Automated System Testing of Dynamic Web Applications

Hideo Tanida[1,⋆], Mukul R. Prasad[2], Sreeranga P. Rajan[2], and Masahiro Fujita[1]

[1]The University of Tokyo, Tokyo, Japan
[2]Fujitsu Laboratories of America, Sunnyvale, CA, U.S.A.
tanida@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp,
{mukul.prasad,sree.rajan}@us.fujitsu.com

Abstract. Web applications pervade all aspects of human activity today. Rapid growth in the scope, penetration and user-base of web applications, over the past decade, has meant that web applications are substantially bigger, more complex and sophisticated than ever before. This places even more demands on the validation process for web applications. This paper presents an automated approach for the system testing of modern, industrial strength dynamic web applications, where a combination of dynamic crawling-based model generation and back-end model checking is used to comprehensively validate the navigation behavior of the web application. We present several case studies to validate the proposed approach on real-world web applications. Our evaluation demonstrates that the proposed approach is not only practical in the context of applications of such size and complexity but can provide greater automation and better coverage than current industrial validation practices based on manual testing.

Keywords: Dynamic analysis, Validation, Web application.

1 Introduction

Web applications are ubiquitous today. The last decade has witnessed rapid growth in both the scope and the penetration of web applications. On one hand, the wide-scale adoption of web applications in all spheres of human activity has brought validation and quality assurance of such applications into focus. On the other hand, the development of WEB 2.0 technologies such as AJAX (Asynchronous JavaScript and XML) and Flash has resulted in feature-rich and highly interactive web applications, which are even more difficult to validate.

Current industrial practice for the functional validation of web applications still continues to largely rely on manually written test cases which exercise and check the application behavior one trace at a time. There is a growing gap between the coverage, automation and scalability of traditional testing-based validation methodologies and the validation requirements of modern WEB 2.0 applications, which has been acknowledged by validation researchers and practitioners alike. Research on automated

⋆ This author was a research intern at Fujitsu Laboratories of America, when this work was done.

M.J. Escalona, J. Cordeiro, and B. Shishkov (Eds.): ICSOFT 2011, CCIS 303, pp. 181–196, 2013.

model generation [9], model-based testing [10,8,7,3], and model checking [1] offers the promise to address this gap. Specifically, there has been a recent work on automated model generation [9] and model-based testing [10] of AJAX applications that looks especially promising.

This paper addresses the problem of developing a better and practical validation solution for WEB 2.0 application development. We propose a methodology for functional validation of WEB 2.0 AJAX applications that is based on an efficacious combination of some previously proposed techniques in the validation literature and our own novel extensions to these techniques. We present several case studies of applying this methodology to the validation of WEB 2.0 applications. The main objectives and contributions of this paper are as follows:

- We propose a solution for automated system testing of modern industrial-strength dynamic web applications. This solution employs a combination of automated dynamic crawling to extract a model of the navigation behavior of the web application and model checking techniques to check this model for various properties of interest.
- We extend and adapt the dynamic crawling and model checking techniques in several novel ways to fit our application domain and to ensure that the validation solution is simple, scalable, automated and applicable in an industrial context. We believe our solution is fairly complementary to current industrial practices of web application validation and at least in some respects, superior to them.
- We present several case studies of applying the proposed approach to the validation of real web applications and report on both the successes and short-comings of our proposed approach. We feel these lessons would be vital in developing and delivering the next generation of industrial practices for web application validation.

The rest of the paper is organized as follows. In the next section we survey related work. Section 4 presents our proposed validation approach. In Section 5 we describe the implementation of this approach. Section 6 presents three case studies evaluating our approach on real-world web applications, followed by a discussion of the lessons learnt in Section 7. We summarize and conclude the paper in Section 8.

2 Related Work

This work is aimed at system testing of dynamic web applications and specifically, validating the navigational aspects of their behavior. The vast body of research on web application validation spans several other important areas such as validation of the server-tiers of web applications, or that of security or performance aspects of the behavior. Nevertheless, these areas are beyond the scope of this paper and are therefore not surveyed in this section.

Current industrial practice for system testing of web applications primarily involves the use of capture-replay tools such as Selenium [1], WebKing [2] and Sahi [3]. Using these

[1] http://seleniumhq.org/

[2] http://www.parasoft.com/jsp/solutions/soa_solution.jsp?itemId=86

[3] http://sahi.co.in/w/

frameworks, users manually exercise the application through various test scenarios, one at a time. These actions are recorded by the tool and can be replayed back at a later time, usually with user-defined assertions expressing expected behavior, inserted at various steps. This mode of validation, however, requires a substantial amount of manual effort.

Our proposed method for system testing of dynamic web applications involves extracting a state-based navigation model of the web application behavior by automatically crawling the deployed web application. This model is then checked against a temporal logic specification, represented as a set of properties, using model checking [4] techniques. There are several works which overlap with one or more aspects of our approach but differ in other respects.

The Target Applications. Several previous works [12,3,2] target traditional (WEB 1.0) web applications, employing some form of automatic crawling to extract a navigation model for validation. However, as also pointed out by others [10,8], the crawlers used there would not be applicable to WEB 2.0 applications. Further, the nature and scope of the model extracted from traditional web applications as well the properties to be validated on them would differ substantially from those of the dynamic web applications we target. This is also true of previous work on GUI Application testing by Memon *et al.* [13], which, while qualitatively similar in many respects, cannot be directly applied to our application domain. The tool MCWEB [1] is one of the few instances of the direct application of model checking to web application navigation behavior. However, the work was also targeted towards WEB 1.0 applications. Further, the lack of support for automated model extraction and the use of *μ-calculus* for specifying properties makes the tool difficult to use for non-formalists. Our approach emphasizes automation, scalability and ease of use in an industrial setting.

The Verification Methodology. Almost all previous papers rely on trace-by-trace testing as the end means to validate the behavioral model. The authors of [10,8] automatically generate test-benches which exercise one trace at a time from the model. While this definitely increases the level of automation compared to the current industrial practice of manually written test-cases, the underlying validation is still test-case driven and hence the requirements and their checking very trace-specific. We submit that our approach, which is based on model checking, is much more natural, given that we have a pre-generated navigation model. Further, we can pose and check more general and global properties of the application. We present several instances of this and the advantages it provides, in Section 6.

3 Background

In this section we review some technologies that form the foundations of our approach for web application validation.

3.1 Automated Crawling of AJAX Applications

We use the technique proposed in the CRAWLJAX work [9] as the basis for exploring the behavior of the web application under test. CRAWLJAX is a tool for automatically

exploring the dynamic state space of modern web applications. It is capable of interacting with the client-side code of a web application through programmatic interfaces that are available for most of the popular web browsers. CRAWLJAX analyzes a web page to detect widgets to click on (clickables), and systematically exercises them to explore dynamic web application behavior. Changes in the dynamic DOM tree of the page are detected and recorded as new states of the behavior. By systematically detecting new states and executing clickables on them the crawler is able to build a finite-state model of the navigation behavior of the web application. CRAWLJAX provides a set of options to configure the crawling behavior. For example, the set of widgets to click on or not click on, during crawling, can be specified. For more details about the algorithms, architecture and features of CRAWLJAX the interested reader is referred to [9]. We have extended the basic CRAWLJAX in several ways for the purposes of this work. These extensions are discussed in Section 4.

3.2 Model Checking

Model Checking [4] is a set of automated techniques for checking if the behavior of a hardware or software system satisfies a certain property. This is typically done by extracting a finite-state abstraction M, of the relevant behavior of the system under test. The property to be checked is expressed as a logical formula f in a *temporal logic*. Subsequently, model checking algorithms are applied to check if M satisfies f. A temporal logic is a formalism for expressing sequential properties of dynamic systems, for the purposes of automated reasoning through techniques such as model checking. There are several temporal logics that have been proposed in the literature, varying in their syntax, expressive power as well as the complexity of the model checking algorithms that work on them. CTL (computation tree logic) and LTL (linear temporal logic) are the two most popular ones. Our property checking approach, proposed this paper, is a simplification of traditional temporal logic model checking.

4 Proposed Method

This work addresses the validation of modern, *dynamic* applications. These applications usually support a feature-rich, highly-interactive, client-side user-interface, typically by the use of technologies such as AJAX and Flash. Further, this work focuses on validating the navigational aspects of the behavior of such application (as opposed to, for example, the performance, security or concurrency aspects of the behavior).

Our overall approach is a two step process. The first step is to extract a finite-state model of the navigation behavior of the web application. This is done by automatically crawling the web application in the style of CRAWLJAX and capturing the observed behavior as a navigation model. In order to facilitate a more comprehensive, yet scalable crawling behavior, applicable to industrial-strength applications, we propose an extension to CRAWLJAX's basic crawling, called *guided crawling*. This is described in Section 4.1. The second step is to validate various functional requirements of the application against the navigation model. To do this we employ a variant of traditional temporal logic model checking, called template-based property checking. This is described in Section 4.2.

This two-step approach has several advantages. First, it allows us to isolate the relatively expensive crawling step from the actual validation and do the crawling once (or a few times) in a mostly requirement and validation independent manner. Second, it allows us to extract a compact model of the navigation behavior, by a judicious choice of the model representation as well as by discarding irrelevant application-level details during crawling. This accelerates both the crawling and the downstream model checking. Third, since all the requirements to be checked are often not known during the initial stages of validation, performing the validation offline obviates the need to repeat the expensive crawling step.

4.1 Model Generation

The Navigation Model. The navigation model represents the structure and screen content observed by the crawler while dynamically crawling the web application. It is comprised of a *state transition graph (STG)*, representing the topology of the crawled behavior, as well as the content of each crawled state.

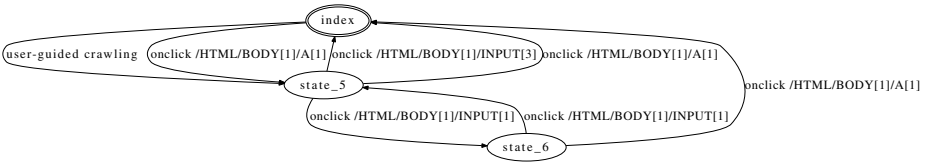

Fig. 1. Graphical view of a State Transition Graph

Fig.1 shows an example of an STG. An STG is a labelled directed graph where each node (state) corresponds to a web page viewable on a web browser. Each state is represented by the DOM (document object model) of its corresponding web page, in the navigation model. A change in the DOM of a web page, typically through the execution of a user action (such as a button click), constitutes a new state. Edges in the STG represent transitions from one state to another and are typically labelled with the user action (e.g., a `click`) and the XPath of the element on which it was executed. This is the case with most edges in Fig.1. Some edges are labelled "user-guided crawling" and correspond to a transition made by a sequence operations from a *guidance directive*. Guided crawling and guidance directives are discussed in the next section.

Note that some web applications may, theoretically, have an infinite state space (for example based on infinite different sets of data inputs). However, due to obvious practical constraints of crawling time and navigation model size we only crawl and validate a finite but behavior-rich subset of the state space of a given web application. A finite state model is also a requirement of the downstream property checking algorithms.

Guided Crawling. The CRAWLJAX tool [9] offers users some control over the crawled behavior by specifying the overall set of widgets to click or not click during crawling. Users can also specify one or more sets of data for each HTML `<form/>` element encountered during crawling. However, the crawling of real-life enterprise web applications often requires a more tighter control over the crawling, for example, in the following practical scenarios.

1. *User Authentication:* This is a common requirement in several web application interactions, when viewing confidential information or executing transactions. However, typical web application interactions are a complex mix of unauthenticated and authenticated behavior, with authentication being activated under specific scenarios (*e.g.* some applications like `Amazon.com` do not require a login till the checkout stage).
2. *Form Data Filling:* HTML forms are commonplace in modern web applications (a user authentication panel is a special instance of this). CRAWLJAX allows form-data filling but always fills a given form with the same data (or data-sets). For more intelligent crawling, it would be desirable to fill a given form with one of several data sets driven by the scenario being navigated.
3. *Excluding Behavior from the Model:* When crawling real web applications, the crawling time as well as the size of the crawled model needs to be managed, according to available computation resources. One strategy is to surgically exclude features and crawl scenarios outside the scope of the ensuing validation, from the crawling.

It is very difficult, if not impossible, to adequately service the above scenarios (and many others), using the default crawling controls provided by CRAWLJAX. We propose a technique called *guided crawling* to provide the user with more direct control over the crawled behavior. It allows the user to specify scenario-based desired crawling behavior. This is done by creating one or more *guidance directives*, specific to the target application being crawled. The crawling alternates between the fully automatic default crawling and the scenario-specific behavior specified by the guidance directives.

Definition 1 (Guidance Directive). *A Guidance Directive $\mathcal{G} = (p, \mathcal{A})$ is an ordered pair that consists of a predicate p that is evaluated on a web application state, and an action sequence $\mathcal{A}$. $\mathcal{A} = (\alpha_1, \alpha_2 \ldots, \alpha_k)$ is a sequence of atomic actions α_i. Each atomic action $\alpha = (e, u, \mathcal{D})$ is a triple consisting of a DOM element e, a user-action u and a set of data-instances $\mathcal{D}$ (potentially empty) associated with u.*

As per Definition 1, a guidance directive $\mathcal{G}$, includes the predicate p that determines when $\mathcal{G}$ should be activated. p is evaluated on the current state of the web application, during crawling, i.e., on the DOM of the current page loaded in the web browser. If p is `true` in the current state, the crawling action sequence $\mathcal{A}$ is executed on the web application. Each atomic action α in $\mathcal{A}$ is a simple (browser-based) user action u on a particular DOM element e on the current web-page/screen. For example, u could be a click and e could be a button. Such actions have no associated data. Hence, $\mathcal{D} = \emptyset$ (the empty set) in this case. Another example of an action would be selecting an option from a `<select/>` element or assigning a string value to an `<input/>` element etc. In these cases $\mathcal{D}$ would be the set of data values to exercise the element with.

Algorithm 1 presents the pseudo code for our model generation, incorporating guided crawling. The main procedure, `GuidedCrawl` accepts a target web application, W and a set of associated guidance directives, $\mathcal{G}^{set}$. It initializes the navigation model M, loads the initial web-page (InitPage) of W in the web browser and invokes procedure *GuidedCrawlFromState* on it. *GuidedCrawlFromState()* does the actual crawling and recursively calls itself on new successor states. *GuidedCrawlFromState()* starts with a check

(*IsVisited(S)*) to see if state S has been visited by a previous invocation of *GuidedCrawlFromState*. This check accounts for any specified state-abstractions (explained in Section 19). If so, the crawling returns back to calling state. If not, *MarkVisited()* marks state S as visited, to exclude it from future guided crawls, and *AddState()* records S in the navigation model M as a newly discovered state. Next, the state S is analyzed to compute the set of actions (*Actions*), to execute on it, to continue the crawling. First, function *FindActions()* (line 7) computes the set of basic (non-guided) user actions, which can be executed on it, based on the clickables specified to the crawler. Next the set of guidance directives, $\mathcal{G}^{set}$ is processed to find additional actions to execute on S (lines $8-11$). For each guidance directive $\mathcal{G}$ that can be activated on S (line 9) function *ComputeActionSequences()* computes concrete action sequences of actions from $\mathcal{G}$ by picking specific data values in its constituent atomic actions α. All possible sequences that can be created by various choices of the specified data-values are constructed and added to the *Actions* set. Subsequently, each action a in *Actions* is fired on S (lines $13-18$). *Execute()* Executes the action (or action sequence) a on W to discover a next state (nextState) and *AddTransition()* records this transition in model M. *GuidedCrawlFromState()* is then recursively called on *nextState* (line 14). *UndoTransition()* functionally reverses the transition $S \rightarrow nextState$ on W to restore it to state S.

Model Reduction. The size of the navigation model has a direct bearing on the efficiency of the property checking. We employ the following two features, to derive a compact and meaningful model for validation.

1. *Specifying User Events:* CRAWLJAX provides several mechanisms for specifying the set of widgets to be exercised (or excluded) during crawling. Our guided crawling technique further supplements these mechanisms. The specification is done by the user on application-specific basis, as an input to the model generation step.
2. *State Abstraction:* Since the crawler uses the screen DOM as the unique identifier for a state, identical looking screens can often be mapped to different states in the model because of slight differences in their DOMs. This can happen because of entities such as visit counters or date/time stamps, included in the DOM or even minor differences in white-spacing or the attribute order in dynamically generated web-pages. Therefore, we have implemented a state abstraction technique which accepts a set of user-given XPaths and removes all matching elements and their descendents from the DOM tree of each state and uses the resulting abstracted DOM as the state identifier, specifically in determining equivalence of two states. This technique is implemented within the *IsVisited()* function in Algorithm 1[4].

Our experience regarding the specific use of these features, in the light of our case studies, is discussed in Section 6.

4.2 Model Validation

As mentioned in Section 2 one of the key differences between our approach and prior art in this area is that while other approaches resort trace-by-trace checking of the behavior

[4] CRAWLJAX provides a similar, albeit independently developed, mechanism called *oracle comparators* for state abstraction.

Algorithm 1. Guided Crawling

```
/* GuidedCrawl(W, G^set) -- main procedure */
Input  : W: Web application under test
         G^set: Set of guidance directives
Output: M: Crawled navigation model
1  begin
2      M = ∅
3      InitPage ← LoadBrowser(W)
4      GuidedCrawlFromState(InitPage, M)
5      return M
6  end
```

```
/* GuidedCrawlFromState(S, W, G^set, M) */
Input  : S: Current state for guided crawling
         W: Web application under test
         G^set: Set of guidance directives
         M: Navigation model being built
1  begin
2      if IsVisited(S) then
3          return
4      end
5      MarkVisited(S)
6      AddState(S,M)
7      Actions ← FindActions(S)
8      foreach G(p,A) ∈ G^set do
9          if p(S) = true then
10             Actions ← Actions ∪ ComputeActionSequences(A)
11         end
12     end
13     foreach a ∈ Actions do
14         nextState ← Execute(a,W,S)
15         AddTransition(nextState, S,M)
16         GuidedCrawlFromState(nextState)
17         UndoTransition(a,W,S)
18     end
19 end
```

a la traditional testing, we propose to check the navigation model as a whole using the formal technique of model checking [4]. The use of a pre-generated, finite state navigation model makes the application of model checking both easy and very efficient. We claim that the expected navigational behavior of web applications can be quite naturally expressed as properties in temporal logic [4], the input language of model checkers. In the following we present a few examples of such classes of requirements and specific instances in each class.

1. *Screen Sequence/Transition Requirements:* The simplest and most common check on web applications is of the form: *A user input i with the web application on Screen **A** takes it to Screen **B***. Here, screens ***A*** and ***B*** may be screens or pages of the web application, identified by the presence or absence of certain features, widgets or DOM elements, while input *i* may be a simple input like a mouseover or button/link click or a more complicated sequence of such actions interspersed with data inputs to various widgets on the screen. This kind of requirement may be further generalized in checking (for example) that Screen ***B*** follows ***A*** in one, all or none of the valid execution sequences of the web application. Some specific examples of this class of requirements could be:
   - In a web application with user authentication: *The* `LOGOUT` *screen is always preceded by the* `LOGIN` *screen*
   - On a utilities web-site under the bill-payment section: *If the* `CONFIRM` *button is clicked on the* `PAYMENT-DETAILS` screen then the next screen is always the `RECEIPT` screen.
2. *Global Navigation Structure or Usability Requirements:* This kind of requirement typically apply to the overall structure of the web application's navigation behavior. Thus, they are by their very intent, *global* and ideally suited for checking on a single, consolidated navigation model (versus conventional trace-by-trace checking). Some examples of requirements in this class are:
   - *All features of are accessible within* 5 *clicks, starting from the home page*
   - *The initial page is accessible from every screen*

Temporal Property Templates. Since each screen of the web application and all features, widgets or DOM elements on each screen are represented as part of our navigation model, it would in theory be possible to express our requirements as properties in a temporal logic [4] such CTL (Computation Tree Logic) or LTL (Linear Temporal Logic). With minor modifications to our navigation model, these properties could then be checked on it using one of the commonly used model checkers, e.g., NuSMV [11].

However, writing properties in temporal logic is quite difficult, error-prone and unintuitive for non-formalists, such as the average software developer or quality assurance engineer. Further, as demonstrated in a study by Dwyer *et al.* [5] most validation requirements observed in practice, can be captured by properties in a limited number of temporal classes. A temporal class refers to a set of temporal logic formulas that share a common temporal structure and differ only in the propositional expressions within this temporal structure. For example, the LTL temporal formulas $G(a \wedge b)$ and $G(\neg p_1)$ share the common temporal structure $G(exp)$, differing only in the value of expression exp.

Composing Properties. Motivated by the above arguments, our approach uses a set of temporal property templates, rather than a complete temporal logic, for specifying properties. For the purposes of this work, we used the three templates shown in Fig.2. Here p,p_1 and p_2 are expressions which are evaluable in a given single state, based on the contents of that state. They are essentially assertions about the state of different DOM elements on the page or relationships between them, very similar to assertions used in

1. *Global Template:: $G(p)$:* Globally p is *true*
2. *Screen Transition Template:: $p_1, i \rightarrow p_2$:* After transiting from a state where p_1 is *true*, with an input or a guidance-directive-driven input sequence which matches i, or with any input ($i = nil$), p_2 is always satisfied
3. *Precedence Template:: $p_1 \rightarrow P p_2$:* Need to reach a state where p_2 is *true*, prior to reach a state where p_1 is *true*

Fig. 2. Temporal property templates used

existing automatic web application system testing frameworks such as Selenium. Therefore, it is quite easy for developers in field to migrate to our methodology. Examples of such expressions include availability of a node with given type and attribute, and availability of text content which matches a given regular expression.

Our Global Template is essentially an assertion p that would be checked on each state of the STG. A counter-example to this would be a state S where p is *false*. The Screen Transition Template relates to a pair of neighboring states S_1, S_2 and a user action i that caused the transition between them. i is a tuple of a user event (e.g., a click) and a unique identifier of the DOM element on which it was performed. This template is particularly useful to check the function of a specific widget, a very common testing scenario. A counter-example to this property would be a tuple (S_1,T,S_2), where S_1 is a state in which p_1 is *true*, S_2 is a state in which p_2 is *false*, and T is a transition from S_1 to S_2 made by an input or a guidance-directive-driven input sequence which matches i. The last template, the Precedence Template, is intended to check more global relationships in the navigation structure of the web application, for example, that a `logout` event in a web application is always preceded by a `login` event. A counter-example for this template would be a sequence of transitions which starting from the initial state, reaches a state where p_1 is *true*, without going through any state where p_2 is *true*.

Model Checking Algorithm. Rather than using a general purpose model checking algorithm for one of the common temporal logics (e.g. CTL or LTL) we found it more efficient to implement the actual model checking itself through a set of state-traversal checkers, one for each template. It is easy to see that any property expressed using the templates in Fig. 2 can be checked using a single, linear-time traversal of the STG in the navigation model. This also gives a flexible and extensible model checker, to which more templates can be easily added in future or existing ones modified to fit practical situations.

5 Tool Implementation

We have implemented our proposed validation approach in a system, which is comprised of two components. One of the components of the system is an extended version of open-source web application crawler CRAWLJAX. The second is a custom model checker called GOLIATH, which supports checking of template-based properties on the navigation model generated by the crawler, as explained in Section 4.2.

Model Generation. Our principal extension to CRAWLJAX is an implementation of the guided crawling feature described in Section 4.1. Our extended crawler supports Java

APIs for instantiating an object encoding a guidance directive (as per Definition 1). The user can instantiate one or more such application specific guidance directives and add them to the driver for crawling a specific target web application.

Another significant extension is a modification to CRAWLJAX's behavior discovery mechanism, to ensure that all behavior reachable up to a specified crawling depth is included in the model, regardless of the order of firing user events. This was previously not the case.

Model Validation. Our model checker, GOLIATH is implemented in Ruby and accepts the crawled navigation model as well as a set of properties formulated in terms of the property templates of Fig.2, with the expressions (e.g. (p, p_1 and p_2 in Fig.2) specified as Ruby expressions. These expressions can refer to specific DOM elements using the standard DOM API. We use the Nokogiri [5] HTML parser in our implementation, to parse and interpret such references.

The following is a very simple instance of a supported expression, which is true if and only if there is some `<a/>` element with `id` attribute value `login`. Here `doc` refers to the `document` DOM object of the page.

```
doc.xpath('//a[@id="login"]').any?
```

A little more complex and useful expression instances can be constructed as the following sequence of expressions:

```
login_xpath = '//a[@id="login"]';
logout_xpath = '//a[@id="logout"]';
login_avail = doc.xpath(login_xpath);
logout_avail = doc.xpath(logout_xpath);
login_avail.any? != logout_avail.any?
```

The last expression (hereafter referred to as $p_{not\_together}$) is *true* only in states where precisely one of `login`/`logout` button exists. Note that in Ruby, sequences of expressions can be evaluated as an expression which yields the value of the final expression. Thus, the property $G(p_{not\_together})$, composed using the Global Template, checks that there is no state in the extracted navigation model, where both the `login` and `logout` buttons simultaneously exist.

6 Case Study

The proposed method has been implemented and evaluated by applying them to applications including an industrial one as well as an open source well-known application. To assess the efficacy and utility of our approach and the corresponding implemented tool, we have conducted a number of case studies following guidelines from [6]. All experiments are performed on a workstation with Intel Xeon CPU W5590@3.33GHz.

[5] `http://nokogiri.org/`

6.1 Subject Web Applications

We conducted our case studies on three web applications. The first subject (ORGANIZER) is a schedule organizer application, called **MyOrganizer**, taken from a textbook [14] on AJAX-based web application development. It is composed of 8,004 lines of Java code, 2,885 lines of JavaScript code, and 1,137 lines of JSP code. The second web application (BPM) is a commercial business process manager comprised of 58,701 lines of Java code, 61,541 lines of JavaScript code, and 90,742 lines of JSP code. It uses the YUI[6] AJAX library. The last subject (REDMINE) is the free and open-source, project management and bug-tracking tool **Redmine** (v1.2.1). It is composed of 66,778 lines of Ruby code, 18,402 lines of JavaScript code, and 5,751 lines of RHTML code. It is implemented based on the Ruby on Rails [7] framework.

6.2 Model Generation

Appropriate guidance directives, which are the keys for meaningful and relatively quick model generations, are given to generate the corresponding screen transition diagrams for the three test subjects. The directives include operations for logging in by providing usernames and passwords whenever there is a screen image having login prompts, or operations for creation of data entries such as bug reports in Redmine at appropriate stages during crawling.

Model reduction techniques are used with all of the applications to allow generation of models in reasonable time. Namely, elements to be clicked during fully automatic crawling are specified based on validation requirements of each application. In addition, for ORGANIZER, we have employed a state abstraction technique to ignore changes on element attribute from `mouseover` events on elements.

The server-tiers of the applications are also controlled to restore their state every time the crawler goes back to the initial page to achieve extraction of models with higher accuracy. For ORGANIZER, a hook to ensure the availability of the user account to be used during model generation is used. For REDMINE, a hook to revert backend database to the state with one normal user created from its initial installation is employed.

Table 1 shows the model (screen transition diagram) generation configurations and results for the examples. Please note loops are excluded on obtaining #path and maximum depth. Although we also tried to generate tests using the approach shown in [10], it did not finish generations for neither of the examples, simply because there are so many cases from its exhaustive analysis even with the depth limit.

6.3 Model Checking

We prepared multiple properties for each of benchmarks. Table 2 contains number of properties for the benchmarks categorized by their temporal templates and verification results, *i.e.* satisfied or unsatisfied. The table also contains maximum / average / minimum *check count* observed during model checking . *Check counts* are provided with the definitions shown in Fig.3 for each of temporal property templates.

[6] `http://developer.yahoo.com/yui/`
[7] `http://rubyonrails.org`

Table 1. Model generation configurations and results

	Crawling depth	#Guidance direc.	#State	#Transition	#Path	Max. depth	Avg. depth	Time (min.)
ORGANIZER	11	1	38	232	65	13	6.385	133
BPM	8	1	765	4039	830	41	22.161	2769
REDMINE	11	7	1580	2528	1220	15	12.937	2634

Table 2. Model checking configurations and results

	$G(p)$			$p_1, i \rightarrow p_2$			$p_1 \rightarrow Pp_2$		
	Sat. prop.	Unsat. prop.	#Check	Sat. prop.	Unsat. prop.	#Check max/avg/min.	Sat. prop.	Unsat. prop.	#Check max/avg/min
ORGANIZER	0	0	0	0	7	23/20.4/5	0	0	0
BPM	9	0	765	2	0	113/61.5/10	2	0	13504317615/ 6752159070/525
REDMINE	2	1	1580	3	2	4/2.2/1	10	0	294/154.4/2

1. $G(p)$*:* Number of states within the model
2. $p_1, i \rightarrow p_2$*:* Number of transitions made by inputs or guidance-directive-driven input sequences which matches i, from states which make p_1 *true*.
3. $p_1 \rightarrow Pp_2$*:* Number of transition sequence to states which make p_1 *true* from states which makes p_2 *true*

Fig. 3. Check count for the property templates

All 7 properties for ORGANIZER including Prop. 1a shown in Fig.4, which are all in form of $p_1, i \rightarrow p_2$ are violated and yielded counter examples. Playing back input sequences to reach and activate the counter examples, actually results in transitions which do not meet the properties. Execution of the counter examples for 5 of the properties generate an error message dialog in client-side web browser and a record indicating "Null Pointer Exception" in the log of Java server software. On execution of counter example for another property, state transition on a user input does not occur. Execution of counter example for the other property results in a state which does not satisfy post condition p_2. They are real bugs in a program shown in the textbook.

Prop. 1a.

```
doc.xpath('//img[@id="dayAtAGlance"]').any? , //img[@id="dayAtAGlance"]:onclick ->
doc.xpath('//img[@src="img/head_dayAtAGlance.gif"]').any?
```

Prop. 3a.

```
G (["home","projects","help"].map { |c|
 doc.xpath('//a[@class="' + c + '"]').any? }.all? )
```

Prop. 3b.

```
doc.xpath('//input[@value="Create"]').any? &&
doc.xpath('//input[@id="issue_subject" and @value and (0<string-length(@value))]').any? &&
doc.xpath('//input[@id="issue_parent_issue_id" and @value="10"]').any? ,
//input[@value='Create']:onclick ->
doc.xpath('//a[contains(@class,"issue") and contains(text(),"10")]').any? ||
doc.xpath('//div[@class="errorExplanation"]').any?
```

Fig. 4. Properties used in the experiments.

All 13 properties for BPM are satisfied. *Check counts* while model checking properties in temporal form of $p_1 \rightarrow Pp_2$, are large due to large number of traces to navigate

back to states which make p_2 *true* from states which make p_1 *true*. It is impossible to track through all of the traces in test-based approaches.

In REDMINE, Prop. 3a shown in Fig.4 which is in the form of $G(p)$ is violated. The property is violated in a state where the web browser is showing a file in a special format. 2 properties in the form of $p_1, i \rightarrow p_2$ are also violated in REDMINE. The violated properties including the one shown as Prop 3b , require the application to transit to a DOM page containing a reference to a ticket (bug report etc.) or to a page indicating an error, from a form with a reference to the ticket input by the user, on "Create" button click. However, the application proceeds without any error even if the referred ticket does not exist, and the resulting pages do not contain any reference to the ticket specified.

Each checking of properties in the experiments finished within 10 seconds. As you can see from the results, once the model is generated, model checking is relatively very quick, and various properties can be examined with reasonable time even for large examples.

7 Discussion

Completeness. Since our technique relies on the finite state navigation model extracted by the crawler, erroneous behaviors not included in this model cannot be exposed by the subsequent model checking step. Although we attempt to capture the largest possible set of relevant behaviors, through crawler enhancements like guided crawling, and by the judicious choice of clickable widgets and crawling depth, the model generation step is inherently incomplete and in practice limited by computation resources.

However, compared to prevailing industrial practices of using trace-by-trace testing based on manually written tests, our technique is able to automatically explore and test a significantly larger set of behaviors and thereby expose many more errors.

Scope. Like other black-box validation techniques, in order to detect errors, our technique requires erroneous behaviors to be propagated and observable at the user interface of the web application, i.e., on the client-tier content displayed on the web browser. In other words, the technique cannot *directly* detect problematic server-tier behaviors, such as perhaps those pertaining to security or performance aspects of behavior. However, it is an ideal fit for testing validating behavior, which typically manifests at the client-tier.

Further, compared to other black-box system testing techniques for web applications, namely those based on user-given test cases and assertions, our approach which makes use of more expressive user-given guidance directives and temporal properties can target extensive classes of behavior.

Automation Level. Our model generation technique require the user to specify a crawler configuration, which is composed of set of elements to be clicked during fully-automatic crawling, state abstraction configuration to ignore some part of DOM page, and a set of guidance directives to partially control the crawling behavior. Our model checking technique is supplied from the user with the properties to be checked. User inputs required for the validation is small, considering large number of execution traces covered with a single configuration, as observed in our case studies.

Scalability. Our case studies confirm that our technique is applicable to large real-world applications in real use. However, our experience also showed that the model generation (crawling) time far exceeds the model checking time. Most of time required for the crawling is due to the communication latency between the crawler and the target application. As the computation burden of the process is relatively small, parallelizing and distributing the crawling is an attractive option for reducing the crawling latency.

Model reduction techniques (for example those described in Section 19) are another option for pruning the state-space and hence the crawling time. We did employ these techniques in our case studies, but to a limited extent. Although more aggressive model reduction, based on the properties to be verified, could reduce the model generation time further, it needs to be implemented and confirmed through more case studies.

Correctness. Since our crawling only observes the client-tier of the web application, the state computed and recorded by it is actually an abstraction of the true system state (which should include the state of the server tiers as well). Thus, the STG computed by the crawler represents an over-approximation of the "'true"' possible set of traces of the web application. Thus, it is theoretically possible that an error trace produced by our model checking is not reproducible on the actual web application, i.e. a false positive.

However, all the counter examples reported in our case-study are confirmed to reproduce. Reproducibility of counter examples is expected to depend also on the degree of abstraction, which is expected be useful to scalability of the technique.

Threats to Validity. We have discussed some of issues related to the external validity of our evaluation in the discussions above. The internal validity of our evaluation may depend on implementation of software tools used. We have minimized the chance by making use of test sets for the tools, which are completely separated from benchmarks used in the evaluation.

8 Conclusions and Future Work

We have proposed a new approach for the automated system testing of modern, dynamic web applications. Our method employs automatic crawling to extract a finite state navigation model of the web application behavior. The user authors a set of properties, expressing desired navigation behavior, using a simple and intuitive template-based specification language. These are then efficiently checked on the extracted model. Our experience with this approach, through several case studies, confirms that it both applicable to industrial strength applications, as well as superior to current industrial practice based on manual testing.

Future works include more verification trials with larger applications for more robust evaluations of the proposed techniques as well as their extensions.

References

1. de Alfaro, L.: Model Checking the World Wide Web. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 337–349. Springer, Heidelberg (2001)

2. Andrews, A.A., Offutt, J., Alexander, R.T.: Testing Web Applications by Modeling with FSMs. Software and Systems Modeling 4, 326–345 (2005)
3. Benedikt, M., Freire, J., Godefroid, P.: VeriWeb: Automatically Testing Dynamic Web Sites. In: Proceedings of 11th International World Wide Web Conference (2002)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420. ACM, New York (1999)
6. Kitchenham, B., Pickard, L., Pfleeger, S.L.: Case Studies for Method and Tool Evaluation. IEEE Softw. 12(4), 52–62 (1995)
7. Marchetto, A., Ricca, F., Tonella, P.: A Case-Study Based Comparison of Web Testing Techniques Applied to AJAX Web Applications. International Journal on Software Tools for Technology Transfer (STTT) 10(6), 477–492 (2008)
8. Marchetto, A., Tonella, P., Ricca, F.: State-Based Testing of Ajax Web Applications. In: ICST 2008: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp. 121–130. IEEE Computer Society, Washington, DC (2008)
9. Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling AJAX by Inferring User Interface State Changes. In: ICWE 2008: Proceedings of the 2008 Eighth International Conference on Web Engineering, pp. 122–134. IEEE Computer Society, Washington, DC (2008)
10. Mesbah, A., Deursen, A.V.: Invariant-Based Automatic Testing of AJAX User Interfaces. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009 (May 2009)
11. NuSMV, `http://nusmv.irst.itc.it/`
12. Ricca, F., Tonella, P.: Analysis and Testing of Web Applications. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, pp. 25–34. IEEE Computer Society (2001)
13. Strecker, J., Memon, A.M.: Testing Graphical User Interfaces. In: Encyclopedia of Information Science and Technology, 2nd edn. IGI Global (2009)
14. Zammetti, F.: Practical Ajax Projects with Java Technology. Apress (2006)