Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines

Sylvain Hallé[*]
Université du Québec à Chicoutimi, Canada
shalle@acm.org

Taylor Ettema, Chris Bunch and
Tevfik Bultan[†]
University of California, Santa Barbara, USA
{tettema,cgb,bultan}@cs.ucsb.edu

ABSTRACT

The enforcement of navigation constraints in web applications is challenging and error prone due to the unrestricted use of navigation functions in web browsers. This often leads to navigation errors, producing cryptic messages and exposing information that can be exploited by malicious users. We propose a runtime enforcement mechanism that restricts the control flow of a web application to a state machine model specified by the developer, and use model checking to verify temporal properties on these state machines. Our experiments, performed on three real-world applications, show that 1) our runtime enforcement mechanism incurs negligible overhead under normal circumstances, and can even reduce server processing time in handling unexpected requests; 2) by combining runtime enforcement with model checking, navigation correctness can be efficiently guaranteed in large web applications.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*

General Terms

Theory, verification

Keywords

Navigation, web applications, model checking

[*]This work was done when Sylvain Hallé was at the University of California, Santa Barbara.
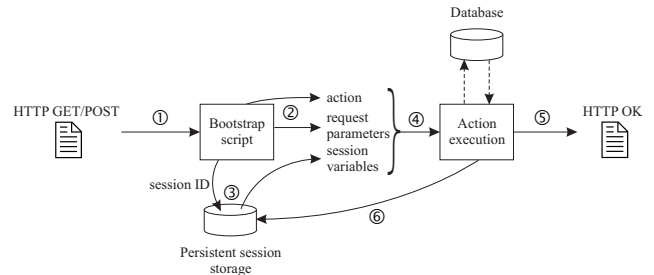
Figure 1: Processing of an HTTP request in an MVC web application.

1. INTRODUCTION

Browser functionalities such as bookmarks and the "back" button allow users to request pages from web applications in unpredictable ways. When incorrectly handled, an unexpected request from the user can execute an unintended action, produce a cryptic and confusing error message, or even expose details about the application that can be exploited for malicious purposes. In fact, our experiments on real-world web applications demonstrate that more than half of unexpected navigation sequences are not properly caught and processed by web applications. The enforcement of navigation constraints is therefore a critical issue in web application development.

An increasing number of web applications are being developed using the Model-View-Controller (MVC) design pattern through frameworks like Zend, Ruby on Rails and CodeIgniter. MVC pattern facilitates the separation of control flow logic (controllers), business logic and data (model) and user interface logic (view). As a consequence, MVC web applications break the concrete link between a URI and a specific resource residing on the server. All incoming requests are rather directed to a single "bootstrap" script that parses and evaluates them, and then determines the actions to execute (see Figure 1).

By clicking on a link in a page, or by typing a URL in the browser's location bar, a user actually requests a new page on the application server through an HTTP GET or POST request (Figure 1, label (1)). This request is relayed to the application's "bootstrap" script. Based on the requested URL, the script extracts the name of the *action* to be executed and any *request parameters* carried in the HTTP request (2); if one of these parameters is an identifier called

a session cookie, the script also fetches from the server's persistent storage an additional set of *session variables* and their values (3). All this data is then used to execute the action proper (4); this may include back-end database read and write access. Finally, the application returns an output page back to the browser as the HTTP response to the initial request (5), and possibly modifies the session information in the server's persistent storage (6).

By navigating through an MVC web application, a user produces a sequence of triplets ⟨action, request parameters, session variables⟩, with each such triplet representing the information related to one "button click" on the browser. We call such a sequence a *navigation trace*.

In a standard desktop application, not all sequences of actions are considered valid. For example, it is nonsensical to call the "save" command if no document has been opened beforehand. Well-designed products prevent such odd sequences by disabling or hiding from the user interface any menu items or buttons which do not make sense in the current state of the application.

In web applications, however, this is far more difficult to achieve. Even when the pages returned by an application after each request only provide links to valid actions from the current state, the browser itself provides many ways of bypassing these measures through the use of the back button, bookmarks, multiple windows, and the location bar where arbitrary URLs can be typed. None of these GUI elements can be reliably disabled, controlled or even hidden by a web application. The task is made even more difficult by the HTTP protocol, which is by design *stateless* and treats requests independently of each other.

In an MVC application, the handling of navigation constraints should clearly be the responsibility of the controller. However, MVC-based applications do not fully exploit this feature. In particular, they do not support the enforcement of constraints across multiple controllers and actions, and they do not provide a mechanism to store the navigation history that is necessary to enforce these navigation constraints. The developer therefore becomes responsible of writing customized mechanisms for these two features, a tedious and cumbersome task.

In this paper, we propose a solution to this problem with a runtime enforcement mechanism that restricts the control flow of a web application to a navigation state machine model specified by the developer by exploiting the existing structure of the MVC pattern. We then use model checking to verify temporal properties of the navigation state machine itself. We implemented our approach by developing 1) a navigation state machine language for specification of navigation constraints; 2) a plugin for MVC-style PHP applications that enforces the navigation state machine constraints at runtime; 3) a translator that generates SMV models from navigation state machines for model checking.

Our experiments show that the runtime enforcement plugin incurs negligible overhead under normal circumstances, and can even reduce server processing time when handling unexpected requests by avoiding the generation of error pages. Moreover, with our runtime enforcement mechanism, the verification of temporal properties about navigation can be done efficiently using model checking, even on web applications containing more than 400,000 lines of code.

2. NAVIGATION ERRORS

Application	Modules	Controllers	Actions	KLOC
Digitalus	2	14	60	401
BambooInvoice	1	9	26	159
Capstone	3	12	33	41

Table 1: Number of modules, controllers, actions and thousands of lines of PHP code for each of the three applications studied in this paper.
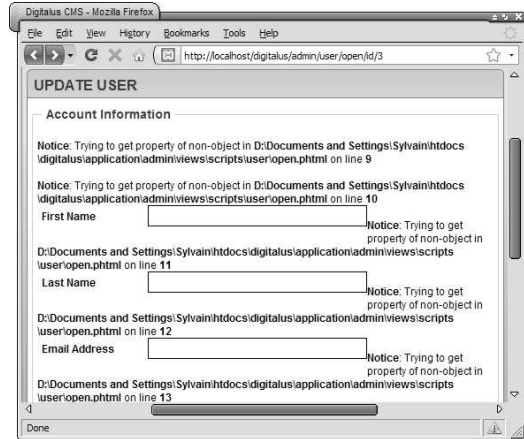

Figure 2: Output of Error 1 in Digitalus.

Although stateful web applications should keep track of navigation traces and validate each request based on that history, most of them do it partially, if at all. Therefore, it is generally easy for a user to produce non-compliant sequences of actions that are improperly handled by a web application.

To illustrate this point, we studied three real-world applications (see Table 1): Digitalus[1] 1.8, an open-source content management system (CMS) written using the PHP Zend framework; BambooInvoice[2] 0.8.9, an invoice management system for small businesses written using the PHP CodeIgniter framework; Capstone Manager, a project management system used at UCSB written using the PHP Zend framework. Below, we describe examples where these applications erroneously handle non-compliant navigation traces, producing what we call *navigation errors*, classified according to the method by which they are created.

Basic Errors. This first category corresponds to errors that can be reproduced by simply using the application's navigation links, and possibly the web browser's back button and bookmark functionality. A first example, taken from Digitalus, involves the following sequence of steps:

ERROR 1 (DELETE YOURSELF). 1) Login as any user, say Bob, with superadmin status. 2) Click on the "Site" tab. 3) In the "Admin Users" sidebar, click on Bob to get to the user edit page. 4) Click on "Delete" in the top-right corner. 5) Bob is deleted from the user list; yet, you are still logged in as Bob. 6) Click on the browser's back button; this will take you back to the "Edit" page for Bob, which no longer exists. The resulting page is shown in Figure 2.

---

[1] http://www.digitaluscms.com
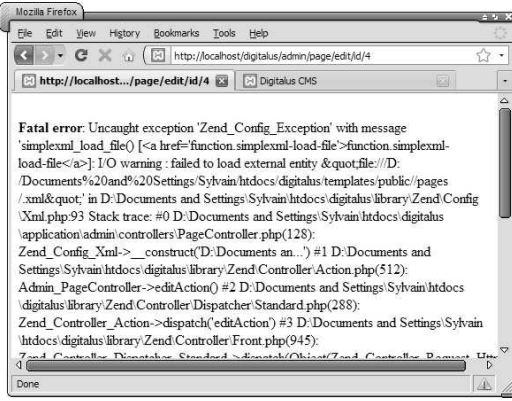[2] http://bambooinvoice.org

Figure 3: Output of Error 3 in Digitalus.

This navigation error has two consequences: First, the user interface is disrupted with PHP error messages; in particular, the messages show the absolute location of Apache's root directory on the web server. This can be used, e.g. to deduce that the server is a Windows XP machine running under the user "Sylvain" off local drive "D". Most importantly, since the deleted user is the one that is actually logged in, the application should immediately log out and redirect to the login page. Instead, the deleted user is still logged in and has access to all administrative functions. A similar error also exists in BambooInvoice.

A second example, taken from the BambooInvoice application, involves the browser's bookmarking functionality:

ERROR 2 (EDIT A DELETED INVOICE). 1) Login to the system. 2) Go to the invoice list and click on an invoice to get to the edit page. 3) Bookmark this page. 4) Go back to the invoice list by pressing the browser's "Back" button. 5) Click on "Delete invoice". 6) Invoke the bookmark. This summons a page with PHP errors similar to Figure 2.

Multi-window Errors. As in the previous category, these errors can be reproduced solely by using the application's and web browser's buttons; however, they require two different browser windows. For example, in Digitalus:

ERROR 3 (UPDATE A DELETED PAGE). 1) Login in as an administrator and go to the "Pages" tab. 2) Select any page and right-click on the link for this page in the "Pages" tab at the right; choose "Open the link in a new tab". 3) In the first window, left-click on the link for this page. There are now two tabs containing the edit form for the same page. 4) In the first tab, click on "Delete". 5) Go to the second tab, click on "Update". The resulting page is shown in Figure 3.

This navigation error involves even further disruption of the user interface. This time, the basic page template does not load at all, and the user is presented with the white page that PHP generates in case of a fatal error. Similar errors can be created with BambooInvoice when trying, in a first window, to create an invoice for a user that was previously deleted in a second window.

Direct URL Errors. This type of errors are caused by typing a URL in the web browser's location bar from the "wrong" context. For example, in Digitalus:

ERROR 4 (NO USER LOGGED). Reproduce all the steps for Error 1. Then try to return to the login page by typing admin in the browser's URL bar. This summons a PHP white "Fatal error" page similar to Figure 3.

ERROR 5 (FORBIDDEN PAGES). In Digitalus, typing any of the URLs mentioned in Figure 6 either causes a page with PHP warnings (similar to Figure 2), or a white page with PHP fatal errors (similar to Figure 3).

This time, the problem lies in the fact that the application assumes that these pages are called from a previous page that gives values to some mandatory request parameters. Calling the page outside that context creates errors.

All in all, we found 11 different ways to cause warnings and fatal errors.

Navigation Constraints. The previous examples show sequences of actions where the web application fails to realize that an action has been called in the wrong context. Most of the navigation errors presented above could be prevented by simply keeping track of the last triplet ⟨action, request parameters, session variables⟩, and by granting or blocking the execution of the requested action according to that triplet.

For example, Navigation Error 1 can be prevented by a) checking that the session ID is different from the user ID in the delete request or b) otherwise, only allowing admin/auth/logout as the next page; this also takes care of Navigation Error 4. Navigation Error 2 can be prevented by making sure that the "edit" action is only accessed if the last action is "view", with both actions having the same "id" parameter. Navigation Error 3 can be prevented by making sure that the "page delete" action is only accessed if the previous page is "page edit". Finally, the solution for Navigation Error 5 is similar for each page mentioned; for example, the problem for admin/page/edit can be solved by accessing it only from admin/page/new or admin/page/index.

In the following sections, we formally describe the problem of navigation errors in web applications, and introduce a mechanism to enforce navigation constraints like the ones above in web applications.

3. A FORMAL MODEL

We formalize the global behavior of an MVC web application (which we will simply refer as an application) as follows:

- M is a set of *data model* states, where the data model can include any stateful representation of application data, such as a database,

- V is a set of *session variables*, i.e., data stored on the server on a per-client basis,

- I is a set of *sessions* used by the server to associate clients with session variables,

- A is a set of *actions*, i.e., the program segments that are invoked based on the HTTP requests sent by the user,

- P is a set of *request parameters*, i.e., input data from the user received as part of the HTTP requests via GET or POST.

To simplify our presentation we will denote the set of values a session variable or a request parameter can take as D.

The *state* of an application is uniquely defined by the combination of a model state and the valuations of the session variables for each session. The *requests* from the user, on the other hand, are used to uniquely identify a transition from one state of the web application to another state. Each request consists of an action and a set of request parameters.

DEFINITION 1. *An application is a tuple $\mathcal{A} = \langle Q, B, \Delta \rangle$ where:*

- $Q \subseteq M \times D^{|I| \times |V|}$ *is the set of states. Given a state $q \in Q$, we use $q(i, v)$ to denote the value of session variable $v \in V$ for session $i \in I$ in state q, and $q(i)$ to denote the vector $\vec{v} \in D^{|V|}$ containing values of all session variables for session i in state q.*

- $B \subseteq Q$ *is the set of* initial *(or begin) states, where for all $q \in B$ and $i \in I$ $q(i) = \perp^{|V|}$ ($\perp$ denotes null).*

- $\Delta : Q \times A \times D^{|P|} \times I \to Q$ *is the* transition function.

The function Δ satisfies the following two constraints:

1. *If $q' = \Delta(q, a, \vec{p}, i)$, then for any session $j \in I$, $j \neq i \Rightarrow q'(j) = q(j)$.*

2. *If $q' = \Delta(q, \epsilon, \perp^{|P|}, \perp)$, then $q'(i) = q(i)$ for any session $i \in I$.*

In an application, each session can only change its own session variables, as specified by the first constraint on the transition function Δ. The second constraint states that the ϵ-transition does not change any session variable and only accounts for changes in the data model state, independent of any session activity. We use the ϵ-transition to represent the fact that a web application can update the data model state (e.g. the back-end database) without receiving a request from the user.

Starting from an initial state $q_0 \in B$, a succession of tuples $(q_0, \epsilon, \perp^{|P|}, \perp), (q_1, a_1, \vec{p_1}, i_1), (q_2, a_2, \vec{p_2}, i_2), \ldots$ is called a *global trace* of $\mathcal{A}$ if, for every $k \geq 0$: $q_{k+1} = \Delta(q_k, a_k, \vec{p_k}, i_k)$. Each element represents the current application state, the *previous* action and the request parameters, and the session for which the previous action was executed. Since there is no previous action in the initial state we use the ϵ-transition as default in the first tuple of the trace.

Our definition of a global trace implies that the actions are executed atomically. This is not true in a real web application. Two actions from different sessions can be executing at the same time. However, the interleaved trace semantics we defined above is equivalent to such an execution for the following reasons: 1) Session variables are disjoint and each session only modifies its own session variables; 2) Although the model is shared among the sessions, it is typically managed through a database, and conflicts and race conditions are resolved through the use of database transactions.

3.1 A Simple Example

We define a simple web application (based on the account management in Digitalus) to demonstrate our formal model. The set of actions is $A = \{\text{index}, \text{open}, \text{delete}, \text{edit}, \text{create}\}$, where the index action returns a page that lists the available account management

options, open returns a page for editing, deleting or creating an account, and the remaining actions are self-explanatory.

There is a single request parameter $P = \{\text{rid}\}$ denoting the identity of the account that the user wants to execute the action on, and there is a single session variable $V = \{\text{sid}\}$ denoting the identity of the current user. The domain for these variables is $D = \{\text{u}, \text{v}, \perp\}$ denoting two user accounts and the null value. Typically, the data model would store detailed information about the user accounts and the edit action would require extra request parameters to update an account which are ignored in this simple example.

The set of sessions is $I = \{1, 2, \perp\}$ (where $\perp$ does not correspond to an actual session). Given a state q of this application, $q(1, \text{sid})$ identifies the user who initiated the session 1 (which would be set using a login action that is not modeled here). A transition $q' = \Delta(q, \text{edit}, \text{u}, 2)$ corresponds to the account u being edited by the user who initiated the session 2. Another transition $q' = \Delta(q, \text{delete}, \text{v}, 1)$ corresponds to the account v being deleted by the user who initiated the session 1. Note that, in this transition we would probably want $q(1, \text{sid}) \neq \text{v}$ so that the user does not self-delete.

In this paper, we focus on navigation constraints that restrict the possible traces of individual sessions, such as:

- The edit, delete and create actions can only be executed immediately after an open action.

- For the open, delete, and edit actions to be executed, the rid parameter has to be non-null.

- The rid parameter and the sid variable cannot be equal when delete action is executed.

3.2 Session Traces

To express navigation constraints, we need to define session traces that only track the actions, request parameters and session variables of individual sessions. To do this, we define a function π that projects a global trace to an individual session by removing all the session variables and actions of other sessions.

DEFINITION 2. *Let $\sigma = (q_0, \epsilon, \perp^{|P|}, \perp)$, $(q_1, a_1, \vec{p_1}, i_1)$, $(q_2, a_2, \vec{p_2}, i_2)$, $\ldots$ be a global execution trace of an application $\mathcal{A}$, and $i \in I$ be a session of $\mathcal{A}$. Let $\sigma' = (q_0, \epsilon, \perp^{|P|}, \perp), (q_1', a_1', \vec{p_1}', i_1'), (q_2', a_2', \vec{p_2}', i_2'), \ldots$ be the subsequence of the trace σ obtained by deleting all tuples $(q_k, a_k, \vec{p_k}, i_k)$ from σ where $i_k \neq i$ and $k > 0$. The* navigation trace for session i in σ, *denoted as $\pi(\sigma, i)$, is the following sequence obtained from σ': $\pi(\sigma, i) = (q_0(i), \epsilon, \perp^{|P|})$, $(q_1'(i), a_1', \vec{p_1}')$, $(q_2'(i), a_2', \vec{p_2}')$, $\ldots$.*

Note that given an execution trace σ of a web application, the session trace $\pi(\sigma, i)$ lists only the actions executed by session i and the request parameters for those actions, and it only keeps track of the session variables of session i. We define a *navigation constraint* as a constraint that can be specified using the information available in a session trace.

This definition has several benefits. First, it clearly separates constraints on the data model from the navigation logic. Second, this separation enables us to enforce navigation constraints independently on each session without any interference from other sessions and without querying the state of the data model. Third, since all the session trace information is visible to the bootstrap script (as shown in Figure 1), we are able to develop a runtime enforcement
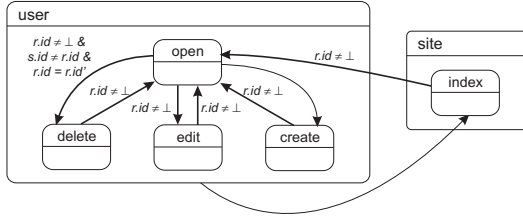
Figure 4: A portion of the navigation state machine for the Digitalus system.

mechanism that does not require us to change any part of the application for navigation constraint enforcement.

4. NAVIGATION STATE MACHINES

All solutions to navigation errors discussed in Section 2 have one point in common: they can be represented by a finite state machine which takes into account the last executed action, the session variables and the request parameters. Unfortunately, analogous to the stateless nature of HTTP, MVC-based web application frameworks lack mechanisms for tracking state. Each request is routed completely independently of history, based purely on the content of the current request. Thus, while the MVC framework vastly improves web application organization and control flow management, it is left to the developer to implement any stateful behavior that may be desired. In this section, we introduce a specification mechanism for characterizing stateful navigation behavior.

A *navigation state machine* (NSM) specifies the allowable execution orderings for actions. Its states correspond to actions, and a transition from one state to another state indicates that the action in the target state can be executed immediately after the execution of the action of the source state (this relation need not be deterministic). Additionally, the transitions contain guard expressions that specify the pre and post-conditions for execution of an action on the session variables and request parameters.

DEFINITION 3. *A* navigation state machine *(NSM) $\mathcal{N}$ is a tuple $\langle S, s_0, \delta \rangle$ where: $S \subseteq D^{|V|} \times A \times D^{|P|}$ is the set of states, $s_0 = (\perp^{|V|}, \perp, \perp^{|P|})$ is the initial state, $\delta : S \times A \times D^{|P|} \to 2^S$ is the transition relation.*

A state of a navigation state machine stores the last action performed in a session and the request parameters for that action, as well as the values of the session variables resulting from that last action. Given the last action, request parameters for that action and the current values of the session variables, the transition relation of the NSM identifies which actions and request parameters can come next in the session trace, and how session variables can change by executing those actions.

Figure 4 shows an NSM for the account management example we discussed above. It contains additional high-level constructs that ease the representation of NSMs and makes them less verbose. First, states are hierarchical: Each "super-state" corresponds to a module of the application and each atomic state represents the last action that was executed. For example, if the NSM is in state `open`, then this means that the `open` action of the `user` module was the

last action executed by this session. This way, one can use an arrow from the `user` box to the `index` state to represent a group of transitions, one from each action in `user` to `index`. Similarly, a single `open` state can be used to represent all combinations of session variables and request parameters for that action. The same goes for other actions.

The transitions specify pre-conditions on the previous request parameters ($\mathtt{rid}'$), the current session variables ($\mathtt{sid}$) and the current request parameters ($\mathtt{rid}$). For example, the arrow from state `open` to state `delete` is associated with the pre-condition $\mathtt{rid} = \mathtt{rid}' \wedge \mathtt{rid} \neq \perp \wedge \mathtt{rid} \neq \mathtt{sid}$ which means that, if the last action executed was `open`, then the `delete` action can be executed as long as the current request parameter $\mathtt{rid}$ is equal to the previous request parameter $\mathtt{rid}'$, the current request parameter $\mathtt{rid}$ is not null, and the current request parameter $\mathtt{rid}$ and the current session variable $\mathtt{sid}$ are not the same. Although not shown in the figure, constraints on how session variables are updated can be specified by writing post-conditions on the current and next session variables and current and previous request parameters. We assume that if there is no post-condition specified, then the session variables do not change their values.

Apart from this syntactical sugar, the semantics of the state machine shown in Figure 4 precisely matches the formal model we defined above. Note that this state machine enforces all the navigation constraints we identified for this application.

4.1 Navigation Conformance

A navigation state machine (NSM) recognizes the session traces that are compliant to the expected use of the web application.

DEFINITION 4. *Let $\sigma = (\perp^{|V|}, \epsilon, \perp^{|P|})$, $(\vec{v_1}, a_1, \vec{p_1})$, $(\vec{v_2}, a_2, \vec{p_2})$, ... be a session trace and let $\mathcal{N} = \langle S, s_0, \delta \rangle$ be a navigation state machine. We say that $\mathcal{N}$ accepts σ if and only if $(\vec{v_1}, \epsilon, \perp^{|P|}) \in \delta((\perp^{|V|}, \perp, \perp^{|P|}), a_1, \vec{p_1})$, and for all $k > 0$, $(\vec{v_{k+1}}, a_k, \vec{p_k}) \in \delta((\vec{v_k}, a_{k-1}, \vec{p_{k-1}}), a_k, \vec{p_k})$.*

Based on the above definition, we can define what it means for a web application to conform to the navigation constraints specified as a navigation state machine.

DEFINITION 5. *Given an application $\mathcal{A}$ and a navigation state machine $\mathcal{N}$, we say that $\mathcal{A}$* conforms *to $\mathcal{N}$, if and only if all the session traces generated by $\mathcal{A}$ are accepted by $\mathcal{N}$.*

5. RUNTIME ENFORCEMENT

Note that at any point during the execution of the web application, if an action and a set of request parameters are requested by the user that causes a session trace to be rejected by the navigation state machine, then the web application should not execute that action. All the errors we discussed in Section 2 are caused by execution of actions that violate the navigation constraints. If we can find a mechanism that guarantees that such actions are not executed by the web application, then we can eliminate navigation errors.

5.1 The NSM Plugin

We achieve this by implementing a navigation state machine (NSM) plugin, which takes an NSM specification as input and enforces the navigation constraints specified by the input NSM during the execution of the web application.

We developed a simple XML based language for specification of NSMs. The NSM plugin is basically an interpreter for this language. A state is defined as a Module/Controller/-Action tuple. Given the hierarchical relationship between modules, controllers, and actions, it is also possible to specify a class of states as a Module/Controller tuple (contains a set of states, one for each action present in the specified controller), or even just a Module (contains a set of states, one for each action present in each controller present in the specified module).

The plugin has 1,100 lines of PHP code, which adds less than 3% to the source code size in the smallest application we tested. It operates proactively. As each request is routed, a decision is made as to whether or not the next state requested by the client is an acceptable transition from the previously visited state. In the event the request is prohibited, the current action is re-executed, effectively producing a "page refresh". In the particular case where the current action is undefined (which can only happen at the very beginning of a transaction), or when a page reload is flagged in the specification as not appropriate (such as when the last action executes a cash transfer), the initial state of the NSM is loaded.

Once the requested action has executed, control returns to the NSM plugin, in order to verify that the action has conformed to the specification. The plugin compares a snapshot of the session variables taken before the execution of the action with the values of the session variables after the action has executed, and ensures that: The action has modified only the variables it is allowed to modify, according to the specification, the variables have been updated to a "valid" value (a value identified in the specification), and the post-condition of the action has been established. If the executed action is a page refresh caused by a navigation error that was blocked, this second pass into the NSM plugin is skipped.

5.2 Compliant Web Applications

The NSM plugin enables us to convert a web application to a *compliant application* that does not allow navigation errors. This is achieved simply by inserting the NSM plugin right before the application's controller applies its navigation logic. For the Zend framework, the NSM plugin is registered with the application's front controller by a single line of code in the `Bootstrap.php` file. For the CodeIgniter framework, the plugin invocation is added just before the point where actions are called in the "bootstrap" file `CodeIgniter.php`.

Below, we formalize the semantics of a compliant web application that has been extended with the NSM plugin. In order for the plugin to remember the state of the application, an additional session variable is stored for each session, keeping track of the last state of the NSM in that session.

DEFINITION 6. *Let $\mathcal{A} = \langle Q, B, \Delta \rangle$ be an application and $\mathcal{N} = \langle S, s_0, \delta \rangle$ be a navigation state machine. The compliant application $\mathcal{A}_\mathcal{N} = \langle Q', B', \Delta' \rangle$ is an application where: $Q' \subseteq Q \times S^{|I|}$, $B' = B \times s_0^{|I|}$, and $(q', \vec{s'}) = \Delta'((q, \vec{s}), a, \vec{p}, i)$ if and only if $q' = \Delta(q, a, \vec{p}, i)$, $s_i' \in \delta(s_i, a, \vec{p})$ and $s_j' = s_j$ for each $j \in I$ and $j \neq i$ where s_i and s_j denote the i'th and j'th elements of the vector $\vec{s}$, respectively.*

A compliant application restricts the navigation behavior so that each session trace conforms to the navigation state machine $\mathcal{A}$. We formalize this property as follows:

THEOREM 1. *Given a web application $\mathcal{A}$ and a navigation state machine $\mathcal{N}$, the compliant application $\mathcal{A}_\mathcal{N}$ conforms to the navigation state machine $\mathcal{N}$.*

6. MODEL CHECKING AN NSM

By automatically analyzing an NSM using model checking, we can iteratively refine its specification and discover and fix potentially critical errors in navigation constraints before they manifest themselves at runtime. Without a specification mechanism like navigation state machines it would be extremely difficult to automatically analyze navigation constraints of a web application at the implementation level, since navigation errors may only manifest themselves after a series of client-server interactions and they may be hard to detect.

6.1 ACTL Model Checking

Navigation constraints can be expressed in terms of the actions, session variables and request parameters used by the application. To this end, we employ the universal fragment of CTL, called ACTL. The ground terms of an ACTL formula are defined as follows: if x, y are session variables or request parameters ($x, y \in P \cup V$) and a is an action ($a \in A$), then $x = y$ and $x \neq y$, a and $\neg a$ are ground terms. ACTL formulæ are then built from ground terms using the standard Boolean connectives $\vee$ and $\wedge$ and the CTL temporal operators **AX** (in the next state of every trace), **AG** (in every state of every trace), **AF** (in some state of every trace) and **AU** ($\mathbf{A}(\varphi \mathbf{U} \psi)$ holds when, for every trace, φ is true until ψ is true).

Intuitively, the satisfaction of a ground ACTL formula depends on the *last* action taken and the request parameters for that action, and the current values of the session variables. An ACTL formula φ is satisfied by an NSM $\mathcal{A}$, denoted as $\mathcal{A} \models \varphi$, if for every initial state s of $\mathcal{A}$ we have $s \models \varphi$. Since the session projection is also a form of NSM, the same semantics applies to it as well. Given an NSM, our plugin produces an SMV model that is sent to the NuSMV model checker [6].

To relate the verification results obtained by analyzing a NSM to the application extended by that NSM, we need to define a projection from an application to a session.

DEFINITION 7. *Let $\mathcal{A} = \langle Q, B, \Delta \rangle$ be an application where $Q \subseteq M \times D^{|V| \times |I|}$, and let $i \in I$ be a session. The projection of $\mathcal{A}$ to session i is an NSM, i.e., $\Pi(\mathcal{A}, i) = \mathcal{N} = \langle S, s_0, \delta \rangle$ where*

- $S \subseteq D^{|V|} \times A \times D^{|P|}$ *is the set of states.*

- $s_0 = (\perp^{|V|}, \perp, \perp^{|P|})$ *is the initial state.*

- $\delta : S \times A \times D^{|P|} \to 2^S$ *is the transition relation where $(\vec{v'}, a', \vec{p'}) \in \delta((\vec{v}, a, \vec{p}), a', \vec{p'})$ if and only if there exist $q, q' \in Q$ such that $q' = \Delta(q, a', \vec{p'}, i)$ and $q'(i) = \vec{v'}$ and $q(i) = \vec{v}$.*

The projection of an application $\mathcal{A}$ to session i is the NSM that recognizes exactly the set of navigation traces for session i in $\mathcal{A}$.

THEOREM 2. *Let $\mathcal{A}$ be an MVC application, and $i \in I$ be some session. A navigation trace σ is a trace of $\Pi(\mathcal{A}, i)$ if and only if there exists a trace σ' of $\mathcal{A}$ such that $\pi(\sigma', i) = \sigma$.*

	Digitalus	BambooInvoice	Capstone
States	66,400	181,600	306
BDD nodes	3,157	2,457	1,654
Memory	4.44 MB	4.42 MB	4.38 MB
Model building	0.4 s	0.4 s	0.1 s

Table 2: Summary of model checking results.

Application	States	Transitions	Variables
Digitalus	32	48	7
BambooInvoice	63	80	8
Capstone	8	16	1

Table 3: Size of navigation state machines for each application

The following theorem shows that for a compliant application $\mathcal{A_N}$, the navigation projections for each session are guaranteed to satisfy φ, provided that the NSM $\mathcal{N}$ satisfies it.

THEOREM 3. *Let $\mathcal{A}$ be an application, $\mathcal{N}$ be an NSM and φ be an ACTL formula. If $\mathcal{N} \models \varphi$, then for every session $i \in I$, $\Pi(\mathcal{A_N}, i) \models \varphi$.*

PROOF. Suppose the contrary; that is, $\mathcal{N} \models \varphi$, but there exists some session i such that $\Pi(\mathcal{A_N}, i) \not\models \varphi$. Since φ is in the universal fragment of CTL, this entails that there exists a trace of $\Pi(\mathcal{A_N}, i)$ which is a counter-example of φ. But by Theorem 1, we know that every trace of $\Pi(\mathcal{A_N}, i)$ is a trace of $\mathcal{N}$, hence there is also a counter-example of φ in $\mathcal{N}$, a contradiction. $\square$

This theorem has an important consequence. Although navigation state machines provide a mechanism for specification and verification of navigation constraints, any conclusions drawn from such a verification effort are valid only if we can guarantee that a web application conforms to a given navigation state machine specification. By the previous result, the seemingly daunting task of guaranteeing conformance to a specific navigation model across an entire application becomes a manageable task for web applications built based on the MVC architecture. In other words, the verification results we obtain by model checking a navigation state machine are guaranteed to hold for a compliant web application that uses that navigation state machine.

6.2 Sample Navigation Properties in ACTL

Equipped with this theorem, it is now possible to perform formal verification of some navigation properties in the web applications we studied. The plugin we developed automatically translates an NSM into an input file to the NuSMV model checker. The results are summarized in Table 2.

As one can see from these figures, building and loading the model from each NSM is a trivial task, and takes NuSMV less than a second for all three applications. Indeed, none of the NSMs generated SMV models of more than 200,000 states and used more than a few megabytes of memory to be processed.

We identified several generic navigation properties that can be used for many applications. We regard these properties as good indicators of the correct navigation design of an application; consequently, a routine check of the following ACTL formulas could be used to identify potential navigation issues if they are violated.

1) An "index" action must be eventually executed:

$$\mathbf{AF} \, (\text{action} = \text{index})$$

The "index" actions are generally intended as the starting point; consequently if an "index" action is never executed this hints at a navigation problem.

2) Once you login, the only way to go back to the login page is by traversing the logout page:

$$\mathbf{AG} \, (\text{controller} = \text{login} \rightarrow$$
$$\neg \mathbf{A} \, (\text{controller} = \text{login} \, \mathbf{U} \, \text{controller} = \text{logout}))$$

This property summarizes the expected behavior of a login system; we have seen that it is false both in the plain Digitalus and BambooInvoice applications. However, the NSM of both applications satisfies this property; by the previous theorem, the compliant extensions of Digitalus and BambooInvoice satisfy it as well.

3) Each controller c has the "index" action as its only entry point from other controllers:

$$\mathbf{AG} \, (\text{controller} \neq c \rightarrow$$
$$\mathbf{AX} \, (\text{controller} \neq c \, \vee \, \text{action} = \text{index}))$$

This property is similar in spirit to some form of visibility mechanism: clearly, two controllers do not implement separate functionalities if they refer to pages other than their respective entry points. The property is false in Digitalus, both with and without the NSM plugin. Indeed, the action admin/site/index calls admin/user/open; this is expected by the application and must be allowed by the NSM. The property is also false in BambooInvoice, where the "clients" controller calls non-index actions in the "clientcontacts" controller, and the "invoices" controller calls non-index actions in the "clients" controller.

Due to the small size of the models (at least according to model checking standards), verification times for these properties are negligible. NuSMV's internal `time` function returned an elapsed time of 0 seconds for all properties we tested. This is due to session modularity, and should be contrasted with the expected model of the application, if all sessions had to be considered.

7. EMPIRICAL EVALUATION

For each of the three applications described in Section 2, we defined a navigation state machine. This NSM was created manually, in parallel to an exploration of the possible navigation paths in the application. Guards were later added to the transitions, when request parameters or session variables were required to execute a particular action. Statistics about these state machines are summarized in Table 3. Equipped with these NSMs, we evaluated the performance of the NSM plugin on the resulting extended MVC applications.

Building the NSM for each application took roughly 4 hours; this approximation is generous, as it also includes the time required to carefully write down sample navigation errors for the purpose of this paper, as well as the capture of screenshots for each step of these errors. Yet, in general, we believe that the specification of an NSM, which should be done along with the development of the application, should

not be too daunting. Following a natural design principle, each module should be self-contained and implement a unit functionality; hence, it should have an "index"-like action as its only entry point from the other modules (this is one of the properties we checked in Section 6.2). In doing so, cross-module dependencies are limited; what remains to specify is the behavior of each module separately, which consist each of only half a dozen actions and transitions in our examples. Case in point, lots of errors in the web applications we studied are caused by cross-module dependencies for pages other than the "main" module page.

7.1 Error Prevention

A first, qualitative component of evaluation is the plugin's capability to prevent navigation errors, and in particular the ones described in Section 2. Therefore, our first experiment was to reproduce the sequences of actions for navigation errors Error 1-5 while the NSM plugin is in action, and to compare the pages received with the original application's response. We observed that all five navigation error types were caught by the plugin; on the last step of each sequence, instead of returning an error page (or error messages within a page), the NSM plugin prevents the faulty action from being executed and simply recalls the last valid page.

This readily presents advantages in terms of user experience (by allowing a graceful handling of all navigation errors) and security (by preventing implementation details from leaking through uncaught error messages). However, the NSM plugin does not merely catch and hide error messages once they are triggered; it actually *prevents* them by diverting the control flow before anything happens. Hence, in the case of Navigation Error 1, the "user-delete" action's side effect (removing a user from the database) is avoided instead of coped with after the fact.

Apart from these qualitative considerations, we were also interested in estimating the proportion of navigation errors that each application would let through, were it not for the presence of the plugin. As we have seen, without a navigation state machine, it is possible to jump from any action to any other. If we denote by $N(s, k-1)$ the number of *valid* traces of length $k-1$ ending in state s, then the number of traces of length k, which are valid for the first $k-1$ actions and which violate the NSM at the k-th is given by:

$$N_v(k) = \sum_{s \in S} N(s, k-1) \cdot |\{s' \in S : s \not\rightarrow s'\}|$$

i.e. we extend each valid trace of length $k-1$ ending in s by one of the actions s' that are not adjacent to s in the NSM, giving us a "bad" trace of length k.

However, in some occasions, violating the NSM has no undesirable side effects; this is the case for all actions whose incoming transitions are not guarded. However, the actions that have guards, if called from the wrong previous page, will have their precondition violated; the results of such violation have been shown in Section 2. Let S_g be the set of such "guarded" actions in the NSM. The number of traces which violate a guard at the k-th step is given by:

$$N_g(k) = \sum_{s \in S} N(s, k-1) \cdot |S_g \cap \{s' \in S : s \not\rightarrow s'\}|$$

i.e. we restrict ourselves to non-adjacent actions that are guarded. Therefore, the proportion of "bad" traces of length

Application	No Plugin	With Plugin
Digitalus	11	12
BambooInvoice	183	199
Capstone	90	122

Figure 5: Server processing time, in milliseconds, for NSM-compliant actions, both with and without the NSM plugin.

k which actually cause errors is given by $N_g(k)/N_v(k)$. This ratio represents the proportion of traces for which a navigation constraint is assumed, but not checked, by the application. We computed this ratio for increasing values of k, using both the NSM for the Digitalus CMS and for BambooInvoice.

The resulting curve converges to a stable value after k reaches 5, which is approximately the depth of the application's state machine in both cases. With Digitalus, when a user (or an automated web crawler) invokes the "wrong" action from its current point in the application, there is approximately a 52% chance that this error will not be caught by the application and create a page containing PHP errors or warnings, as shown in Figures 2-3. With BambooInvoice, this chance is 64%. Alternately, this indicates that in both applications, the majority of the pages blocked by the NSM plugin would actually cause errors without the plugin.

One can also consider the number of traces that the NSM prevents, but which would not have caused an error in the original application. While the number of errors caught by the NSM gives us information about the application's success (or lack thereof) at enforcing its own navigation logic, the converse only gives us information about how conservative is a particular instance of NSM for that application. For this reason, we did not compute this measurement.

7.2 Impact on Processing Time

We then measured the incurred costs of using the NSM plugin on all three applications. In a first time, a *valid* sequence of operations, producing no errors and visiting all the actions (including create/delete operations), was manually executed on each application. The application was instrumented to record and write to a file the start and end time of each PHP processing request. The same sequence was then re-executed, this time with the NSM plugin registered and running. The average processing time per request in each application is shown in Figure 5; the times were measured on an AMD 1.4 GHz Windows XP machine, using 768 MB of RAM and running Apache 2 and PHP 5.1.

While this figure shows that the NSM plugin does induce processing overhead, during normal use of an application, this overhead is negligible: 2% in the case of Digitalus, and 9% in the case of BambooInvoice. In absolute numbers, this amounts to at most 20 milliseconds per request in the worst case. Moreover, in its present version, the plugin has to load and parse the state machine's XML file on each page request; therefore, these figures should be seen as an upper bound as they could easily be improved.

In addition to processing overhead in valid requests, we studied the performance consequences of error processing. To this end, we selected actions in each application and called them out of context to deliberately violate the NSM specification. We retained the situations where the application produced uncaught errors: errors where the application

Page	No Plugin	With Plugin
admin/media/create-folder	28	26
admin/media/upload	18	32
admin/user/edit	22	32
admin/user/open	14	24
invoices/newinvoice	938	574

(a) Warnings

Page	No Plugin	With Plugin
admin/page/edit	416	32
admin/media/delete-folder/folder	424	36
admin/user/delete	434	22
invoices/view	564	594

(b) Errors

Figure 6: Server processing time, in milliseconds, for pages that produce PHP warnings and fatal errors, both with and without the NSM plugin.

is still able to load part of the page, or which display only PHP errors of the class "warning" or "notice", as in Figure 2, and errors where the application crashes and the error page is produced by PHP itself; this is generally the case for "fatal errors", as in Figure 3.

For each action, we calculated its processing time (including the PHP error handling), repeating the operation 5 times and taking the average of those measurements. We then proceeded to the same invocation, this time with the NSM plugin enabled. In that situation, no PHP error is produced, and the application simply reloads the last valid page. Sample processing times for various warnings and errors are shown in Figure 6.

While in some cases the NSM plugin induces the same overhead as before, surprisingly, in some cases the running time with the plugin is *lower* than without the plugin. In these situations, it takes more time for PHP to execute the script and generate the warnings, than it takes to simply run the NSM plugin, determine that the requested action violates the state machine, and reload the last valid page. This phenomenon is even more marked in the case of fatal errors, where the action does not execute at all and PHP simply returns a white page with an error message. In the case of Digitalus, running the NSM plugin is more than ten times faster than letting the application provoke a fatal error.

Therefore, although the plugin induces a slight overhead over the normal use of an application, its capability to catch navigation errors can actually save server processing time.

8. RELATED WORK

The problem of navigation inconsistencies in web applications has been described in earlier work by Licata et al. [17], where it has been shown that multiple browser windows can lead the user of a popular travel reservation site to purchase the wrong flight. To alleviate the problem, the authors suggest the development of a new programming language based on Scheme, in which such navigation inconsistencies reduce to type checking errors [16]. In contrast, the approach presented in this paper neither warrants the introduction of a new programming language, nor does it require any substantial modification to an existing application. Moreover, since NSMs are based on statecharts, a common specification formalism that is part of UML, we believe that their adoption will not require a significant learning effort.

Modeling web applications as state machines was suggested a decade ago [20], and has already been used to automatically generate test sequences [21] and perform some form of model checking [19]. The verification of standard, non-MVC web applications, has also been attempted numerous times in recent years [8–10, 14, 18]. There has also been work on specification of interfaces between two components as state machines [3, 7]. However, the usefulness of these works is mitigated by one key limitation: one cannot assume that the navigation flow intended by the programmer is the only possible path of execution in the implementation. Indeed, as much as formal, finite state models are used for testing and verification, none of the previous works actually use this model to *enforce* proper navigation sequences on a running application, as the NSM plugin allows.

Guha et al. [11] statically analyze an Ajax application on the client side and extract a control-flow graph from its source code. A monitor then enforces, on the server side, that the requests received follow the extracted client model. This approach cannot be used to prevent navigation errors, since the specification is based on the client's code, which may not correspond to what the server expects. Han and Hofmeister [13] define a formal navigation model for web applications based on statecharts. However, the model does not include request parameters such as HTTP GET or POST fields; consequently, most navigation constraints presented in our work can not be handled. This remark also applies to the Dialog Flow Notation [5]. The Spring Framework[3] allows the definition of a state machine called "web flow"; however the flow's identifier is passed as a parameter in the URL for each action, which makes the application vulnerable to sequences involving bookmarks, as in Navigation Error 2. Book et al [4] present a formal model for specifying input validation rules for web applications and present a framework where an implementation can be generated from the formal specification. In contrast, our approach combines the constraints on sequencing of user actions with constraints on input data. Moreover, our approach can be integrated in existing applications and does not require users to adopt a new framework for developing web applications. Finally, none of the above approaches combine model checking with runtime enforcement or provide an empirical analysis of the overhead incurred by runtime enforcement.

The enforcement of NSMs on a web application can also be viewed as a form of *runtime monitoring* of linear temporal logic properties [2]. Such a monitor has been developed for Ajax web applications communicating with web services through SOAP [12]. Its working principle is similar to the NSM plugin, but on the client side instead of the server side; therefore, it does not have access to session variables like the NSM plugin. Moreover, the use of LTL as a specification language can make the specification of some navigation constraints problematic, as LTL is too weak to encode a finite-state machine. It also raises the issue of model checking $ACTL$ properties, as in Section 6.1, over a system defined by a conjunction of LTL formulas.

Finally, the use of the MVC design pattern to centralize navigation decisions bears some similarity to aspect-oriented programming (AOP). The NSM plugin, registered in the MVC application, acts as a "pointcut" that is triggered whenever a new page is requested; the advice associated

---

[3]http://springsource.org/webflow

with that pointcut is used to enforce the navigation state machine. In that respect, the NSM plugin could also nicely fit into an aspect-oriented web application framework [15].

9. CONCLUSION

We have shown that web applications mishandle unexpected navigation sequences caused by the uncontrolled use of a web browser's navigation functionalities. In particular, we calculated that more than half of all possible unexpected navigation sequences are improperly handled in three real-world applications we studied.

We showed that the navigation state machines presented in this paper can help relieve these problems by restricting the control-flow of an application to sequences intended by the developer. Our main conclusion is that the benefits of the extra layer of verification largely outweigh its overhead. This is particularly true if some unexpected user requests produce PHP error pages. In this case, in addition to producing a cryptic message possibly revealing implementation details, processing the error can incur a CPU overhead of as much as 10 times the time required to check the error and redirect to a correct page.

As future work, we plan to investigate automatic extraction of navigation state machine skeletons, either through automated crawling or by analyzing the source code. In addition, the implementation of the plugin as a reverse proxy directly in the HTTP server could detach its application from any language or MVC platform.

10. REFERENCES

[1] *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria.* IEEE Computer Society, 2004.

[2] D. A. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In R. Hariharan, M. Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 49–60. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[3] A. Betin-Can and T. Bultan. Verifiable web services with hierarchical interfaces. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2005*, pages 85–94, 2005.

[4] M. Book, T. Brückmann, V. Gruhn, and M. Hülder. Specification and control of interface responses to user input in rich internet applications. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 321–331, 2009.

[5] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *ASE* [1], pages 100–109.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[8] L. Desmet, P. Verbaeten, W. Joosen, and F. Piessens. Provable protection against web application vulnerabilities related to session data dependencies. *IEEE Trans. Software Eng.*, 34(1):50–64, 2008.

[9] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 772–774. ACM, 2006.

[10] F. M. Donini, M. Mongiello, M. Ruta, and R. Totaro. A model checking-based method for verifying web application design. *Electr. Notes Theor. Comput. Sci.*, 151(2):19–32, 2006.

[11] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, editors, *WWW*, pages 561–570. ACM, 2009.

[12] S. Hallé and R. Villemaire. Browser-based enforcement of interface contracts in web applications with BeepBeep. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 648–653. Springer, 2009.

[13] M. Han and C. Hofmeister. Relating navigation and request routing models in web applications. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 346–359. Springer, 2007.

[14] M. Haydar. Formal framework for automated analysis and verification of web-based applications. In *ASE* [1], pages 410–413.

[15] K. Hokamura, N. Ubayashi, S. Nakajima, and A. Iwai. Aspect-oriented programming for web controller layer. In *APSEC*, pages 529–536. IEEE Computer Society, 2008.

[16] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen. *Modeling Web Interactions and Errors*, pages 255–275. Springer, 2006.

[17] D. R. Licata and S. Krishnamurthi. Verifying interactive web programs. In *ASE* [1], pages 164–173.

[18] H. Miao and H. Zeng. Model checking-based verification of web application. In *ICECCS*, pages 47–55. IEEE Computer Society, 2007.

[19] E. D. Sciascio, F. M. Donini, M. Mongiello, R. Totaro, and D. Castelluccia. Design verification of web applications using symbolic model checking. In D. Lowe and M. Gaedke, editors, *ICWE*, volume 3579 of *Lecture Notes in Computer Science*, pages 69–74. Springer, 2005.

[20] P. D. Stotts, R. Furuta, and C. R. Cabarrus. Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Trans. Inf. Syst.*, 16(1):1–30, 1998.

[21] S. Yuen, K. Kato, D. Kato, and K. Agusa. Web automata: A behavioral model of web applications based on the MVC model. *Information and Media Technologies*, 1(1):66–79, 2006.