# Model-Based Testing of Thin-Client Web Applications and Navigation Input

Pieter Koopman, Peter Achten, and Rinus Plasmeijer

Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{pieter, P.Achten, rinus}@cs.ru.nl

**Abstract.** More and more software systems use a browser as the universal graphical user interface. As a consequence these applications inherit browser navigation as part of their interface. Typical browser actions are the use of the back- and forward-button and the cloning of windows. Browser navigation is difficult to deal with because it has effects that are noticed indirectly by the application logic. It is easy to forget or misunderstand the consequences of this aspect in the construction of a program. Hence, testing the correct behavior of the application is very desirable, preferably with an automatic model-based test tool. For this kind of model-based testing a specification including browser navigation is needed. We introduce a transformation to lift the specification of a program without browser navigation to one with browser navigation. This reduces the specification effort considerably. The distinguishing feature of our method is that it allows the test engineer to specify only the exceptions to the general rule. We show how this lifting of specifications is used for some examples and how errors are found in real web applications. The described system builds on the model-based test tool G∀ST.

## 1 Introduction

Equipping software systems with an HTML-based browser interface has many advantages: the interface becomes platform independent, the look and feel is familiar to new users, and it is often less work to implement a browser based GUI instead of some traditional GUI library. Moreover, the new application obtains browser navigation (cloning of windows and the possibility to go back and forward between previous states of the GUI) for free. The browsers provide this new GUI navigation without any help from the web application by maintaining a stack of previous pages. The possibility to review previous states in the interaction can be very convenient for the user. It is even possible to go to a previous page and give a new input in that state of the GUI to investigate several options or to undo mistakes.

A consequence of using a browser as GUI is that the application, and hence its state, becomes divided between the browser (handling rendering and browser navigation) and the web application (handling the events and all other interfaces of the program) on the server. In thick-clients the browser handles even a larger

part of the application by executing (Java) scripts, but that is outside the scope of this paper. We focus on thin client web applications. By returning to a previous page the current part of the state stored at the client site is replaced by a previous version as well. Since the web application at the server is unaware of this browser navigation, the part of the state stored at the server is unaffected. Whether some part of the state should be changed on browser navigation is problem dependent. Parts of the state representing actions or objects in the real world, like purchases in a web shop, can usually not be undone by the user. Hence these parts of the state should not be changed by going to a previous page, this is achieved by storing them on the server. Other parts of the state, like the contents of the basket in a web shop, can safely be changed by browser navigation and hence should be stored in the page. Storing some part of the state at the wrong place is usually harmless without browser navigation, but using browser navigation reveals the problem, see also [4]. Hence, it is desirable to include browser navigation in the tests of software with a web interface.

In this paper we extend our approach for model-based testing of web applications [6] to the additional behavior imposed by browser navigation. A web application to be tested is modeled by an extended state machine. The test system automatically determines conformance between the web application and its model by generating inputs. The corresponding output is checked by the test system using the model. Many other existing models used for testing browser navigation, like [1], cover only the input elements available in the web-page and are not able to check the correctness of the new page obtained.

The effects of browser navigation on the models needed for testing are severe. Even if the original implementation under test, iut, can be adequately modeled by a *finite* state machine, an *infinite* state machine is needed for testing with browser navigation buttons. Each state in the model without browser navigation has to be replaced by an unbounded number of states where the difference between those states is the history of previous pages reachable with the back-button and the forward-button. It would be tedious if the behavior corresponding to this browser directed navigation must be specified for each and every system. Fortunately the behavior is very similar, but not necessarily identical, for most web applications. This enables us to define a *model transformer* that adds default behavior for the back and forward-button to each state in the model. We have chosen to take the model where all state information is local in the current web page as the default. This default corresponds to the browsing of simple old fashioned web-pages. If other behavior is required for specific states or inputs, the test engineer has to specify only the exceptions to the default behavior.

The organization of this paper is as follows. Section 2 rephrases testing of thin-client web applications in order to make this paper more self contained. In section 3 we elaborate on the special behavior associated with the back- and forward-button. Section 4 introduces a general model transformer to turn a model of a web application without browser navigation into a model with default behavior for the back- and forward-button. This is illustrated by two simple examples, for the first one (section 4.1) the automatic transformation

does everything wanted, for the second example (section 4.2) exceptions of the default behavior are specified. Other forms of browser navigation are briefly touched in section 5. In section 6 we discuss related work. Finally we draw conclusions.

## 2 Model Based Testing of Thin Client Web Applications

In this paper we use the automatic model-based test tool G∀ST [5]. G∀STuses functions in the functional programming language Clean[1] [8] as specification. Distinguishing features of G∀ST are the fully automatic generation of tests, their execution and the generation of verdicts. Input generation for arbitrary types can be derived automatically using generic programming as well as specified explicitly by the test engineer.

Reactive systems such as web applications are modeled with an Extended State Machine, ESM. An individual transition is written as $s \xrightarrow{i/o} t$, where $s$ is the source state, $t$ is the target state, $i$ is the input value, and $o$ the associated output. An ESM is similar to a Mealy Finite State Machine, FSM, but can have an infinite number of states, inputs and outputs. Moreover, an ESM can be *nondeterministic*, i.e. there can be multiple transitions for each state and input. Sometimes the output determines the target state, but there exist also systems with transitions $s \xrightarrow{i/o} t_1$ and $s \xrightarrow{i/o} t_2$ for some $s$, $i$ and $o$ with $t_1 \neq t_2$. From a single source state $s$ there exist two (or more) transitions with identical labels (input and output), but different target states. An ESM used as specification in model-based testing can be nondeterministic for two reasons. Either the system specified is nondeterministic, or the system is deterministic but there is incomplete state information in the specification. Consider for instance the purchase of an item from a webstore. If the goods in stock are unknown in the specification, the model has to allow the response to situation where the item is in stock as well as the situation where the item is not available. The webstore itself will be deterministic; if the item is available it will be sold. The specification of this transition in model-based testing however, must be nondeterministic due to the incomplete state information. Such a situation with incomplete state information in the specification is common in model-based testing.

A specification is *partial* if there is a combination of a reachable state and a valid input that does not occur in any of the specified transitions. The conformance relation defined in section 2.1 states that any behavior of the system under test is allowed if the specification does not cover the current state and input. Since anything is allowed, testing such a transition is useless. G∀ST is able to handle these partial specifications.

For the specification of a web application the state can be freely chosen by the test engineer. The type `HtmlInput` represents input elements in a web-page like: buttons, links, drop-down lists and edit-boxes. One can use any de-

[1] See http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf for the main differences between Clean and Haskell.

sired type for inputs in the model if one provides an instance of the class `transInput i :: i → HtmlInput` for that type. Using a tailor made type instead of `HtmlInput` is convenient in the generation of test data. For instance, the test engineer can construct a data type to generate only integers values between 0 and 10, and define an instance of `transInput` that puts these values in the desired edit-box. The output is always an element of type `Html`. This type is an abstract syntax tree for HTML-code rather than a textual representation. We reuse the type for HTML from the iData toolkit [7], Clean's tool for generic web-page generation.

For the specification of the output of a web application we do not want to specify the HTML after each input completely. That would be much too detailed and restrictive. Instead of modeling a single transition $s \xrightarrow{i/o} t$ by a tuple $(s, i, o, t)$ and the entire specification $\delta_r$ by a set of tuples $\delta_r \subseteq S \times I \times O^* \times S$, we use a function. This specification function $\delta_F$ takes the current state and input as argument and yields a function that takes the output of the web application as argument and yields the set of allowed target states. In this way, the output can be used to determine the target states. Instead of a single function, $\delta_F$ yields a list of functions. Hence, the type of the specification is $\delta_F(s, i) \in S \times I \to \mathbb{P}(O^* \to \mathbb{P}S)$. In this representation the empty set conveniently models partial specifications. The set of functions is convenient in the composition and transformation of specifications as shown below. Moreover, in this representation it is much easier to determine the set of inputs that are allowed in a state $s$ (the init defined below) then in a representation of the specification as a function $\delta_F$ of type $S \times I \to O^* \to \mathbb{P}S$). Mathematically these types of specifications are equivalent, but for a test system the first version is much more convenient. Finally, this representation makes it easier to mix it with the existing specification format used by G∀ST where one specifies the combinations of outputs and target states $S \times I \to \mathbb{P}(O^* \times S)$.

A specification of the form $S \times I \to \mathbb{P}(O^* \times S)$ is similar to a classical Mealy machine were the output function $S \times I \to \mathbb{P}(O^*)$ and state transition function $S \times I \to \mathbb{P}S$ are joined to a single function. Classical Mealy machines are deterministic, i.e. these functions have types $S \times I \to O^*$ and $S \times I \to S$. Moreover, classical Mealy machines handle finite state machines, that is the sets $S$, $I$, and $O$ should be finite. For a nondeterministic specification it is essential to join the output and transition function to a single specification function in order to make the connection between outputs and target state on nondeterministic transitions. Using functions of type $S \times I \to \mathbb{P}(O^* \to \mathbb{P}S)$ for the specification instead of functions yielding a list of tuples has as advantage that is is possible to use a small function as specification instead of a very long list of tuples. In the specification of a web application the type $O$ represents the HTML-pages allowed. In general one does not want to specify the produced output of a web application until the last bit of the HTML output. Details like the background color of the page are usually completely irrelevant and one does not want to specify those particulars. Listing all allowed outputs would at least be nasty and annoying. In such situations a function of type $S \times I \to \mathbb{P}(O^* \to \mathbb{P}S)$ is much more convenient,

the functions $O^* \to \mathbb{P}S$ can implement a predicate over the produced $\mathsf{HTML}$. Of course it is still possible to model a finite state machine in this representation. If the web application at hand should be a finite state machine, we can still test it as a finite state machine. In general the web application is modeled as an ESM, which shows richer behavior.

For a single transition our specification reads: $s \xrightarrow{i/o} t \Leftrightarrow \exists f \in \delta_F(s,i) \wedge t \in f(o)$. For web applications the functions $f$ yielded by $\delta_F(s,i)$ are predicates over the $\mathsf{HTML}$ output of the web application. Such a predicate typically verifies some key aspects of a web-page, like the availability of buttons and specific text fields. We show some examples in section 4.1.

Although technically superfluous, it turns out to be convenient to have the possibility to specify one additional predicate $P$ relating the output and target state in each and every transition. This predicate checks whether the combination of $\mathsf{HTML}$ and target state is well formed[2]. That is $s \xrightarrow{i/o} t \Leftrightarrow \exists f \in \delta_F(s,i) \wedge t \in f(o) \wedge P(o,t)$.

The set of inputs allowed in a state $s$ is $\mathsf{init}(s) \equiv \{i | \delta_F(s,i) \neq \emptyset\}$. A *trace* is a sequence of inputs and associated outputs from some start state. The empty trace connects a state to itself: $s \stackrel{\epsilon}{\Rightarrow} s$. A trace $s \stackrel{\sigma}{\Rightarrow} t$ can be extended with a transition $t \xrightarrow{i/o} u$ to the trace $s \stackrel{\sigma;i/o}{\Longrightarrow} u$. If we are not interested in the target state we write $s \xrightarrow{i/o} \equiv \exists t. s \xrightarrow{i/o} t$ or $s \stackrel{\sigma}{\Rightarrow} \equiv \exists t. s \stackrel{\sigma}{\Rightarrow} t$. All traces from state $s$ are: $\mathsf{traces}(s) = \{\sigma | s \stackrel{\sigma}{\Rightarrow}\}$. All reachable states after a trace $\sigma$ from state $s$ are: $s\,\mathsf{after}\,\sigma \equiv \{t | s \stackrel{\sigma}{\Rightarrow} t\}$. We overload $\mathsf{traces}$, $\mathsf{init}$, and $\mathsf{after}$ for sets of states instead of a single state by taking the union of the individual results. When the transition function, $\delta_F$, to be used is not clear from the context, we add it as subscript to the operator.

### 2.1 Conformance

Here the implementation under test, $\mathsf{iut}$, is a web application. The $\mathsf{iut}$ is modeled as a black box transition system. One can observe its traces, but not its state. The $\mathsf{iut}$ and its specification need not have identical input output behavior in all situations to say that the web application conforms to the specification.

*Conformance* of the $\mathsf{iut}$ to the specification $\mathsf{spec}$ is defined as:

$$\mathsf{iut}\ \mathsf{conf}\ \mathsf{spec} \equiv \forall \sigma \in \mathsf{traces}_{\mathsf{spec}}(s_0), \forall i \in \mathsf{init}(s_0\ \mathsf{after}_{\mathsf{spec}}\,\sigma), \forall o \in O^*.$$
$$(t_0\ \mathsf{after}_{\mathsf{iut}}\,\sigma) \xrightarrow{i/o} \Rightarrow (s_0\ \mathsf{after}_{\mathsf{spec}}\,\sigma) \xrightarrow{i/o}$$

Here $s_0$ is the initial state of $\mathsf{spec}$. The initial state of the $\mathsf{iut}$, $t_0$, is generally not known. The $\mathsf{iut}$ is in this abstract state when we select its $\mathsf{url}$ for the first time. Intuitively the conformance relation reads: if the specification allows input $i$ after trace $\sigma$, then the observed output of the $\mathsf{iut}$ should be allowed by the

[2] In the implementation this function is able to yield an error message if the combination of output and target state is invalid, rather than having a plain boolean as result.

specification. If spec does not specify a transition for the current state and input, anything is allowed.

For the specification spec it is perfectly legal to be partial. That is nothing is specified about the behavior for some state and input combinations. The interpretation in the conformance relation is that all behavior of the iut is allowed. Since everything is allowed, it makes no sense to test this.

The iut cannot refuse inputs. In every state the iut should give some response to any input. This response can be an error message or an empty sequence. During testing for conformance only inputs occurring in the specification after a valid trace will be applied.

This conformance relation is very similar to Tretmans ioco (*I*nput *O*utput *Co*nformance) relation. The original ioco relation [9] handles conformance of labeled transition systems (LTS). The essential difference between a LTS (as used in the ioco relation) and a ESM is that the input and output are separate actions in a LTS. This implies that a LTS allows for instance two consecutive inputs without an (probably empty) output of each of these inputs. Exactly one input and the associated output are combined to a single transition in an ESM. Moreover, an ESM has rich states where the "state" of an LTS is given by the current location in the LTS and the value of a set of global variables. For the abstraction level of web applications used here, the ESM-based view is more appropriate than a LTS-based view: we always want to consider the output associated to a given input.

### 2.2 Testing Conformance

The conformance relation states that for all inputs allowed after all traces of the specification, the input-output pair obtained from the iut should also occur in the specification. Since the number of different traces is unbounded for a general ESM, it is impossible to determine conformance by exhaustive testing. For a general ESM the real conformance can only be determined by model checking the specification and a model of the web application. That is a completely different technique from model-based testing. Here we want to treat the web application as a black box. We can apply an input to the web application and observe the generated output. No detailed model of its behavior is known. Hence, model checking is not an option.

Testing can however give a fair approximation of conformance. Moreover, a large number of correct transitions increases the confidence in the correctness of the iut. Experience shows that nonconformance is very often found rather quickly. Errors are found most often within thousands or even hundreds of transitions, rather than millions of transitions. Nevertheless, it is easy to design an iut with an error that can only be found with an enormous testing effort, but these kind of errors appear to be rare in practice.

Since checking the conformance by testing all possible traces is generally impossible, we test conformance by verifying a fixed number of traces. For each trace we check a finite number of transitions with the following algorithm:

$$\begin{aligned}
&\mathsf{testConf} : \mathbb{N} \times \mathbb{P}\, S_{spec} \times\ S_{iut} \rightarrow \mathsf{Verdict} \\
&\mathsf{testConf}\,(n, s, u) = \textbf{if } s = \emptyset \\
&\qquad\qquad\qquad\quad \textbf{then } \mathsf{Fail} \\
&\qquad\qquad\qquad\quad \textbf{else if } n = 0 \vee \mathsf{init}\,(s) = \emptyset \\
&\qquad\qquad\qquad\quad \textbf{then } \mathsf{Pass} \\
&\qquad\qquad\qquad\quad \textbf{else } \mathsf{testConf}\,(n-1, t, v) \\
&\qquad\qquad\qquad\quad \textbf{where } i \in \mathsf{init}(s);\ (o, v) = \mathsf{iut}\,(u, i);\ s \xrightarrow{i/o} t
\end{aligned}$$

In this algorithm $n \in \mathbb{N}$ is the number of steps still to be tested in the current trace, $s \in \mathbb{P}\, S_{spec}$ is the set of possible states in the specification, $u \in S_{iut}$ is the abstract state of the iut, and Verdict is the result type that contains the elements Fail and Pass. Since the iut is a black box, the state $u$ cannot be observed. We assume that the iut is available as a function of type $(S_{iut} \times I) \rightarrow (O^* \times S_{iut})$. The consistency predicate $P(o,t)$ is incorporated in the transition $s \xrightarrow{i/o} t$. Since the transition function yields a function, the new set of possible states is actually computed as $t = \{x \,|\, \forall s_i \in s, \forall f \in \delta_f(s_i, i), \forall x \in f(o), P(o,x)\}$. Due to the overloading of the transition notation we can write it concisely as $s \xrightarrow{i/o} t$.

Testing of a single trace is initiated by testConf $(N, \{s_0\}, S^0_{iut})$, where $N$ is the maximum length of this trace, $s_0$ the initial state of the specification, and $S^0_{iut}$ the initial abstract state of the iut. The input $i$ used in each step is chosen arbitrarily from the set init$(s)$. In the actual implementation it is possible to control this choice. The default algorithm generates input elements in pseudo random order and uses the first input element that is in init$(s)$, i.e. $\exists x \in s.\delta_f(x, i) \neq \emptyset$.

### 2.3 Implementation

Apart from the representation of the specification the conformance relation used here is identical to the conformance relation implemented previously in G∀ST. The type of specifications handled by G∀ST are functions of type `Spec s i o`. Given a state `s` and an input `i`, the specifications yields a list of functions. Each of these functions yields a list of allowed targets states `[s]` after it receives the output obtained from the iut.

```
:: Spec s i o :== s → i → [[o]→[s]]
```

Since `s`, `i`, and `o` are type variables, one can choose any type for states inputs and outputs. In order to test web applications the test system has to behave as a browser for the web application. The web application expects an input and yields a new HTML-page as response. G∀STis extended by a module that selects the indicated input element from the current page and sends an input to the web application as if it were generated by a real browser. The page received as answer from the iut is used in the test of conformance. The additional predicate to check the consistency of the page and the states of the specification can be given as an option to the main test function:

```
testHtml :: [TestSMOption s i Html] → (Spec s i Html) → s →
            (*HSt → (Html,*HSt)) → *World → *World
```

The first argument is the list of test options. The options can control the number of traces used in the test and their length, the amount of detail written in log-files, the input selection algorithm etcetera. It is also possible to add a predicate checking the consistency of the current HTML-page and the state of the specification. G∀ST has reasonable defaults for all these parameters. Hence, it is often possible to leave this list of options empty. The second argument is the specification as shown above. The next argument is the initial state of the specification. The final arguments are the web application and Clean's world.

In order to facilitate testing of the received page, there are functions to query the data structure representing the HTML-page. For instance it is possible to obtain the list of all texts from the page, and to retrieve all text labeled with some name. These names are the anchors that are used in the page. Requiring anchors with specific names appears to be very convenient, but not necessary, in the testing of web-pages. For example `htmlTxts "Answer"`, applied to the current page yields the list of all strings labeled `"Answer"`.

# 3 Browser Navigation

In order to use our model-based test tool G∀ST to test web applications with browser navigation, we need a model of the desired behavior. First we determine the requirements for such a specification. An unattractive option is to add inputs `Back` and `Forward` to each specification and let the test engineer completely specify the semantics of these actions. *Requirement 1*: it should be easy to transform a specification ignoring browser navigation to a specification prescribing default behavior for browser navigation.

Even if the web application itself is a simple finite state machine with one or more loops the specification needs an unbounded amount of memory to record the potentially infinite number of browser navigation actions. *Requirement 2*: there is no artificial limit on the number of browser navigation actions.

Next we argue that using the back-button is not an undo of the previous transition in the specification. Consider the left state machine in figure 1. Suppose we know that the specification is in state $S_0$. When we apply the input $i_1$ to the iut and observe the output $O_1$ the set of possible states is $\{S_1, S_2\}$. After observing a transition $i_2/O_3$ the state is $S_4$. Hence the previous state was $S_2$. The back button brings the specification in this state $S_2$ via the transition $S_2 \xleftarrow{Back/O_1} S_4$ and not in $\{S_1, S_2\}$. This implies that if we apply the input $i_2$ again the only allowed output is $O_3$. *Requirement 3*: the back-button is a normal transition for the specification, not an undo action for the specification.

If the transition labeled $i_2/O_2$ should be allowed after the trace $i_1/O_1$; $i_2/O_3$; $Back/O_1$, we need a specification as depicted on the right in figure 1.

## 3.1 Persistent states

In the discussion above we have assumed that the web application stores its state in the web-page. This state can either be stored in visible parts like edit boxes

and radio buttons, or in so-called *hidden fields* in the web-page. This content is invisible for the user in a normal browser window, but readable and writable for the web application. If the entire state of the web application is stored in the web-page, the web application goes to its previous state if the user goes a page back with the back-button of the browser.

Consider what would happen if the base converter (introduced in section 4.1) stores the number to be converted, the latest contents of the edit-box, in a persistent memory location, like in a cookie or some storage location on the web-server. If the user uses the back-button the browser displays the previous page, but the number to be converted is not changed even if it is different on the previous page which becomes actual. When the user chooses now a different base on this page not the displayed number is converted, but the number from the persistent memory.

For the number to be converted this is considered as undesirable behavior, but for a counter that keeps track of the total number of conversions done in this session this is exactly what is needed. This implies that the designer of the web application should determine the desired behavior for each variable in the web application.

In order to perform a model-based test of such a web application the model should reflect the persistent or volatile behavior of the modeled components of the state. Instead of splitting the state of each web application explicitly in a persistent and a volatile part, we ask the user to define a function that composes the new state of the specification from the last state and the previous state on a transition corresponding to a back-button. The default function yields the previous state. This corresponds to a complete volatile memory without any persistent component.

## 4 A Model Transformer to Model Browser Navigation

In this section we present a specification transformer that converts a specification without browser navigation to a specification that covers browser navigation. The transformed specification states that a `Back` just returns to the previous state in the specification without any state change corresponding to persistent

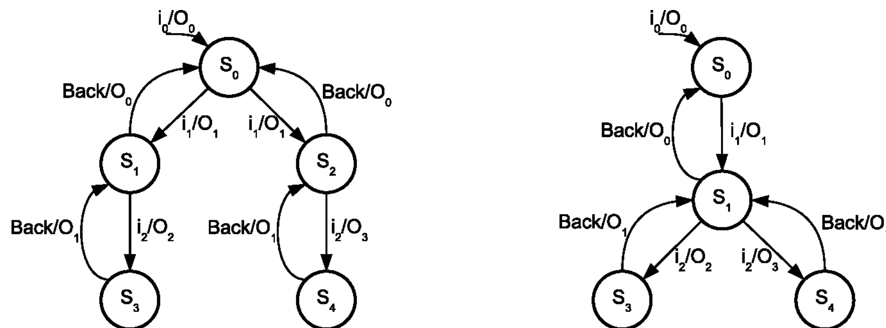


**Fig. 1.** Two nondeterministic state machines with back transitions.

state components. The `Forward` in the specification is always the undo of the corresponding `Back`-action, independent of the existence of persistent components in the state of the web application. The user can easily specify exceptions to this rule.

Web applications are specified most concisely using tailor made types for their input and state. Instances of the class `transInput` transform a tailor made type for inputs to the type `HtmlInput` required by the test system. It is undesirable to extend all types for inputs with their own back and forward buttons, and all types for state with their own history. This problem is solved by using a polymorphic type `BFInput i`. This type states that the input is either an element of the input type of the specification, a backward button from the browser, or a forward button from the browser.

```
:: BFInput i = SpecIn i | BrowserBack | BrowserForward
```

In a similar way we lift the type of a single state of the specification to a state with a history, `past`, and a future, `next`.

```
:: BFState s = {past :: [s], next :: [s]}
```

By convention the current state is the first state in the past:

```
toBFState :: s → BFState s
toBFState s = {past = [s], next = []}
```

Lifting a specification from an ordinary state `s` to a state `BFState s` that is also able to handle browser navigation is done by the function `toBackForwardSpec`. This function transforms both the state and the input to the types given above.

```
toBackForwardSpec ::    (Spec s i Html) (s→Bool) (Maybe (s→s→[Html]→s))
                      → (Spec (BFState s) (BFInput i) Html)
toBackForwardSpec s v Nothing = toBackForwardSpec s v (Just (λc p o→p))
toBackForwardSpec spec v (Just back) = BackForwardSpec
where
  BackForwardSpec {past=[c,p:r],next} BrowserBack
    = [λo→[{past=[back c p o:r],next=[c:next]}]]
  BackForwardSpec {past,next=[n:r]} BrowserForward
    = [λo→[{past=[n:past],next=r}]]
  BackForwardSpec {past=[c:r],next} (SpecIn i)
    = [λo→[{past=if (v c) [n,c:r] [n:r],next=[]}\\n←f o]\\f←spec c i]
  BackForwardSpec {past,next} input = []
```

The predicate `v` checks whether the state to be stored is valid. Only valid states are stored in the history. Particularly the initial state used to startup the web application is usually regarded to be invalid. The optional function `s→s→[Html]→s` can is used to copy persistent parts of the current state to the previous state for a `BrowserBack` input. Note the rather complex behavior specified for an ordinary input: each new state receives its own copy of the history.

### 4.1 Example: base converter

The first example is a web application that converts a decimal number to some other base. The bases supported are 2 (binary), 8 (octal), 12 (duodecimal), and

16 (hexadecimal). The base to be used is selected by the corresponding button. The number to convert is given in an integer edit box. In figure 2 a screen shot and the transition diagram of the Extended State Machine are given.

This example is clearly not a classic HTML-page, it contains buttons and performs computations rather than simple text and links. The state of the specification of this web application contains just the number to be transformed and the base to be used for this transformation. The state is represented by a record of type `State` containing the number and the base as an element of the enumeration type `Base`.

```
:: State = {n :: Int, b :: Base}
:: Base = Bin | Oct | Duo | Hex
```

The behavior of this web application is specified by the function `BaseConvSpec`:

```
BaseConvSpec :: State In → [[Html] → [State]]
BaseConvSpec s (BaseButton b) = [checkN2 {s&b=b} b s.n]
BaseConvSpec s (IntTextBox i) = [checkN2 {s&n=i} s.b i]

checkN2 :: State Base Int [Html] → [State]
checkN2 state b n html
    | findHtmlTexts "n2" html == [convert (toInt b) n] = [state]
    | otherwise                                        = []

convert :: Int Int → String
convert b i
    | i<0       = "-" + convert b (~i)
    | i<b       = baseDigits b !! i
    | otherwise = convert b (i/b) + baseDigits b !! (i rem b)
```

After each transition the specification checks whether the string labeled `"n2"` in the HTML-code received from the web application is equal to the string obtained by transforming the current number to a string in the current base. This example shows how we can use information from the parameterized state of the model and the actual HTML-code of the web application in the construction of the target state of the model. The initial state of the specification is:

```
initState = {n = 0, b = Bin}
```

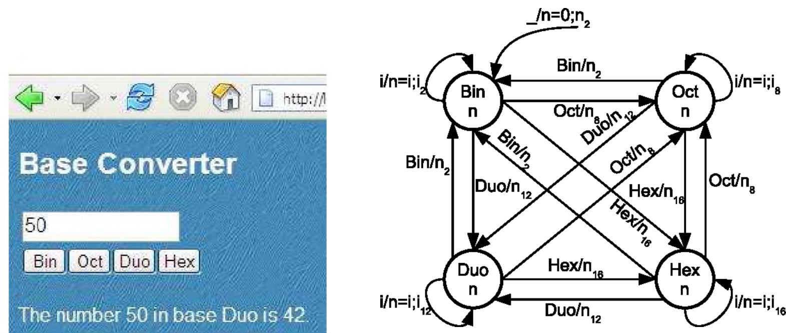


**Fig. 2.** Screen shot of the base converter and its Extended State Machine.

After these preparations we can test an iData-based implementation of the base converter, called `converter`, by executing:

```
Start :: *World → *World
Start world = testHtml [] BaseConvSpec initState converter world
```

The web application is run by executing:

```
Start :: *World → *World
Start world = doHtmlServer converter world
```

Testing the web application `converter` with the specification `BaseConvSpec` does not reveal any issues. All observed traces are allowed by the specification.

**The base converter with browser navigation** For the base converter we are happy with an implementation that treats the back-button of the browser as an undo of the last change. This implies that the implementation can store its entire state in the current page. That is the default choice of the iData, so converter can ignore browser navigation completely.

Using the specification transformer introduced here it is very easy to test whether the web application `converter` shows indeed the desired behavior. The specification transformer `toBackForwardSpec` imposes exactly the desired behavior. In order to execute the model-based tests one executes:

```
Start :: *World → *World
Start world = testHtml [] (toBackForwardSpec BaseConvSpec Nothing)
                          (toBFState initState) converter world
```

The results of this test show that the converter has the behavior prescribed by the specification lifted to the domain of browser navigation.

### 4.2 Example 2: a number guessing game

As a slightly more advanced example we show a number guessing game implemented as web application. The player has to determine a number chosen by the web application in the least number of guesses possible. A smart player will use binary search to complete this task in $O(\log n)$ steps where $n$ is the number of possible values. After entering the right number the web application shows a list of fame: the names of players and the number of guesses needed by them. This application is one of the standard examples in the iData-library. After successful manual testing it was assumed to be correct.

The required behavior of this web application is described by the function `ngSpec` below. The specification keeps track of the upper and lower bound of the possible correct answer. The specification also counts the number of tries and checks if this is correctly reported by the web application for a correct answer.

```
:: NGState = {upB :: Int, lowB :: Int, tries :: Int}

newNGState = {upB = up, lowB = low, tries = 0}
initState  = {newNGState & tries = −1}
```

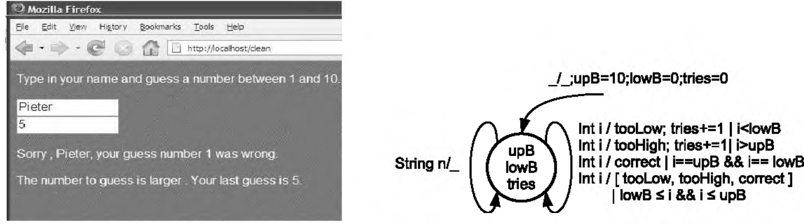


**Fig. 3.** The number guessing web application and a sketch of its state machine.

```
ngSpec :: NGState In → [[Html]→[NGState]]
ngSpec s  input | s.tries<0 = [λ html = [newNGState]] // used at startup
ngSpec s (StringTextBox n)  = [λ html = [s]]
ngSpec s (IntTextBox i)
  | i < s.lowB = [tooLow t]
  | i > s.upB  = [tooHigh t]
  | i == s.upB && i == s.lowB = [correct]
  | otherwise  = [tooLow {t & lowB = i+1},tooHigh {t & upB = i-1},correct]
where t = {s & tries = s.tries+1}

tooLow t html  |    htmlTxts "Hint" html == ["larger"]
               && htmlTxts "Answer" html == ["Sorry"] = [t]
               | otherwise                           = []
tooHigh t html |    htmlTxts "Hint" html == ["smaller"]
               && htmlTxts "Answer" html == ["Sorry"] = [t]
               | otherwise                           = []
correct   html | htmlTxts "Answer" html==["Congratulations"] = [newNGState]
               | otherwise                                   = []
```

Using this specification we tested the iData-implementation of this game. Testing shows an error: in contrast to the specification the web application chooses a new number to be guessed if the user enters a new name. Both choices for the behavior after a name change can be defended, but the choice in the specification and implementation has to be consistent. This shows that model-based testing of web applications is able to find data-related inconsistencies.

**The number guessing game with browser navigation** Using browser navigation, it seems attractive to cheat in this game in order to enter one's name high in the list of fame. A player guesses numbers until she knows the correct value. Then the user presses the back button until she is back to the HTML-page showing this task for the first time. Now she can enter the correct number immediately. In order to prevent this kind of cheating we require that the number of tries is stored in persistent memory (file, database or cookie) instead of in the page. If the number of tries is stored outside the HTML-page, going to the previous page does not affect the number of tries. Hence the player is not able to cheat in this way. In an iData-application such a change is just a matter of indicating a different storage option for the associated counter.

When we lift the specification `ngSpec` to the level of browser actions we need to deviate from the general rule that states that all components of the state

are stored in the page. The number of tries in the previous state `p` should be identical to the number of tries in the current page `c`. Using our specification transformer this is expressed as:

```
liftedSpec          = toBackForwardSpec ngSpec valid (Just backNumGuess)
valid s             = s.tries ≥ 0
backNumGuess c p o = {p & tries = c.tries}
```

Several other issues were found using model-based testing with this specification. **1)** The first issue is found by the predicate that the number of tries as text in the page should always be identical to number of tries in the specification. The first use of the back-button spoils this property. **2)** If one does not play the game to the end and starts a new one later, the web application continues with the old number of tries in persistent memory. **3)** Entering the same input twice in a row is counted as one try in the web application, but as two tries in the specification. **4)** The number to be guessed is stored in persistent memory by the web application, but the bounds are part of an ordinary state in the specification. This leads to inconsistencies in answers if one browses with the back-button over a correct guess and then continues with guessing. The specification prescribes consistency with the old bounds, while the answers of the web application are based on a new target. During the correction of these problems several small errors were introduced accidently and found by G∀ST.

Even testing a simple application that was assumed to be correct raised a number of serious issues. This shows that this framework is able to spot issues in real web applications. Testing with browser navigation often reveals issues related to the kind of storage used for parts of the state of the web application.

To demonstrate the possibilities of introducing additional changes in the lifted specification, we show how we can prevent going back to a previous game in the tests (corresponding to additional issue 4 above).

```
liftedSpec2 ls=:{past=[s,t:r]} | s.tries==0 = liftedSpec2 {ls & past=[s]}
liftedSpec2 s = toBackForwardSpec ngSpec valid (Just backNumGuess) s
```

If the number of tries in the current state is zero this implies that a new game is started. Going back to previous states is prevented by removing these states from the lifted specification. Note that only this additional requirement is defined, the general specification transformation keeps track of everything else.

By this additional requirement we only omit this behavior in the model. A real user might still provide this trace. When the model does not specify any behavior for such a trace, anything is allowed as behavior of the web application. So, if we have an opinion an the behavior of the iut in such a situation, we should specify it rather than remove it from the model.

## 5 More browser navigation

Most browsers provide more ways of browser navigation than just going a single page back or forward. A fairly standard option is to select one of the pages from

the history or future to go $n$ pages back or forward. It is straightforward to model such an transition and hence to test it using our model-based testing approach.

Most browsers allow also the possibility to clone a page. At any moment a user can decide to switch from the current page to such a clone or back and starts giving inputs from that page. This is slightly different from just going $n$ pages back since a new input removes all forward pages (if they exist). A cloned page is not effected by a new input in some other page. In order to test this behavior we have to model it. This implies that we have to maintain a set of `BFStates` instead a single `BFState`. At the input `Clone` we copy the current state from the current `BFState` to a new `BFState`. At the input `Switch` the model makes another `BFState` the current state. Testing proceeds as before using this new state in the specification.

## 6 Related Work

Many other test tools for web applications exist, see www.softwareqatest.com for an overview. They often execute user-defined test scripts (like HttpUnit [2] see also httpunit.sourceforge.net). Browser navigation is only tested if it is explicitly included manually in the scripts. Our tool generates traces on-the-fly from the specification instead of executing predefined scripts. Other tools verify the HTML-code generated by the application, or the existence of the links in a generated page.

The paper by Andrews *et al.* [1] also specifies web applications by state machines in order to test the web applications. They argue that a huge number of errors is introduced by browser navigation. Hence, it is very worthwhile to test this. An important restriction of their approach is that they do not solve the *oracle problem*: their specification only specifies the inputs that should be accepted by the web application. The result of applying an input is not specified and cannot be tested. The reason the oracle problem is not solved is that an input to a web application can have many effects: e.g. an order can be printed or stored in an database. Hence, it is next to impossible to determine all effects of applying an input to a web application. Moreover, even the effect of input to the new HTML-page can be large. Usually we do not want to pinpoint every detail of the HTML-code that should be produced by the web application.

In our approach we cannot specify all effects of an input to a web application. Instead, we write a predicate over the resulting HTML-page. With a predicate we can specify the amount of detail needed, ranging from nothing at all to every detail of the HTML-code. Moreover, we use parameterized state machines rather than finite state machine like Andrews. This implies that we can store much data-related information (like counters, values and names) compactly in a parameterized state. This paper shows that we can specify elements of the page using this information and test these data by the model-based test tool G∀ST. As future research we will investigate how other effects of the web application on the world can be specified and tested.

Another popular approach to specifying transition systems is based on labeled transition systems, most notably the ioco approach and its variants as introduced by Tretmans [9]. In such a system the input and associated output have to be modeled by two separated actions. In our model of web applications the input and output are directly coupled: each input produces a new page as result. This is very convenient for the level of abstraction we want to consider here. In [3] Frantzen et al. describe the basis of an approach to test web services (instead of the web applications handled here). They use a Java like language as carrier for their specification which makes it less suited for function transformations that are the key of our approach. The state of the specification in their current implementation is mapped to a single integer. This implies that both examples used in this paper cannot be handled by their tooling.

We plan to extend G∀ST with asynchronous transitions in order to specify and test systems where the input and output are not necessarily tightly coupled. This would allow the handling of timeouts and web applications based on AJAX-technology.

## 7 Conclusions

It is a trend that new applications start using a browser as their universal graphical user interface. By design or not, these applications receive an interface with browser navigation. It is important to specify and test this behavior. We introduce a specification transformer that makes it easy to lift a specification that ignores browser navigation to a version that includes browser navigation. Only exceptions to the general behavior have to be specified explicitly. In this paper we demonstrate that this technique is capable of spotting errors in real web applications. We have shown that it really matters where the state of an application is stored. If the entire state is stored in the page the back-button corresponds to an undo-action. A state stored at the server is not influenced at all by using the back-button. The desired behavior of such a web application needs to be prescribed in a specification.

Specifying web applications by extended state machines has been shown to be a good basis for model-based testing. Representing the extended state machines by functions in a functional programming language yields very compact and elegant specifications. Due to the use of parameterized types and computations with these parameters, the specifications are clearer and more compact than corresponding graphical representations of the specification. Moreover, the representation of specifications by a function is much better suited for transformations, like lifting to the domain of browser navigation.

We show that this technique is able to spot issues in real web applications. In this paper we used web applications constructed with Clean's iData library, but that is not an inherent limitation of the technique described. For an arbitrary web application the test system will receive a textual version of the page in HTML rather than the data structure used here. The iData system contains a parser that is able to transform the textual representation to the data structure used

here. If the test engineer would prefer, she can also use the textual representation of HTML (or any other representation preferred) in the specification and hence in the tests.

The formal treatment of conformance is improved in this paper. In our previous work [6], the predicate that checks the consistency of the output and the target state was added rather ad-hoc to the test algorithm. In this paper this predicate is part of the transition relation, and in that way smoothly integrated in the conformance relation and the test algorithm.

## References


1. A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modelling with FSMs. *Software Systems and Modeling*, 4(3), August 2005.
2. E. Burke and B. Coyner. *Java Extreme Programming Cookbook*. O'Reilly, 2003.
3. L. Frantzen, J. Tretmans, and R. d. Vries. Towards model-based testing of web services. In A. Polini, editor, *International Workshop on Web Services - Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Italy, June 9th 2006.
4. P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling Web Interactions. In P. Degano, editor, *Proceedings 12th European Symposium on Programming*, volume 2618 of *LNCS*, pages 238–252, April 7-11 2003.
5. P. Koopman and R. Plasmeijer. Testing reactive systems with GAST. In S. Gilmore, editor, *Trends in Functional Programming 4*, pages 111–129, 2004.
6. P. Koopman, R. Plasmeijer, and P. Achten. Model-based testing of thin-client web applications. In K. Havelund, editor, *Proceedings Formal Approaches to Testing and Runtime Verification (FATES/RV06)*, volume 4262 of *LNCS*. Springer, 2006.
7. R. Plasmeijer and P. Achten. iData For The World Wide Web - Programming Interconnected Web Forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *LNCS*, 2006.
8. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/~clean/.
9. J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J. Baeten and S. Mauw, editors, *CONCUR'99*, volume 1664 of *LNCS*, pages 46–65. Springer, 1999.