A Model for Browser/Server Interaction

Andi Scharfstein, 2006

Saarland University, 66041 Saarbruecken, Germany

Abstract. Interactions between web servers and clients are still not well-understood, since no formal descriptions exist for them. A paper by Krishnamurti et al. [1], which gives a first attempt at a formal model for these interactions, is presented and discussed.

1 Introduction

"Programming for the Web is essentially a solved problem."

Well, is it? It is true that we have come a long way since the conception of the web. Scripting languages used to construct dynamic web sites have seen a great surge in interest, large stores like Amazon or eBay have been successfully erected, and most users surf these sites without ever encountering problems.
Now, why should the last item be treated like an achievement? Could it be the case that somehow, users are expected to run into problems in a way they never would in a regular store, and that preventing them from doing so is a cause for celebration and admiration of this technical (or sociological) feat? As a matter of fact, it is, and the rest of this paper will deal with an attempt to clarify and remedy this situation – that is, explain what bugs the user is supposed to encounter, why these bugs exist at all, and how to fix them. Let's begin with a typical problem a user might run into!

2 The Orbitz Bug

Suppose our hypothetical user – let's call him Hans – wants to book a flight online. Hans opens orbitz.com, a popular site offering flights from multiple airlines, and chooses his flight destination. Since Hans is well-versed in the use of browsing techniques, he employs tabs (or *multiple windows*) to compare several flights at once, each offering different benefits for different prices. After carefully choosing the one that suits his needs best, Hans closes all windows but this one, concentrating on his selected flight. However, when he goes on to book this flight, he discovers to his surprise and dismay that the confirmation page doesn't display his chosen flight at all! Instead, it tries to sell him another flight – invariably the one from the most recently opened details window, even though it was subsequently closed. Hans discovers that "going back" from a choice is impossible: whenever he opens a window to look at a flight's details, this flight is set as the one to be booked later on, regardless whether the booking process was initiated from this or any other flight's details page. Disgruntled, Hans leaves the site and books his flight at a competitor's web page.

3 Modelling the Web

Fixing any bug necessitates its complete understanding. The Orbitz Bug was first identified by Krishnamurti et al. (see [1], terminology also from that paper) in 2003, but still persists as of this writing, three years later. One could argue that this signifies a lack of proper understanding of the issues involved in building such an application, so in order to gain this understanding, Krishnamurti et al. developed a formal model describing web interactions.
Some reservations have to be made: The model does not deal with multiple clients accessing a single server, nor with any "concurrency" issues (deadlocks on shared resources, etc.). It also neglects static web pages, instead focusing on the (more interesting) case of dynamic sites. However, each of these concerns can be addressed: Multiple clients can be distinguished via sessions, effectively allowing to model them as single entities (which are covered by the model). Concurrency, while a valid research interest in its own right, is orthogonal to the problem at hand and so can be ignored for the moment. As for static pages, they can easily be modelled as special cases of dynamic pages, so this is really not an issue. Starting from a very abstract view and going into details later on (waterfall-style), a *web configuration* W is just a pair consisting of a single web server and a single client: $W = S \times C$. We shall now look at each component of the pair in detail.

3.1 The Server

Obviously, the web server needs some kind of internal storage to hold user data and the like. It will be modelled by a function $\sigma \in \Sigma$, where Σ is the set of all functions with the type $Id \to V_b$. Id and V_b in turn designate identifiers and values, $V_b = \text{Int} \mid \text{String}$. The σ in use can be thought of as the current server state, since it captures all mutable entities accessible by the server.
The other necessary server component is required by what we "normally" think of as the web server: A dispatcher that deals with the process of looking up the pages and delivering them. In particular, since the web pages are dynamic, the dispatcher has to evaluate a looked-up program with respect to a yet to be defined language, and return the results of this computation. So, aside from a lookup table that assigns programs to URLs, some evaluation function is needed. Formally: The lookup table $P = \text{Url} \to M^\circ$ is a function from URLs to valid programs in some language, denoted by M°. A server is a tuple that consists of storage state and lookup table ($S = \Sigma \times P$), and has an associated dispatch operator d_p that will be properly defined later on in terms of the reductions it allows, which define program evaluation.

3.2 The Client

Taking a formal definition for web pages F for granted (it is given in the very next section), the client can be modelled quite easily as a tuple consisting of the currently shown page (as in, displayed on the screen in the browser window) and

a collection of all pages formerly visited during this session: $C = (F \times \overrightarrow{F})$. The latter can be thought of as the browser cache, although strictly speaking this is not the exact truth (as shall be seen later on).

3.3 Web Pages/Forms

Since the only opportunity for true client/server interaction arises when the client sends information to the server (as the other way round is deterministic, if state-dependent), it is sufficient to only consider pages where the client has the opportunity to do so. This is the case with HTML forms, and nothing else.[1] Hence, for our purposes, we identify web pages with HTML forms, and model only the elements needed to describe a form. To this end, we employ a constructor **form** that takes some URL and a collection of key/value pairs, and constructs their respective HTML representation: $F = (\textbf{form}\ \text{Url}\ \overrightarrow{(Id\ V_b)})$. The URL denotes the location where the information from the key/value pairs is sent and is called *submit URL*. The key/value pairs model text fields where user input can occur (in HTML, `<input type="text">`), and their respective content. So, for any pair (k_0, v_0), an HTML tag `<input type="text" name="`k_0`" value="`v_0`">` will be generated and inserted into the according HTML form construct `<form action="Url">`$\ldots$`</form>` at evaluation time. Typically, all values will start out empty and be filled in by the user later on.

4 Web Interactions

Now that we have the necessary definitions down, let's consider all possible actions a user could perform in such a setting. Surprisingly, three distinct rewriting rules suffice to model the whole range of these possibilities. Entering data into a form input field is the first one. The other two options concern changing the page shown in the current browser window: The user may use the browser's back button or switch between tabs to display any previously visited page at any time, or he may load a new page by submitting the form data on the currently active one. We'll discuss each of these options in detail:

4.1 Filling Out Forms

The first rule is called `fill-form`. It is stated as follows:

`fill-form`: $W \to W$
$\langle \text{s}, \langle(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_0)}),\ \overrightarrow{f}\rangle\rangle \hookrightarrow \langle \text{s}, \langle(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_1)}), \{(\textbf{form}\ \text{u}\ \overrightarrow{(k\ v_1)})\} \cup \overrightarrow{f}\rangle\rangle$

In essence, form values can be modified as desired. Since the form is added

[1] With modern web programming techniques such as AJAX, where other interaction paradigms are introduced, this no longer holds true. The discussed paper completely fails to address this issue.

to the "cache" at once after the modification (even before a submit), it is in fact not technically accurate to call it a cache (at least not in the sense used in today's browsers, where only submits indicate cache updates). This doesn't impair the model's functionality for the use cases we are interested in, however, so we'll ignore the issue from now on.

4.2 Switching to Cached Pages

If at first it's not obvious why the model includes the browser cache at all, a look at the side condition to the second rule should clarify this concern: Switching to any page without loading it can only be done if it was previously seen by the client, so naturally the client has to keep track of its visited pages. This is done in the cache. The rule is quite easy to grasp:

`switch`: $W \to W$
$\langle \mathrm{s},\, \langle f_0,\, \overrightarrow{f} \rangle \rangle \hookrightarrow \langle \mathrm{s},\, \langle f_1,\, \overrightarrow{f} \rangle \rangle$, **where** $f_1 \in \overrightarrow{f}$.

It merely states that users may switch to any previously visited page, including the one that is currently shown.[2]

4.3 Submitting Forms

The third and most involved rule captures the notion of submitting form data, including the server's reaction to this.

`submit`: $W \to W$
$\langle \langle \sigma_0, p \rangle,\, \langle f_0,\, \overrightarrow{f} \rangle \rangle \hookrightarrow \langle \langle \sigma_1, p \rangle,\, \langle f_1, \{f_1\} \cup \overrightarrow{f} \rangle \rangle$, **where** $\langle \sigma_1,\ f_1 \rangle \ = \ d_p \langle \sigma_0,\ f_0 \rangle$

The already mentioned lookup function d_p is used to compute a new server state and the next form that will be sent to the browser, depending on the old state and current client form. The server is assigned this new state, and the new form is delivered to the client. On the client side, the currently active page is updated to the new form, which is simultaneously added to the cache. Note that the previous form is already in the cache, because it was either modified (and automatically cached) by `fill-form`, or left unchanged, in which case it was added to the cache during `submit`.

4.4 The Scripting Language

A formal definition of the scripting language is omitted at this point, since it is not particularly enlightening with respect to the problem at hand. Basically, it behaves like the λ-calculus extended with records – see [1] if you're interested in details. However, we will cover its capabilities in a short summary: Besides the

[2] This means that the reduction relation is not terminating. Also note that in `fill-form`, v_0 and v_1 are not required to be distinct.

basics (function application, abstractions, constants, variables), it can handle forms by creating them (with the **form** constructor seen before) and by taking them apart (i.e., getting the value designated by some key). Besides β-reduction, this is the only semantic action defined for the basic language.
The dispatcher d_p works as follows: when a form is submitted, its "successor" is fetched from the form's submit URL. This successor is an abstraction (the only valid program type, M^o) that takes as input the data from the old form, and returns the new one, so all that's left to do for the dispatcher is to apply it accordingly. The new form is then delivered to the client.
The basic language can be extended with the notion of server storage. This is done by adding **read** and **write** directives, which modify the state accordingly, for instance $\langle \sigma,\ E[(\textbf{write}\ Id\ v_b)]\rangle \longrightarrow \langle \sigma[Id \backslash v_b],\ E[v_b]\rangle$, where E is a reduction context, $Id \in dom(\sigma),\ v_b \in V_b$. Note that the storage is server-global.

5 Dissecting the Bug

A careful look at the three rewriting rules should already reveal an interesting fact: only one of them actually modifies the server state, and of the changes that the other two perform, only one can be noticed by the server at some later point in time (i.e., when submitting user data). The problem at the heart of the matter is that the server cannot know if the user has multiple windows opened, since the HTTP protocol is inherently stateless. It doesn't support the "Observer" design pattern [3] (as there is no way to implement a *push* method to get a notification from client to server on a page `switch`), so Krishnamurti et al. call this the *observer problem*. Basically, modern browsers afford the users previously unknown degrees of freedom, while at the same time making web programmers despair of the complexity introduced by not knowing what the user did, or more to the point: where he came from. Certain invariants that programmers implicitly assume while developing the application ("The user will only look at one given flight at a time", "The user only has the opportunity to click on 'Cash cheque' once") no longer have to hold true. A regular store can always rely on the fact that the customer will not buy his products on multiple lanes at once; in an online store, you can't be so sure.
The Orbitz Bug is introduced by a violated assumption, namely that the customer only ever will book the flight he was last interested in (which holds in a sequential model, but not in this setting). It is fixed easily enough by making explicit the distinction between local and global storage (where local refers to the environment defined by a form), and placing the information about which flight should be booked in the environment, that is, the form. The key/value pairs already supported by forms are all that's needed to model local storage: it suffices to make the relevant input fields constant, e.g. by changing the HTML type to `hidden`.

6 Preventing Further Bugs

Now that we have gained a thorough understanding of the issues involved, we have found that fixing the bug isn't really all that hard. Maybe it can even be done automatically?
Using a slightly extended model, the answer is a reasonably qualified yes – while true bug fixes are beyond the scope of any automated computational process, at least warnings can be given when something's gone wrong (in this case, when client and server have run out of sync because of outdated information on the client side). For this to work, first of all the server needs a notion of time. The total number of submits during one session suits this need well. Next, the server needs to know which information can become outdated, so a registry is added to record all fields accessed (i.e., **read** or **written**) during the evaluation of every program. The registry is called that program's *carrier set*. Making use of these facilities, every form is timestamped during its creation, and its carrier set added to its internal storage.
Now, all that's left to do is to keep track of changes to the server state (which only occur during **writes** to some Id). Whenever these changes affect items from any carrier set of an opened form, the form associated with that set is considered outdated, and a warning can be emitted at its submission. Keeping track of **writes** works by modifying the definition of Σ to describe functions of the type $Id \to Time \times V_b$, so every Id now has an associated "last write" timestamp as well as its familiar stored value. Checking works by comparing this timestamp with a submitted form's timestamp for every item of the form's carrier set. If in any of these comparisons the timestamp from the server storage is larger, the submit is potentially outdated and should be treated accordingly.

7 Conclusion

A formal model for a particular kind of web interactions has been presented and discussed, the Orbitz Bug has been explained and fixed, and it has been shown how the type of bugs represented by it can be detected automatically using an updated version of the basic model.
By this, the usefulness of the model as well as the need for further work in this area have been demonstrated. Let's hope that these efforts will enable customers one day to browse **any** online shop with the same ease as a normal store!

References

1. Krishnamurti, S., Findler, R.B., Graunke, P., Felleisen, M.: Modeling Web Interactions and Errors. *Proc. of 12th European Symp. on Programming*, LNCS **2618** (2003) 238–252
2. Licata, D., Krishnamurti, S.: Verifying Interactive Web Programs. *IEEE International Symposium on Automated Software Engineering* (2004)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. *Addison-Wesley* (1994)