Web Automata: A Behavioral Model of Web applications based on the MVC model

Shoji Yuen  Keishi Kato  Daiju Kato  Kiyoshi Agusa

We propose a behavioral model of web applications, called 'Web Automata', based on the MVC(Model View and Control) model architecture. The MVC model architecture separates design concerns to improve the overall software quality. Since the architecture only defines the abstract outline of the configurations, there is a broad gap between web application codes and their behavioral properties.

We model the behavior of a web application with dynamic contents as an extension of links-automata proposed by Stotts et al. with the constraint-logic feature of the Extended Finite Automata, EFA for short, by Sarna-Starosa and Ramakrishna. As extended in the model checking techniques, we view a web application as a data-independent system, where variables appearing in link parameters and form inputs are attached to each page.

We present a testing framework for web applications based on the behavioral model. We show it provides reasonable testing criteria for web applications when we focus on the loops in the Web automaton model. We apply our framework to the Jakarta Struts by presenting the extended configuration schema of Struts in order to describe the web automata directly.

1 Introduction

Web applications has now become one of the major forms of the information systems over the computer network based on the World Wide Web technology. The fundamental feature of web applications is that its behavior is specified by the interaction between the environment and the system. An execution of a web application does not simply follow the source descriptions.

One of the major platform to build web applications is J2EE [3] based on the MVC model [11]. The J2EE platform is the set of basic APIs to construct the web applications specified by Sun Microsystems. The MVC model gives the architectural pattern to build up a web application from the components over the APIs. A framework gives a concrete method for the building pattern. A framework based on the MVC model such as Struts [13] configures the components over Servlets and JSPs to build up a web application.

Because of the reactive feature of components in web applications, the behavioral description results in being scattered over the components. Therefore we are motivated to give a simple behavioral model in the framework.

Our basic idea is that a web application behaves as an automaton where the states are the pages and the state transitions are invoked by the requested URLs. Since in a web application a page may be dynamically generated, links in the page may depend on the execution history. To model this behavior, we extend a classic automaton to at-
tach variables to control executions following EFA proposed by Sarna-Starosta and Ramakrishnan [10], where we consider a web application is a data-independent system [15]. By adding a control variable to states, (called locations in [10]), we model the dynamic behavior of the contents. We call EFAs applied to web application ‘Web Automata’.

Based on the model, we present a testing framework for web application. Given a behavioral model by our web automaton, testing has following two procedures: (1) test if all intended words are included in the language of the web automaton, and (2) for each word test if the behavior and appearances are correct.

We assume that there is an entry page for the application such as the welcome page and there are some final pages like thank-you pages or error pages. We consider this assumption is enough for the testing purpose. A testing sequence is characterized as a word of the behavior automaton. The language comprised of such words is generally infinite with respect to the following two aspects. First, because the forms can accept infinitely many data, input requests can be infinitely many and so are the output responses. Second, because the automaton may have loops, the length of the sequences can be infinite.

For the first, we regard the pages generated from one JSP page are equivalent. We distinguish transitions to different JSP pages. Since we are focusing testing aspect, this abstraction is justified in that more refined tests may be performed within each JSP pages. We abstract the sequences with concrete data to those with finite abstract data, where the abstract data just distinguish the transitions. We use the technique to obtain abstract runs from concrete runs in EFA [10] to achieve this abstraction. For the second, in web applications, returning to the page previously visited often gives the similar transitions afterwards. We consider revisiting pages as refining the tests.

Based on this idea, we give a systematic way to generate testing criteria for a web application. Each criterion contains finitely many words being classified by the number of loops. We illustrate the effectiveness of our approach by a simple example of a library management system. By an experimental implementation [8] to generate testing sequences given a web automaton, it is shown that the number of generated testing sequences is reasonably small.

The remainder of this paper is organized as follows. In section 2 briefly recalls the MVC model architecture and the struts framework. Section 3 presents our behavioral model and Section 4 proposes the testing criteria based on our extended framework. Section 5 presents the actual application of our extension to Jakarta Struts. We review the related work in Section 6 and in Section 7 we state concluding remarks and future work.

## 2 MVC Model of Web Applications

The MVC model of web applications consists of three types of components: Model, View and Controller. The MVC model is originally proposed for GUI applications as seen in Smalltalk. The MVC model, often called the “MVC model 2”, of web applications is different as follows: the Model-typed components represent the data domain, the View-typed components treat the outputs to the clients, and the Control-typed components define the operations over the data. The MVC model is commonly implemented by the Servlet/JSP technology over the J2EE platform. Figure 1 shows the basic architecture we refer to as the MVC model over J2EE. In this paper, we simply call this architectural model ‘the MVC model’.

The typical scenario in this architecture is as follows: A request is first handed to the controller. The controller processes some data in the Model-typed components according to the request. While processing the Model-typed components, the components may access the background databases. As the output, the controller will dispatch a JSP and the dispatched JSP is presented as the response to the request.

![Fig.1 MVC Model over J2EE using JSP/Servelt](image-url)
Using the JSP/servlet technology, the Model-typed components are implemented by JavaBeans, the View-typed components by Java Server Pages, and the Controller-typed components by the Action servlets. The J2EE container, such as tomcat, combines the components according to a configuration information that specifies the connection between components.

Struts is an open-source web application framework being developed in Jakarta Project. Struts uses the JSP/Servet technology following the MVC model stated above.

Figure 2 shows the concrete class structure of the Struts framework. The behavior based on the MVC model is implemented as follows in Struts: ActionServlet is a general factory class that takes the external request and dispatch an appropriate 'Action' servlet. Action class defines an abstract class invoked by the ActionServlet. A developer specifies the control behavior by defining the subclass of Action class. An Action servlet refers an 'ActionForm' object and do some operation. The result is wrapped in an 'ActionForward' object to be sent back to the ActionServlet. The ActionServlet dispatches a JSP according to the ActionMapping where the mapping between the ActionForward to JSP is described. Finally the dispatched JSP is presented as the response. While this presentation, the JSP may do some processing according to its dynamic contents.

3 Web Automata

A primitive view of external behavior of web applications based on the MVC model is the sequence of pairs of requests and JSPs. Following a programming convention, a dispatched JSP is naturally considered as a state of the program. Sending a request to the application causes the state transition. We model the behavior of web applications characterized by a type of automata extended with variables. We formalize our behavioral model as a simplified version of EFA[10].

In the followings, we restrict our attention to the JSP/servlet technology in J2EE framework, especially Struts[13]. The notion should be mapped easily to other framework such as Turbine[14] as long as it follows the MVC model. The states are the set of JSPs. The alphabet of the automata is the set of a request and a response to and from the action servlet, where those requests and responses are described in the configuration file.

Following EFA, we start with the constraint language definition. In our model, all the constraints are the comparison with strings.

Definition 1 (Web Constraint Language)

A set of primitive constraint \( PC_{(=)} \) is defined as follows:

\[
PC_{(=)} = \{ v_1 = v_2, v_1 \neq v_2 \mid v_1, v_2 \in V \cup C \}
\]

where \( V \) is a set of variables and \( C \) is a set of string constants;

A constraint language over the primitive constraint is a smallest set that satisfies following conditions:

1. \( PC \subseteq L_{(=)} \)
2. If \( \varphi_1, \varphi_2 \in L_{(=)} \), then \( \varphi_1 \land \varphi_2, \varphi_1 \lor \varphi_2 \in L_{(=)} \)
3. If \( \varphi \in L_{(=)}, v \in V \), then \( \exists v. \varphi \in L_{(=)} \)

A set \( [\varphi] \) of substitution mapping variables in \( \varphi \) to \( C \) with each assertion \( \varphi \). A substitution \( \sigma \in [\varphi] \) satisfies an assertion \( \varphi \) and is written as \( \sigma \models \varphi \).

When an execution follows a link, some variables may be instantiated by assignment.

Definition 2 (Assignment)

An Assignment is written as \( x := s \) where \( x \in V, s \in V \cup C \). The meaning of an assignment \( x := s \) is a binary relation of substitution such as

\[
(\sigma, \sigma') \in [x := s] \text{ iff } \sigma' = \sigma[s/x]
\]

The following definition gives the characterization of web applications.

Definition 3 (Web Automaton)

Given a web application \( A \), an extended finite au-
Web Automaton

\( Web_A = \{ \text{View}, \delta, \iota, \text{Trans}, FV \} \) is a Web Automaton where:

- View: a set of views;
- \( \delta = \text{View} \times 2^V \): the variable map, is a function that maps each view \( v \) to a set of variables local to the view \( v \);
- \( \iota: \) a function that maps each view to an assertion. \( \iota(v) \in L_{(-)} \) and \( \iota(v) \) is satisfied by \( \delta(v) \) whenever \( v \) is reached;
- \( iv \in \text{View}: \) The initial view page in \( A \);
- Trans: the transition relation such that for all \( \langle v_i, v_j, \langle \gamma, \alpha, \rho \rangle \rangle \in \text{Trans} \)
  - \( v_i, v_j \in \text{View} \) are the source and destination view of the transition respectively
  - \( \langle \gamma, \alpha, \rho \rangle \) is a transition label such that:
    - * \( \gamma \in L_{(=)} \) is an enabling condition; an assertion which specifies the condition over the set \( \delta(v_i) \);
    - * \( \alpha \in \text{Act} \) is associated with transition; \( \text{Act} \) is consist of url?variables where url \( \in C \), variables \( \subseteq V \) and \( \sigma \) is a substitution. url represents an URL request, variables represent parameters along with the URL request; and
    - * \( \rho \) is an update relation: a set of assignments defining the values in the destination view \( v_j \) by the values in the source view \( v_i \); and
  - \( FV \subseteq \text{View}: \) a set of the final views in \( A \)

For Struts, Views and Trans correspond to JSPs and action-mapping respectively. Variables are the parameter variables appearing on links and forms.

Starting from a view, a concrete run is defined as an actual sequence of actions invoked by links. In a concrete run, all variables are instantiated by some strings.

**Definition 4 (Concrete run of Web Automaton)**

Given a web automaton

\( Web_A = \{ \text{View}, \delta, \iota, \text{iv}, \text{Trans}, FV \} \), a concrete run \( \text{exec} \) of \( Web_A \) is defined as follows:

- \( \text{exec} \in \text{Exec} \): an alternating sequence of concrete views and actions, that is,
  \( \text{exec} = \langle v_0, \theta_0 \rangle \alpha_0(v_1, \theta_1) \alpha_1 \cdots (v_n, \theta_n) \)
- \( v_0 = \text{iv} \)
- for all \( i(v_i, v_{i+1}, \langle \gamma, \alpha, \rho \rangle) \in \text{Trans} \) such that:
  - Transition is enabled: \( \theta_i \models \gamma \)
  - Parameter values are bound: \( \theta_i' \) is a ground extension of \( \theta_i \) such that \( \alpha_i = \alpha[\theta_i'] \)
  - Data is transferred from source to destination: \( \theta_i, \theta_i' \in [\rho] \)
  - Parameter values are transferred: \( \theta_i'_{i+1} = \theta_i' \circ \sigma \) where \( \sigma \) is such that \( \theta_i = \theta_i' \circ \sigma \)
  - Destination assertion holds: \( \theta_{i+1} \models \iota(v_{i+1}) \)

For the last view \( \langle v_n, \theta_n \rangle \) of a concrete run, if \( v_n \in FV \), then the run is called a concrete experiment. The set of all concrete experiments of \( Web_A \) is written as \( E_c(\text{Web}_A) \).

As for EFA, an abstract run is defined as follows.

**Definition 5 (Abstract run of Web Automaton)**

Given a web automaton

\( Web_A = \{ \text{View}, \delta, \iota, \text{iv}, \text{Trans}, FV \} \), an abstract run of \( Web_A \) is defined as follows:

- \( \text{exec}_a \in \text{Exec}_a \) is a sequence of alternating states and actions, that is,
  \( \langle v_0, \varphi_0 \rangle \alpha_0(v_1, \varphi_1) \alpha_1 \cdots (v_n, \varphi_n) \)
- \( v_0 \in \text{iv}, v_i \subseteq FV \) and \( \varphi_0 = \iota(v_0) \)
- for all \( i(v_i, v_{i+1}, \langle \gamma, \alpha, \rho \rangle) \in \text{Trans} \) such that:
  - Transition is enabled: \( \varphi_i \land \gamma \) is satisfiable
  - Constraint is transferred to destination view: \( \varphi_i' = \Xi_\rho(\varphi_i) \) and \( \varphi_{i+1} = (\exists ! \varphi_i') \land \iota(v_{i+1}) \)
  - Parameter values are transferred:
    - \( \varphi_i = \varphi_i' = \Xi_\rho(\varphi_i) \) and \( \varphi_{i+1} = (\exists ! \varphi_i') \land \iota(v_{i+1}) \)

For the last view \( \langle v_n, \theta_n \rangle \) of an abstract run, if \( v_n \in FV \), then the run is called a abstract experiment or simply an experiment if no confusion arises. The set of abstract experiments of \( Web_A \) is written as \( E_a(\text{Web}_A) \).

**Proposition 6**

- If a concrete state \( \langle v_i, \theta_i \rangle \) is reachable in a concrete run, then there exists an abstract run in which an abstract state \( \langle v_i, \varphi_i \rangle \) is reachable such that \( \theta_i = \varphi_i \) and
- if an abstract state \( \langle v_i, \varphi_i \rangle \) is reachable in an abstract run, then for all \( \theta_i \) such that \( \theta_i = \varphi_i \), a concrete state \( \langle v_i, \theta_i \rangle \) is reachable in a concrete run.
Example (Library Management System)

Figure 3 shows an example of the behavior model of a simple library management system. The behavior model WebLib for figure 3 is constructed in figure 4.

This system manages the database of book titles when purchasing new books and modifying the existing entries for books already in the library. This system consists of three functions. One is to search books registered in this system. This function is executed by search.jsp, QueryAct, list.jsp, and detail.jsp without certification. The others are to register new books and to modify registered ones. These functions are executed by modify.jsp, regist.jsp, ModifyAct, RegistAct, success.jsp, and failure.jsp. They need certification which consists of CertifyAct and login.jsp. Modify.jsp cannot be reached from search.jsp via CertifyAct because modification is unable unless a book is specified. So search.jsp can reach login.jsp, regist.jsp via CertifyAct. On the other hand, detail.jsp can reach modify.jsp because a book which one wants to modify is specified. We assume that detail.jsp cannot reach regist.jsp.

We show a concrete run and an abstract run of WebLib as follows. Note that if $\delta(v)$ is empty, we abbreviate $\langle v, \phi \rangle$ to $\langle v \rangle$. And if there is no variables, we simply write url for url?variables $\sigma$.

- A concrete run:

  $\langle \text{search} \rangle$
  $\langle \text{detail} \rangle$
  $\langle \text{login} \rangle$
  $\langle \text{modify} \rangle$

  We can get this run by considering as follows:
  First, The initial view; search transfers to detail by queryAct. QueryAct has a free variable book_ID. Here, we consider book_ID to be replaced by a string '5'.
  Second, detail transfers to login by certifyAct. certifyAct has a variable act and it must be replaced by a string 'mod'. An assignment along with the certifyAct is book_ID := book_ID. So the reached view; login has ['mod'/act,'5'/book_ID].
  Third, login transfers to modify by certifyAct. certifyAct has a free variable; user. We consider this to be replaced by a string 'kato'. And
\[
\text{View} = \{\text{search, list, detail, login, regist, modify, success, failure}\}
\]

\[
iw = \text{search}
\]

\[
FV = \{\text{list, detail, success, failure}\}
\]

\[
\delta(\text{detail}) = \{\text{book}\_ID\}\delta(\text{login}) = \{\text{act, book}\_ID\},
\]

\[
\delta(\text{regist}) = \{\text{act, user}\}, \delta(\text{modify}) = \{\text{act, user, book}\_ID\}
\]

\[
\iota(\text{regist}) = \{\text{act = 'reg'}\}, \iota(\text{modify}) = \{\text{act = 'mod'}\}
\]

\[
\text{Trans} = (\text{search, login}, \{\text{true, certifyAct?act[reg'/act], \{\}}\}),
\]

\[
(\text{search, list, (true, queryAct?\phi, \{\}}),
\]

\[
(\text{search, detail, (true, queryAct?book}\_ID, \{\}}),
\]

\[
(\text{list, detail, (true, link\_detail?\phi, \{\}}),
\]

\[
(\text{list, search, (true, link\_search?\phi, \{\}}),
\]

\[
(\text{detail, search, (true, link\_search?\phi, \{\}}),
\]

\[
(\text{detail, login, (true, certifyAct?act[mod'/act], \{bookID := book\_ID\})),
\]

\[
(\text{login, regist, (true, certifyAct?user, \{act := act\})),
\]

\[
(\text{login, login}, (true, certifyAct?\phi, \{act := act, book\_ID := book\_ID\})),
\]

\[
(\text{login, modify}, (true, certifyAct?user, \{act := act\})),
\]

\[
(\text{regist, success, (true, registAct?\phi, \{\}}),
\]

\[
(\text{regist, failure, (true, registAct?\phi, \{\}}),
\]

\[
(\text{modify, success, (true, modifyAct?\phi, \{\}}),
\]

\[
(\text{modify, failure, (true, modifyAct?\phi, \{\}}),
\]

\[
(\text{success, search, (true, link\_search?\phi, \{\}}),
\]

\[
(\text{failure, search, (true, link\_search?\phi, \{\}})
\]

Fig.4 Web Automaton for Library Management System

assignments with it are \(\text{act := act, bookID := book\_ID}\). Therefore, modify has substitutions; ['mod'/act,' przegląd'/book\_ID,' kató'/user].

Finally, modify transfers to success and the run ends because success is one of \(FV\).

In this run, book\_ID and user are free variables. These are not bound by any constraints. So these are represented as arbitrary concrete strings in concrete runs. However, based on notion of abstract run, such a variable is represented as “one” variable.

Following abstract run shows that concrete strings; '5' and 'kató' are represented as variables; SomeID and SomeUser, respectively.

An abstract run:

\[
\begin{align*}
\langle \text{search} \rangle \\
\text{queryAct?bookID[SomeID]/bookID} \\
\text{detail, \{bookID = SomeID\}} \\
\text{certifyAct?act[mod'/act]} \\
\text{login, \{act = 'mod', bookID = SomeID\}} \\
\text{certifyAct?user[SomeUser/user]} \\
\text{modify, \{act = 'mod', bookID = SomeID, user = SomeUser\}} \\
\text{modifyAct} \\
\langle \text{success} \rangle
\end{align*}
\]

4 Testing Framework

4.1 Testing Criteria

Checking all experiments in \(E_a(\text{WebA})\) is considered as an appropriate test of \(A\). \(E_a(\text{WebA})\) is still infinite in general. There are many ways to check all \(E_a(\text{WebA})\) finitely since usually \(E_a(\text{WebA})\) is a regular set because views are finite and links from each view are finite. However, from the software point of view, \(E_a(\text{WebA})\) is a very rough characterization of behavior. In the development process, the developers need to check the page details following the element of \(E_a(\text{WebA})\). In this paper, we characterize the importance of testing according to the nature of Web applications.

In view of testing, we focus on the property of web applications: a loop in \(\text{WebA}\) is a sequence to return to the page previously visited, and such revisiting page usually results in similar application of the logic components in the application. With this observation, the most interesting tests in \(E_a(\text{WebA})\) is the word with loops at most once. Since the logic components involve in state transitions, one loop is necessary in order to cover all transitions. By increasing the visits of JSPs, we test more properties of the web application. We
formalize this idea as a testing criterion.

WebLib has two kinds of loops: by the links between the final pages and the top page, and by the links for the authentication for update. The former kind of loops are the repetition of the similar transactions and the latter kind of loops are the idling loop until a user gets the authentication. To test both kinds of loops, it is enough to test the loops once. When a loop has some side effect such as the shopping cart, the iterated test may be necessary.

Given Web_A = (View, δ, τ, Trans, FV), let an experiment e ∈ E_a(Web_A). Also let occ(v_i, e) be the number of occurrences of v_i in e. We write Act(e) for the sequence of actions removing all View from e and V(e) for the set of views appears in e.

**Definition 7 (Loop Testing Criterion)** The Loop testing criterion for A with degree n, written as T^n_A defined as follows:

\[ T^n_A = \{\text{Act}(e)| e \in E_a(\text{Web}_A), \forall v \in V(e) : \text{occ}(e, t) \leq n + 1\} \]

We drop A from T^n_A when A is clear from the context.

It is obvious that T^0_A, T^1_A, ... is an increasing sequence and \(\bigcup_{n=0}^{\infty} T^n_A = E_a(\text{Web}_A)\).

In many cases T^0_A gives a set of substantial testing sequences. Figure 5 shows testing framework for web applications.

We apply the test following our test criteria from T^n according to the degree until the satisfactory result is obtained. The maximum degree of the testing criteria depends on the web application under testing. If there is no side effect in any loop, T^0 is enough.

**4.1.1 Example**

We apply our testing criteria to WebLib shown in the previous section.

The runs to construct T_0 is shown in figure 6. We have the loop testing criteria T_0 by removing V(e) in each e.

Note that the action servlet by “CertifyAct” is reinvoked many times but since it is not a JSP no effects on the testing criteria.

**4.2 Automated Test Generation**

We implemented an experimental prototype system[8] to generate testing sequence based on the Web automaton model in Prolog Cafe[2] where Pro-
(P1) \(\text{(search) queryAct}(\text{list})\)
(P2) \(\text{(search) queryAct?book_ID|ID_o|\text{detail, \{ID_o\}}})\)
(P3) \(\text{(search) queryAct}(\text{list})\text{link_detail|\text{detail, \{ID_o\}}})\)
(P4) \(\text{(search) queryAct}(\text{list})\text{link_detail|\text{detail, \{ID_o\}}})\)
\(\text{certifyAct?act}[\text{mod}/\text{act}]\{\text{login, \{mod\} = act, \{ID_o\}}\})\)
\(\text{certifyAct?user}[\text{User_o}]\{\text{modify, \{mod\} = act, \{ID_o, User_o\}}\})\)
\(\text{modifyAct}(\text{success})\)
(P5) \(\text{(search) queryAct?book_ID|ID_o|\text{detail, \{ID_o\}}})\)
\(\text{certifyAct?act}[\text{mod}/\text{act}]\{\text{login, \{mod\} = act}\})\)
\(\text{certifyAct?user}[\text{User_o}]\{\text{modify, \{mod\} = act, \{ID_o, User_o\}}\})\)
\(\text{modifyAct}(\text{success})\)
(P6) \(\text{(search) certifyAct?act[\text{reg}/\text{act}]})\)
\(\{\text{login, \{reg\} = act}\})\text{certifyAct?user}[\text{User_o}]\)
\(\{\text{regist, \{reg\} = act, \{User_o\}}\})\text{registAct}(\text{success})\)
(P7) \(\text{(search) queryAct}(\text{list})\text{link_detail|\text{detail, \{ID_o\}}})\)
\(\text{certifyAct?act}[\text{mod}/\text{act}]\{\text{login, \{mod\}/\text{act}\}}\})\)
\(\text{certifyAct?user}[\text{User_o}]\{\text{modify, \{mod\} = act, \{User_o, \{ID_o\}}\})\)
\(\text{modifyAct}(\text{failure})\)
(P8) \(\text{(search) queryAct?book_ID|ID_o|\text{detail, \{ID_o\}}})\)
\(\text{certifyAct?act}[\text{mod}/\text{act}, \{ID_o\}]\)
\(\{\text{login, \{mod\} = act \{ID_o\}}\})\text{certifyAct?user}[\text{User_o}]\)
\(\{\text{modify, \{mod\} = act, \{User_o, \{ID_o\}}\})\)
\(\text{modifyAct}(\text{failure})\)
(P9) \(\text{(search) certifyAct?act[\text{reg}/\text{act}]})\)
\(\{\text{login, \{reg\}/\text{act}\})\text{certifyAct?user}[\text{User_o}]\)
\(\{\text{regist, \{reg\} = act, \{User_o\}}\})\text{registAct}(\text{failure})\)

where
\(\text{User_o} \overset{\text{def}}{=} \text{user} = \text{SomeUser}\)
\(\text{User_r} \overset{\text{def}}{=} \text{SomeUser/user}\)
\(\text{ID_o} \overset{\text{def}}{=} \text{book_ID = SomeID}\)
\(\text{ID_r} \overset{\text{def}}{=} \text{SomeID/book_ID}\)

Fig.6 \(T^0\) sequences for Library Management System

Log is easily coupled with Java. The Prolog feature is useful for solving constraints and the Java feature is useful for various output formats of the test generation. Given a web automaton specification, the prototype system generates the set of testing sequences according to the specified testing criteria. Since the system directly solves the constraints on the web automata, it is able to check the reachability between pages. By using this function, the reachability property from the start page is easily checked.

4.2.1 Web Automaton Specification
A web automaton is specified by the following four types of predicates. Each state is given a unique constant to be identified. \(A\) is replaced by a constant to identify the web automaton\(^1\).

- **init**(\(A, \text{initial state id}\))
  asserts that state \(\text{initial state id}\) is the initial state.

\(^1\) Although this argument is not essential in solving constrains, it is added to load multiple web automata at a time for convenience.
init(lib2, search).
final(lib2, [list, detail, success, failure]).

inv(lib2, search, []). 
inv(lib2, login, [act(Act), id(BookID)]).
inv(lib2, regist, [act(Act), user(User)]: Act = reg.
inv(lib2, modify, [act(Act), id(BookID), user(User)]: Act = mod.
inv(lib2, list, []). 
inv(lib2, detail, [id(BookID)]).
inv(lib2, success, []). 
inv(lib2, failure, []). 

trans(lib2, search, [], queryAct, list, []). 
trans(lib2, search, [], queryAct([id(BookID)]), detail, [id(BookID)]). :- BookID_d = BookID. 
trans(lib2, list, [], link_detail, detail, [id(BookID)]). 
trans(lib2, detail, [id(BookID)], certifyAct([act(Act)]), login, [act(Act), id(BookID)]): Act = mod, BookID_d = BookID_s.
trans(lib2, search, [], certifyAct([act(Act)]), login, [act(Act), id(BookID)]: Act = reg.
trans(lib2, login, [act(Act), id(BookID)]), certifyAct([act(Act)],[id(BookID)]), user(User): Act = mod, BookID_d = BookID_s.
trans(lib2, login, [act(Act), id(BookID)],[certifyAct([act(Act)],[id(BookID)]), user(User)]: Act = mod, BookID_d = BookID_s, User_d = User.

Fig.7 Specification of Library Management System in Web Automaton

- **final(A, list_of_final_states)**
  asserts the members of list_of_final_states are the final states.
- **inv(A, state, list_of_parameters)**
  ::= constraint1,...,constraintn
  declares the constraints to hold at state. Each parameter is of the form:
  parameter_name(value), where value may be a variable. Each constraint is a list of equations to hold at the state.
- **trans(A, source, list_of_source_parameters, action, destination, list_of_destination_parameters)**
  ::= constraint1,...,constraintn
  declares that there is a transition between source and destination labeled by action under the constraints of constrainti. The parameters are given in the same way as inv predicates.
4.2.2 Test Generation and Reachability Analysis

Testing sequences are generated by solving the constraints with the unification mechanism. At the beginning of generation, the testing criterion shall be given by the user. Our system counts the number of visiting pages to see if the testing criterion is satisfied.

The direct reachability between page1 and page2 is checked by solving the constraint of:
\[
\text{trans}(A, \text{page1}, \ldots, \text{page2}, \ldots)
\]
By chaining the trans constraints under the specified number of loop, the reachability is checked.

4.2.3 Example

Figure 7 is the specification of the web automaton of the library management presented in the example of section3. Figure 7 is almost directly derived from the definition shown in figure 4 following the rules in the previous part of this section. In the example, the web automaton is named as lib2.

Figure 8 is an example of generating the \(T^0\) test sequences. The predicate main is defined to generate testing sequences. Given a web automaton, it generates all the testing sequences in the form of abstract runs by interactively entering the criterion for the prompt. (In the figure, the criteria is given as 0.) The sequences are given in the form of Prolog lists. The anonymous variables (beginning with _) show the abstract runs. For this web automaton, the number of testing sequences for criteria \(T^0\), \(T^1\) and \(T^2\) are as follows.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>(T^0)</th>
<th>(T^1)</th>
<th>(T^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Test Sequences</td>
<td>9</td>
<td>132</td>
<td>1947</td>
</tr>
</tbody>
</table>

Figure 9 is a part of the output in checking the reachability. The predicate outputs the list of reachable state and the path. It succeeds if all the states are reachable in the specified web automa-
5 Application to Jakarta Struts

In this section, we show the correspondence to Jakarta Struts framework. Although we have already been referring to Jakarta Struts as one of the MVC model implementation, we see closer correspondence to our model and Struts.

5.1 Building A Web automaton from Struts

In Struts framework, JSPs are views and transitions between JSPs are defined by the configuration file `struts-config.xml`. We describe the correspondence informally as follows.

View: JSPs appearing in the application.

δ: Parameter Variables on links and properties of formbeans are the constraint variables.

ι: Analyzing the action-class servlet invoked by an action, constraints are constructed according to the strings the servlet forwards in `<action>` tags of the configuration.

iv: A starting JSP

FV: JSPs where an intended sequence of pages terminates.

Trans: We extract `<a>`, `<html:link>`, and `<form>` tags from JSPs. Looking up at `<action-mappings>` in `struts-config.xml`, `Trans` is constructed.

The most difficult construction is ι. In the examples we can deal with at the moment, the action-class servlets are rather simple like certifying users. In general, getting ι may not be easy. However, as the design principle, the functionality of each action-classed servlets should be fairly simple. We believe the analysis is possible in many cases.

5.2 Extension to Configuration Schema

We extend the configuration schema of Struts to describe the behavior directly. This extension mainly adds the combination information between JSPs and the action mapping. By this extension, we directly define a web automaton for a web application described in Struts. This enables to present the behavior model in an explicit manner.

We create a configuration file (`wsm-config.xml`) for View-typed components. This extension describes transition relation between JSPs.

The following xml elements are additionally de-
fined in our extended schema.

- **initial-state, final-state** elements
  An initial-state element expresses the page which a user requests first. A final-state element expresses a final state within a final-states element.

- **view** element
  A view element expresses the specification about the link of the page generated from JSP. In a view element, view-link element expresses the link to JSP and an action-link element expresses the link to action servlet. A param element expresses the parameters along with the links and the forms. If the values of the parameters can be determined, a value attribute is described.

- **condition** element
  A condition element expresses the condition that a forward included in the condition element can be occurred.

Figure 10 shows the extended configuration file (wsm-config.xml) for the library management system extending struts-config.xml. wsm-config.xml file describes a basic page transition system.

struts-config.xml file describes the actions when a page transition occurs and the basic data structure that affects the executions. It is straightforward to translate this extended configuration file into the web automaton specification by which the test sequences are automatically generated according the loop test criteria.

### 5.3 Action Generation

In the above, we intentionally included form-property elements in the form-bean element. This gives the typing information for the bean. By looking up the source code of the beans, this information can be derived, except a generic FormBean class, such as DynaActionForm. In our testing criteria, an action is a pair of a request and a response to and from the action servlet. The request
is externally observable, while the response is not observable. Therefore, in practice, we have to create external test values to generate every response. We need to analyze the action servlets to have the all response within the components. In general, it is impossible to give such input values to the form bean since the action servlet is a general Java program. But in many cases, it is relatively easy to cover all response values since the response is usually used as a flag for the values given by the form bean and all responses are given in the configuration.

6 Related Work

The automaton model for hyperdocument systems[12] is close to our approach. Their model only deals basically with the static contents. We extend their model to the data-independent system to deal with the dynamic contents.

The page transition model[9] proposes to build up path expression from their model from web applications based on the meta model to generate test cases with respect to the path coverage criteria. While their approach needs to generate the input data to have different page transitions, we focus on the result of control action objects and generate the test data in a reversed manner.

MAWL project proposes a concept of web applications which a page is embedded in a session[1][5]. A session is used in order to give states in a web application. Session-oriented web applications are not easy to apply existing web applications. The approach aims at more static analysis based on typing. The following <bigwig> project introduces the formal validation for dynamic page generation such as CGI[4].

As a more general approach as the software engineering, an extended UML syntax[6] is proposed in order to represent the characteristic architecture in web applications. Since this extension does not intend only to represent the elements of the MVC model, it is difficult to apply this extension for the web applications based on the MVC model. Behavior of web applications is modeled by state transitions where a page is a state defining a view and the control is modeled by page transition sequences. Since a web application with dynamic contents may change the behavior communicating with the environment, states of web automata have variables. By viewing our model as a data-independent system, we present our behavioral model as a direct extension of automaton model for static hyperdocument systems[12].

We have shown an application to Jakarta Struts framework where the MVC architecture is implemented using the JSP/servlet technology over the J2EE platform. Our behavior model provides the overall behavior of the web application intuitively. States are JSPs and state transitions are labeled by the requests and the results from the action servlet invoked at the transition.

Generally a data-independent system[15] is used to verify a reactive system. It is possible to apply our model to a verification technique of web application. In this paper, however, we focus on the development process of web applications. Web automata abstract the executions of web applications. For example, our model abstracts the real contents of views. Thus, in the real development, even if a web application is verified on our model, a developer still needs more refined tests or closer verification techniques in order for the application to work correctly.

We focus on the property of web applications that revisiting same pages often results in the similar transitions of pages. We have shown testing criteria based on the number of revisits of pages. The more a testing sequence revisits same pages, the more refined the test becomes. Although the criterion depends on the property of the web application under the test, in many cases we expect that the simplest criterion covers the interesting test sequences where the number of revisits is at most once.

By giving an experimental prototype for testing sequence generation, we counted the number of testing sequences according to our criterion. In the theoretical point of view, the number increases exponentially. This implies the general complete test gets unrealistic if the constraint is weak. However, for \( n \) such that the number of test in criterion \( T^n \) is small, the complete test is still possible.
Finally, we extended the configuration schema in Struts. The extension itself is redundant information since the information is described in JSPs and Action-class servlets. This extension enables to explicitly present the essential information about behavior in a uniform abstraction level. Although the extension is described by hand at the moment, in future some supporting tool should be developed.

Due to the gap between the static description and dynamic properties in the MVC model, it is quite useful to have an abstract model in the middle. We expect our behavior model can be used to check the consistency between the architecture and the implementation. Using our behavior model, it is possible to check the consistency in the component-wise way. We are also working to develop a dependency analysis technique for web applications based on our behavior model. The data dependency analysis is also important to decide to what extent the testing criteria should be applied.

Another future work is that the flow analysis based on our behavior model. As in the library management system, in a testing sequence, there is a part needed to be authenticated and not to be authenticated. The security aware testing is also an important property in our testing framework.

Acknowledgement

The authors would like to thank the members of Agusa laboratory for their valuable advice and Prof. Shin Nakajima for encouraging comments.

This research has been partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B)(2), 14380141, 2002-2004 and Scientific Research (C)(2), 13680408, 2001-2004 and Scientific Research (C)(2), 16500027, 2004. This research also has been partially supported by the E-Society Foundation Software program of the Ministry of Education, Science, Sport and Culture.

References