Automated Exploration and Analysis of Ajax Web Applications with WebMole

Gabriel Le Breton
Université du Québec à Chicoutimi, Canada
gabriel.le-breton@uqac.ca

Fabien Maronnaud
Université du Québec à Chicoutimi, Canada
fabien.maronnaud@uqac.ca

Sylvain Hallé
Université du Québec à Chicoutimi, Canada
shalle@acm.org

ABSTRACT

WebMole is a browser-based tool that automatically and exhaustively explores all pages inside a web application. Contrarily to classical web crawlers, which only explore pages accessible through regular `<a>` anchors, WebMole can find its way through Ajax applications that use JavaScript-triggered links, and handles state changes that do not involve a page reload. User-defined functions called *oracles* can be used to bound the range of pages explored by WebMole to specific parts of an application, as well as to evaluate Boolean test conditions on all visited pages. Overall, WebMole can prove a more flexible alternative to automated testing suites such as Selenium WebDriver.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*web-based services*

General Terms

Theory, verification

Keywords

Web applications, reverse engineering, navigation

1. CRAWLING WEB 2.0 APPLICATIONS

Modern-day web applications are formed of an intricate network of code-generated pages, whose interactions are often subtle and difficult to debug. As a web browser provides only weak means of enforcing specific navigation sequences, some pages may be accessed through paths unforeseen by the application developer, possibly introducing security threats. Moreover, since the application's display is dynamically generated based on the history of past requests, it becomes hard to verify that some conditions (such as the layout of elements) hold for all reachable pages.

The systematic exploration of a web application for testing purposes can therefore prove a crucial help to application developers. Crawling the application offers the possibility to generate a "navigation map" that can be used to quickly identify and resolve navigation issues, as has been shown in [6]. Moreover, one can take advantage of such an exhaustive exploration to automatically run a variety of tests on the contents of each page as it is visited, thereby sparing the developer from tedious and time-consuming manual verifications.

WWW'13, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

1.1 Related Work

Unfortunately, current testing tools have not kept up with the increasing complexity of web applications, and offer few in the way of systematically exploring its network of pages automatically. Traditional crawlers such as Heritrix or Nutch may mine a web application to fetch its list of pages, but none of them maintains a graph of links usable for debugging navigation problems.

Other tools, such as WAFA [1] and WANDA [2] work on the server side and can be classified as trace mining tools: they record information about page calls made by visitors by instrumenting the application's source code, and process this log *a posteriori*. VeriWeb [3], while not being labelled a "crawler", explores interactive web sites using a special browser that systematically explores all paths up to a specified depth; SiteHopper [5] also takes into account the parameters inside the URLs. None of these tools interact with the client-side GUI to trigger JavaScript events; they instead directly work at the HTTP request level. Therefore, all events are amalgamated into a single page until the next page reload. This makes these tools unable to properly explore and analyze even the simplest Web 2.0 application.

On the contrary, Selenium WebDriver[1] is a collection of language specific bindings to drive a browser, including client-based, GUI-triggered JavaScript code. However, it does not perform an automated exploration of a web application, but merely provides facilities to interact with the browser. Therefore, most solutions based on Selenium consist of elaborate, yet static testing scripts that are simply played back at intervals.

Closer to our needs is a tool called WebMate [4], which, however, has yet to be publicly released at the time of this writing. WebMate claims to systematically explore an application to extract its "usage graph". However, the tool is geared towards the discovery cross-browser incompatibilities, and apparently cannot be used to run arbitrary, user-defined tests as an application is being explored.

1.2 Contributions

To address the issues mentioned above, we developed WebMole, an automated crawler and tester for web applications

[1] `http://seleniumhq.org/`

that implements a number of novel features. First, WebMole handles JavaScript applications seamlessly. Links between pages are not restricted to standard `<a>` anchors; WebMole clicks on all elements of a page to look for a change of state. Moreover, the state of an application is not defined by the URL of the current page, but rather by the shape of its DOM tree; WebMole therefore properly recognizes changes in a page caused by Ajax requests that do not reload a page.

Second, WebMole provides a wider range of exploration and testing capabilities than existing tools. For example, upon visiting each new state of the application, WebMole can be told to evaluate a user-supplied function called a *test oracle*. The test oracle can be made of any valid JavaScript code, refer to the DOM tree of the current page, and even compare contents of the current page with elements recorded from past calls to the function. In addition, the exhaustive exploration of an application can be bounded through a variety of configurable parameters, including a second kind of user-defined function called a *stop oracle*. Finally, WebMole is publicly available right away under an open source license.[2]

To the best of our knowledge, WebMole is the first tool that allows the systematic exploration of Ajax-based web applications, and the automated testing of very expressive, user-configurable conditions. This combination of features makes it unique among current web testing software.

2. A MODEL OF STATE EXPLORATION

A the heart of WebMole is an algorithm for systematically exploring the contents of a web application automatically. Although slightly more complex than a regular graph discovery algorithm, WebMole's exploration strategy makes sure that the whole application is eventually explored, while at the same time never requiring to go back to a previous page; rather, WebMole prefers to reset the application to its initial page and move forward to an unexplored page. This avoids reverting the application to a past state (akin to using the browser's "Back" button), with the undesirable side effects that such a backtracking could produce. This feature is unique to WebMole.

2.1 Web State Machines

At the lowest level of this model are *DOM nodes*, each of which is formally defined as a function $\nu : P \to V$, where P is a set of parameter names and V is a set of values. The possible parameters and values for DOM nodes are those specified in the W3C Document Object Model Level 2. A DOM *tree* is a tree whose nodes are DOM nodes. A DOM *path* is a slash-separated sequence of node names indicating the location of a given node in a tree structure; for example, the path `body[1]/p[3]/i[1]` designates the first `<i>` element within the third `<p>` inside a document's (only) `<body>`.

WebMole's algorithm progressively builds a data structure called a Web State Machine (WSM), which can be seen as a graph connecting all possible states of the application according to links found in a page. Formally, a WSM is a directed graph $G = \langle V, E, \delta, v_0 \rangle$, where its vertices V are DOM trees, its edges E are DOM paths, and the function $\delta : V \times E \to V$ is such that $\delta(v, e) = v'$ if in DOM tree v, clicking on the element in path e results in the DOM tree v'.

[2] `http://github.com/GabLeRoux/WebMole`.

```
function explore(v)
  while some node e has not been clicked in v
    v' := click(v, e)
    if v' = v then continue
    add v' to V (if not present)
    add e to E (if not present)
    add (v, e, v') to δ
    if v' is not completely explored then
      explore(v')
    end if
    exit function
  end while
  mark v as completely explored
end function

function main
  add v0 to V
  while some tree v ∈ V is not completely explored
    goto(v)
    explore(v)
  end while
end function
```

Table 1: The exploration algorithm in WebMole.

The starting point of an exploration is singled out as DOM node v_0.

As one can see, the WSM uses DOM contents to detect changes of state in an application, rather than page URLs. Therefore, state changes that do not result in a new page being loaded by the browser are still regarded as two nodes despite the page URL remaining the same. This is crucial for a proper handling of Ajax applications, and is a departure from the representation used by most web crawlers. In the same way, links between pages are represented by the path to some node in a DOM tree; therefore any piece of a page, including elements with some click handlers registered, can be modelled as a potential link. This again differs from most tools where links must be static `<a>` anchors.

The exploration algorithm, shown in Table 1, describes how WebMole crawls an application from a given start page v_0. In a given page v, the algorithm first finds a DOM node that has not yet been clicked. That element is then "clicked", marked as such, and the resulting page v' is compared to v; if the DOM tree is the same, the algorithm loops to the next unclicked DOM node. On the contrary, if the page contents have changed, the algorithm adds v' to the set of vertices, then adds the path to element e to the set of edges, and the pair (v, e, v') to the transition function δ. If the resulting page has not yet been completely explored, the algorithm then starts over by exploring v'. When no edge remains unclicked in a page, it is marked as completely explored and the process ends.

It shall be noted that the exploration finishes as soon as a page is completely explored; that is, the algorithm, although recursive, always moves forward and never backtracks to the last processed page to resume its exploration from there. To ensure that the application is still exhaustively explored, the algorithm repeatedly calls explore as long as some vertex v has not been completely explored. Each time, the algorithm restarts from the application's initial page v_0, clicks on the proper elements until v is reached, and then calls explore(v).

2.2 Oracles

The unbridled exploration of a web application, although automatic, can amount to fetching a very large amount of

pages. For example, in the case of an online store where pages can be dynamically generated based on inventory, the previous algorithm would ultimately process the entire contents of the store's database. Moreover, the previous algorithm still does not provide facilities for automatically testing conditions on visited pages.

A second unique feature of our tool is that it provides facilities to bound the set of reachable pages and provide diagnostics through a mechanism called *oracles*. Formally, an oracle is any function $\omega : V \rightarrow \{\top, \bot\}$, that takes as input a DOM tree and returns either true ($\top$) or false ($\bot$).

The first type of oracle is the *stop oracle*. When a stop oracle evaluates to true on a given page, the algorithm does not analyze that page and behaves as if it encountered a dead end. The *once oracle* works like the stop oracle, except it allows to evaluate to true exactly once before behaving like the stop oracle. Using stop and once oracles can be used to circumscribe the scope of an exploration.

Oracles can also be used to perform runtime testing of a web application —that is, evaluation of some conditions on-the-fly, as the application is being explored. A *test oracle* is a Boolean condition on the contents of a page, where the value $\bot$ is by convention the indication of an error.

3. DEMONSTRATION

To illustrate how WebMole can be used to automatically explore and test an application, we shall show how to use it on a web site making use of Ajax. WebMole itself is a JavaScript application that runs in any browser; it does not require any specific plugin and is not tied to any rendering engine.

Figure 1 shows the main interface for WebMole. A user summons the main page, and provides the URL of the start page to start the exploration from. To perform an exhaustive exploration of an application, no other information is required from the user.

As the application is being explored, the left pane shows the list of pages accessed by WebMole (in the order they are processed); a different icon distinguishes between pages accessed through static anchors, Ajax calls or internal JavaScript handlers. The right pane is itself an `iframe` showing the current page from the application that is being processed.

When the exploration of the application is automatic (the default), the pages in the right pane scroll in rapid succession. However, the user has the choice of switching to a *manual* mode, where the right pane becomes unlocked and users have the choice of clicking on any elements they wish. In manual mode, WebMole still records the elements clicked, builds its WSM in memory and evaluates all oracles, but the exploration of the pages is driven by the user.

It shall be noted that WebMole interprets neither the HTML contents of pages received from a server, nor the JavaScript code that these pages may include. Rather, WebMole uses the host browser's own engine to render this content in an `iframe`, and directly queries the DOM tree resulting from that process. This both simplifies the implementation of the tool, and makes it see pages exactly as the host browser renders them to a user.

Onece the exploration is over, the WSM can be saved to a file in JSON format for later processing. It has been shown, for example, that model checkers can be run on such a graph to verify complex navigational constraints based on temporal logic, and that the same graph can also be used at

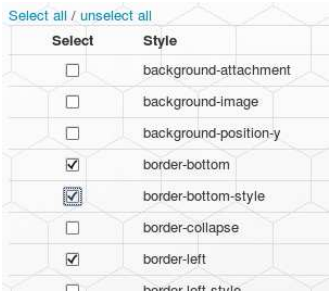

Figure 2: Selecting the DOM attributes to record in the WSM

runtime as a *guard* to prevent a user from veering outside of some predefined navigation graph [6]. However, since a DOM tree is a large data structure, and that a WSM is made of a large number of such trees, one can choose what attributes, among all the available defined in the DOM, should be kept by WebMole. For example, if one only wishes to analyze the positioning of elements inside a page, only the `style.left` and `style.top` attributes of each element can be saved, and all other remaining information be discarded. Figure 2 shows how the user can choose the DOM attributes to be kept in the WSM in WebMole.

Adding oracles to WebMole can be done easily using its user interface. For example, if one does not want to explore pages containing shopping cart information, one can write a stop oracle such as:

```
f = function(doc) {
  if (doc.getElementById("cart") == undefined)
    return false;
  return true;
}
```

This oracle will make WebMole stop exploring some branch as soon as it encounters a page containing a `<div>` with `id` "cart". Similarly, if one wants to explore only one product page (assuming all such pages are similar with respect to some test), one can write a once oracle like the following:

```
f = function(doc) {
  var e = doc.getElementsByTagName("h1")[0];
  if (e.value.indexOf("for product") == -1)
    return true;
  return false;
}
```

This time, WebMole will process (and run tests on) all pages whose title does not contain the substring "for product", and also on the first encountered page containing that substring. The end result is that only one product page will be analyzed, and all the remaining ones skipped.

More importantly, oracles can also be used to perform runtime testing of a web application —that is, evaluation of some conditions on-the-fly, as the application is being explored by WebMole. One particular use of such an oracle is to detect disruptions in the layout of an application. For example, one may make sure that the page's main contents is always the same width by writing the following test oracle:

```
f = function(doc) {
  var e = doc.getElementById("contents");
  if (e.style.width == '960px')
    return true;
  return false;
}
```

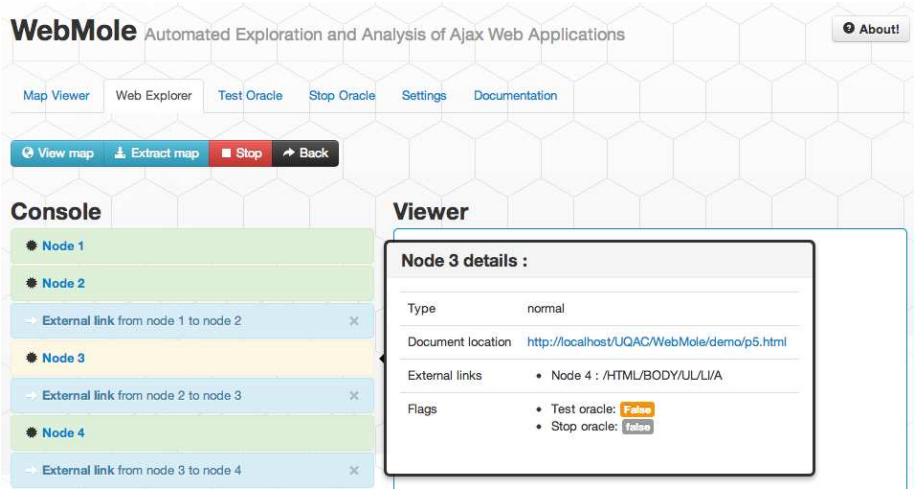

Figure 1: The main interface for WebMole.

Oracles turn out to be a powerful testing feature. Indeed, since an oracle in WebMole is a standard JavaScript function, it can hence contain static variables. This allows an oracle to retain information on past calls and compare it to the state of the current page. For example, the following oracle has a static variable `f.lastx` that can be used to record the x-position of the menu in a page. The oracle returns $\perp$ when the top menu in a page has shifted with respect to its position in the *last* visited page.

```
f = function(doc) {
  var e = doc.getElementById("menu");
  if (f.lastx != undefined && e.style.left != f.lastx) {
    return false;
  }
  f.lastx = e.style.left;
  return true;
}
```

Oracles with static variables can also be used to detect navigation issues within the application. For example, a user should not be allowed to create a shopping cart before logging in first. This can be verified by writing the following oracle:

```
f = function(doc) {
  var e = doc.getElementById("h1");
  if (e.text().indexOf("Welcome,") == -1)
    f.loggedin = true;
  return f.loggedin == true ||
    doc.getElementById("cart") == undefined;
}
```

4. CONCLUSION

Initial testing of WebMole on real-world applications shows that it can prove a powerful tool for the automated crawling and testing of Web 2.0 applications. As the previous examples have shown, complex user-defined oracles can be written to guide the exploration of the tool and perform elaborate testing on pages; the use of static variables even allows the tool to evaluate conditions that correlated many pages along some path.

Following these promising results, we are now integrating into WebMole functionalities that have already been developed in our previous tool called SiteHopper. These include the handling of input forms, parameterized pages. Moreover, additional functions that are currently being implemented include the handling of other user events in addition to mouse clicks and support for jQuery inside the oracles. In time, WebMole can prove a more flexible alternative to automated testing suites such as Selenium WebDriver.

5. ACKNOWLEDGEMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada and by the Fonds de recherche Québec – Nature et technologies.

6. REFERENCES

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean. WAFA: Fine-grained dynamic analysis of web applications. In *WSE*, pages 141–150. IEEE Computer Society, 2009.

[2] G. Antoniol, M. D. Penta, and M. Zazzara. Understanding web applications through dynamic analysis. In *IWPC*, pages 120–131. IEEE Computer Society, 2004.

[3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *World Wide Web Conference Series*, 2002.

[4] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. WebMate: Generating test cases for web 2.0. In D. Winkler, S. Biffl, and J. Bergsmann, editors, *SWQD*, volume 133 of *Lecture Notes in Business Information Processing*, pages 55–69. Springer, 2013.

[5] G. Demarty, F. Maronnaud, G. Le Breton, and S. Hallé. SiteHopper: Abstracting navigation state machines for the efficient verification of web applications. In N. Lohmann and M. H. ter Beek, editors, *WS-FM*, volume 7843 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2013.

[6] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *ASE*, pages 235–244. ACM, 2010.