parser, which does not extract conditions that may affect enabledness of some events at run-time.

Even with overapproximations, the extracted PTG is useful for visualization and detecting potential errors. Figure 5 shows visualization of the page transition portion of the PTG extracted for the registration example, which shows one (white) ghost page. It is calculated when processing `response.sendRedirect()` event on the `loginsuccess.jsp` page. This event refers to the `"../userlogin.jsp"` page, which does not exist; the correct path is `userlogin.jsp`. Evaluating this potential error, we find that this transition has a condition associated with it: `if(userid==null)`. Under normal usage, this error is not exercised, however, it is still possible to see the error at run-time. When accessing the `loginsuccess.jsp` page directly, the server throws an exception with "the requested resource not available" message. Such errors often appear after refactoring or due to typos and are difficult to find using traditional testing techniques.

Note that, the PTG analysis with respect to navigation properties requires no extensions for Struts domain. To perform the PTG-based analysis of Struts-based applications, the generic driver, presented in section 2.2, needs to be extended to handle domain-specific event-handling APIs, for example the method `processEventHandling()` needs to be overridden:

```
public Screen processEventHandling(Event event,  HttpServletRequest req){
  List handlers = event.getEventHandlers();
  for(Action action : handlers){
    forward = action.execute(...,form, req, res);
    nextPage = getNextPage(event, forward); ...
}
```

4 Implementation

We implemented the PTG extractor with the following modules:

Parsers: The JSP parser is automatically generated by JavaCC, based on the grammar capable of parsing JSP/HTML/XML tags, scriptlet blocks, and various types of comments. The JSP parser extracts information based on the tags specified in a separate file (similar to one in Figure 4). One can add additional tags, based on the domain-specific encodings. XML parsers are based on the Digester framework[4]. We currently support parsing of `web.xml`, and configuration files used by Struts, Struts2, and Tiles[5] frameworks.

Generators: Given the AST, generators populate the PTG data structure. We implemented the PTG data structure using the Eclipse Modeling Framework

[4] `http://commons.apache.org/digester/`

[5] `http://tiles.apache.org/framework/index.html`

(EMF)[6]. EMF supports model specification using Java interfaces. Given Java interfaces, EMF automatically generates their implementation and provides additional facilities, such as writing the model in XML and loading it from XML to Java. The PTG APIs can be used as a library.

Printers: Given the populated PTG, printers generate its various representations. Currently, we support PTG generation in Java, used by the PTG-based JPF driver, XML, used to populate page and event names for property specifications, and the Dot[7] representation, used for visualization.

Checkers: We implemented various checkers, which take a populated PTG and collect features that may signal possible errors (e.g., unreachable and ghost pages and undefined transitions). The checkers print their output in XML.

The PTG generator has an extensible architecture: one can plug in their own implementation of parsers and code generators. The entire codebase is around 10K LOC, of which 4K LOC belongs to the auto-generated code (i.e., the EMF-generated PTG code and the JavaCC parser). The PTG generator is fully automated: one only needs to specify the application directory of the example under test.

5 Experience

We evaluated our driver generator on a collection of ten examples. Before setting up the experiments, we had the following research questions:

PTG Quality: What is the quality of the PTG generator? Does it miss any transitions? Does it generate spurious transitions? Does it enable error detection? What is the level of coverage achieved by the PTG-based driver at run-time?

Human/Tool Cost: What is the level of automation? What is the level of manual work?

5.1 Case Studies

To perform evaluation, we downloaded several open-source Struts-based examples: the simplified and the original registration example, described in section 3.2, *cookbook*, *artimus* and *polls* from sourceforge[8], *petstore*[9], *personalblog*[10], and *cart*[11]. In addition, we were given two Fujitsu's internal applications: a sample Project Management application, called in this paper *FujitsuPM*, and a sample Document Management application, called *FujitsuDM*. All examples can be deployed using the Tomcat server.

[6] `http://www.eclipse.org/emf`
[7] `http://www.graphviz.org/`
[8] `http://sourceforge.net/projects/struts/files`
[9] `http://www.jwebhosting.net/servlets/jpetstore5/index.html`
[10] `http://suif.stanford.edu/~livshits/work/securibench/download.html`
[11] `http://www.roseindia.net/shoppingcart/cart1.1.zip`

Table 1. Examples Static Data

ID	AppName	**Frameworks**	**LOC**	**Cl** (All/Act/Form)	**JSP/XML**
1	*simregister*	Struts	615	15/4/2	6/2
2	*register*	Struts	709	34/5/3	12/2
3	*cookbook*	Struts, Tiles	681	18/1/0	17/4
4	*artimus*	Struts, Tiles	799	16/1/1	20/3
5	*petstore*	Struts	1,705	26/5/5	21/2
6	*personalblog*	Struts, Tiles	807	24/11/5	23/3
7	*polls*	Struts, Tiles	8,980	48/13/0	35/3
8	*cart*	Struts, Tiles	3,873	57/16/11	49/3
9	*FujitsuPM*	Struts2	9,876	67/9/0	25/3
10	*FujitsuDM*	Struts, Tiles	18,277	129/17/9	29/3

Table 1 shows static information about the case studies. The **Frameworks** column describes the types of frameworks used by the front end of each application. The **LOC** column gives the number of lines of code; the **Cl** column gives the number of all classes, followed by the number of `Action` classes, followed by the number of `ActionForm` classes. The **JSP/XML** column gives the number of application's JSP pages and XML configuration files relative to the PTG population.

5.2 Experiment

For each case study, we performed the following steps:

1. PTG Extraction: we ran the PTG extractor with the default values used for population of the forms. This step generates PTG and prints it using the following representations: Java (used by JPF), XML, and Dot.
2. PTG Analysis: we enabled PTG checkers looking for various possible violations and features. We present our finding for the following features described in section 2.1: unreachable pages, ghost pages and undefined transitions. Checking these features requires no user specifications.
3. PTG-based Analysis: We ran the driver using the traversal algorithm that avoids loops by keeping track of visited transitions. We used the Java representation of PTG, generated in the first step. We measured page and transition coverage achieved by the driver.

All experiments were run on a Linux machine, with 2G RAM and 2.66 GHz processor. Table 2 shows the collected data. The **PTG** column shows the size of the extracted PTG in terms of pages/nodes and events/transitions. The **Time** column shows the time, in min:sec format, the PTG extractor takes to parse JSP/XML/Java files and build a PTG for each example. The **Errors** column presents possible errors found in each PTG: pages unreachable from the initial page, followed by ghost pages, followed by the number of undefined transitions. The next columns present data for generation of test sequences based on running the PTG-based driver with JPF. The **Sts/Mem/Time** column gives the number of end states as reported by JPF, which corresponds to the number of

Table 2. Examples Experiment Data

ID	AppName	**PTG** (n/tr)	**Time**	**Errors**	**Sts/Mem/Time**	**Cov**(n/tr)
1	*simregister*	7/13	00:02	0/1/0	48/54/00:03	7/13
2	*register*	13/25	00:03	2/1/0	192/75/00:04	11/25
3	*cookbook*	17/18	00:03	5/0/0	15/56/00:02	12/17
4	*artimus*	20/29	00:03	9/0/0	256/70/00:11	10/21
5	*petstore*	22/87	00:04	2/1/8	7,671/106/01:30	20/85
6	*personalblog*	24/83	00:03	3/1/3	11,618/122/01:17	21/61
7	*polls*	36/113	00:07	5/1/1	10,081/106/02:04	28/76
8	*cart*	55/165	00:08	31/6/3	6,345/105/01:11	24/75
9	*FujitsuPM*	27/71	00:05	11/3/11	1,945/76/00:26	16/57
10	*FujitsuDM*	30/97	00:07	7/1/3	7,452/107/01:03	22/73

explored paths, and JPF resources in terms of the memory used (in MB) and time. The last, **Cov**, column gives the run-time coverage over the PTG nodes and transitions as explored by JPF.

The data show that many PTG models contain unreachable pages, ghost pages, and undefined transitions. For example, the *FujitsuPM* PTG contains 11 unreachable pages, all going to the special `Undefined` ghost page, two other ghost pages and 11 undefined transitions. We reported these results to the development team of the application. The team was surprised but confirmed that all errors were real and, possibly, appeared in the sample application after the code refactoring. The team members requested a tool demo and were pleased with the speed of the PTG extractor and the ease of its use. Currently, the tool is being evaluated for possible integration into their testing environment.

The last four columns of Table 2 show the data for the run-time execution of the PTG-based driver, as executed by JPF. The last column shows coverage in terms of the driver's ability to visit pages and execute events. The numbers show that the driver misses some pages and transitions. Manual evaluation shows that most of the missed pages are due to their unreachability from the initial page. The *cart* example shows particularly large amount of unreachable pages. Inspecting the example, we found that the graph contained 2 components, possibly corresponding to the user and the administrator modules. The administrator pages were not reachable from the default initial page of the application. Such issues can be corrected by specifying additional initial pages to the PTG generator.

The driver also misses some transitions due to the limited data values, used to populate `ActionForms`. Since the data population is done fully automatically, using default corner-case values, it is possible for the driver to miss some outcomes of the `execute()` event-handling code. Despite such limitation, the end states number reported by JPF shows that the PTG-based driver is capable of executing hundreds of test sequences within seconds.

5.3 Discussion

In this section, we address our research questions.

PTG Quality: We manually evaluated the extracted PTGs and found no missing transitions with respect to the encodings specified to the tool. We also manually evaluated all reported errors and found no false warnings, except for the `Undefined` ghost page, which is used by the model to visualize the destination page of the undefined transitions. Most reported ghost pages show up on rare executions outside of normal use case scenarios. However, these errors are reproducible at run-time, when accessing some pages directly, without going to the initial page first (as explained in section 3.4, using the *simregister* example).

Even with possible overapproximations, we believe, the PTG model is useful. It enables automatic checking of possible navigation errors and enables validation of the application under test using the PTG-based driver. While the PTG model may encode more executions than possible at run-time, it can serve as a starting navigation and event-handling model of the application PTG, which can be manually edited with conditions to restrict some of the overapproximations.

Human/Tool Cost: Our experiment consists of three fully automated steps. Steps 2 and 3 can be enhanced with user-supplied properties. However, even without user specifications, our technique is capable of finding suspicious features such as unreachable and ghost pages or undefined transitions. We found several such violations in many case studies. These violations would have been hard to find with traditional testing approaches (e.g., testing or run-time crawling would not uncover unreachable pages).

Threats to Validity: The set of case studies is limited yet representative. Case studies were picked by people not involved with the development of the PTG generation approach. The examples were picked based on their accessibility and the frameworks used for their implementation.

6 Related Work

Many approaches to testing web applications require users to specify requirements (e.g., [7,10]). User specifications can capture behavior based on high-level requirements, not system implementation, which may be faulty. However, they require manual work, which could also be prone to errors. The PTG model extracted in our approach can serve as a starting point for specifications in such approaches.

Run-time analysis can be used to extract the model, while executing the application under test. For example, Memon et al. [11] extract an Event Flow Graph (EFG) while executing a GUI application. The extracted EFG is used to generate sequences of events and feed them back to the GUI application under test. Haydar [8] presents an approach to extract a finite automata model from execution traces. Without user guidance (e.g., specification of user input data), run-time approaches may not be able to visit all possible pages of the application and the extracted models may miss some behavior. To avoid such issues, some approaches require user guidance (e.g., [1,13]).

Static analysis techniques work well for specific domains and extracting specific features. For example, Kubo et al. [9] present an automated technique for extracting page transitions from Struts applications. Unlike in our work, this approach extracts and model checks, with SPIN, page transitions only, whereas in our work, we also extract the information about the event-handling classes and event data, which can be used to drive the underlying event-handling code of the application under test with analysis tools such as JPF. Yuen et al. present web automata [14], a behavioral model for MVC-based applications, similar to PTG. They propose to extend `struts-config.xml` into another XML file that represents an automaton, which can be used to check reachability requirements. However, they do not present automation for their approach.

Recent work on interface discovery for web applications [5,6] uses static analysis [6] and symbolic execution [5] to analyze Java servlets to calculate possible data inputs, represented by sets of input parameters (e.g., `username` and `password`) and their possible values. This analysis is designed specifically for servlets, which usually check user input. In contrast, our analysis focuses on generating sequences of user events allowed by the application's interface. In spite of differences, their approach is complementary and can be used to enhance the PTG `EventData` with interesting user values.

7 Conclusions and Future Work

In this paper, we presented a driver generation approach for analysis of JSP-based web applications that (1) using the application implementation, automatically extracts a Page Transition Graph (PTG) that encodes page transitions, possible user and server events, their corresponding event-handlers and user data (2) visualizes and checks the PTG for presence of ghost pages, undefined transitions, and unreachable pages and (3) generates and executes test sequences with JPF. We evaluated the tool on ten Java applications, including two large industrial applications. We uncovered multiple navigation errors in many case studies, including sample applications from the Fujitsu development teams.

The landscape of frameworks used to develop applications is always changing. In this paper, we used the Struts framework as an example for domain-specific automation support, however, our tools can be extended to handle other frameworks and approaches. We are interested in (1) extending the JSP parsers with parsing of event conditions to prune possible spurious transitions, (2) evaluation of symbolic execution [4] as a technique to enhance generation of event data, (3) extending the driver to run with HttpUnit[12], and (4) combining our static approach for PTG extraction with a dynamic approach (e.g., crawling).

Acknowledgments

We are deeply grateful to the JPF team, especially Willem Visser and Peter Mehlitz, for their constant support with JPF.

[12] `http://httpunit.sourceforge.net/`

References

1. Andrews, A.A., Offutt, J., Alexander, R.T.: Testing web applications by modeling with fsms. Software and Systems Modeling 4, 326–345 (2005)
2. Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder – a second generation of a Java model-checker. In: Proceedings of the Workshop on Advances in Verification (July 2000)
3. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: A system for specification and verification of interactive, data-driven web applications. In: SIGMOD 2006: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 772–774. ACM, New York (2006)
4. Ghosh, I., Rajan, S., Shannon, D., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: Proceedings of the International Workshop on Defects in Large Software Systems (July 2009)
5. Halfond, W.G., Anand, S., Orso, A.: Precise interface identification to improve testing and analysis of web applications. In: ISSTA 2009: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 285–296. ACM, New York (2009)
6. Halfond, W.G.J., Orso, A.: Improving test case generation for web applications using automated interface discovery. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 145–154. ACM, New York (2007)
7. Hallé, S., Ettema, T., Bunch, C., Bultan, T.: Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In: ASE, pp. 235–244 (2010)
8. Haydar, M.: Formal framework for automated analysis and verification of web-based applications. In: ASE 2004: Proceedings of the 19th IEEE International Conference on Automated software Engineering, pp. 410–413. IEEE Computer Society, Washington, DC, USA (2004)
9. Kubo, A., Washizaki, H., Fukazawa, Y.: Automatic extraction and verification of page transitions in a web application. In: APSEC 2007: Proceedings of the 14th Asia-Pacific Software Engineering Conference, pp. 350–357. IEEE Computer Society, Washington, DC, USA (2007)
10. Licata, D.R., Krishnamurthi, S.: Verifying interactive web programs. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 164–173. IEEE Computer Society, Washington, DC, USA (2004)
11. Memon, A., Banerjee, I., Nagarajan, A.: Gui ripping: Reverse engineering of graphical user interfaces for testing. In: WCRE 2003: Proceedings of the 10th Working Conference on Reverse Engineering, p. 260. IEEE Computer Society, Washington, DC, USA (2003)
12. Rajan, S.P., Tkachuk, O., Prasad, M.R., Ghosh, I., Goel, N., Uehara, T.: Weave: Web applications validation environment. In: ICSE Companion. Software Engineering in Practice, vol. 2, pp. 101–111 (2009)
13. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering, pp. 25–34. IEEE Computer Society Press, Washington, DC, USA (2001)
14. Yuen, S., Kato, K., Kato, D., Agusa, K.: Web automata: A behavioral model of web applications based on the mvc model. Information and Media Technologies 1(1), 66–79 (2006)