WebCheck

Oskar Wickström

2020-06-17

Contents

Introduction

This is a design document for my web testing platform, tentatively named *WebCheck*. It describes the motivation and design of a temporal logic used to specify the behavior of web applications. Further, it describes the design of tools used to check that web applications satisfy such specifications.

How It Works

In WebCheck, a tester writes *specifications* for web applications. When *checking* a specification, the following happens:

1. WebCheck navigates to the *origin page* of the web application, and waits for the *ready* condition, that a specified element is present in the DOM.
2. It generates a random sequence of actions to simulate user interaction. Many types of actions can be generated, e.g. clicks, key presses, focus changes, reloads, navigations.

3. Before each new action is picked, the DOM state is checked to find only the actions that are possible to take. For instance, you cannot click buttons that are not visible. From that subset, WebCheck picks the next action to take.
4. After each action has been taken, WebCheck queries and records the state of relevant DOM elements. The sequence of observed states is called a *behavior*.
5. The specification defines a proposition, a logical formula that evaluates to *true* or *false*, which is used to determine if the behavior is *accepted* or *rejected*.
6. When a rejected behavior is found, WebCheck *shrinks* the sequence of actions to the *smallest*, *still failing*, sequence of actions. The tester is presented with a minimal failing test case.

You might say "This is just property-based testing!" and "I can do this with state machine testing!" You'd be right, similar tests could be written using a state machine model, WebDriver, and property-based testing.

With WebCheck, however, you don't have to write a model that fully specifies the behavior of your system! Instead, you describe the most important state transitions and leave the rest unspecified. You can gradually adopt WebCheck and improve your specifications over time.

Furthermore, in problem domains where there's *essential complexity*, models tend to become as complex. It's often hard to find a naive implementation for your model, when your modelling a business system with a myriad of arbitrary rules.

Now, how do you write specifications and propositions? Let's have a look at the *Temporal Logic of Web Applications*.

Temporal Logic of Web Applications

In WebCheck, the behavior of a web application is specified using the Temporal Logic of Web Applications (TLWA). It's a first-order temporal logic, heavily inspired by TLA+ and LTL, most notably adding web-specific operators.

Like in TLA+, specifications in TLWA are based on state machines. A *behavior* is a sequence of states. A *step* is a tuple of two successive states in a behavior. A specification describes valid *behaviors* of a web application in terms of valid states and transitions between states.

In TLWA, a syntactic form is called a *formula*, and every formula evalutes to a *value*. A *proposition* is a boolean formula, evaluating to either true or false. Formulae can also evaluate to text, numbers, sequences, and sets.

Syntax

- From propositional logic we have the atoms and the logical connectives:

  `true` is a proposition that is true.

`false` is a proposition that is false.

`not(p)` negates the proposition `p`.

`p and q` is the logical AND, also known as *logical conjunction*, of `p` and `q`.

`p or q` is the logical OR, also known as *logical disjunction*, of `p` and `q`.

`p => q` is the logical implication, where `p` implies `q`. This is equivalent to `q or not(p)`.

`p == q` is a proposition which is true if the values of `p` and `q` are equal. For propositions it is the same as *logical equivalence*, but in TLWA it is more general, as it compares values of any supported type.

- From first-order logic, we have the universal and existential quantifiers:

  `forall x in S: p` is true if for all `x` in the set `S` the proposition `p` holds (universal quantification).

  `exists x in S: p` is true if for at least one `x` in the set `S` the proposition `p` holds (existential quantification).

- The temporal operators affects the temporal mode of a proposition:

  `next(p)` evalutes to the value of the formula `p` in the *next* state. This operator cannot be nested, e.g. `next(next(...))`.

  `always(p)` is true if the proposition p is true in all future states, including the current state.

- The web-specific operators allows for querying the DOM and the state of DOM elements:

  `query(selector)` is the sequence of all elements in the DOM matching a CSS selector.

  `element.text` is the text content of an element.

  `element.attributes(name)` is the value of an element's attribute named `name`.

  `element.style(name)` is the computed CSS value named `name` of an element.

  `element.visible` is true if element is visible.

- Sets and sequences are defined using literals, and set comprehensions allow filtering sequences and sets into new sets:

  `{x1, x2, ..., xN}` defines a set containing values of the comma-separated forms (`x1`, `x2`, and so on).

  `[x1, x2, ..., xN]` defines a sequence containing values of the comma-separated forms (`x1`, `x2`, and so on).

  `{x in S: y}` defines a set including each `x` in the set `S` where `y` is true.

- Finally, user operators are defined at the top level:

  `o(x1, x2, ..., xN) = p` defines the operator `o`, with arguments `x1`, `x2`, and so on, to equal `p`. An operator with no arguments can be defined without parentheses as `o = p`. Operators cannot be self-recursive or mutually recursive.

Example: Deleting Drafts

In this example, we have a list of drafts that can be deleted. The user selects one or more drafts, by checking the corresponding checkboxes, and clicks ”Delete”. A confirmation dialog is shown, and the user can either cancel or confirm the deletion.

We want to show that when drafts are selected for deletion and the user has clicked ”Delete”, entering the *confirming* state, the deletion is either:

1. *cancelled*, meaning that no drafts are deleted, the same set of drafts are selected, and the confirmation dialog is hidden, or
2. *confirmed*, meaning that the set of selected drafts are deleted from the drafts list and that the confirmation dialog is hidden

These are the only two valid actions when in the *confirming* state.

A Specification using TLWA

The following formula defines the `confirming` state as the existence of an element `e` returned by querying the current DOM for the CSS selector ‘.confirm‘, that is visible and has the text content ”Are you sure?”.

```
confirming =
  exists e in query(".confirm"):
    e.visible and e.text == "Are you sure?"
```

We also need the checkbox elements in the DOM:

```
checkboxes == query(".drafts input[type=checkbox]")
```

And the checked and unchecked subsets of checkboxes:

```
checked = { c in checkboxes : c.property("checked") }

unchecked = { c in checkboxes : not(c.property("checked")) }
```

We can now define the `cancel` action. It says that the set of drafts (or their checkboxes, rather) are the same in the current and next state, that the same checkboxes are checked, and that we’re no longer confirming in the next state.

```
cancel =
    checkboxes == next(checkboxes)
    and checked == next(checked)
    and next(not(confirming))
```

The `confirm` action is the other possibility. It says that the resulting set of checkboxes is equal to the currently non-checked ones, and that we're no longer confirming in the next state.

```
confirm =
    unchecked == next(checkboxes)
    and next(not(confirming))
```

Finally, we can compose our building blocks to define the safety property. At all times (`always`), when we're confirming the deletion of selected drafts, we can either cancel or confirm.

```
always(confirming => cancel or confirm)
```

That's it. We've now specified the safety property of the draft deletion functionality using TLWA.

Reading material

- LTL patterns survey
- Intro to TLA
- Specifiying Concurrent Systems with TLA+