PAPER Special Section on Formal Approach

Modeling, Verification and Testing of Web Applications Using Model Checker

Kei HOMMA[†a)], ***Nonmember***, **Satoru IZUMI**[††], ***Student Member***, **Kaoru TAKAHASHI**[†††], ***and*** **Atsushi TOGASHI**[†], ***Members***

SUMMARY The number of Web applications handling online transaction is increasing, but verification of the correctness of Web application development has been done manually. This paper proposes a method for modeling, verifying and testing Web applications. In our method, a Web application is modeled using two finite-state automata, i.e., a page automaton which specifies Web page transitions, and an internal state automaton which specifies internal state transitions of the Web application. General properties for checking the Web application design are presented in LTL formulae and they are verified using the model checker Spin. Test cases examining the behavior of the Web application are also generated by utilizing the counterexamples obtained as the result of model checking. We applied our method to an example Web application to confirm its effectiveness.

key words: *Web application, modeling, testing, automata, model checking, Spin*

1. Introduction

Web applications are evolving rapidly and used in important transactions like online shopping and online banking, and the correctness of Web applications is a primary concern. Therefore, thorough analysis and verification of Web Applications is indispensable to assure the high quality of applications. On the other hand, Web applications need to be developed rapidly in order to offer the latest information to the customer.

In Web application development, a front-end, which is a Web page, is important because it is a contact point to users. But, the specification for a Web page usually takes considerable time to be decided, and it is not rare that the design of a Web page changes frequently. In order to reduce the influence of the changes made to a Web page, the Web page and the business logic should be developed separately, as one of the development styles, in the development of large Web applications.

One process taken for developing in the above mentioned way is to design the page transition first. Secondly, using the page transition, Web pages and business logic are designed. Finally, Web pages and business logic are implemented (e.g. JSPs for pages and Java Beans for business logic). Since the page transition and the business logic are designed in different phases, there is a need to check if they are designed correctly.

Model checking [1], [2], a method for formally checking state transition systems, has now become popular, because it allows the fully automatic analysis of designs of software systems as well as hardware systems. There is much work in which model checking is used on Web application modeling and verification (e.g. [7], [11], [12]), and on Web application testing (e.g. [13], [14]).

This paper proposes a method which uses the model checker Spin [3], [4] for modeling, expressing, verifying and testing Web applications. A Web application is modeled using the page transition and the internal state transition. They are described as finite-state automata, and the whole Web application is modeled using the product automaton of them. The model can be checked with respect to general properties such as deadlock-freeness using Spin, and its implementation can be tested using test cases which are generated from the counterexamples to show that certain properties are false for the model.

The characteristics of our method is summarized as follows, while the contrast with the related works is discussed later.

- The development process of a Web application is totally covered with our method from the modeling stage through testing one, i.e., design by automata; both verification and testing by model checking.
- The entire Web application can be modeled as the synchronized product between page transitions and internal state transitions, where each transition is specified separately with little concern of synchronized actions.
- We devise a specific way to express the model in Promela, the language used in Spin, and give a practical example. This way gives us a direct transformation method from automata in design stage into processes by Promela language in the verification and testing stage.
- General properties for verifying the Web applications are presented, and expressed with LTL (Linear Temporal Logic) formulae. These properties state general assertions to be satisfied by all Web applications independent of the intended functions.

Manuscript received July 22, 2010.
Manuscript revised November 14, 2010.

[†]The authors are with the Graduate School of Project Design, Miyagi University, Miyagi-ken, 981–3298 Japan.
[††]The author is with the Research Institute of Electrical Communication/Graduate School of Information Sciences, Tohoku University, Sendai-shi, 980–8577 Japan.
[†††]The author is with the Sendai National College of Technology, Sendai-shi, 989–3128 Japan.
a) E-mail: khomma@jp.ibm.com

DOI: 10.1587/transinf.E94.D.989

- A way to generate test cases examining state reachability and action feasibility of the Web application is given. Those test cases are generated by utilizing counterexamples that Spin produces for a false property represented in a certain LTL formula. So verification and testing are done in a uniform framework using model checking.
- We have illustrated modeling, verification, and testing using a practical application which represents an online store Web application. This illustrative example imitates a real application on the intended functions and state space without any specific exceptions. So, our formal method covers various Web applications on the view point of design, verification, and testing.
- Using our approach, it is feasible to design page transitions of each page and its internal state transitions separately at first. Then, we extend them to the whole application satisfying requirements for the Web application. This design process saves costs in software development.

To summarize the characteristic, this paper shows practical approaches on how to make use of formal methods for the design, verification and testing of Web systems based on formal techniques such as automata theory, concurrent calculus, and model checking. The methods are applied to a practical Web system with enough size of volume. The illustrated example is equivalent in essence to the real Web system in functions and size. The experimental result of this paper shows the usefulness of the proposed methods. This paper reduces the existing gap between formal theory and practice, and appeals for the importance of formal methods in software development.

The remainder of this paper is structured as follows: Section 2 describes some preliminaries. The proposed model of Web application is given in Sect. 3, followed by verification and test generation presented in Sect. 4; Section 5 shows the result of applying the proposed method to an example Web application (online store) and provide the case studies; In Sect. 6, we discuss related work, and Sect. 7 concludes the paper.

2. Preliminaries

2.1 Model Checking

Model checking [1], [2] is a technique for formally verifying hardware or software systems represented as finite state systems. Given a model of a system, a model checking tool automatically checks whether this model satisfies the specification of a given property such as reachability and deadlock-freeness. The specification is typically given by a formula in LTL or CTL (Computation Tree Logic). In particular, if the model does not satisfy the specification, the model checking tool will produce a counterexample that can be used to detect the source.

In this paper, we will use the model checker Spin [3], [4] as a model checking tool and LTL formulae to specify properties to be checked.

2.2 Linear Temporal Logic

The propositional linear temporal logic or simply LTL extends traditional propositional logic with temporal operators. Formally, an LTL formula φ has the following syntax:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi U \varphi \mid G\varphi \mid F\varphi \mid X\varphi$$

where p is an atomic proposition, U is the until operator, G (or $\Box$) is the always operator, F (or $\Diamond$) is the eventually operator, and X (or $\bigcirc$) is the next operator.

Given a set of atomic propositions AP, let $M = \langle S, T, S_0, L\rangle$ be a Kripke structure, where S is a set of states, $T \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, and L is a labeling function from S to the power set of AP. A state sequence $\pi = \langle s_0, s_1, \cdots\rangle$ is a path of M if $s_0 \in S_0$ and $(s_i, s_{i+1}) \in T$ for all i ($i \geq 0$). We denote $|\pi|$ as the length of a given state sequence π. If π is an infinite sequence of states, then $|\pi| = \infty$, assuming that ∞ is greater than any integer. An empty sequence of states is denoted as ε, and $|\varepsilon| = 0$. A path is called finite if it is a finite sequence of the form $\langle s_0, s_1, \ldots, s_k\rangle$ such that $(s_k, s) \notin T$ for all $s \in S$. $\pi^i = \langle s_i, s_{i+1}, \ldots\rangle$ denotes the suffix of a sequence $\pi = \langle s_0, s_1, \ldots\rangle$ starting at s_i. We assume that $\pi^i = \varepsilon$ for $|\pi| \leq i$. Also, note that $\pi^0 = \pi$.

Given a model M, the semantics of an LTL formula φ is given by defining a satisfaction relation which is a relation between M, a state sequence π of M and φ, and shows that φ is true in the situation M and π, written as $\pi \models \varphi$:

(1) $\pi \models p \Leftrightarrow |\pi| > 0$ and $p \in L(s_0)$
(2) $\pi \models \neg\varphi \Leftrightarrow \pi \not\models \varphi$
(3) $\pi \models \varphi \wedge \psi \Leftrightarrow \pi \models \varphi$ and $\pi \models \psi$
(4) $\pi \models \varphi\ U\ \psi \Leftrightarrow$ there exists i ($0 \leq i < |\pi|$) such that $\pi^i \models \psi$, and for all j ($0 \leq j < i$), $\pi^j \models \varphi$
(5) $\pi \models G\varphi \Leftrightarrow$ for all i ($0 \leq i < |\pi|$), $\pi^i \models \varphi$
(6) $\pi \models F\varphi \Leftrightarrow$ for some i ($0 \leq i < |\pi|$), $\pi^i \models \varphi$
(7) $\pi \models X\varphi \Leftrightarrow \pi^1 \models \varphi$

3. Modeling

3.1 Modeling of Web Applications

The general architecture of a Web application is a client-server system. Communication between the client and the server in the Web application typically revolves around the navigation of Web pages [5]. Therefore, the page transition is a significant ingredient in the Web application design. The Web pages in the page transition can be treated as states and the page transition as state transition. Thereby the page transition can be regarded as a finite-state automaton, and we call it a *page automaton*.

Definition 1: A *page automaton* is a quadruple:

$$M_G = \langle Q_G, \Sigma, \delta_G, q_{0G}\rangle$$

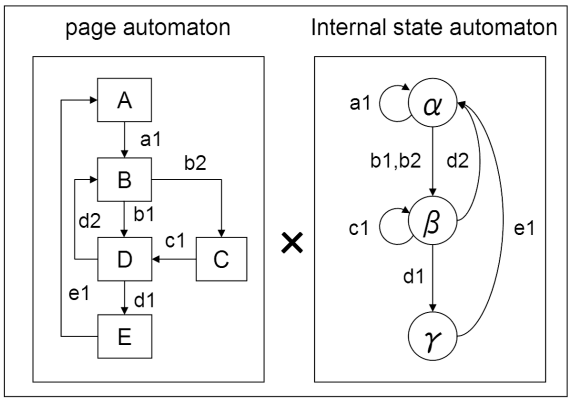

Fig. 1 Model of Web application.

where

(1) Q_G is a finite set of states (*pages*);
(2) Σ is a finite set of input symbols (*actions to Web application*);
(3) δ_G ($\subseteq Q_G \times \Sigma \times Q_G$) is a transition relation (*page transition*);
(4) q_{0G} ($\in Q_G$) is the initial state (*top page*).

An example of a page automaton is shown in Fig. 1.

A Web application executes a business logic and so the most important model of the system focuses on the business logic and business state [5], and we call this an internal state. The internal states are determined by the set of variables and values from input. The internal state transition occurs synchronously with page transition triggered by actions. We call this internal state transition an *internal state automaton*.

Definition 2: An *internal state automaton* is a quadruple:

$$M_e = \langle Q_e, \Sigma, \delta_e, q_{0e} \rangle$$

where

(1) Q_e is a finite set of states (*internal states of Web application*);
(2) Σ is a finite set of input symbols (*actions to Web application*);
(3) δ_e ($\subseteq Q_e \times \Sigma \times Q_e$) is a transition relation (*internal state transition*);
(4) q_{0e} ($\in Q_e$) is the initial state.

An example of an internal state automaton is shown in Fig. 1.

Given a page automaton M_G and an internal state automaton M_e, we define the whole Web application as their product $M_G \times M_e$, and call it a *product automaton* (see Fig. 1).

Definition 3: The *product automaton* of a page automaton $M_G = \langle Q_G, \Sigma, \delta_G, q_{0G} \rangle$ and an internal state automaton $M_e = \langle Q_e, \Sigma, \delta_e, q_{0e} \rangle$ is defined as:

$$M = M_G \times M_e = \langle Q, \Sigma, \delta, q_0 \rangle$$

where

(1) $Q = Q_G \times Q_e$;
(2) $((q_G, q_e), a, (q'_G, q'_e)) \in \delta \Leftrightarrow (q_G, a, q'_G) \in \delta_G$ and $(q_e, a, q'_e) \in \delta_e$;
(3) $q_0 = (q_{0G}, q_{0e})$

with $q_G, q'_G \in Q_G$, $q_e, q'_e \in Q_e$ and $a \in \Sigma$.

For readability purpose, we denote $(q, a, q') \in \delta$ as $q \xrightarrow{a} q'$. For $w \in \Sigma^*$, $q \xrightarrow{w} q'$ is inductively defined as follows:

$q \xrightarrow{\epsilon} q$ where ϵ is an empty sequence;
$q \xrightarrow{aw'} q' \Leftrightarrow$ for some q'', $q \xrightarrow{a} q''$ and $q'' \xrightarrow{w'} q'$

where $a \in \Sigma$ and $w' \in \Sigma^*$. The symbol $\xrightarrow{q}\!\!\!\!\!\!/$ is the negation of the relation $\xrightarrow{a}$, and $\xrightarrow{*}$ means $\xrightarrow{w}$ for some $w \in \Sigma^*$.

3.2 Modeling with Variables

We extend a page automaton and an internal state automaton by introducing variables. An automaton can be defined easily by extending the state transition to depend on the values of the variables in the definition of its transition relation.

Let $X = \{x_1, \cdots, x_n\}$ be a finite set of variables which store values derived from outside (e.g. input from the user through Web pages), and store values used inside the system (e.g. data in a database). With the addition of enabling predicate depending on X and value assignment to X, a page automaton and an internal state automaton with variables are defined as follows.

Definition 4: A *page automaton with variables* is a 6-tuple:

$$M_G = \langle Q_G, \Sigma, \delta_G, q_{0G}, \{R_i\}_{i \in \{1,\cdots,n\}}, \{v_{0i}\}_{i \in \{1,\cdots,n\}} \rangle$$

where

(1) Q_G is a finite set of states (*pages*);
(2) Σ is a finite set of input symbols (*actions to Web application*);
(3) δ_G ($\subseteq Q_G \times (\Pi_{i \in \{1,\cdots,n\}} R_i \rightarrow \{\textbf{true}, \textbf{false}\}) \times \Sigma \times Q_G$) is a transition relation (*page transition*);
(4) q_{0G} ($\in Q_G$) is the initial state (*top page*);
(5) R_i is the finite domain of variable x_i;
(6) v_{0i} ($\in R_i$) is the initial value of variable x_i.

In this definition, the set $\Pi_{i \in \{1,\cdots,n\}} R_i$ represents the direct product $R_1 \times \cdots \times R_n$. The second element $\Pi_{i \in \{1,\cdots,n\}} R_i \rightarrow \{\textbf{true}, \textbf{false}\}$ (equivalently $2^{\Pi_{i \in \{1,\cdots,n\}} R_i}$) of the transition relation δ_G represents the enabling predicate depending on the values of the variables. There is no value assignment to the variables.

Definition 5: An *internal state automaton with variables* is a 6-tuple:

$$M_e = \langle Q_e, \Sigma, \delta_e, q_{0e}, \{R_i\}_{i \in \{1,\cdots,n\}}, \{v_{0i}\}_{i \in \{1,\cdots,n\}} \rangle$$

where

(1) Q_e is a finite set of states (*internal states of Web application*);
(2) Σ is a finite set of input symbols (*actions to Web application*);
(3) δ_e ($\subseteq Q_e \times (\Pi_{i\in\{1,\cdots,n\}} R_i \rightarrow \{\textbf{true}, \textbf{false}\}) \times \Sigma \times \Pi_{i\in\{1,\cdots,n\}}(\Pi_{j\in\{1,\cdots,n\}} R_j \rightarrow R_i) \times Q_e$) is a transition relation (*internal state transition*);
(4) q_{0e} ($\in Q_e$) is the initial state;
(5) R_i is the finite domain of variable x_i;
(6) v_{0i} ($\in R_i$) is the initial value of variable x_i.

In this definition, the fourth element $\Pi_{i\in\{1,\cdots,n\}}(\Pi_{j\in\{1,\cdots,n\}} R_j \rightarrow R_i)$ of the transition relation δ_e represents the assignment of the values to the variables.

3.3 Representation of the Model

Spin which we use as the model checking tool has its own specification language called Promela. Here we give a way to express the page automaton and the internal state automaton using Promela.

Each automaton is expressed as a Promela *process* denoted by using the keyword `proctype`, like "`proctype PageAutomaton() {..}`." To represent each state of an automaton, we use *label*. For example, "`A:`" represents a state named "A." The action accompanied with a state transition is written in the `do..od` statement. A conditional action is expressed using the `if` statement.

The product of the automata is implemented as a means of the synchronous message passing mechanism of Promela which uses a rendezvous port, denoted like "`chan port = [0] of {..}`" for example. We will explain how the page automaton and the internal state automaton synchronize. Let "`a`" be an executable action of the page automaton. Then, it is sent from the page automaton using the rendezvous port, denoted like "`port!a`." It is received by the internal state automaton using the same port, denoted like "`port?a`," and the response "`res`" is sent back, denoted like "`port!res`" which is received by the page automaton, denoted like "`port?res`." Then, both automata move to the next states synchronously.

4. Verification and Testing

4.1 Verification

We will present general properties for Web application design, and a way to express and verify them in Spin where $M = M_G \times M_e = \langle Q, \Sigma, \delta, q_0, \{R_i\}_{i\in\{1,\cdots,n\}}, \{v_{0i}\}_{i\in\{1,\cdots,n\}}\rangle$.

(1) The page reachable from the top page always has a next page in the transition.

> A Web application is defined to be *deadlock-free* if every page has a next page during its execution. In this sense, the property implies the Web application is deadlock-free:
>
> For all $q \in Q$ such that $q_0 \xrightarrow{*} q$,
> there exist $a \in \Sigma$ and
> $q' \in Q$ such that $q \xrightarrow{a} q'$.
>
> To verify this property, we observe the execution of Promela statements. In the Web page more than one action is available, so which statement is executed is decided at random in Promela. It gives rise to non-deterministic choice. If all statements within the page are not executable, then the system is deadlocked. In other words, deadlock does not occur when at least one action is sent and received synchronously.

(2) Every page is reachable from the initial state.

> All pages in the Web application can be reached from the initial state via other pages:
>
> For all $q_e \in Q_e$ and $q_G \in Q_G$,
> there exist $w \in \Sigma^*$ and $q'_e \in Q_e$
> such that $(q_{0G}, q_{0e}) \xrightarrow{w} (q_G, q'_e)$.
>
> To verify this property, we use LTL formulae. It is unable to express "every page" using the LTL formula. Therefore, verification is done by specifying each concrete page. The following formula states that from the top page, there exists a path to a given page:
>
> $$\Diamond (p \wedge \Diamond q)$$
>
> where p is the top page and q is a given page.
>
> Different from CTL, LTL cannot be applied to verify "at least one path" when there are many possible paths. Therefore, using the negation of the LTL formula, Spin will find a counterexample showing the execution path from the top page to a given page:
>
> $$\neg \Diamond (p \wedge \Diamond q)$$
>
> where p is the top page and q is a given page.
>
> By switching the target page, all pages can be checked whether they are reachable from the top page. Therefore, comprehensive verification may take a certain amount of time. But it can be done in a repetitive manner without much effort.

(3) The top page is reachable from all pages.

> There is no end page in the Web application, so that we have to be able to return back to the top page from any Web page in order to deal with the process repeatedly:
>
> For all $q_G \in Q_G$ and $q_e \in Q_e$
> such that $(q_G, q_e) \in Q$,

there exist $w \in \Sigma^*$ and $q'_e \in Q_e$
such that $(q_G, q_e) \xrightarrow{w} (q_{0G}, q'_e)$.

This property can be verified using the same previous method. The LTL formula can be shown as follows:

$$\neg \Diamond (p \wedge \Diamond q)$$

where p is a given page and q is the top page.

(4) Every variable value is under the designated domain.

No variable takes the values beyond the domain during the execution:

Let $dom(x_i)$ be the set of values of the variable x_i while the application is running. Then:
$dom(x_i) \subseteq R_i$ for all i $(1 \leq i \leq n)$.

In the similar way as mentioned above, the LTL formula is used for verifying this property. A value of variable should be between minimum (x_{min}) and maximum (x_{max}) of the designated domain:

$$\Box (p \wedge q)$$

where p is $x >= x_{min}$ and q is $x <= x_{max}$.

4.2 Test Generation

We will show a way to generate test cases, using the Web application model we previously defined. The Web application model we defined is intended for use in the design phase, so the test cases correspond to the application design, which is done in the "system test" phase. Because of system test, the generated test cases are applied in a black box testing for testing the functional structure.

Combined with the Web application model, each state and/or each action can be treated as test target. Therefore, the test cases are a combination of the following patterns:

(1) covering states (pages, internal states, and a combination of pages and internal states)
(2) covering the occurrence of actions

Test case generation can be done by utilizing counterexamples produced by Spin, using the following form of the LTL formula:

$$\Box (\neg p)$$

where a state (page or internal state) or action is set to p as a test object. If the test object is reachable in the Web application model, then an "error" is output as a result of verification in Spin creating the "trail file." Using the trail file, the way to transit from the initial state to the target state is guided, which can be used as a test case.

To be more concrete, let us assume a page Z of a page automaton A as a test state. Then, we execute model checking by Spin with the Promela description of the model and the formula setting p being Z of A. If $\Box(\neg p)$ is not satisfied, i.e. the page Z is reachable, the path leading to Z is obtained from the counterexample. The action sequence included in the path can be used as a test case for state reachability of Z.

As mentioned in Sect. 3.3, each state of an automaton is represented by *label* in Promela. The *label* can be specified in the LTL formula. However, action only appears as a message of rendezvous port. Therefore, we use a temporary variable, which will store action if the action is executed normally. We specify this temporary variable in the LTL formula to create test case.

There are some problems in this test case generation. The first problem is that if there are a large amount of test cases, each state has to be verified and each counterexample is output separately. The second problem is that test cases may become redundant in this way of generation.

These problems can be solved using the following method:

$$\Box(\neg p) \vee \Box(\neg q) \vee \Box(\neg r) \ldots$$

A counterexample for transition to more than one state can be output by verifying multiple states at the same time.

5. Illustrative Examples

5.1 Modeling

As an example of modeling, we use an online store Web application (e.g. amazon.com [6]). This application has eleven Web pages and six internal states. Its page transition and internal state transition are outlined in Figs. 2 and 4, respectively. The initial Web page is "A. Top Page" and the initial internal state is "α. no item selected." A product is selected and the amount of the product is entered in the page "B. Product List." The order is placed in the page "J. Place Order." Finally, the order completes in page "K. Completion." The user can increase and decrease the amount of order, but five orders are the maximum in this example Web application.

The example application can be modeled by two automata. The page automaton M_G (Fig. 3) is defined as follows:

$$M_G = \langle Q_G, \Sigma, \delta_G, q_{0G}, \{R_i\}_{i\in\{1,\cdots,3\}}, \{v_{0i}\}_{i\in\{1,\cdots,3\}} \rangle$$

where

$Q_G = \{A, B, C, D, E, F, G, H, I, J, K\}$
$\Sigma = \{a1, a2, b1, b2, c1, c2, c3, d1, d2, d3,$
$\quad e1, e2, e3, e4, f1, f2, f3, g1, h1, h2, h3,$
$\quad i1, i2, i3, i4, i5, j1, j2, j3, j4, j5, k1\}$
δ_G = refer to Fig. 3
$q_{0G} = A$
$R_1 = R_2 = R_3 = \{1, 2, 3, 4, 5\}$
$v_{01} = v_{02} = v_{03} = 1.$

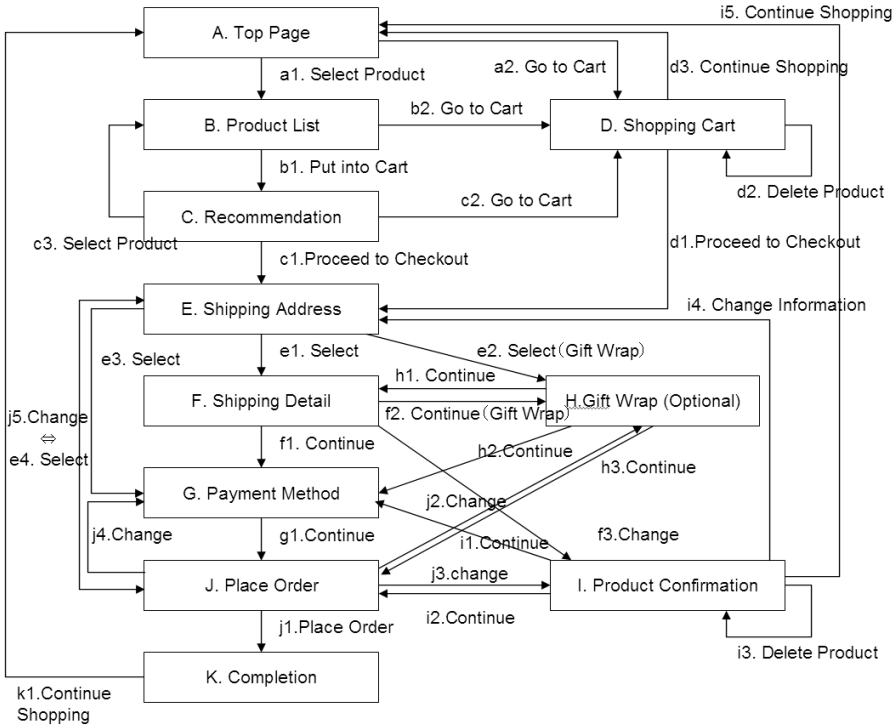

Fig. 2 Page transition.

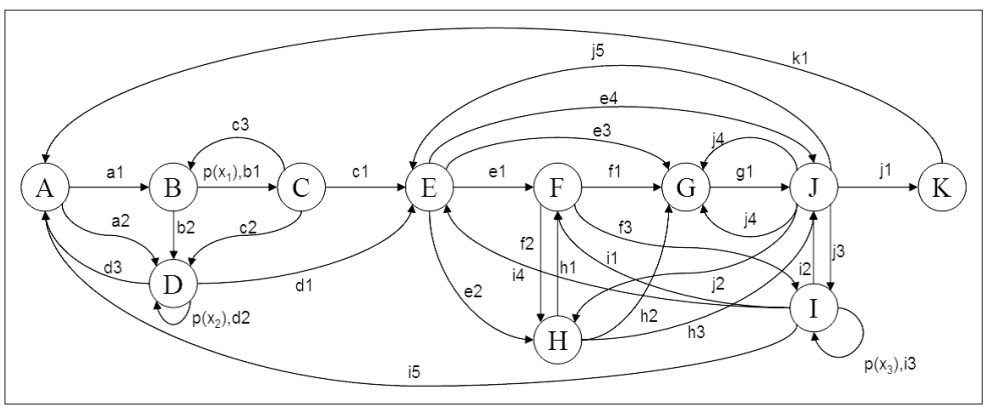

Fig. 3 Page automaton.

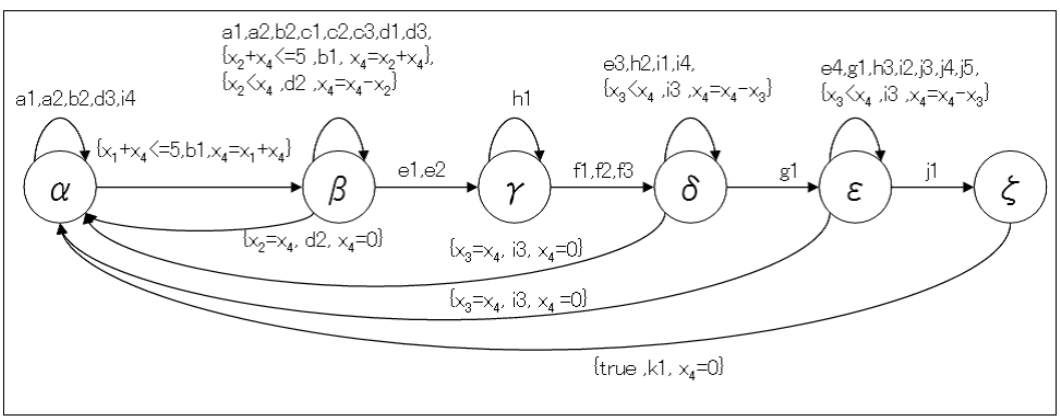

Fig. 4 Internal state automaton.

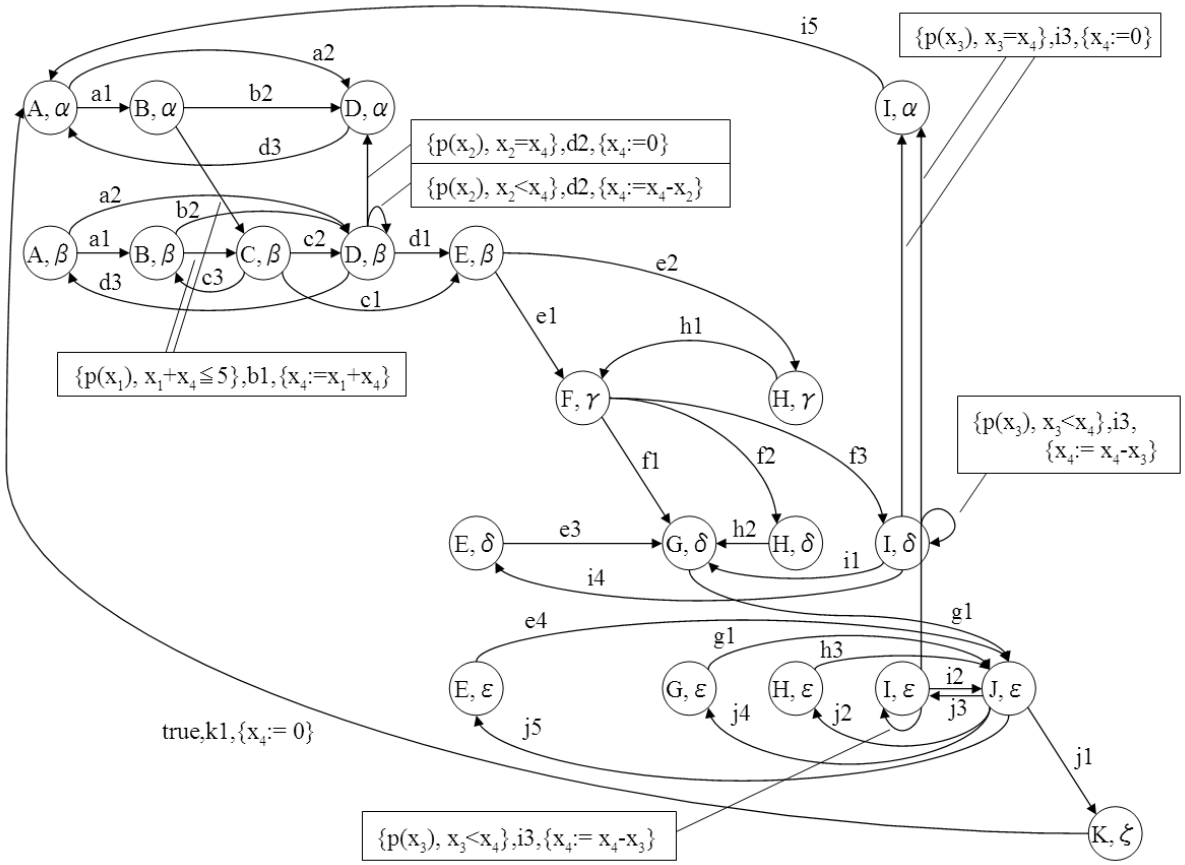

Fig. 5 Product automaton.

The variables x_1, x_2 and x_3 in M_G represent input from "B. Product List," "D. Shopping Cart" and "I. Product Confirmation," respectively. The variable x_1 describes the amount of the product for purchase, and the variables x_2 and x_3 describe the amount of the products for cancellation. p is an enabling predicate which returns **true** if the value of the argument is in its domain, and **false** otherwise. It represents input validation in the application.

The internal state automaton M_e (Fig. 4) is defined as follows:

$$M_e = \langle Q_e, \Sigma, \delta_e, q_{0e}, \{R_i\}_{i\in\{1,\cdots,4\}}, \{v_{0i}\}_{i\in\{1,\cdots,4\}}\rangle$$

where

$$\begin{aligned}
Q_e &= \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta\}\\
\Sigma &= \{a1, a2, b1, b2, c1, c2, c3, d1, d2, d3,\\
&\quad e1, e2, e3, e4, f1, f2, f3, g1, h1, h2, h3,\\
&\quad i1, i2, i3, i4, i5, j1, j2, j3, j4, j5, k1\}\\
\delta_e &= \text{refer to Fig. 4}\\
q_{0e} &= \alpha\\
R_1 &= R_2 = R_3 = \{1, 2, 3, 4, 5\}\\
R_4 &= \{0, 1, 2, 3, 4, 5\}\\
v_{01} &= v_{02} = v_{03} = 1\\
v_{04} &= 0.
\end{aligned}$$

The variables x_1, x_2 and x_3 in M_e represent the same ones as in M_G. The variable x_4 describes the total amount of product for purchase. It stores values input at "B. Product List" page and values are changed at "D. Shopping Cart" and "I. Product Confirmation."

The product automaton M of the page automaton M_G and the internal state automaton M_e is defined as follows:

$$M = M_G \times M_e = \langle Q, \Sigma, \delta, q_0, \{R_i\}_{i\in\{1,\cdots,4\}}, \{v_{0i}\}_{i\in\{1,\cdots,4\}}\rangle$$

where

$$\begin{aligned}
Q &= \{(A,\alpha), (A,\beta), (B,\alpha), (B,\beta), (C,\beta),\\
&\quad (D,\alpha), (D,\beta), (E,\beta), (E,\delta), (E,\epsilon),\\
&\quad (F,\gamma), (G,\delta), (G,\epsilon), (H,\gamma), (H,\delta), (H,\epsilon),\\
&\quad (I,\alpha), (I,\delta), (I,\epsilon), (J,\epsilon), (K,\zeta)\}\\
\Sigma &= \{a1, a2, b1, b2, c1, c2, c3, d1, d2, d3,\\
&\quad ee1, e2, e3, e4, f1, f2, f3, g1, h1, h2, h3,\\
&\quad ei1, i2, i3, i4, i5, j1, j2, j3, j4, j5, k1\}\\
\delta &= \text{refer to Fig. 5}\\
q_0 &= (A,\alpha)\\
R_1 &= R_2 = R_3 = \{1, 2, 3, 4, 5\}\\
R_4 &= \{0, 1, 2, 3, 4, 5\}\\
v_{01} &= v_{02} = v_{03} = 1\\
v_{04} &= 0.
\end{aligned}$$

5.2 Representation of the Model

We have presented the way to express the page automaton and the internal state automaton in Promela in Sect. 3.3. In this section, we will describe the specific way to express two automata by using the automaton model of the example Web application.

Figure 6 shows a part of Promela source code for the example Web application. In lines 01 and 19, each automaton is expressed as a process using the built-in word `proctype`, and in lines 02, 08, and 20 the alphabets such as "A:", "B:" and "a:" show states of the automaton, respectively. The product of the automata is expressed as synchronous communication using `channels` which is "param" in line 04 for example. "`param_var`" is also a synchronous communication channel for sending input values from the Web page. "`tmp_rec`" in InternalStateAutomaton is a temporary variable for storing the action value.

"selectAmount();" in line 10 is an inline function which sets the value between one and five to the variable x1. This inline function represents the values derived from outside (i.e. input from the user), and covers the range of values which meet the requirement, so the exhaustive verification can be done by Promela.

```
01  proctype PageAutomaton(int n)  {
02  A:                                      /*** Top Page ***/
03   do
04    ::param ! a1; param ? res -> goto B
05    ::param ! a2; param ? res -> goto D
06   od;
07
08  B:                                      /*** Product List ***/
09   do
10    ::param ! b1; selectAmount(); param_var ! x1;
11      if
12       :: param ? res -> goto C
13       :: param ? res_ng -> goto B
14      fi
15    :: param! b2; param ? res -> goto D
16   od;
17  ...
18  }
19  proctype InternalStateAutomaton(int n) {
20  a:                              /*** no item selected ***/
21   do
22    ::param ? a1 -> tmp_rec = a1; atomic{ param ! res; skip  }
23    ::param ? a2 -> tmp_rec = a2; atomic{ param ! res; skip  }
24    ::param ? b2 -> tmp_rec = b2; atomic{ param ! res; skip  }
25    ::param ? d3 -> tmp_rec = d3; atomic{ param ! res; skip  }
26    ::param ? i5 -> tmp_rec = i5; atomic{ param ! res; skip  }
27    ::param ? b1 -> param_var ? tmp_x1;
28      if
29       ::(tmp_x1 + x4 <= 5) -> tmp_rec = b1;
30            atomic{ x4 = tmp_x1 + x4; param ! res; goto v }
31       ::(tmp_x1 + x4 > 5) -> atomic{ param ! res_ng; skip}
32      fi
33   od;
34  ...
35  }
```

Fig. 6 Promela source code (part).

We will describe how the page automaton and the internal state automaton communicate by using the action "a1" in Fig. 6. Lines 03 to 06 show the page A and its actions. There are two actions "a1" and "a2" in the page A. The action "a1" is sent (denoted by `''!''`) to the internal state automaton using the channel "param," and it is received (denoted by `''?''`) by the state α (a:) of the internal state automaton in line 22, and the response ("res") is sent to the page automaton using the same channel. The "`skip`" in line 22 describes that no transition occurs by this action. The "res" is received by the page automaton in line 04, and the page moves to "B. Product List" by the method "goto B."

By the way, the variables used in the Promela code, such as x1, x2 and x3, are defined as global variables. So the variables can be used in the page automaton and the internal state automaton without using the channel. However we used the channel for two reasons. One reason is to resemble the implementation of the Web Application. The channel represents the data passing between the browser and the server. The other reason is to get values sent by the channel from the trail file, which is created when the Spin verification is not satisfied. This value can be used for input for testing.

5.3 Verifying the Web Application Model

Using the automaton model of the example Web application, we now verify it with the properties listed in Sect. 4.1.

(1) Deadlock-Freeness in Page Automaton

We verified the property (1) in the example Web application using Spin. The verifier is generated from the Promela source code, and the verifier is compiled to create an executable verifier. We executed it and no error was detected. Therefore, we confirmed there was no deadlock in the example Web application.

(2) Page Reachability from the Top Page

The property (2) in the example Web application was verified using Spin. Taking the following LTL formula, we checked that the page , "I.Product Confirmation," is reachable from the top page "A.Top Page." We created and executed the executable verifier, and an error was detected. The error-trail file is dumped showing a counterexample which is a route from "A.Top Page" to "I.Product Confirmation."

$$\neg \Diamond (p \wedge \Diamond q)$$

```
#define p (PageAutomaton[1]@A)
#define q (PageAutomaton[1]@I)
```

Here, `#define` is a reserved word in Promela which defines the proposition for the LTL formula, e.g. p is true if the current process page is `PageAutomaton[1]@A`. `PageAutomaton[1]` represents the process PageAutomaton with its ID "1." `@A` and `@I` are labels in PageAutomaton which describe the Web page.

The same LTL formula is used for this verification. Therefore, by changing labels in PageAutomaton, all pages were verified in the same way.

(3) Top Page Reachability from every Page

The property (3) is the opposite of the property (2). Using the following LTL formula, we checked the top page "A.Top Page" is reachable from the page "I.Product Confirmation." The executable verifier searched state space and detected error. The error-trail file is dumped showing the counterexample, same way as previous, which is a route from "I.Product Confirmation" to "A.Top Page."

$\neg \Diamond (p \wedge \Diamond q)$

```
#define p (PageAutomaton[1]@I)
#define q (PageAutomaton[1]@A)
```

All pages were verified using the same way.

(4) Taking Values under the Designated Domain

To verify the property (4), we verified that the value of the variable x_4, the total amount of products for purchase in Internal State Automaton, is in its domain. The value of the variable x_4 is not input from the Web page, therefore it depends on inputs from the Web page and how the enabling predicate is designed. Using the following LTL formula, we checked that no error was detected showing the variable x_4 under its designated domain.

$\Box (p \wedge q)$

```
#define p (x4 <= 5)
#define q (x4 >= 0)
```

All variables were verified using the same method.

5.4 Generating Test Cases from the Model

In this section, we will explain how to generate test cases for covering the states and the occurrence of actions using the example application.

(1) Covering States

We will use two pages "H.Gift Wrap" and "I. Product Confirmation" as states. Using the LTL formula listed below, the trail file (Fig. 7) is created to show a counterexample.

$\Box(\neg p) \vee \Box(\neg q)$

```
#define p (PageAutomaton[1]@H)
#define q (PageAutomaton[1]@I)
```

By using the trail file (Fig. 7), we can find a path to reach two pages. This can be done by searching which events are processed before the trail file ends. An easy way of tracing events is to select the events following `param!` in the trail file. In the example application, the trail file shows six events are required for this test:

"a1. Select Product" → "b1. Put into Cart" →
"c1. Proceed to Checkout" → "e2. Select" →
"h1. Continue" → "f3. Change"

```
proc 0 = :init:
proc 1 = PageAutomaton
proc 2 = InternalStateAutomaton
q¥p   0   1   2
  1   .   param!a1
  1   .   .   param?a1
  1   .   .   param!res
  1   .   param?res
  1   .   param!b1
  1   .   .   param?b1
  2   .   param_var!1
  2   .   .   param_var?1
  1   .   .   param!res
  1   .   param?res
  1   .   param!c1
  1   .   .   param?c1
  1   .   .   param!res
  1   .   param?res
  1   .   param!e2
  1   .   .   param?e2
  1   .   .   param!res
  1   .   param?res
  1   .   param!h1
  1   .   .   param?h1
  1   .   .   param!res
  1   .   param?res
  1   .   param!f3
  1   .   .   param?f3
  1   .   .   param!res
  1   .   param?res
spin: trail ends after 47 steps
```

Fig. 7 Trail file (part).

Also, the trail file shows the value set in PageAutomaton. The value following `param_var!` in the trail file shows the value sent from PageAutomaton to InternalStateAutomaton. Therefore, input values for pages are generated specifically for use in the test case.

Test case generation for InternalStateAutomaton is also feasible using the same method.

(2) Covering Actions

We generate a test for covering three events, "d2. Delete Product", "e1. Select", and "i3. Delete Product." In the same way as (1), an LTL formula is used and the trail file is created to show a counterexample.

$\Box(\neg p) \vee \Box(\neg q) \vee \Box(\neg r)$

```
#define p (tmp_var == d2)
#define q (tmp_var == e1)
#define r (tmp_var == i3)
```

`tmp_var` is a temporary variable which holds actions, and `define p (temp_var == d2)` means p is true if the action "d2" occurs.

	memory(MB)	time(sec)
R_4 = {0,1,2,3,4,5}		
(1)	17.3	1.57
(2)	2.50	0.05
(3)	2.50	0.06
(4)	17.6	1.48
R_4 = {0,1,2,3,4,5,6,7,8,9,10}		
(1)	33.4	3.36
(2)	2.50	0.05
(3)	2.50	0.06
(4)	31.8	3.1

Fig. 8 Cost of verification.

5.5 Verification Cost

The verification in Sect. 5.3 was done using a PC with Windows XP for OS, Intel Core2 Duo 2 GHz for CPU, and 2 GB Memory. The cost of verification, the used memory and the elapsed time, for each verification case is shown in Fig. 8. The top four rows show the cost of verification in Sect. 5.3, where the range of x_4 (the total amount of product for purchase) is $R_4 = \{0, 1, 2, 3, 4, 5\}$. The bottom four rows show the cost of verification which the range of x_4 is changed to $R_4 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. The range for the other variables remain the same, i.e. $R_1 = R_2 = R_3 = \{1, 2, 3, 4, 5\}$.

The costs for cases (1) and (4) are different from cases (2) and (3). The verification for (1) and (4) covers all states and transitions. However, the verification for (2) and (3) comes to a stop when a counterexample is found. The result also shows that used memory and the elapsed time increase when the range of variables is expanded in verification (1) and (4). Therefore, the modeling of the application needs to be done with consideration of size and number of variables.

6. Related Work

In this section, we briefly discuss some of the researches related to modeling, verification and testing of Web applications using model checking.

There is much research in which an Web application is modeled as a directed graph. In [7], [8], components of window (e.g., a page, a frame, and a link) are modeled as states. The reachability between states is defined as requirements of a Web application, and they are verified using model checking. In the research, pages, frames, and links are modeled by a single state transition system (a single automaton). On the other hand, our model has two automata, so, not only page transition but also the internal state is considered for the verification of reachability.

In [9]–[11], they took an approach to model Web applications using parallel composition of UML diagrams. The papers [9], [10] propose a model of Web application using the UML class diagram. The model is used for reachability checking and semi-automatic test case generation. Our model differs from their models in the level of abstraction, as our model is more abstract than their models in Web application design. The paper [11] presents an approach that uses statecharts to model adaptive navigation of Web application formally and checks for the unreachable Web page. This model focuses only on users mode (e.g., whether they are logged in or not) and page history (e.g., what pages the user has visited previously). The internal state automaton in our model can deal with the adaptive navigation as it is.

In [12], the authors present a formal approach for modeling Web applications using communicating automata. They observe the external behavior of an explored part of Web application using a monitoring tool. The observed behavior is then converted into communicating automata representing all windows, frames, and framesets of the application under test by intercepting HTTP requests and responses using a proxy server. Their model is different from ours as they focus on external behavior.

There is some research on test case generation using model checker [13], [14]. In [13], the authors propose an automatic test case generation method using the NuSMV model checker. Their method can extract test cases that cover all states or all transitions in the Statechart from the counterexamples produced by the model checker. Our test case generation method is based on their idea. We extend their method according to our model for testing Web application.

The paper [14] presents an approach that uses model checkers to generate test sets without redundant test sets. In this approach, a system is modeled by one Stateflow and test sets are generated based on the Stateflow. Our method is based on two automata. Therefore, our method generates test cases in various perspectives such as only Web page transition or Web page transition with internal business logic.

7. Conclusion

In this paper, we have described a method for modeling, verifying and testing Web applications. We have shown that the entire Web application can be expressed using the product of a page automaton and an internal state automaton. We have devised a way to express the model in Promela and we have given a means of verifying the Web application and generating test cases using Spin and LTL formulae. The proposed method has been validated through an example Web application. The example application and validation showed that our modeling, verification and testing are well applied to it.

If the page transition and internal state transition are specified and integrated in the design phase of Web application development, it is possible to effectively test the implementation using our method as well as verification of its design.

Our future work includes an application of our method for larger Web applications and a solution for the state space explosion problem.

References

[1] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen, Systems and Software Verification —Model-Checking Techniques and Tools, Springer, 2001.
[2] E.M. Clarke, O. Grumberg, and D.E. Long, "Model checking and abstraction," ACM OPLAS, vol.16, no.5, pp.1512–1542, 1994.
[3] G.J. Holzmann, "The model checker SPIN," IEEE Trans. Softw. Eng., vol.23, no.5, pp.279–295, 1997.
[4] G. Holzmann, The Spin Model Checker, Primer and Reference Manual, Addison-Wesley, 2003.
[5] J. Conallen, "Modelling Web applications architectures with UML," Commun. ACM, vol.42, no.10, pp.63–70, 1999.
[6] amazon.com, amazon.com (online): available from http://www.amazon.com/ (accessed 2009-12-20).
[7] E.D. Sciascio, M.F. Donini, M. Mongiello, and G. Piscitelli, "Web applications design and maintenance using symbolic model checking," Proc. Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), pp.63–72, 2003.
[8] E.D. Sciascio, M.F. Donini, M. Mongiello, and G. Piscitelli, "AnWeb: A system for automatic support to Web application verification," Proc. International Conference on Software Engineering and Knowledge Engineering (SEKE '02), pp.609–616, 2002.
[9] F. Ricca and P. Tonella, "Analysis and testing of Web applications," Proc. International Conference on Software Engineering (ICSE2001), pp.25–34, 2001.
[10] F. Ricca and P. Tonella, "Web site analysis: Structure and evolution," Proc. International Conference on Software Maintenance (ICSM2000), pp.76–86, 2000.
[11] M. Han and C. Hofmeister, "Modeling and verification of adaptive navigation in Web applications," Proc. International Conference on Web Engineering (ICWE'06), pp.329–336, 2006.
[12] M. Haydar, A. Petrenko, and H. Sahraoui, "Formal verification of Web applications modeled by communicating automata," Proc. International Conference on Formal Techniques for Networked and Distributed Systems (FORTE2004), LNCS 3235, pp.115–132, 2004.
[13] M. Kadono, T. Tsuchiya, and T. Kikuno, "Using the NuSMV model checker for test generation from statecharts," Proc. International Symposium on Dependable Computing, pp.37–42, 2009.
[14] G. Hamon, L. Moura, and J. Rushby, "Generating efficient test sets with a model checker," Proc. International Conference on Software Engineering and Formal Methods (SEFM'04), pp.261–270, 2004.

Kei Homma received the B. E. degree and M. E. degree from Waseda University in 2001 and 2003, respectively. Currently, he is pursuing his doctoral degree in the Graduate School of Project Design, Miyagi University, and he is also working at IBM Japan. His research interests include Web application, formal approach, and model checking. He is a student member of IPSJ.

Satoru Izumi graduated from the Advanced Engineering Course, Sendai National College of Technology in 2007. He received the M. S. degree in 2009 from Tohoku University, Japan. Currently, he is pursuing his doctoral degree in the Graduate School of Information Sciences (GSIS), Tohoku University. His research interests include specification and verification of system, ontology engineering and its application. He is a student member of IPSJ.

Kaoru Takahashi received the Ph.D. degree from Tohoku University in 1992. From 1993 to 1995, he was a senior visiting researcher in the Advanced Intelligent Communication Systems Laboratories. He is now a professor in the Department of Information Systems, Sendai National College of Technology. Currently, he is doing research on software engineering, Semantic Web, sensor applications and so on. He is a member of IPSJ.

Atsushi Togashi received the B.E. degree from the Yamagata University in 1979, the M.E. and the Dr. of Eng. degrees from Tohoku University in 1981 and 1984, respectively. He worked at Tohoku University (1984–96), University of Sussex as a guest researcher (1995–96), Shizuoka University (1996–1997), and also Kyushu University, Sapporo University, The Open University of Japan, University of Tsukuba as an adjunct professor. Currently, he is a Professor at Miyagi University. His research interests include process calculi, model checking, ICT applications to real fields. He is a member of the IPSJ, JSSS, AI in Japan, ACM, and IEEE.