Model-Based Testing of Thin-Client Web Applications

Pieter Koopman, Rinus Plasmeijer, and Peter Achten

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

Abstract. In this paper we present a novel automated, on-line, model-based testing system for on-the-fly testing of thin-client web applications. Web applications are specified by means of Extended State Machines. To handle dynamic web applications, arbitrarily large and complex state input and output types, and the transport of information from the web-page to the state of the specification, we define a new, ioco like, conformance relation. In this conformance relation a specification is a function from state and input to functions from output to the new states. The implementation builds on the G∀ST test tool and spots errors in real web applications.

1 Introduction

Web-applications have rapidly become popular. Web-interfaces are defined for many ordinary applications. Just like any other software system these web-applications and interfaces tend to contain mistakes. In order to determine the quality of software with a web-based interface it needs to be tested. Testing such software can be done most thoroughly and cost effectively by using an automatic model based test system. Such a system automatically generates test sequences based on a formal specification of the desired behavior of the system, executes the associated tests, and makes a verdict based on the observed behavior of the implementation under test, the iut.

In this paper we present a novel testing system that performs automated, on-line, model-based testing of thin-client (no processing on the client) web applications. Systems are specified using non-deterministic Extended State Machines (ESMs) with arbitrarily rich states, inputs, and outputs. Additionally, the transitions in these state machines are specified by defining them as functions over the output domain to the reachable states. This allows us to concisely express highly dynamic systems with states that depend on the output and eliminates the need to specify and enumerate all possible HTML outputs. For these systems we define a conformance relation that is closely related to the well-known *ioco* relation [20,19]. The system performs on-line testing, as propagated by e.g. Larsen *et al.*[10]. We identify the same advantages: we can employ potentially long test runs, we can limit the state space to a finite portion, and use non-deterministic specifications.

K. Havelund et al. (Eds.): FATES/RV 2006, LNCS 4262, pp. 115–132, 2006.

We intend to perform a black box test and look only at the input and output of the web application. For a web application this implies that the test system performs an input from the current page, e.g. press a button or edit a text field, and receives a new page in HTML. We restrict ourselves to testing the web application based on the HTML input elements available in the page. Hence, we do not consider navigating by back/forward browser buttons, window cloning and history caches. Furthermore, we restrict ourselves to thin clients. The behavior of the web-interface should be determined by exchanging HTML code with the server instead of things like Java(scripts) embedded in the web-page. This restriction enables us to investigate the response to an input just by looking at the HTML code.

We show how a web application can be specified by an ESM. Since there is always a strict relation between selecting an input from the current page and obtaining a new page, we prefer a state machine based specification rather than a specification by a Labeled Transition System (LTS). The web application itself can have an arbitrary complex state, and can contact any system it needs, e.g. a database. As mentioned above, in our black box approach we restrict ourselves to the input to the web-application and the associated new page (in sharp contrast with the approach by Margaria *et al.*[15]) This may appear to be very restrictive, but that is not the case. The test engineer can incorporate any knowledge of the back-end of the system in the ESM specification.

The specification of the web-application can be nondeterministic, either because the iut is not deterministic, or because the specification has only partial knowledge of the world. One of the case studies in this paper tests a web-shop. If it is not known whether an item is available in the web-shop, the specification should handle the situation that the item is available and that it is unavailable. The output of a web-application is typically large, containing a lot of HTML code that is sent to the browser. We do not want to specify each and every detail of this HTML code, nor do we wish to enumerate all allowed responses. Special about our approach is that instead of explicitly describing the allowed outputs in the specification, we use a function that has the actual output as argument and yields the allowed target states. This function can be a predicate that checks aspects of the HTML code. Typical examples are the presence of buttons and key texts. The function can also extract information from the HTML code and store it in the target state. An example of information that we want to store in the state of the specification is the result of queries executed by the web-application. The results can determine future behavior, and should be consistent with later responses of the web-application. In this way we can test the contents of the HTML code produced by very dynamical web-applications, like web-shops.

We define a conformance relation that incorporates parameterized data types for state input and output (infinite number of states), nondeterministic systems, and functions from the output to the target state. The conformance relation is based on the well-known *ioco* relation. As a host language, we use the pure functional programming language Clean [18]. Clean is a state-of-art programming language with support for Algebraic Data Types (ADTs), generic programming

[1], and features the generic test tool G∀ST [6] that is used in this work to implement the testing framework.

The remainder of this paper is structured as follows: we present the formal definitions and the conformance relation in Sect. 2. In Sect. 3 we introduce the test tool G∀ST and explain how it is used to implement the test system based on the formal definitions. Two case studies are presented in Sect. 4: one of a small number guessing game, and one of a dedicated web-shop. Related work is discussed in Sect. 5. Finally, we conclude in Sect. 6.

2 Specification

The test tool G∀ST can handle two kinds of properties. It can test properties stated in logic about (combinations of) functions and it can test the behavior of reactive systems based on an Extended State Machine (ESM). Web applications are reactive systems.

An ESM consists of states with labeled transitions between them. A transition is of the form $s \xrightarrow{i/o} t$, where s, t are states, i is an input which triggers the transition, and o is a, possibly empty, sequence of outputs. The domains of the states, S, inputs, I, and outputs, O, are given by arbitrarily complex, recursive ADTs. These types can be used to model parameterized states, inputs and outputs. None of these types is required to be finite. The model of the system can be nondeterministic, it is possible to define several transitions for one combination of state and input. The conformance relation defined in Sect. 2.2 states that the tested system is free to choose one of these transitions. This constitutes the main difference with traditional testing with state machines where the testing algorithms can only handle finite domains and deterministic systems [12].

A transition $s \xrightarrow{i/o} t$ is represented by the tuple (s, i, o, t). A relation based specification δ_r is a set of these tuples: $\delta_r \subseteq S \times I \times O^* \times S$. Since none of these types is finite, there can be infinitely many transitions. Our specification describes synchronous systems. As reaction on input i the system produces a list of outputs. We assume that we are able to detect the end of this list of outputs. This is similar to detecting *quiescence* in many *ioco* based approaches [19].

For instance, a system that has natural numbers as state, input and output can have transitions of the form: $\forall s, i : \mathbb{N} \cdot s \xrightarrow{i/[s,s+i]} i$ which is equivalent to the set $\{(s, i, [s, s+i], i) | s \in \mathbb{N}, i \in \mathbb{N}\}$. The output of this system consists of the previous input and the sum of the previous input and the current input. The new state is the current input. This rule describes obviously infinitely many individual transitions. Usually we omit the universal quantifiers and write $s \xrightarrow{i/[s,s+i]} i$.

Such an infinite set of transitions is fine for a mathematical specification, but unsuited as a specification for model based testing. Listing all transitions in a table, as is often done for FSM based testing, is impossible. For our ESMs this would yield an infinite table. A predicate that given the source state, input, output and target state tells whether the transition is allowed is also not suited for several reasons. First of all, we want an easy way to determine for which

inputs a transition is defined given the current state s. Secondly, we want to compute the target state, t, from a known source state, the supplied input and the observed output.

2.1 Transition Functions

In [7,23] we defined a transition function that meets the requirements that were mentioned in the previous section. The transition function δ_f is defined by $\delta_f(s,i) = \{(o,t)|(s,i,o,t) \in \delta_r\}$. Hence, $s \xrightarrow{i/o} t$ is equivalent to $(o,t) \in \delta_f(s,i)$. The type of δ_f is $s \times i \to I\!P\,(o^* \times s)$, with $I\!P\,x$ powerset of x. The system containing only the transition $s \xrightarrow{i/[s,s+i]} i$ can be specified by $\delta_f(s,i) = \{([s,s+i],i)\}$.

The transition function δ_f works very well as specification in model based testing if the number of output-target state tuples, (o,t) in the specification is small. In a number of situations the number of output-target state tuples can become very large. A typical example is an authentication protocol. On the input get-challenge, the protected system should produce a number from a large set, say a 64-bit number. This would require 2^{64} output-target state tuples. For web based specifications the situation is even worse. We do not want to specify each and every detail of the HTML code obtained from the server. We only require some details like the title of the web page and the availability of certain buttons. This would require an unbounded number of output-target state tuples.

In order to cope with these requirements we replace [1] the output-target state tuples by a function from output to the allowed target states. This yield a new kind of transition function called $\delta_F(s,i)$ of type $s \times i \to (o^* \to (I\!P\,s))$:

$$\exists f \in \delta_F(s,i) \wedge (o \mapsto T) \in f \Leftrightarrow \forall t \in T : (s,i,o,t) \in \delta_r.$$

or in other words $s \xrightarrow{i/o} t \Leftrightarrow \exists f \in \delta_F(s,i) : t \in f(o)$.

For our example $s \xrightarrow{i/[s,s+i]} i$ we can use the transition function

$$\delta_F(s,i) = \{f\} \textbf{ where } f\ o = \textbf{if } (o == [s,s+i]) \textbf{ then } \{i\} \textbf{ else } \emptyset$$

If we require that the output is a value between the current state and current input we have: $\delta_F(s,i) = \{\ o \to \textbf{if } (s \leq o \wedge o \leq i \vee i \leq o \wedge o \leq s)\ \{i\}\ \emptyset\}$ This system is much harder to describe by a function yielding a set of tuples, the number of tuples and their contents depends on s and i. Enumerating all possibilities is cumbersome and can yield a very large set of tuples. Hence, the specification by transition functions that yield a function instead of a set of output-target state pairs really adds descriptive power.

A specification is *partial* if for some state s and input i we have $\delta_F(s,i) = \emptyset$. A specification is *deterministic* if for all states and inputs all functions from the corresponding set of functions contain at most one function and there is at most one target state for each output. Formally: $\forall s\, \forall i, \forall o : \#\bigcup f(o) | f \in \delta_F(s,i) \leq 1$.

[1] The test tool G∀ST allows that the transition function yields tuples or functions. This gives maximum freedom in the specification of the system. For simplicity we assume here that the new transition function always yields a function.

A *trace* σ is a sequence of inputs and associated outputs from a given state. Traces are defined inductively: the empty trace connects a state to itself: $s \stackrel{\epsilon}{\Rightarrow} s$. We combine a trace $s \stackrel{\sigma}{\Rightarrow} t$ and a transition $t \xrightarrow{i/o} u$ from the target state t, to trace $s \stackrel{\sigma;i/o}{\Longrightarrow} u$. We define $s \xrightarrow{i/o} \equiv \exists t.s \xrightarrow{i/o} t$ and $s \stackrel{\sigma}{\Rightarrow} \equiv \exists t.s \stackrel{\sigma}{\Rightarrow} t$. All traces from state s are: $traces(s) = \{\sigma | s \stackrel{\sigma}{\Rightarrow}\}$. The inputs allowed in a state are given by $init(s) = \{i | \exists o : s \stackrel{i/o}{\Longrightarrow}\}$. The states after trace σ in state s are given by $s \,\mathsf{after}\, \sigma \equiv \{t | s \stackrel{\sigma}{\Rightarrow} t\}$. We overload *traces*, *init*, and after for sets of states instead of a single state by taking the union of the individual results. When the transition function, δ_F, is not clear from the context, we add it as subscript.

2.2 Conformance

The basic assumption for testing is that the iut has the same input/output behavior as a state machine: all output is initiated by an input. This implies that it is possible to obtain a trace from the iut. Since we do black box testing, the state of the iut is invisible. It is assumed that the iut accepts any trace of the specification. This is a weaker requirement than *total* or *input enabled* which is often assumed in similar conformance relations. These traces only contain inputs/output pairs covered by the specification. This means for instance that if the specification allows to push a button on a web-page after a sequence of transitions, that the iut should accept this input as well.

Conformance of the iut to the specification spec is defined as (s_0 is the initial state of spec, and t_0 the initial state of iut):

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

Intuitively: if the specification allows input i after trace σ, then the observed output of the iut should be allowed by the specification. If spec does not specify a transition for the current state and input, anything is allowed. This notion of conformance is very similar to the *ioco* relation [20,19] for LTSs. In a LTS each input and output is modeled by a separate transition. In our approach an input and all induced outputs up to *quiescence* are modeled by a single transition.

2.3 Testing Conformance

The conformance relation *conf* tells when an implementation iut conforms to a specification spec. In practice it is usually impossible to determine conformance by testing. Both the number of traces of the specification, $traces_{\mathsf{spec}}(s_0)$, and the length of individual traces can be infinite. This implies that determining conformance by experimentation generally requires the execution of infinitely many transitions, and hence takes infinitely long. Instead of determining the conformance of all transitions from all possible traces, we determine the correctness of a limited amount of transitions in a limited number of traces. As usual, testing approximates the conformance relation. If we find an error during testing the

conformance relation does not hold. When no errors are found we gain confidence in the conformance of the iut to the specification, but errors may remain.

For the implementation of a test system it is very inconvenient to record all traces of the specification corresponding to the observed trace of the implementation. There can be a huge number, in fact even infinitely many, of these traces of the specification. Instead of keeping track of all traces of the specification that conform to the observed trace, our test algorithm records all states in the after set of the specification given the observed trace. By a well engineered specification, this set can always be sufficiently small.

In the test algorithm we assume that the iut is available as a function of type $(S_{iut} \times I) \rightarrow (O^* \times S_{iut})$. In this function S_{iut} is the abstract state of the iut that is carried around as a black box. The test algorithm for a single trace is:

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

Since the transition function yields a function, the new set of possible states is actually computed as $t = \bigcup\{f(o) \,|\, \forall f \in \delta_f(s_i, i), \forall s_i \in s\}$. 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 $\mathsf{testConf}\ (N, \{s_0\}, S^0_{iut})$, 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 can be chosen arbitrarily from the set $\mathit{init}(s)$. In the actual implementation it is possible to control this choice. In a complete test the nondeterministic computation $\mathsf{testConf}\ (N, \{s_0\}, S_{iut})$ is repeated M times. Before each of these test runs, the iut is brought to its initial state by applying the function $\mathit{reset} : S_{iut} \rightarrow S_{iut}$ to the state of the iut. If one of these test runs yields Fail, the iut is known to be not conforming to the specification, otherwise it passes the conformance test.

Due to the dynamic choice of the input to be used in the next transition the testing is called *on-the-fly*. This means that input generation, test execution, and result analysis are performed in lock-step, so that only the inputs actually needed are generated.

2.4 Testing Consistency of Outputs

For large and rich outputs, like HTML code, the internal consistency of the output as well as the consistency of the output with the target state requires some attention. For instance, if one goes to the next page in a series of pages in a web-shop, it is required that the items displayed in the HTML code are indeed the items on the desired page.

In principle it is possible to handle this in the transition function. If the output does not correspond to the intended target state, the transition function can simply yield an empty set of states. If there are no other transitions specified, there will be no target state and hence our test algorithm will determine an error. However, it can be pretty hard to spot the error in the given trace. We can improve this by introducing a separate predicate over the observed output and the set of target states of the specification. If the predicate holds, testing continues as usual. Otherwise, we have found an error and testing terminates [2]. To capture this notion we define a new transition function δ_P that is very similar to δ_F. The extension is that a transition $s \xrightarrow{i/o;p(o,t)} t$ implies $s \xrightarrow{i/o} t \wedge p(o,t)$. Written in terms of the transition function this is: $s \xrightarrow{i/o;p(o,t)} t \Leftrightarrow \exists f \in \delta_F(s,i) : t \in f(o) \wedge p(o,t)$. The corresponding testing algorithm makes clear why it is more convenient to have a predicate of type $O^* \times I\!P\, S \to \text{Bool}$ than $O^* \times S \to \text{Bool}$:

$$
\begin{aligned}
&\mathsf{testConf}_P : \mathbb{N} \times (\ I\!P\ S) \times\ S_{iut} \to \mathsf{Verdict} \\
&\mathsf{testConf}_P\ (n, s, u) = \textbf{if}\ s = \emptyset \\
&\qquad\qquad \textbf{then}\ \mathsf{Fail} \\
&\qquad\qquad \textbf{else if}\ \mathit{init}\ (s) = \emptyset \vee n = 0 \\
&\qquad\qquad \textbf{then}\ \mathsf{Pass} \\
&\qquad\qquad \textbf{else if}\ P_{\text{consistent}}(o,t) \\
&\qquad\qquad \textbf{then}\ \mathsf{testConf}_P\ (n-1, t, v) \\
&\qquad\qquad \textbf{else}\ \mathsf{Fail} \\
&\qquad\qquad \textbf{where}\ i \in \mathit{init}(s);\ (o,v) = \mathsf{iut}\ (u,i);\ s \xrightarrow{i/o} t
\end{aligned}
$$

G∀ST implements this algorithm extended with the collection of data indicating the trace and the error if testing yields Fail. Moreover, the test engineer is able to influence testing details like the choice of the input i from $\mathit{init}(s)$.

3 G∀ST

The test tool G∀ST executes conformance tests according to the conformance relation in Sect. 2. In order to execute such a conformance test we use: **(1)** a specification in some executable form; **(2)** an implementation of the conformance test algorithm; and **(3)** an interface to the iut. We discuss these topics briefly.

In Sect. 2 we have shown that specifications are represented by functions over user defined, and problem dependent, ADTs for state, input and output. Instead of defining a new language for this purpose, we use the high level functional programming language Clean as carrier for these specifications. Modern functional programming languages are known for their high expressive power and concise function definitions. We consider it much better to reuse decades of language design and compiler technology than to define a new language.

[2] In the actual implementation of G∀ST, this predicate is replaced by a function yielding success or a list of error messages.

For the implementation of the test system we also use Clean. This prevents a language border between the specification and its use. Moreover, Clean provides polymorphism, overloading and generic programming. These techniques enable us to use functions over various types in a very convenient way. This is particulary useful for the functions used as specification. The types used in these functions for state, input and output are tailor-made for the system at hand. Using generic programming the generation of input elements [8], the printing and comparing of elements of all types needed can be generated automatically.

The test tool G∀ST implements the test algorithm presented above with a few additional bells and whistles. For instance, the system records the trace leading to an error. Most importantly, it controls the choice of the input to be applied to the iut. By default G∀ST generates a list of elements and pseudo randomly selects an input element, i, that is accepted by the specification. That is, there is a state s_i in the set of possible states of the system such that $\delta_f(s_i, i) \neq \emptyset$. The test engineer can provide a user defined selection algorithm. A default algorithm is provided to select all traces needed to fully test a FSM. The test engineer can provide an algorithm to guide the test to specific targets.

In order to apply an input to the iut and to obtain the answer, the test system needs an interface to the iut. G∀ST assumes that there are two functions in this interface. The first function takes the input to the iut as argument and yields the corresponding output from the iut to G∀ST. In the case of testing web applications typical inputs are pushing buttons and editing text boxes. The output is the HTML code that corresponds with the new web page. The second function, `reset`, brings the iut to its initial state at the start of a new trace.

4 Testing Web Applications

We test web applications from the viewpoint of a user. The user enters a URL in a browser and obtains an initial web-page. In such a page there can be various ways to give input, like buttons, edit fields, and dropdown menus. If the user supplies such an input, the browser sends the current page and information about the input to the web application. In response the web application sends a new web-page in HTML to the browser.

For automatic model based testing, our test system G∀ST provides the input and checks the HTML code received as response. We use a data structure representing the HTML code instead of a textual representation. The data structures for HTML from the iData approach [16,17] are reused. Without restricting the general approach in any way we test web applications constructed with iData. Compared with testing an arbitrary web application it has as advantage that it enables us to make a shortcut that increases the speed of testing. Instead of transforming the data structures generated by the web-application to HTML text, transmitting this text over the web, parsing the text, and converting it to a suitable data structure to inspect the code in a structural way, we directly pass the HTML data structure to the web-interface of G∀ST. Also the input is sent directly as data structure from G∀ST to the web-application under test.

Within the HTML data structure all viewable information is stored in a list of body-tags. The recursive ADT for body-tags contains separate cases for items like strings, tables, buttons, and edit fields. To retrieve information from these data structures easily we have created functions to select strings, tables and table contents from HTML or body-tags. The function `findBodyTags` finds the named list of body-tags in a specification.

In the examples below we assume that we have limited information of the iut. In the number guessing game the specification does not know the number to be found, and in the CD-shop the specification does not know the content of the CD database at the back-end of the application. Nevertheless, we are able to do useful tests and to spot errors in both cases. Including the CD database in the specification allows us to check more details of the obtained web-pages.

4.1 Example 1: A Number Guessing Game

The first example is a number guessing game that randomly selects a number between integer bounds *low* and *up*. After each guess, the game provides feedback: if the number is too low (high), the guess count is incremented, and the player is told that the number to guess is larger (smaller); if the number matches, then the player's name and used number of guesses are entered and displayed in the Hall of Fame. At any time, a different player name can be entered.

Although this is a small example, there are many aspects that can be tested. To mention just a few of them: **(1)** the game should give consistent answers to guesses; **(2)** the Hall of Fame should add the player with the given name and number of guesses; **(3)** the Hall of Fame should be persistent and not alter existing entries; **(4)** entering a different player name should not change the state. Here we test aspect **(1)** and **(4)**.

The specification is a state transition function written in Clean [18] is given in figure 1. The function `spec` is the heart of the specification. The state used in this specification consists only of the integer to be guessed. The transition from initial state to running state (line 2) is a standard idiom for web applications. In this line `Init` is some integer value outside the range of valid numbers to be guessed. Line 3 captures every switch to a new name. Lines 4-7 are concerned with numerical input. Lines 5 and 6 handle incorrect inputs. Line 5 states that if the input `i` is smaller than the goal `g` only the transitions described by the function `tooLow` are allowed. Line 6 states that only the transition described by `tooHigh` is allowed when `i>g`. If `i` is neither smaller nor larger than `g`, it will be equal to `g`. This is handled in line 7. In this situation the guess should be correct. This is handled by the function `correct`.

The functions `tooLow`, `tooHigh`, and `correct` are the functions that compute the reachable states from the associated input and output page. They are very similar. They inspect the HTML text elements that are tagged with labels `"Hint"` and `"Answer"`. For instance, `correct` demands that the text line labeled with `"Answer"` has content `"Congratulations"` and resets to a new guess state. Note that each function alternative yields a list of functions of type `[Html] → [Int]`. This is the instance of $O^* \rightarrow I\!P\, S$ for this test.

```
spec :: Int In → [[Html] → [Int]]                                          1
spec Init input = [ FTrans (λhtml = newGuess)]                             2
spec r (StringTextBox s) = [λhtml = [r]]                                   3
spec g (IntTextBox i)                                                      4
    | i<g       = [(tooLow [g])]                                           5
    | i>g       = [(tooHigh [g])]                                          6
    | otherwise = [(correct newGuess)]                                     7

tooLow r [html]                                                            8
    | htmlTexts (findBodyTags "Answer" html) == ["Sorry"] ∧                 9
      htmlTexts (findBodyTags "Hint" html) == ["larger"] = r               10
    | otherwise = []                                                       11
tooHigh r [html]                                                           12
    | htmlTexts (findBodyTags "Answer" html) == ["Sorry"] ∧                 13
      htmlTexts (findBodyTags "Hint" html) == ["smaller"] = r               14
     | otherwise = []                                                      15
correct r [html]                                                           16
    | htmlTexts (findBodyTags "Answer" html) == ["Congratulations"] = r     17
    | otherwise = []                                                       18
```

Fig. 1. The specification of the number guessing game

The function `newGuess` yields the list of states for a new game. Since we assumed that the specification has no knowledge about the choice of numbers to be guessed, it yields the list of all numbers from the lower bound up to the upper bound: `newGuess = [low..up]`.

The states of numbers, `g`, that appear to be incorrect will be eliminated as soon as the iut gives a reply that is not consistent with the behavior of `spec` for that `g`. Suppose that `low` is 1, `up` is 10. This implies that all number from 1 to 10 are allowed states in the specification after initialization. Assume that we supply the input 5 and the iut replies `Sorry, larger`. This will eliminate states 1 to 5. When the next input is 4 and the iut would answer `Sorry, smaller` this is clearly inconsistent behavior. The specification will notice this since there is no transition matching this HTML-output on input 4 for states 6 to 10. As a result the set of allowed states in the specification becomes empty. Hence, the iut did a transition that is not covered by the specification, i.e. an error occurred.

Input Generation. The inputs for this web-application are either a new name in the string text box, or a new guess in the integer text box. This is modeled by the algebraic data type `In`.

```
:: In = StringTextBox String | IntTextBox Int
```

During testing instances of this type are needed in order to determine the next input. G∀ST is able to derive all possible inputs values from the type definition for `In` automatically. However, the generic generation algorithm used for this has no notion of the intended use of these values and will produce many values that are not very sensible for testing this web-application. Instead of deriving values

for the type `In`, we specify to use only the name `Tester` and the integer values from `low-1` to `up+1`.

```
ggen{|In|} = [StringTextBox "tester": [IntTextBox i \\ i ← [low-1..up+1]]]
```

The border values `low-1` and `up+1` are added to include some invalid numbers in the tests. A single name appears to be sufficient in the tests. Using different names would be very simple. G∀ST tries these values in a pseudo random order. In each state G∀ST applies the first input element that is accepted by one of the current states of the specification (e.a. is an element of $init(s)$). Since all inputs of type `In` are accepted by the given specification, the sequence of inputs used in the tests is a pseudo random choice of elements from the values defined above.

Test Results. We have run the test against an iut that interprets the switching of player names differently than the specification does: whenever a new player name is entered, the iut starts with a new number to guess. This violates the behavior specified at line 3 of the test specification: nothing should change. After entering a new name the iut gives answers that are not consistent to previous guesses. G∀ST spots that there are no transitions according to the reactions observed from the iut for the remaining states. Hence an issue is reported. When testing against a maximum trace length of 100 transitions, the system requires on average 3 paths to reveal the error (more precisely, 325 transitions). The average testing time was 0.80 sec per detected error. Testing was done on an AMD Athlon XP 2200+, 1.80GHz PC, 512MB RAM, running Microsoft Windows XP.

This very simple example shows that G∀ST is able to find real errors in web-applications. In order to find this inconsistent behavior, the test system has to gather information from the HTML-page generated by the iut and compare it with information from previous responses.

Efficient State Representation in the Specification. The specification in Fig. 1 uses one state for each value that is still a possible number to be guessed. For ranges up to hundreds of allowed numbers this is no problem. When the range of numbers would be extended to many thousands of values, handling all these individual numbers in the test system states takes a noticeable amount of time. Fortunately, it easy to change the specification such that also a huge range of numbers to be guessed can be handled. The numbers that might be correct is always the entire sequence of numbers from the largest guess that was too low, up to the smallest guess that was to high. Instead of storing all possible numbers, we can better store the bounds of this sequence. The corresponding specification is given in Fig. 2. The type `SpecState` defined in line 1 stores the bounds of the correct numbers in the arguments of the constructor `RunningS`. Line 2 states that the bounds of the possible correct numbers are initial the bounds given in the game. Line 3 and 4 handles the initial game and entering a new name, these are direct mirrors of line 2 and 3 of `spec` in figure 1. Line 7 and 8 state that for a guess outside the bounds only the corresponding output with `Sorry` is allowed. When the input is equal to both bounds, it has to be correct

```
:: SpecState = InitS | RunningS Int Int                                          1
newGame = RunningS low up                                                        2

spec2 :: SpecState In → [[Html]↦[SpecState]]                                     3
spec2 InitS input          = [λhtml = [newGame]]                                 4
spec2 r (StringTextBox s) = [λhtml = [r]]                                        5
spec2 (RunningS l u) (IntTextBox i)                                              6
    | i < l = [tooLow [RunningS l u]]                                            7
    | i > u = [tooHigh [RunningS l u]]                                           8
    | i == l ∧ i == u = [correct [newGame]]                                      9
    | l≤i ∧ i≤u = [tooLow [RunningS (i+1) u], tooHigh [RunningS l (i-1)]         10
                  ,correct [newGame]]                                            11
```

Fig. 2. The specification of the number guessing game using one single state

(line 9). Otherwise the guess might be too low, too high or correct. The state is adapted correspondingly in line 10 and 11.

Testing with this specification produces the same issue as the tests with the previous specification. Since the number to be guessed in in the range from 1 to 10, choice of the seed for the pseudo random numbers in G∀ST dominates the effects of the more compact representation of the states.

More Controlled Tests. When the test engineer wants more control over the test there are several options. By using a partial specification one can exclude parts of the behavior from the tests. For instance if the right hand side of the function alternatives of line 7 and 8 are replaced by [] (undefined) no tests for input values outside the range of possible correct numbers will be done. In this example the error is found quicker by inputs that have to yield *too low* or *too high*. These inputs are excluded in the test by the partial specification. Hence it takes about 20% more transitions to find the error.

Another possibility to control the test process is by specifying a function that determines the possible inputs for a given state. This function can be supplied as optional argument to G∀ST. In this way we can force G∀ST to test only guessing by binary search and changing names:

```
iFun (RunningS l h) = [StringTextBox "tester2", IntTextBox ((l+h)/2)]
iFun InitS          = [StringTextBox "tester1", IntTextBox ((low+up)/2)]
```

Using binary search and the original specification form Fig. 2, the error is found in about 20% less transitions. No matter what variant of testing we use the error is always found pretty quickly. The longest test run to the first error observed is 1605 transitions and takes less than 5 seconds. For more complicated examples it might be worthwhile to guide the testing process more precisely. This section just indicates that G∀ST offers the tools to do this easily.

4.2 Example 2: A Web-Shop

Our second example is a highly dynamic web-shop selling CD's. This application contains four main views: **(1)** the initial home-view; **(2)** the shop-view to browse, search and order the CDs in the shop; **(3)** the basket-view to examine and change the CDs the user is ordering; **(4)** the order-view to make the order definitive and pay. The actual contents of the shop-view is determined by the contents of a database. The contents of the basket-view and order-view are determined by the CDs selected by the user.

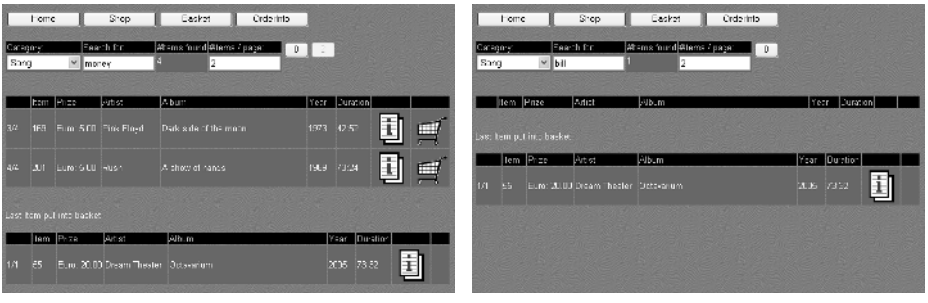

Fig. 3. Screen shots of the web-shop. On the left page 3 of the shop-view, on the right the graphical representation of the error found by G∀ST

The specification does not know the contents of the database, so we cannot check whether the right CDs are displayed. Still, the specification does prescribe consistent behavior during the navigation and searching in the shop-pages, and takes care that ordered items appear in the basket and the final order.

Also in this web-application G∀ST found an error. If the user is not on the first page with CDs and makes a selection (on artist name, album or song), the web application does not go to the first page of CDs. This can cause that an empty page with CDs is shown although there are CDs in the selection.

The complete specification is too large for this paper. Fig. 4 contains a self-contained specification that is just capable of finding the described error. This

```
shopSpec :: ShopState ShopInput → [[Html] → [ShopState]]                    1
shopSpec s=:{view=InitView} input = [λo → [{s&view = HomeView}]]            2
shopSpec s ShopButton                                                       3
  = [ λ[html] → [{s&view = ShopView, cds = findCdCount html}]]              4
shopSpec s=:{view = ShopView} (PageButton (PageNum n))                      5
  | n ≠ s.pageNum ∧ n*s.itemsPage < s.cds                                   6
              = [λ[html] → [{s & pageNum = n}]]                             7
  | otherwise = []                                                          8
shopSpec s=:{view = ShopView} (SearchTextBox str)                           9
  = [ λ[html] → [{s & pageNum = 0, cds = findCdCount html}]]                10
shopSpec s i = []        /* default: undefined */                           11
```

Fig. 4. The partial specification of the web-shop

is only part of the complete specification, but it can be used on its own by G∀ST and finds the error quickly. Line 2 covers the standard transition from the initial state to the home page. The lines 3 and 4 states that the shop-button brings you from any state to the shop-view. The number of CDs is retrieved from the HTML code and stored in the `cds` field of the shop state record of type `ShopState`. Lines $5-8$ handle navigation through the various pages in the shop-view. Such a transition is only possible if the target page is different from the current page and exists. Entering a new text in the search box is specified in line $9-10$. The specification states that the number of CDs in the state must be read from the page and the page number should be set to 0.

The inconsistency is spotted by a predicate over the output and the new state. This predicate checks whether the CDs with desired numbers, represented as string like `"3/7"` (third of seven CDs), are listed on the current page.

5 Related Work

Testing web applications is experiencing an increased interest. A wide variety of existing testing techniques and theories are being extended and modified for the web. It is beyond the scope of this paper to discuss them all.

In van Beek and Mauw [22] black box conformance testing of thin (no local client based computations) Internet applications is presented. In their approach, Internet applications are modelled with MRRTS-es (*multi request-response transition systems*). In order to create specifications conveniently, they use the process algebraic DiCons [21] specification language. DiCons has been developed specifically for *distributed consensus* applications. These are applications in which several users have a common goal that needs to be reached. In their test system, they run the implementation under test and consider the link-activations and form submissions. Differences with our approach are that we use a functional specification style with rich algebraic data types; the implementation under test is a function that yields HTML code; we test only form submissions.

In Sect. 1 we have argued that interactive applications are modelled naturally with Extended State Machines, which are LTSs over input/output pairs. Conformance of these systems is well studied by Latella and Massink [11]. They prove that a *quiescence* supporting semantics is crucial to obtain substitutivity properties: implementations conforming to a specification can be safely replaced with a testing equivalent implementation without breaking conformance, and implementations conforming to a specification also conform to testing equivalent specifications. Our approach is geared towards practical situations in the sense that we consider states, input and output labels to be values of arbitrarily complex, recursive ADTs. It is an interesting and open question whether the theoretical results also hold for our approach.

Frantzen *et al.*[4] study black box conformance testing with symbolic state. This is related to our work because they address the issue of working with arbitrarily complex data structures. In their approach the data structures are specified by means of first order logic specifications. Their approach is more

general than our approach, but this leads to a number of open issues, such as finding the solution to a logical formula (if it exists at all), and the actual computation of concrete input values to the iut. Our approach is based on ADTs, and functional term graph rewriting. Confluence holds for these systems, and our ESMs can rely on arbitrarily complex state transition functions to describe complex systems.

Andrews *et al.*[2] employ FSMs with constraints to model and test web applications. Hierarchical decomposition and constraints are used to control the usual state space explosion problem: with hierarchical decomposition the FSM can be decomposed recursively into subsystems. For each subsystem tests can be generated and assembled into compound tests up to the entire application level. Constraints for sequencing and sets remove the need to tediously specify all different possible input sequences in terms of state transitions. The hierarchical decomposition is done manually by the tester, as well as defining the constraints. The inputs on which the constraints are defined correspond with standard form elements, such as (multi-)lines, URLs, links, (radio) buttons, and so on. As in our approach, they model the web application at the user level.

Wu and Offutt [24] model web applications by identifying the structure of web pages in terms of atomic sections that are composed with process algebraic like operators such as sequential composition, choice, and aggregation. Interactions, such as link transitions, composite transitions, and operational transitions, define the relationship between different pages. From these models, tests can be derived. As with our approach, the authors restrict themselves to monitoring HTML output only. In contrast with our approach, they deliberately ignore state. This is argued by the fact that the HTTP protocol is stateless. However, a standard way to include state is to pass additional information along with the HTML.

Jia and Liu [5] present a general framework to automatically test several key aspects of web applications, such as functionality, page structure (which is what our approach concentrates on), security and performance. XML is chosen to formally specify the test because it also provides access to specify page structure properties using standard utilities such as DOM and XPath. A test specification is a set of test suites. A test suite is a set of test steps. A test case is a tree of test steps. A test sequence is a traversal from root to leaf of a test case. A test step is a (possibly guarded) request-response pair that is executed only if the guard is true. The request is a pattern of HTTP request that need to be matched. The response is an assertion on the HTTP output of the web application. XML is also used by Lee and Offutt [13] as a vehicle for test specifications and data transmissions. In our approach web pages are modelled by means of ADTs, and access to these pages is provided by means of functions. Advantages of our approach are that specifications are type correct, and that the user can specify arbitrarily complex computations on these pages (for instance, extract the complete content of a table and return it as a matrix of values).

Although we have not considered incorporating testing of browser functionality such as window cloning and the use of the back/forward browsing buttons as done e.g. by Di Lucca and Di Penta [3], our framework can be used for these

purposes. It is up to the test engineer to model the desired behavior of the application under these circumstances. This is even the case when testing the behavior of web applications in the presence of users who manually edit links or even alter page codes. Usually for these kinds of robustness tests white box testing techniques are used (e.g. Liu *et al.*[14] and Kung *et al.*[9]). Our system is independent of the concrete implementation language(s) of the web application.

6 Conclusions

The automatic, model based, testing of web applications is an important topic since the number applications is growing rapidly. Thin-client web applications send a complete new web page in pure HTML to the browser in response to each input. Usually it is undesirable to specify each and every aspect of this HTML code. For most specification techniques this is troublesome since they commonly require to explicitly list the combinations of allowed output and target state. In this paper we introduced a specification technique and the associated, *ioco*-like, conformance relation to tackle this problem. The key step is to replace the combination of allowed outputs and target states by a function from output to allowed target states. This function can check aspects of the output, as well as retrieve information to be stored in the target state.

This technique is implemented as an extension of the on-the-fly test tool G∀ST. In this paper we illustrate with two examples that it is possible to (partially) specify the desired behavior of highly dynamic web applications in this way and to find errors in the concrete implementations of these web applications.

References

1. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, Sept. 2002.
2. A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modelling with FSMs. *Software Systems and Modeling*, 4(3), August 2005.
3. G. Di Lucca and M. Di Penta. Considering Browser Interaction in Web Application Testing. In *Proceedings of the 5th International Workshop on Web Site Evolution*, Amsterdam, The Netherlands, October 2002. IEEE Computer Society, USA.
4. L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Proceedings 4th International Workshop, FATES 2004, Revised Selected Papers*, volume 3395 of *LNCS*, pages 1–15, Linz, Austria, September 21 2004. Springer-Verlag.
5. X. Jia and H. Liu. Rigorous and Automatic Testing of Web Applications. In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, pages 280–285, Cambridge, MA, USA, Nov. 2002.
6. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.

7. P. Koopman and R. Plasmeijer. Testing reactive systems with GAST. In S. Gilmore, editor, *Trends in Functional Programming 4*, pages 111–129, 2004.
8. P. Koopman and R. Plasmeijer. Generic Generation of Elements of Types. In *Sixth Symposium on Trends in Functional Programming (TFP2005)*, Tallin, Estonia, Sep 23-24 2005.
9. D. Kung, C. Liu, and P. Hsia. An Object-Oriented Web Test Model for Testing Web Applications. In *IEEE Proceedings of the 24th Annual International Computer Software & Applications Conference (COMPSAC'00)*, pages 537–542, Taipei, Taiwan, Oct. 2000.
10. K. Larsen, M. Mikucionis, and B. Nielsen. Online Testing of Real-Time Systems Using UPPAAL. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004 - Revised Selected Papers*, volume 3395 of *LNCS*, pages 79–94. Springer, September 21 2004.
11. D. Latella and M. Massink. On Testing and Conformance Relations for UML Statechart Diagrams Behaviours. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 144–153, New York, NY, USA, 2002. ACM Press.
12. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. In *Proc. IEEE*, volume 84(8), pages 1090–1126, 1996.
13. S. Lee and J. Offutt. Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis. In *12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, pages 200–209, Hong Kong, November 2001.
14. C. Liu, D. Kung, P. Hsia, and C. Hsu. Object-Based Data Flow Testing of Web Applications. In *Proceedings First Asian Pacific Conference on Quality Software (APAQS 2000)*, pages 7–16, Oct. 2000.
15. T. Margaria, O. Niese, and B. Steffen. Automated Functional Testing of Web-based Applications. In *Proceedings of the 5th International Conference on Software and Internet Quality Week Europe*, pages 157–166, Brussels, March 2002.
16. R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. Technical Report No: TCD-CS-2005-60.
17. 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*, Fuji Susono, Japan, Apr 24-26 2006. Springer Verlag.
18. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.
19. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
20. 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-Verlag, 1999.
21. H. van Beek. *Specification and Analysis of Internet Applications.* PhD thesis, Technical University Eindhoven, The Netherlands, 2005. ISBN 90-386-0564-1.
22. H. van Beek and S. Mauw. Automatic Conformance Testing of Internet Applications. In A. Petrenko and A. Ulrich, editors, *Proceedings Third International Workshop on Formal Approaches to Testing of Software, FATES 2003*, volume 2931 of *LNCS*, pages 205–222, Montreal, Quebec, Canada, October 6 2003. Springer-Verlag.

23. A. van Weelden, M. Oostdijk, L. Frantzen, P. Koopman, and J. Tretmans. On-the-fly formal testing of a smart card applet. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of the 20th IFIP TC11 International Information Security Conference SEC 2005*, pages 564–576, Makuhari Messe, Chiba, Japan, May 2005. Springer. Also available as Technical Report NIII-R0428.
24. Y. Wu and J. Offutt. Modeling and Testing Web-based Applications. GMU ISE Technical ISE-TR-02-08, Information and Software Engineering Department, George Mason University, Fairfax, USA, Nov. 2002.