CLIENT-TIER VALIDATION OF DYNAMIC WEB APPLICATIONS

Hideo Tanida[1], Masahiro Fujita[2], Mukul Prasad[3] and Sreeranga P. Rajan[3]
[1]*Dept. of Electrical Engg. & Information Systems, The University of Tokyo, Tokyo, Japan*
[2]*VLSI Design & Education Center, The University of Tokyo, Tokyo, Japan*
[3]*Trusted Systems Innovation Group, Fujitsu Laboratories of America, Sunnyvale, CA, U.S.A.*

Keywords: Dynamic analysis, Validation, Web application.

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 a case study of the validation of Ajax web Applications, where a combination of dynamic crawling-based model generation and back-end model checking is used to comprehensively validate the client-tier of the web application. Our experience shows that such an 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 testing. A couple of experimental results are presented to show the effectiveness of the proposed approach.

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 business activity has brought validation and quality assurance of such applications into focus. On the other hand, the use of WEB 2.0 technologies such as AJAX (Asynchronous JavaScript and XML) results in feature-rich and highly interactive web applications, thereby further complicating the task of validating such applications.

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 model generation (Mesbah et al., 2008), model-based testing (Mesbah and Deursen, 2009; Marchetto et al., 2008b; Marchetto et al., 2008a; Benedikt et al., 2002), and model checking (Alfaro, 2001) offers the promise to address this gap. Specifically, there has a been recent work on automated model generation (Mesbah et al., 2008) and model-based testing (Mesbah and Deursen, 2009) 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 case studies of applying this methodology to the validation of a WEB 2.0 AJAX web applications. The main objectives and contributions of this paper are as follows:

- We propose a solution for the validation of the client-tier behavior of modern industrial-strength dynamic web applications. This solution is a combination of several previously proposed technologies as well as several novel extensions of our own which we believe are critical to its applicability in an industrial context. Further, this approach is fairly complementary to current industrial practices of web application validation and at least in some respects, superior to these approaches.
- We apply the proposed approach to the validation of WEB 2.0 AJAX applications and report on both the successes and short-comings of our proposed

Tanida H., Fujita M., Prasad M. and P. Rajan S..
CLIENT-TIER VALIDATION OF DYNAMIC WEB APPLICATIONS.
DOI: 10.5220/0003510500860095
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 86-95
ISBN: 978-989-8425-77-5

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 works. Section 3 presents our proposed validation approach. In Section 5 we describe the validation case study, including s description of target applications, the experimental setup, procedure and results. We summarize and conclude the paper in Section 6.

2 RELATED WORK

One of the key objectives of this paper is to serve as a bridge between current industrial practice in web application validation and the research literature on this topic. Thus it is relevant to review related works in these two broad areas. However, the following survey will be restricted to works addressing validation of the client-tier of web applications and specifically that of the navigational aspects of the behavior (in contrast to security or performance aspects). The vast body of research on web application validation spans several other interesting and important areas such as validation targeted at the server-tier(s) of web applications, or the vast area of security validation. Nevertheless, these areas are beyond the scope of this paper.

Current industrial practice for the functional validation of web applications, and specifically for validating the client-tier of dynamic web applications (such as those employing AJAX) primarily involves the use of capture-replay tools such as Selenium [1], WebKing[2] and Sahi[3]. Using these 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 regarding expected behavior, inserted at various steps. This mode of validation, however, requires a substantial amount of manual effort. Frameworks such as JsUnit[4] , can be used to perform unit testing of the JavaScript code in the application. However, this kind of testing is by its very nature very localized, language specific and also manually intensive.

Our proposed method for client-tier validation of dynamic web applications involves creating a state-based model (in our case, a finite state machine) of the client-tier behavior of the web application by automatically and dynamically crawling the deployed web application. This model is then checked against a temporal logic specification (represented as a set of properties), using model checking (Clarke et al., 1999). There are several works which overlap with one or more aspects of our approach but differ in one or more of the following respects.

[1]http://seleniumhq.org/

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

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

[4]http://www.jsunit.net/

2.1 The Target Applications

Many of these papers (Alfaro, 2001; Ricca and Tonella, 2001; Benedikt et al., 2002; Andrews et al., 2005) target traditional (WEB 1.0) web applications, which differ substantially from the dynamic (e.g. AJAX-based) WEB 2.0 applications we target, especially in terms of client-tier behavior and its validation. Specifically, (Ricca and Tonella, 2001; Benedikt et al., 2002; Andrews et al., 2005) do use automatic crawlers to extract a navigational model for validation. However, we concur with the assessment of (Mesbah and Deursen, 2009; Marchetto et al., 2008b) among others that these crawlers would not be applicable to WEB 2.0 applications. VeriWeb (Benedikt et al., 2002) does claim to have some support for client-side scripts but there is not sufficient detail in the paper to conclude if it would apply to modern enterprise AJAX applications. VeriWeb also uses a model checking based verification back-end (VeriSoft) similar to us, but the paper is scant on experimental results on real applications. Another class of relevant work, relevant in this context is the work on GUI Application testing (Strecker and Memon, 2009) where they reverse engineer a model of the desktop GUI application, with the objective of generating test cases. However, while modern dynamic web applications share the rich user-interface and interactivity of desktop GUI applications, they have several unique features as well (such as asynchronous client-server communication and a DOM-based user interface). Thus GUI testing techniques cannot be applied as such to the problem at hand *i.e.* the validation of dynamic web applications.

2.2 The Model Generation Process

Several papers (e.g. (Marchetto et al., 2008b)) rely on a completely or mostly manual specification of the behavioral model being checked. We submit that this would place unreasonable specification burden on the designers and verification would not be practical, especially in an industrial setting. An automatic crawler for AJAX application which we have used as a basis for model extraction is developed (Mesbah et al., 2008). However, as explained in Section 3.1 the crawler needed several enhancements to yield a comprehensive model for industrial applications.

2.3 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 (Mesbah and Deursen, 2009; Marchetto et al., 2008b) automatically generate test-benches which exercise one trace at a time from the model. While this definitely increase 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 navigational 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 5. The tool MCWEB (Alfaro, 2001) is one of the few instances of the direct application of model checking to web application navigational behavior. However, the work was targeted towards WEB 1.0 applications. Another key difference is that in our approach the model generation step is distinctly separated from the subsequent model checking (*i.e.* checking is not done on the fly). Thus, while MCWEB relies on *contractive μ-calculus* we use considerably simpler specification formalisms (a finite state machine for the navigational model and temporal logic model checking).

3 PROPOSED METHOD

This paper deals with the validation of modern *dynamic* WEB 2.0 applications. Such applications are characterized by a feature-rich, client-side user-interface and have a high degree of user interactivity, typically by the use of technologies such as AJAX and Flash. Further, this work focuses on validating the behavior of the *client-tier* of the web application, and specifically the navigational aspects of the behavior.

Our overall approach is a two step process. The first step is to extract a behavioral model of the client-tier of the web application. This is done by dynamically and automatically exercising the web application through a process called *guided crawling*, capturing the resulting behavior and representing it as a finite-state-machine model. Guided crawling is implemented as extension to the CRAWLJAX tool (Mesbah et al., 2008) and is described in Section 3.1.2. The second step is to use *temporal logic model checking* (Clarke et al., 1999) to validate various functional requirements of the application against this model.

This two-step approach has several advantages. First, it allows us to isolate the relatively expensive dynamic crawling 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 navigational behavior, achieving compaction both by a judicious choice of the model representation as well as by discarding irrelevant application-level details during crawling. This accelerates both of the crawling process and the downstream model checking. Third, since all the requirements to be checked are often not known when the first rounds of validation is done, performing the validation offline obviates the need to repeat the expensive model generation step.

3.1 Model Generation

This section describes generation of the model. As mentioned earlier, the behavioral model of the web application client-tier is generated by dynamically, automatically and comprehensively executing the web application and capturing the observed behavior in a mathematical model.

3.1.1 The Navigational Model

The model extracted during the crawling process primarily captures the navigational aspects of the web application's client-tier behavior (as opposed to, for example, behavior specific to its security policies or its performance). Hence we will refer to this model as the *navigational model* in the sequel.

Navigational models take form of FSM (finite state machine), with their internal states defined by content of DHTML (Dynamic HTML) document and transitions representing changes to content of DHTML document before and after user operations. DHTML documents which reflect dynamic changes made by on-browser script and page reloading, represent views in AJAX-based web applications, and are therefore suitable for defining states within our FSM model to verify navigational aspects. The models intended to represent navigational aspects of web application, contain only inter-state transitions before and after user operations. Further, while the web application may theoretically allow for an infinite number of *different* screens or web pages it is neither practical nor particularly useful to attempt to capture, represent or check these infinite screens. Thus we focus on automatically navigating (through our guided crawling) a rich but finite number of screens within the web application, while using innovative state representation and abstraction mechanisms to represent this information. Thus our navigational model can be summarized as follows. We use a state representation to capture the content of each DHTML page/screen visited by the application and a finite state machine (FSM) to represent the navigational behavior of the web application.

Figure 1 shows a graphical representation for a fragment of navigational model corresponding to a simple web application. Each state (node) is annotated by reference to DHTML document it corresponds to, and transition (edge) is annotated by input which has triggered the transition. Note that the each input may be an event on an element which is identified by index-number-based XPath or a sequence of operations performed based on a guidance directive (described in detail in Section 3.1.2). The edge labeled “user-guided crawling”, corresponds to a transition made by operations from a guidance directive.

3.1.2 Guided Crawling

The CRAWLJAX tool (Mesbah et al., 2008) proposes a fully automated crawling of the web application under test. The tool does allow for some customization of the crawling to the given web application. For example by specifying classes of DOM elements to execute actions on (*e.g.* HTML `<input/>` or `<a/>` elements) or the set of actions to execute (*e.g.* `click`, `mouseover`). However, while this level of customization is sufficient for automating a very basic level of exploration, based on button-clicks and link navigation on simple web applications, it is usually not sufficient to produce meaningful comprehensive models on real-life enterprise web applications. Some typical scenarios where the above kind of “automatic” crawling falls short are the following:

1. *User Authentication:* The most common situation is where the web application requires some kind of user authentication (*i.e.* login) in order to proceed to the next step. This may require, for example, filling in specific, valid username and password information into specific input boxes on the current page and clicking a specific login button. Further, this authentication step may not occur as the first step of the navigation but at a later point (*e.g.* some applications like `Amazon.com` do not require a login till the checkout stage). Another variant on this scenario is that applications typically have different behaviors (privileges, features, actions) for administrative versus regular users and a good behavioral model should encompass both these use-cases. Thus the tool should do one set of exploration with an administrative login and another with a regular user login.
2. *Processing Forms with Several Sets of Data:* HTML forms are commonplace in modern web applications (a user authentication panel is a special instance of this) and typical exploration would require exploring these forms with different data-sets (for example one erroneous data set and one correct one) for checking the response of the application to correct and incorrect user inputs.
3. *Excluding Behavior from the Model:* In many instances practical resource constraints dictate that only specific actions on specific pages be exercised when crawling the application. One reason for doing this could be that the excluded features and actions may be outside the scope of the ensuing validation step and thus need not be included in the model.

The above scenarios necessitate that the tool have a mechanism for specifying and executing (during crawling) a *specific* sequence of actions, with specific and possibly multiple sets of data, which are executed at *specific* stage in the application evolution, *i.e.* based on the state of the web application. There could be several such sequences, based on the different scenarios exemplified above. Further, the behavior produced by this kind of *custom* crawling should be seamlessly and correctly integrated into the other parts of the model produced by the blind crawling (such as fully automatic link navigation).

Our solution to the above requirement is the mechanism of *guided crawling*. The guidance is based on a set of *Guidance Directives* that are specific to the target application and supplied as input to our crawler.

Definition 3.1 (Guidance Directive)**.** *A Guidance Directive $\mathcal{G} = (p, \mathcal{A})$ is an ordered pair that consists of a predicate p that is evaluated on the current web application state and an action sequence $\mathcal{A}$. The action sequence $\mathcal{A} = (\alpha_1, \alpha_2 \ldots, \alpha_k)$ 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.*

A guidance directive $\mathcal{G}$, as per Definition 3.1 includes the predicate that determines when $\mathcal{G}$ should be activated. The predicate is evaluated on the current state of the web application during the crawling. In our case the state is essentially DHTML document shown in current screen/web-page on the browser. Therefore, predicate p can be supplied as an expression which yields a binary value given a

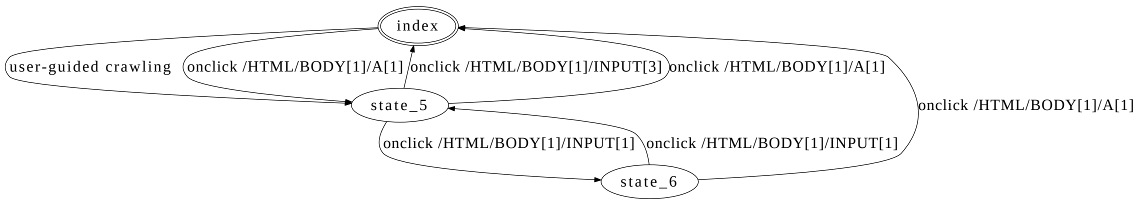

index
user-guided crawling
onclick /HTML/BODY[1]/A[1]
onclick /HTML/BODY[1]/INPUT[3]
onclick /HTML/BODY[1]/A[1]
onclick /HTML/BODY[1]/A[1]
state_5
onclick /HTML/BODY[1]/INPUT[1]
onclick /HTML/BODY[1]/INPUT[1]
state_6

Figure 1: Graphical view of a screen transition diagram.

DHTML document. The action sequence $\mathcal{A}$ represents the sequence of user actions that must be executed on the web application upon activation of $\mathcal{G}$ (*i.e.* when p is true in the current state/screen), along with the associated data, if applicable. 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, a typical action u would be a click or mouse-over on a DOM element (such as a link or button). Such actions have no associated data. Hence the $\mathcal{D}$ component of the atomic action would be the empty set $\emptyset$. Another class of actions are those that correspond to the choice or input of some data string, for example selecting an option from a `<select/>` element or assigning a string value to an `<input/>` element etc. In these cases the $\mathcal{D}$ component would represent the set of data values we would like to test/exercise the element with. Upon activation of the guidance directive, the tool constructs a concrete action sequence from $\mathcal{A}$ by picking specific data-values from the set $\mathcal{D}$ corresponding to each α in $\mathcal{A}$ and executes it on the web application. It does this systematically, for each combination of data-values represented in $\mathcal{A}$.

Algorithms 1 and 2 present the pseudo code for the overall model generation algorithm based on guided crawling. The main procedure, `GuidedCrawl` (Algorithm 1) is supplied a web application, W and a set of associated guidance directives, $\mathcal{G}^{set}$. It initializes the navigational model M, loads the web application in the browser and invokes the inner procedure, *GuidedCrawlFromState* (Algorithm 2) on the initial web-page *(*InitPage). *GuidedCrawlFromState* does the actual crawling and recursively calls itself on new successor states/screens. The various other functions used in the algorithm are described as follows:

- *IsVisited:* Checks if the state S has been visited by a previous invocation of *GuidedCrawlFromState*. This check takes into account any state-abstractions implemented for this step (explained further in Section 3.1.3).
- *MarkVisited:* Marks state S as visited to exclude it from future guided crawls
- *AddState:* Records the state S in the navigational model M as a newly discovered state

Algorithm 1: GuidedCrawl($W, \mathcal{G}^{set}$).

```
1: $M = \emptyset$
2: $InitPage \leftarrow$ LoadBrowser($W$)
3: GuidedCrawlFromState($InitPage$)
3: **return** $M$
```

Algorithm 2: GuidedCrawlFromState(S).

```
 1: **if** IsVisited($S$) **then**
 1:     **return**
 2: **end if**
 3: MarkVisited($S$)
 4: AddState($S, M$)
 5: $Actions \leftarrow$ FindActions($S$)
 6: **for all** $\mathcal{G}(p, \mathcal{A}) \in \mathcal{G}^{set}$ **do**
 7:     **if** $p(S)$ = `true` **then**
 8:         $Actions \leftarrow Actions \cup$ ComputeActionSequences($\mathcal{A}$)
 9:     **end if**
10: **end for**
11: **for all** $a \in Actions$ **do**
12:     $nextState \leftarrow$ Execute($a, W, S$)
13:     AddTransition($nextState, S, M$)
14:     GuidedCrawlFromState($nextState$)
15:     UndoTransition($a, W, S$)
16: **end for**
```

- *FindActions:* Computes the set of primitive (non-guided) user actions (e.g. clicks, mouseovers etc.) that can be executed on the current screen S.
- *ComputeActionSequences:* Computes concrete sequences of actions from the guidance directive $\mathcal{G}$ by picking specific data values in its constituent atomic actions α. Computes all possible sequences that can be created by various choices of the specified data-values.
- *Execute:* Executes the action (or action sequence) a on the web application W, which is currently in state/screen S.
- *AddTransition:* Records the transition from state S to state *nextState* in the model M.
- *UndoTransition:* Functionally reverses the transition $S \rightarrow nextState$ on W to restore it to state S.

3.1.3 Model Reduction

One of the principal advantages of our two-step approach, *i.e.* model generation followed by model checking, is the ability to perform the validation on a much more compact and abstracted model than the full HTML content available at the browser-level during model generation. This enables a much more efficient check. There are several sources of compression we utilize in generating the final model to be checked:

1. *Validation-Directed Inclusion/Exclusion of Events:* It is obvious that not all of transitions resulting from firing a certain kind of event on a certain class of element would be of relevance from a validation perspective. Therefore, we specify to the crawler a list of tuples of HTML elements set and event types, that should definitely be *included* and those that should be definitely be *excluded* during crawling. The specification is done by the user on application-specific basis, as an input to the model generation step. In each of specifications above, set of HTML elements are specified using XPath.
2. *State Abstraction:* In our proposed navigational model, the determination of whether given two states are equivalent or not is performed based on content of DHTML document. While states corresponding to DHTML documents with same content is considered to be equivalent in our navigational model, there are several instances of situations where two “similar-looking” DHTML pages should be considered to be equivalent. The followings are some of the scenarios where “exact” determination of equivalence between DHTML pages falls short.
   (a) Current time (variable from run-time environment) included in page
   (b) Server-side variables (which we do not explicitly reset), and are not of our interest exposed to DHTML content

   Therefore, we have introduced state abstraction techniques which accepts user-given XPaths of elements to be removed from DHTML documents observed. When determining equivalence between two given states, document elements which match the given XPaths and their descendants are removed from DHTML document structures corresponding to the states and thereafter compared. This criteria for *state-equivalence* are implemented within the *IsVisited()* function in Algorithm 2.

3.2 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 a la traditional testing, we propose to check the navigational model as a whole using the formal technique of model checking (Clarke et al., 1999). As pointed out in Section 3 the use of a pre-generated navigational model, compactly represented as a finite state machine makes the application of model checking both easy and very efficient. We further claim that several navigational and other types of requirements that are typically checked on web applications can be quite naturally formulated as properties in temporal logic (Clarke et al., 1999), 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 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. *Navigation Structure/Usability Requirements:* This kind of requirement would apply checks to the overall structure of the navigational model, with the intent of checking the ease with which the user can access various functionalities offered by the application. This class of requirements is by its very intent, *global* and hence ideally suited for model checking on a pre-extracted model (versus real-time or 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*

Since each screen of the web application and all features/widgets or DOM elements on each screen are captured as part of the state representation in our navigational model, it is possible to formulate and check the requirements above (and several other categories of requirements typically checked on web applications) as properties in a temporal logic (Clarke et al., 1999) (*e.g.* Computation Tree Logic (CTL) or Linear Temporal Logic (LTL)) with the following steps.

3.2.1 Validation Scheme

First, we annotate each state within the navigational model with values of "atomic propositions" used in the properties. In our methodology, we allow atomic propositions be any expression which hash same DHTML document to same binary value. After the annotation, there will be two kinds of variables within the model: unique ID for each state and annotating variables corresponding to value of atomic propositions in each state. Then we convert the FSM model including annotating variables, into the format which is accepted by external model checkers. While inter-state transition relations are described using values of ID variables, property can be described using any annotating variables containing values of atomic formulas which hashes equivalent state into same value.

Atomic propositions which are hash functions of DHTML documents into binary output, can be given by the user as a procedure in some programming language. And they can take forms which are quite similar to assertions used in existing automatic testing tools for client-tier behavior of AJAX-based web applications such as Selenium. Therefore, it is quite easy for developers in field to migrate to our methodology. Examples of atomic expressions include availability of a node with given type and attribute, and availability of text content which matches a given regular expression.

3.2.2 Temporal Property Templates

Still, writing complex temporal logic formula based on those atomic propositions is troublesome work for developers. So we have decided to introduce template-based approach which is based on a few temporal classes of properties in property specification. The idea is based on the notion from a study that most verification requirements observed in practice, can be captured by properties in a limited number of temporal classes (Dwyer et al., 1999). Temporal classes of properties we have targeted in our framework are shown in Figure 2, where p,p_1 and p_2 are atomic propositions which are evaluable in a given single state, based on DHTML page contents corresponding to the state. In the second class of properties, inputs in i are specified as a tuple of a unique ID in each DHTML document, and event type or input data. Forms of counter example for the temporal property templates are shown in Figure 3.

1. *$G(p)$:* Globally p is *true*
2. *$p_1, i \rightarrow p_2$:* 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. *$p_1 \rightarrow F p_2$:* After transiting from a state where p_1 is *true*, eventually reaches a state where p_2 is *true*
4. *$p_1 \rightarrow P p_2$:* Need to reach a state where p_2 is *true*, prior to reach a state where p_1 is *true*

Figure 2: Temporal property templates targeted.

1. *$G(p)$:* A state S where p is *false*
2. *$p_1, i \rightarrow p_2$:* 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
3. *$p_1 \rightarrow F p_2$:* A tuple (S_1,$\mathcal{T}$,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 $\mathcal{T}$ is a non-empty sequence of transitions from S_1 to S_2
4. *$p_1 \rightarrow P p_2$:* A sequence of transitions $\mathcal{T}$, which reaches a state where p_1 is *true* from initial state, without going through any state where p_2 is *true*

Figure 3: Counter examples for the property templates.

4 TOOL IMPLEMENTATION

We have implemented our model-based testing approach, in a system which is comprised of two components. One of the components of the system is an extended version of open-source CRAWLJAX tool, and the other is a custom model checker called GOLIATH, which is dedicated for checking requirements on screen transitions explained in Section 3.2.

4.1 Model Generation

We have extended CRAWLJAX-1.7, to include the following features required for its application to our flow.

1. *Guided Crawling (Section 3.1.2):* The feature was implemented so that predicate p in Definition 3.1 can be supplied as a Java method which accepts Document object and returns boolean value. And each of actions α_n in $\mathcal{A}$, can be specified as a tuple of XPath string specifying target element (e), and event type string (u) or input data string set ($\mathcal{D}$).
2. *On-The-Fly Model Reduction (Section 3.1.3):* The feature was implemented in such a way, that user can specify a set of strings each corresponding to XPath of elements to abstract out, and all of elements matching the XPath and their descendants are removed from the document structure corresponding to the state.

4.2 Model Validation

We have implemented a custom model checker GOLIATH, which is dedicated for checking properties with atomic propositions and temporal forms described in Section 3.2.1 and 3.2.2 respectively.

GOLIATH is implemented using Ruby programming language, and atomic propositions (p, p_1 and p_2 in Figure 2 and 3) are expected to take form of Ruby expressions and may refer to DHTML document object with identifier `doc` with Nokogiri[5] HTML parser API. The following is a very simple instance of atomic proposition supported, which is true *iff* there is some `<a/>` element with `id` attribute value `login`.

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

A little more complex and practical expression example can be found in the following.

```
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 expression above (hereafter referred to as $p_{not\_together}$) is *true* only in states where only one of `login`/`logout` DOM object exists and they do not co-exist (Note in Ruby, that sequences of expressions can be evaluated as an expression which yields value of their final expression). We can write the following property with $p_{not\_together}$ for example.

$$G(p_{not\_together}) \qquad (1)$$

The property can be used to make sure that there is no state, where both of `login`/`logout` exist or do not exist, in the application behavior extracted to the model.

[5]http://nokogiri.org/

5 CASE STUDY

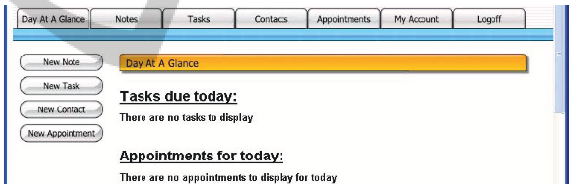
The proposed method has been implemented and evaluated by applying them to examples including a real industrial one. To assess the efficacy and utility of our approach and the corresponding implemented tool, we have conducted a number of case studies following guidelines from (Kitchenham et al., 1995).

5.1 The Examples Targeted

We used two examples, the first one (Example 1) is taken from a textbook (Zammetti, 2006) on AJAX based web application developments and the other (Example 2) is a real industrial application.

- Example 1 has 8,004 lines of Java codes
- Example 2 is a business process manager comprised of 58,701 lines of Java code, 61,541 lines of JavaScript codes, and 90,742 lines of JSP codes. YUI[6] is the AJAX library used.

Typical screen images of user interface for the two examples are shown in Figure 4 and Figure 5, respectively. All experiments are performed on Intel Core2 Duo CPU E8400@3.00GHz.



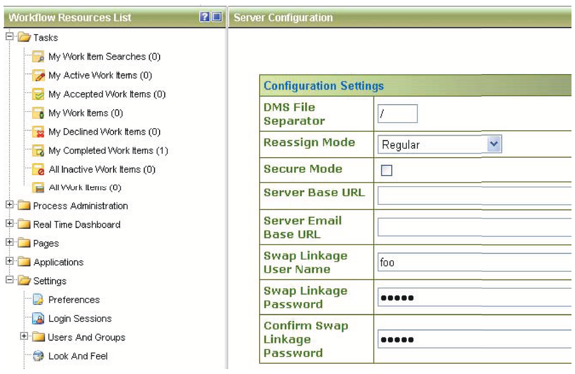
Tasks due today:

There are no tasks to display

Appointments for today:

There are no appointments to display for today

Figure 4: Typical screen image of Example 1.



Configuration Settings	
DMS File Separator	/
Reassign Mode	Regular
Secure Mode	☐
Server Base URL	
Server Email Base URL	
Swap Linkage User Name	foo
Swap Linkage Password	●●●●●
Confirm Swap Linkage Password	●●●●●

Figure 5: Typical screen image of Example 2.

5.2 Model Generation

Appropriate guidance directives, which are the keys for meaningful and relative quick model generations,

[6]http://developer.yahoo.com/yui/

are given to generate the corresponding screen transition diagrams for the two examples. The directives include operations for logging in by providing usernames and passwords whenever there is a screen image having login prompts. Also, when generating the screen transition diagrams, the maximum depth of operations for each application is set to 9.

Table 1 shows the model (screen transition diagram) generation results for the two examples. Please note that although we also tried to generate tests using the approach shown in (Mesbah and Deursen, 2009), 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.

Table 1: Model generation results.

	Example 1	Example 2
Time (sec)	3175	166114
#State	49	763
#Transition	360	4037
Avg. HTML size (kB)	11	2890

5.3 Properties for Model Checking

For Example 1, we extracted 7 properties, including the one shown in Figure 6, from the textbook relating to the manipulations of tabs shown in the top of Figure 4. The property in Figure 6 which is in the form of $p_1, i \rightarrow p_2$ in Figure 2 says, if the tab “Day At A Glance” in Figure 4 is clicked, the orange area which shows the current screen name must display “Day At A Glance”. The other properties also check if a tab is clicked, its corresponding screen must be shown.

For Example 2, 14 properties each of which corresponds to each test case that the program developers have used for their testing, are prepared. As Example 2 is a real industrial application, their details cannot be shown here, but they are somehow similar to the ones for Example 1 in the sense of manipulations.

5.4 Model Checking Results

All 7 properties for Example 1 have been model checked within one second. Only one of the 7 properties shown in Figure 6 generates a counter example shown in Figure 7. The counter example is automatically converted to a test program which performs inputs required to reach and activate it, by a helper tool. Invoking the test program obtained actually generates an error shown in Figure 8 within a few seconds. Also in the log of the Java server software, an error message which indicates “Null Pointer Exception” is reported as shown in Figure 9. This is a real bug in a program shown in the textbook.

For Example 2, all properties have been checked in 337 seconds, and all 14 properties have been proved to be correct. As the time for model checking is rather short, many more properties can be checked within practical time.

As you can see from the above, once the model is generated, model checking is relatively very quick, and various properties can be examined with reasonable time even for large industrial examples.

```
doc.xpath('//img[@id="dayAtAGlance"]').any? ,
/HTML/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/\
TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/IMG[2]:onclick
->
doc.xpath('//img[@src="http://localhost:8080/\
theorganizer/img/head_dayAtAGlance.gif"]').any?
```

Figure 6: One of properties used for Example 1.

```
State:["state_534-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/\
TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/\
TABLE[1]/TBODY[1]/TR[1]/TD[1]/IMG[2]:onclick"]
State:["state_536-raw.html"]
```

(a) A counter example for the property
(A transition which does not meet the property).

```
State:["index-raw.html"]
Transition: ["/HTML/BODY[1]/CENTER[1]/ ... /INPUT[1]:onclick"]
State:["state_5-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_6-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_9-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_13-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_18-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_24-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[6]:onclick"]
State:["state_398-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[2]:onclick"]
State:["state_400-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /IMG[1]:onclick"]
State:["state_466-raw.html"]
Transition: ["/HTML/BODY[1]/TABLE[1]/ ... /INPUT[1]:onclick"]
State:["state_534-raw.html"]
```

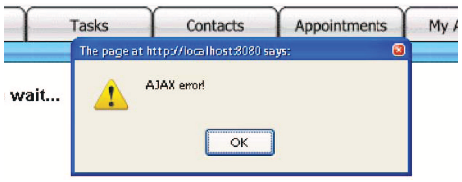
(b) A trace to reach the counter example from initial state.

Figure 7: Counter example for the property in Figure 6.

6 CONCLUSIONS & FUTURE WORK

We have proposed a verification approach for AJAX based web applications consisting of two steps:

- Generation of the models in terms of finite state



The page at http://localhost:8080 says:

AJAX error!

Figure 8: Screen of error message observed after playing back an input sequence to exercise the counter example.

```
[ERROR] DispatcherUtils - Could not execute action
<java.lang.NullPointerException>java.lang....
at com.apress.ajaxprojects.theorganizer.actions.
   DayAtAGlanceAction.execute ...
at sun.reflect.GeneratedMethodAccessor80.invoke ...
...
```

Figure 9: A part of server log observed after playing back an input sequence to reach the counter example.

machines (FSMs) that correspond to the behavior of the AJAX based web applications.

- Model checking the generated FSM based on the user-given properties

Once the models are generated, various properties can be checked very quickly. Also, for the model generations, the proposed technique can accept "guidance directive" to target user-specific features and behaviors of the AJAX based web applications. These have been confirmed through experiments. Industrial sized applications can be processed within reasonable time.

Future works include more verification trials with larger applications for more robust evaluations of the proposed techniques as well as their extensions. On verification trials with larger-scale applications, the performance of model generation stage in our technique which is taking relatively long time, can be improved with several strategies.

While we have introduced model reduction techniques in Section 3.1.3 to reduce run time, use of the state abstraction technique which removes DHTML substructure unrelated to verification requirements, was limited in the experiments. Although more aggressive model reduction configuration based on verification requirements is expected to reduce time required for model generation stage, the prospect has to be confirmed through case studies.

Most of time required for crawling-based model generation stage is from communication latency between the crawler and the target application. As computation burden of the process is relatively small, it is desirable to introduce parallel crawling technique used in conventional web crawler. There is an ongoing work of parallelizing the process in CRAWLJAX project, and the outcome is expected to be effective for reducing run time of our verification flow.

REFERENCES

Alfaro, L. D. (2001). Model Checking the World Wide Web. In *Computer Aided Verification*, pages 337–349. Springer-Verlag.

Andrews, A. A., Offutt, J., and Alexander, R. T. (2005). Testing Web Applications by Modeling with FSMs. *Software and Systems Modeling*, 4:326–345.

Benedikt, M., Freire, J., and Godefroid, P. (2002). VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of 11th International World Wide Web Conference*.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in Property Specifications for Finite-State Verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA. ACM.

Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case Studies for Method and Tool Evaluation. *IEEE Softw.*, 12(4):52–62.

Marchetto, A., Ricca, F., and Tonella, P. (2008a). 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.

Marchetto, A., Tonella, P., and Ricca, F. (2008b). State-Based Testing of Ajax Web Applications. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 121–130, Washington, DC, USA. IEEE Computer Society.

Mesbah, A., Bozdag, E., and Deursen, A. V. (2008). Crawling AJAX by Inferring User Interface State Changes. In *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*, pages 122–134, Washington, DC, USA. IEEE Computer Society.

Mesbah, A. and Deursen, A. V. (2009). Invariant-Based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*.

Ricca, F. and Tonella, P. (2001). Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 25–34. IEEE Computer Society.

Strecker, J. and Memon, A. M. (2009). Testing Graphical User Interfaces. In *Encyclopedia of Information Science and Technology, Second ed.* IGI Global.

Zammetti, F. (2006). *Practical Ajax Projects with Java Technology*. Apress.