Script InSight: Using Models to Explore JavaScript Code from the Browser View

Peng Li and Eric Wohlstadter

University of British Columbia
{lipeng, wohlstad}@cs.ubc.ca

Abstract: As Web programming standards and browser infrastructures have matured, the implementation of UIs for many Web sites has seen a parallel increase in complexity. In order to deal with this problem, we are researching ways to bridge the gap between the browser view of a UI and its JavaScript implementation. To achieve this we propose a novel JavaScript reverse-engineering approach and a prototype tool called Script InSight. This approach helps to relate the semantically meaningful elements in the browser to the lower-level JavaScript syntax, by leveraging context available during the script execution. The approach uses run-time tracing to build a dynamic, context-sensitive, control-flow model that provides feedback to developers as a summary of tracing information. To demonstrate the applicability of the approach we present a study of an existing open-source Web 2.0 application called the Java Pet Store and metrics taken from several popular online sites.

Keywords: Reverse-Engineering, Software Maintenance, Rich Internet Applications, JavaScript

1 Introduction

The user interface (UI) is a key aspect of most Web sites. As Web browser programming standards such as JavaScript and the W3C Document Object Model (DOM) have matured, the implementation of UIs for many sites has seen a parallel increase in complexity. These rich Web applications have the advantage of providing a seamless and interactive experience for end-users. However, these applications also require more development effort to build and maintain than older Web UI. As the Web has become more interactive and complex, we are researching a more interactive, model-based approach for Web application reverse-engineering and debugging.

Most existing work on modeling of UI-intensive Web applications focuses on development but not specifically maintenance and debugging. For example, [1] introduces a framework for the integration of presentation components in mashup applications. Trigueros et al. present a model driven approach, the RUX-Model, for the design of rich Internet applications [2]. Valderas et al. introduce an approach to support the coordinated work between Web UI designers and analysts during the development of a Web application [3]. In [15], Rossi et al. use a model-driven approach to transform conventional Web applications into rich Internet applications by applying refactoring at the model level. Meliá et al. propose a model-driven

development methodology which extends a traditional Web modeling methodology for use with the Google Web Toolkit [16]. Some research in software maintenance and reverse-engineering has been used for testing of Ajax applications [14], but not specifically for interactive debugging, as we focus on in this paper.

As with any complex software development task, creating a user interface requires an iterative cycle of design and implementation. Starting with an initial design, an interface would first be prototyped and then refined over several cycles into a final product. At each stage, some design decisions may need to be reconsidered and the implementation adjusted accordingly. The UI might even evolve after the release of an application in order to fix bugs or add new features.

After each cycle, developers can determine the quality of the current application by executing the implementation and evaluating the UI appearance and functionality in a Web browser. If they notice anything wrong with the browser view of the UI, they would need to map the problem back to some part of the implementation, to enact the appropriate change.

Unfortunately, reversing engineering a rich interactive Web page and mapping the appearance or behavior of some element in the Web page to the corresponding implementation can be quite difficult. This is because today's Web UI are stateful and reactive. Their appearance and behavior vary over time based on mutations of state made from JavaScript. This problem is exacerbated by the fact that a developer working on the UI might not have written the original code for all parts of the Web site. In that case, they may need to dig through unfamiliar code to try and reverse-engineer the source. This process is especially difficult since code for some systems on the Web is poorly documented. As described by Hassan et al. [4], "Currently, [code] inquiries can only be answered by scanning the source code for answers using tools such as grep, consulting documentation, or asking senior developers."

In order to deal with this problem, we are researching an interactive approach to bridge the gap between the browser view of a UI and the JavaScript piece of the implementation. This is motivated by the fact that the browser view is usually easy to understand and semantically meaningful, unlike the implementation code. We want to help developers use the live UI as an entry-point into the lower-level implementation details.

To achieve this, we propose a novel JavaScript tracing approach. To a first approximation, when a change is made by a script statement to a visual DOM attribute (e.g. color, height, etc...), we record a link between the effected browser element and the code responsible. The intuition is that a developer can now easily navigate through the code by hyperlinking directly from browser elements. However, a basic implementation of this approach is vulnerable to two problems.

First, mapping semantically meaningful events, such as the mutation of a visual attribute directly to a location in the source code (e.g. a statement) may not be helpful, because that one statement might be reused for several different purposes in the execution of the script. For example, informing a developer that an attribute was changed in a "setter" method for that attribute provides little useful information. The "setter" method could be called many times in the execution of a script, in different contexts, for a variety of different purposes.

For this reason we are researching the use of context-sensitivity to help provide a mapping. A context-sensitive approach captures not only the execution of individual

statements, but also the state of the call stack, which can help distinguish between multiple executions of the same statement.

Second, the visual behavior of a Web page (e.g. the way widgets are animated) is often achieved by a set of coordinated DOM attribute mutations. For example, a button's appearance may change to reflect the button is active when a panel is closed, and change again to reflect it is inactive when the panel is open. The changes to the button appearance and panel appearance have a causal relationship. If a developer wants to change the widget animation they may need to make coordinated changes to several DOM nodes. For this reason we are researching the use of a custom control-flow model, the *DOM mutation graph* (DMG), that developers can use to leverage their understanding of these causal relationships, seen in the browser view, in mapping from the browser view to script source code.

To demonstrate our approach of using this DMG to explore script code, we present a study of an existing open-source Web 2.0 application called the Java Pet Store [9] and metrics taken from several popular online sites. We show how this model is used to understand animation effects in the application which require coordinated changes to several page elements. The metrics taken from other pages provide evidence supporting the need for context-sensitivity in Web application reverse-engineering.

The rest of the paper is organized as following: Section 2 presents a motivating example and an overview of our approach, Section 3 presents technical details, Section 4 presents metrics from online sites, Section 5 presents a further detailed example, and in Section 6 we give related work and we conclude in Section 7.

2 Motivating Example and Approach Overview

In order to motivate our approach, we use a case-study of an existing open-source Web application called Java Petstore 2.0 (henceforth, JPS). This online pet store offers the end-user several interactive widgets to control the application, as shown in Figure 1. Here we see the "Catalog Browser" page from which the end-user can browse prospective pets. This one page alone makes use of 1232 lines of JavaScript code spread across 3 files.

Running down the left-side of the page is an *accordion bar*. This widget is a stylized tree-view for browsing categories of pets and their respective sub-categories (e.g. the specific kinds of cats). The table rows for the categories interactively expand/deflate to reveal/hide sub-categories when the mouse cursor is positioned/removed from a category name. This "accordion" animation requires JavaScript programming to mutate the DOM in an event loop. In Figure 1, the "Cats" row is expanded and the other categories remain deflated.

Consider the perspective of a front-end developer who would like to make changes to this Web page. They have to remember or understand how the 1232 lines of code is mapped to elements of the page and their behavior.

In the original JPS, each *accordion row* is expanded and deflated at a constant speed. Here we consider a change task where a front-end developer wants to change

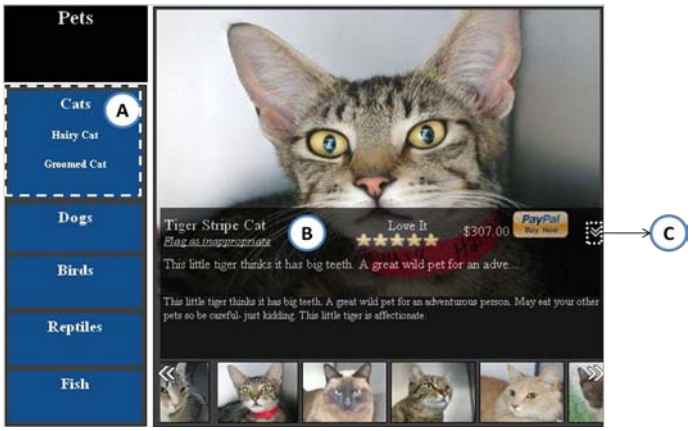

Fig. 1. A snapshot for the "Catalog Browser" from the Java Pet Store. Label (A) is an expanded accordion row, "Cats". The labels (B) and (C) will be described later, in Section 4.

the animation to accelerate at a decreasing/increasing rate when a row is expanding/deflating. During the task the developer is confronted with three problems.

First, they would need to determine which DOM nodes and which attributes of those nodes are responsible for the animation. This could be difficult because the implementation details could vary. For example, the animation might involve any combination of style attributes such as `height`, `top`, `clip`, etc...

Second, suppose a developer figures out that `height` is the key to change the animation. However, when they search through the code, there are two assignment statements to the height of some node in the JavaScript implementation. One of them is shown in Figure 2 and another one turns out not to be relevant. By looking at each statement individually, it is not always clear if the statement is relevant to the task at hand. They may also have to search the code to understand the *calling context* of each height setting statement. In other words, the function calls which lead to the statement's execution.

```
Row.prototype.setHeight = function(nH) {
  this.h = nH;
  this.div.style.height = nH + "px";
}
```

Fig. 2. The function `setHeight` on its own lacks the calling context which is needed to properly associate the function with the accordion bar animation.

Third, suppose the developer determines the function as shown in Figure 2 contains the assignment statement they are interested in. In order to create the new

acceleration/deceleration effect, they would want to change the argument value that was passed to a function call to `setHeight`, but not the definition of the `setHeight` code itself. But now, when a developer searches `setHeight` in the code, they find two places where the `setHeight` function is called, as shown in Figure 3. Each one is relevant for the change task, but for different reasons.

```
147. if(...) {
148.   nHeight = nHeight + INCREMENT;
149.   divs[nExpandIndex].setHeight(nHeight);
150.   if(...) {
151.     if(...)  {
152.       ...
153.     }
154.     else {
155.       oHeight = oHeight - INCREMENT;
156.     }
157.     divs[oExpandIndex].setHeight(oHeight);
158.   }
159. }
```

Fig. 3. Two function calls related to the accordion bar animation. The developer will need information to disambiguate the purpose of each function call. Some code is elided for illustration purposes.

After some investigation, they may find that the first one (line 149) is involved with expanding an accordion row and the second (line 157) is involved with the deflating.

Using our proposed approach, a developer could have chosen to see a model of the accordion row's execution. The model generated by our tool is shown in Fig. 4. In the model, each node represents a statement that mutated some visual DOM attribute and the calling context in which that statement execution. Notice that the model contains two nodes, although we are only concerned with one source code statement (the height setting statement in Fig. 2). From the model a developer could determine that the animation was created by alternating, repeated executions of the context represented by `height0`, followed by repeated executions of the context represented by `height1`.

By selecting each node in our tool, the developer can perform two functions. First, the developer can view a trace of the values which were set in each context. From the trace, it is clear which one is responsible for expanding and which one is responsible for deflating. Having the information in mind, the developer can hyperlink to the corresponding source for the one they are interested in. In the model, `height0` links to the executions of Fig.2, which were made from line 149 in Fig 3; `height1` links to the following repeated executions of Fig. 2, which were made from line 157 in Fig. 3.

Now, the developer can find the correct locations to change argument values for each call to implement the desired acceleration/deceleration change. In the remainder of the paper we describe more precisely the details regarding using a DMG for exploring JavaScript code using Script InSight.

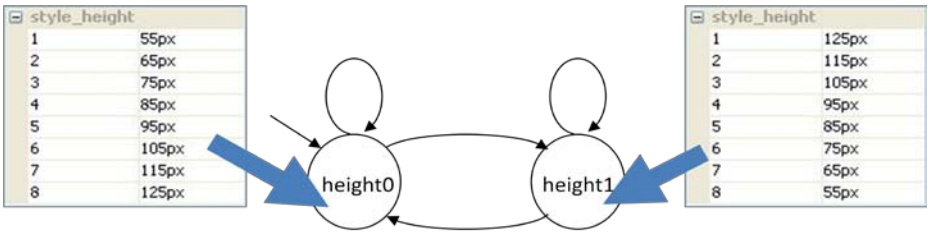

Fig. 4. The abstract behavior of an accordion row presented as a DMG. The two traces of the height values (overlayed on the model with block arrows, in the figure simply for illustration) show the information displayed to a developer when selecting one of the two nodes in the model.

3 Implementation Details

Our prototype is implemented as a JavaScript front-end, to execute within a standard Web browser, and a separate HTTP proxy executable. A developer using our tool will install and point their browser to the HTTP proxy which provides instrumentation of existing JavaScript code. First, we describe our prototype tool from a developer's perspective to provide an overview of the lower-level details involved in our run-time tracing infrastructure, which is described in Section 3.1. The DMG model presentation for UI execution history is presented in Section 3.2.

Using Script InSight, developers can switch the Web browser between normal execution mode and inspection mode. In script inspection mode, a developer can select an element in the browser view. For example, the developer might select a particular image or table row they are interested in. Next, a list of the event handlers that have affected that node during execution are displayed. When the developer selects one of the handlers, a DMG of its previously recorded behavior is displayed.

By selecting a node in the DMG, the developer is hyperlinked to the file for the associated JavaScript statement in a special text editor, as shown in Figure 5. In the editor, the cursor position is set for the line number of the statement for convenience. This text editor includes a drop-down menu for the developer to navigate the calling context for a given statement execution. This allows the developer to jump up and down the call stack that was captured precisely for that instance of statement execution in the trace history.

3.1 Tracing JavaScript Execution

Run-time tracing is implemented as a set of JavaScript functions which are called by tracing code injected into existing scripts. Scripts are intercepted and manipulated by a client-side HTTP proxy. We use the open-source Rhino [10] JavaScript compiler framework to convert scripts into an abstract syntax tree (AST) which is then transformed to add the tracing code. In the remainder of this section we describe the details of this tracing procedure.

During program execution, our tool monitors a subset of the JavaScript statements executed. We refer to these statements as DOM *mutators*. A *DOM mutator* is a JavaScript statement which mutates the state of the DOM. This can be either by directly setting an attribute of a node (e.g. `node.id = 'submit'`) or through any one of the functions in the W3C DOM standard (e.g. `node.appendChild(..)`).

For example, in JPS, the `height` attribute of some nodes is mutated dynamically. Our tool records this fact so that a developer concerned with an animation concerning the height can quickly locate the corresponding implementation.

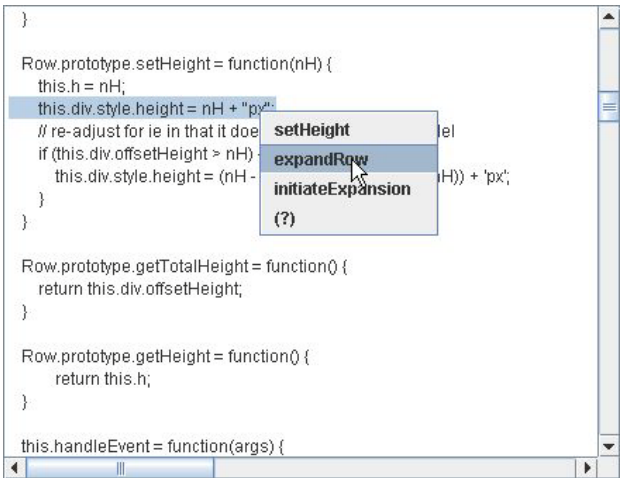

Fig. 5. Selecting a function call location from the calling context. The (?) entry references an anonymous JavaScript event handler function. A mutation of the `style.height` attribute for some DOM node was made in the function `setHeight` which is shown at the top of call stack. This mutation corresponds to the `height0` node from Fig. 4. The stack contents serve to distinguish this execution of `setHeight` from those corresponding to node `height1`.

In many cases, dynamic information is needed to distinguish the calling context in which some statement executed. For this reason, our tool captures the calling context of each DOM mutator execution instance. The *DOM Mutator Context* is an ordered list containing the location of all JavaScript function calls active at the moment of execution for some DOM mutator. This context captures the path of function calls from some event handler invoked by the browser, to the statement.

Consider an example from eBay where JavaScript library code is used to build "widgets". These widgets are an aggregation of DOM nodes which are encapsulated behind a high-level widget interface.

Suppose a developer is interested in a particular instance of an eBay drop-down menu. They might wish to modify the parameters that were used in the construction of the menu. Using our tool they could click on some part of the menu to be hyper-linked to the DOM mutator where that part of the menu was created. However, since these nodes were created as an internal part of the widget library, the developer would not want to actually change the library code but rather find where it was called from for this menu instance. This could be achieved using the captured context modeled in the DMG.

3.2 DOM Mutation Graph

Many Web 2.0 and Ajax style sites use JavaScript to control dynamic UI effects and animations. We want to help developers navigate directly to the code responsible for controlling this part of the UI. In this case, it could be hard for a user to determine precisely the moment when the UI transitioned between states which are responsible for creating the effect or animation.

To help developers review mutations in an animation which occur over the span of some time, we need to consider the history of DOM mutations related to each DOM node attribute. Our tracing infrastructure captures a complete trace of all DOM mutator contexts, including the value (e.g. 10, 'red','http://..') which is assigned by the mutator for each context. However, it is well known that dynamic traces can sometimes overwhelm a user with a large magnitude of data, making the information not valuable.

To abstract large execution traces for developers, we designed a mechanism to represent JavaScript execution as a variation of a traditional control-flow model, the DOM mutation graph. Each DMG is an abstract representation of the execution history for a specific instance of a JavaScript event handler (e.g. `onclick`, `onhover`). This execution history captures all mutations made during the activation of the handler (i.e. while the handler is on the call stack).

We use this partitioning of trace information because each particular event-handler is commonly responsible for creating one particular animation or dynamic effect on the page. Scoping the generation of models to align with event-handlers, allows a developer to focus on a particular animation or effect, and the way it may affect multiple attributes of multiple DOM nodes, in a coordinated fashion.

Our model is similar to traditional control-flow models, such as a control-flow graph or call-graph, in that each node represents some implementation level artifact. However, we consider only the set of statements which affect the visual appearance of the UI and distinguish those statements based on dynamic context information. These statements serve as a bridge between the browser view and the implementation. This is because a developer can plainly observe their effect from the live UI.

In the model, each node corresponds to a mutator context, abstracting over all the particular values which may have been assigned in that context. The trace of concrete mutations, including the attribute values assigned, can be retrieved by interrogating

each node (as illustrated in Fig. 4). Edges in the model correspond to the sequencing of statement execution. A directed edge is created from node, *u*, to node, *v*, if there is a trace entry for *u* followed by a trace entry for *v*. This allows the model[1] to become a bridge between the flow of changes that a developer can see directly in the browser, and the implementation which is causing those changes.

Using the DMG as a bridge is effective because the implementation-level statements which can cause visual changes to the UI in standards-compliant Web applications are limited to a standard set of HTML/CSS attributes and DOM operations. Thus we are able to capture, and focus on, just these attributes and operations. If implementation code was non-standardized or able to directly draw to the browser window using pixel-level operations, such a mapping would be much more difficult or even impossible to create.

4 JavaScript Metrics

In order to better understand if our approach is truly motivated by the complexity of today's JavaScript implementations for several existing Web applications, we gathered metrics from JPS and several popular Web sites. These measurements were taken using Mozilla Firefox 3.0.3 for Microsoft Windows.

Web Page	**# of Files**	**Total Lines**	**Context (see caption)**	**Memory (MB)**
Petstore	3	1,232	2.2 (1.6) /118	36 / 38
eBay	4	19,682	1.5 (.84) /43	40 / 44
Facebook	7	37,310	1.7 (1.3) /485	68 / 72
Yahoo	1	10,218	2.3 (1.4) /164	42 / 43
Amazon	4	5,903	2.0 (.95) /91	45 / 46
Priceline	9	11,667	3.5 (1.8) /73	38 / 40

Table 1: (**# of Files**) lists the number of JavaScript files downloaded for each page and (**Total Lines**) is the sum of their file line counts (in some cases the code is obfuscated so we cannot give an accurate estimate of non-commented lines of code). (**Context**) lists the average number of distinct contexts which a mutator statement was executed in (standard deviation in parentheses) / and the total number of DOM mutator statements executed for the page after the slash. (**Memory**) is the original memory used by Firefox for page execution / with the memory used for our instrumented page after the slash.

Table 1 shows four columns of metrics for each page. The second column, number of files, counts the JavaScript files which were referenced by the page. The total lines, column three, is the sum of the files sizes (in terms of lines) for those files.

[1] To generate the visual appearance of the model, we use a GraphViz-based extension for Firefox.

The column labeled Context describes information about the DOM mutator statements which were executed. The first number lists the average number of distinct calling contexts in which a statement executed. For example, considering the Petstore, each assignment statement to a DOM attribute was executed in 2.2 different contexts on average. The second number shows the standard deviation. The third number lists the total number of DOM mutator statements executed for the page.

The final column lists the memory usage of Firefox with a page loaded, after having its UI exercised; first without our tool in use and second with our tool being used. Memory consumption is discussed further in Section 4.2.

For JPS we use the Catalog Browser which has already been described in detail. The eBay page is a simple list of results for searching auctions related to "iPods". The FaceBook page is the default "Profile" page for a new Facebook user. For Yahoo, Amazon, and Priceline, we used the default homepages.

We took the metrics by triggering a measurement function injected into the code. Since these metrics measure properties of the JavaScript execution, we needed to exercise the UI of the page before taking measurements. We did this by simply manually manipulating any part of the UI which did not cause the page to be changed (hence losing the script state for the page).

4.1 Discussion

By looking at the results for the Context metrics, we see for which pages our calling context capture could be useful. Here we see that these pages either: frequently execute mutators in more than one context and/or execute some mutators in many different contexts.

In general, we see that it was common for a mutator of a DOM node to be used in more than one context. At first this could seem unintuitive because even most interactive Web pages tend to have a large amount of static content. However, this makes sense since we are only including mutations made in the JavaScript code and not any HTML attributes which are set in the static HTML or HTML generated by the server. If some attribute was going to be set only one time and never mutated, it would make sense that the developer chose to generate the value on the server. Thus for JavaScript execution, the reuse of code from different contexts appears to be prominent for these pages.

Developers working on a particular Web page without the help of a model, will need to create a mental map which connects an element of the Web page to a particular location in code. This would currently be done in an ad-hoc fashion. Two possible examples are as follows.

First, a developer could scan the code to identify relevant code. From the # of files and total LOC in Table 1, we believe that this approach is not scalable. There is simply too much code to consider across the files.

Second, a developer could associate an identifier such as a JavaScript function name or file with each element of the Web page. For example, they might use a particular file for all "information pane" functions. In this way, when they want to work on some code related to a particular element, they could use a text-based search to find the relevant code. However this one-to-one mapping does not appear scalable

in light of the Context metrics from Table 1, because a distinct page element may be associated with code reused by several elements or for different purposes. Next, in Section 5 we turn to an example in our JPS case-study to demonstrate how our approach could be leveraged to deal with these problems.

4.2 Performance Considerations

Since our tool collects a history trace of DOM mutations, we wanted to determine how much memory overhead was used for the example Web pages in Table 1. These measurements are listed in last column. Here we see that the amount of memory used was never more than 4MB. Since we only exercised the parts of the UI that were obvious to us, it is possible we had missed some button, menu, or other widget that was not clearly marked. Still, since the amount of memory used was small relative to that in today's desktop machines, we did not consider this to be a large issue.

Certainly the memory used will depend on the code for the page itself. For example, looking back at Fig. 4, we see that the history for expanding and deflating one accordion bar, one time, required 16 trace entries. If JPS was programmed differently, this number could certainly increase but we believe that JPS and the example pages in Table 1 are a fair representation of UI programming practices for many of today's Web applications.

We have used our tool extensively in the exploration of JPS and also as part of collecting the measurements for Table 1. Using the tool we did not notice any perceptible slow down caused by the run-time tracing while interacting with the page.

5 Catalog Browser Example

In order for us to be able to describe some details of our study in depth, we choose to focus on the "Information Pane" (B) and "Collapse Button" (C) on the Catalog Browser page of JPS in Fig. 1.

In this section, we will first introduce the behavior of this information pane and collapse button at a high level. Then, we will give a more low level description from the developer's perspective. Finally, we describe the model that is generated by using our approach to bridge these two different perspectives. A developer can use this model as linked from the browser view, to quickly get into the script programming details.

The information pane (B) describes the detail information for a selected pet (e.g. name, description and rating). This widget is mapped to a `div` element in the DOM. In Fig 1., the information pane appears raised, partially obscuring a pet image. When the pane is lowered, it appears to slide behind the scrollbar (positioned beneath it). This animation is performed by mutating `clip`, `height`, and `top` attributes in coordination.

The collapse button (C) controls the raising and lowering of the information pane. It is an `img` element in the DOM. There are two places in JavaScript which set the `src` attribute. The collapse button's icon is changed to a down arrow when the

information pane becomes fully raised and changed to an up arrow when the information pane becomes fully lowered.

DOM Mutator Context	Trace Values
height0	[75px…177px]
top0	[452px…350px]
clip0	[75px…177px]
src0	up-to-down.gif
height1	[177px…75px]
top1	[350px…452px]
clip1	[177px…75px]
src1	down-to-up.gif

Table. 2. The various contexts in which attributes of the information pane and collapse button are mutated. The trace information of value changes associated which each context are shown in the second column (some are elided for illustration). Note that as is common, the coordinate for `top` is measured as the pixel distance from the top of the screen, hence it is decreasing. The clip value actually includes four coordinates but only one changes in this example so the others are elided.

Table 2 summarizes the three mutated attributes of the information pane and one attribute of the collapse button. Each attribute is mutated in two contexts, which correspond to each of the nodes in Fig. 6. The figure encapsulates changes made to multiple attributes of multiple DOM nodes, to show the flow of execution which was monitored.

From Fig. 6, we can see that the two sets of nodes related to the information pane (at the top and bottom of the figure) are separated by the nodes related to the button icon, which reflects the raising and lowering of the information pane. For each set of information pane attributes, the mutation of the three different attributes, `height`, `top`, `clip` have been executed continuously in an event-loop, shown by the recursive edges out of `clip0` and `clip1`.

By examining the trace of values captured for different DMG nodes we can observe the changes which occur to create the raising and lowering effect. For example, by looking at the entry in Table 2 for `height0`. Here we see the height increases. Without examining the source code, we can already tell that this context is responsible for raising the information pane.

After discerning this information, then by an understanding of the information pane and collapse button behavior from the browser view, and examining the topology of the flow relationships between the DMG nodes, we can plainly determine that `src0` is the context responsible for setting the image of the down arrow; `top0` and `clip0` must then be responsible for part of the information pane raising effect; so then, `height1`, `top1`, and `clip1` must be responsible for the lowering effect; and finally we can surmise that `src1` changes the down arrow to up arrow. Now, we

can link to the code associated with any of the DMG nodes we are interested in for performing any changes during maintenance or debugging.

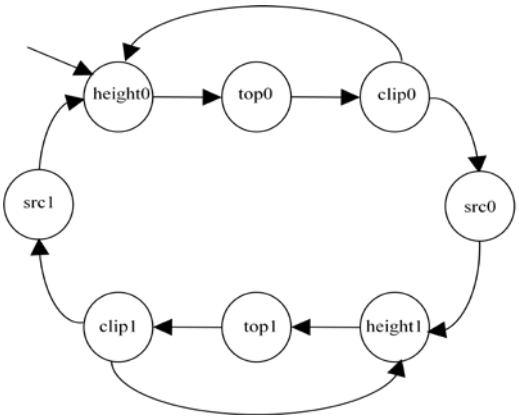

Fig. 6. The flow of the information pane and collapse button presented as a DMG; each node corresponds to the entries from Table 2.

6 Related Work

JavaScript Programming Tools

Due to the popularity of Ajax based applications, there is an increasing demand for JavaScript programming tools. One representative tool for developing Ajax applications is the Firebug [13] extension for the Mozilla Firefox browser. Using Firebug, a developer can simply click on a rendered element in the browser and be hyperlinked to an expanded tree-view of the corresponding DOM element. Now, a developer can inspect the low-level attributes of that specific DOM object and also understand its context relative to its ancestor and children objects.

Although this practice is useful, Firebug still does not provide any help for the developer to understand the connection between a DOM node and the JavaScript which acts on the DOM. Essentially, our research addresses this mapping between the DOM and JavaScript which is not addressed in existing practice.

GUI Maintenance

In [11], McMaster et al. present how to use calling context information collected during a GUI program's execution to solve the GUI test suite reduction problem (i.e. finding a minimal satisfactory test set). Their research considers two GUI test cases to be equivalent if they generate the same set of call stacks after execution. This new call-stack coverage criterion can be used to address the challenges for GUI-intensive applications, which are difficult to be handled by some other criteria such as statement or branch coverage. Similar with their research, we also use calling context to distinguish two artifacts. However, our research is used to resolve the ambiguity of

the different UI changes instead of GUI test cases, for example, accordion row expanding and deflation.

In [17], Michail introduced a tool to provide GUI-guided browsing of source. Their objective was to allow developers to find where in the code a feature was implemented, based on how code was related to the GUI. For example, to find "spell checking" code, they could locate the code which executed when the spell checking menu was selected. Similar to our approach, they use a GUI as an entry-point into the lower-level implementation details. However, they user the GUI to understand its relation to other program features and not the GUI implementation itself.

Model-Based Approaches

Several projects looked into the possibility of recovering a high-level architecture for a Web application from its implementation [4, 12]. In [4], Hassan and Holt describe a set of semi-automated tools that parse the source code and binaries of Web applications and extract relations between the different components to create a model. Their model helps Web developers to understand the high level architecture of traditional HTML and server-side template based Web applications.

Using a finite state machine model to present GUI behavior has been studied in [7]. Their paper describes a Java toolkit called SwingStates which is used to assist in the development of GUIs for non-expert developers. The novel part of their research is that they use finite-state machines to describe the behavior of interactive UI systems. However, their research is concerned about how to create a user interface instead of reversing engineering from an existing UI.

In [8], Shehady and Siewiorek introduced how to use a Variable Finite State Machine (VFSM) interface model to present the behavior of the user interface. Each node in the VFSM is the state of the GUI, and an edge represents the possible events that can be triggered in that state. This model is useful for determining the flow of user-triggered events which change the state of the GUI. In contrast, our model is useful for mapping the live DOM nodes which make up the GUI to implementation-level statements.

Ali et al. introduces a tool called CrawlJax in [14]. Their research uses a dynamic approach to crawl Ajax based applications by triggering the event handlers in the code. After crawling, a state-flow graph is constructed. In this graph, each node represents the snapshot of the DOM tree for a Web UI after some event handler is triggered; each edge in the state-flow graph represents the clickable elements that transform one state to another state. This state-flow graph can be used to provide automated testing of Ajax applications. Similar to the research in the previous paragraph, their research is not concerned with providing a mapping for a programmer to the implementation level details of the UI.

7 Conclusion

In this paper we have studied the problem of JavaScript implementation complexity for interactive Web UI. These details of the UI are easy to understand from the perspective of the Web browser view but can be hard to map to the related code. We proposed an approach which leverages execution history and calling context so that

developers can explore the code from the browser view. The DMG model was introduced to present the obtained history and context information to developers for a better understanding of the behavior of the UI. We presented some script complexity metrics for popular Web sites to further motivate the need for our interactive script development approach. We found that many of the sites that we measured included significant complexity based on the number of calling contexts for a given statement. To demonstrate how the DMG could help, we presented examples from the open-source Java Pet Store Ajax application.

References

1. Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., and Matera, M.: A framework for rapid integration of presentation components. In: Proc. of the International Conference on the World-Wide Web (2007)
2. Trigueros, M.L., Preciado, J.C., Sánchez-Figueroa, F.: A Method for Model Based Design of Rich Internet Application Interactive User Interfaces. In: Proc. of the International Conference on Web Engineering, pp:226-241 (2007)
3. Valderas, P., Pelechano, V., Pastor, O.: Introducing Graphic Designers in a Web Development Process. In: Proc. of International Conference on Advanced Information Systems Engineering,pp: 395-408 (2007)
4. Hassan, A. and Holt, R.: Architecture recovery of web applications. In: Proc. of the International Conference on Software Engineering (2002)
5. Dojo JavaScript Toolkit, http://dojotoolkit.org/
6. jQuery JavaScript Library, http://jquery.com/
7. Appert, C., Beaudouin-Lafon, M.: SwingStates: adding state machines to Java and the Swing toolkit. Softw., Pract. Exper. 38(11): 1149-1182 (2008)
8. Shehady, R.K., Siewiorek, D.P.: A Methodology to Automate User Interface Testing Using Variable Finite State Machines. In: Proc. of the International Symposium on Fault-Tolerant Computing,pp:80-88 (1997)
9. Java Pet Store, Sun Microsystems, http://java.sun.com/developer/releases/petstore/
10. Rhino JavaScript compiler framework. Mozilla, http://www.mozilla.org/rhino/
11. McMaster, S., Memon, A.M.: Call Stack Coverage for GUI Test-Suite Reduction. In: Proc of the International Symposium on Software Reliability Engineering, pp:33-44 (2006)
12. Ricca, F., Tonella, P.: Analysis and Testing of Web Applications. In: Proc. of the International Conference on Software Engineering, pp: 25-34 (2001)
13. FireBug, http://getfirebug.com/.
14. Mesbah, A., Bozdag, E., Deursen, A.V.: Crawling AJAX by Inferring User Interface State Changes. In: Proc. of the International Conference on Web Engineering, pp: 122-134 (2008)
15. Rossi, G., Urbieta, M., Ginzburg, J., Distante, D., Garrido, A.: Refactoring to Rich Internet Applications. A Model-Driven Approach. In: Proc. of the International Conference on Web Engineering, pp: 1-12 (2008)
16. Meliá, S., Gómez, J., Pérez, S., Díaz, O.: A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In: Proc. of the International Conference on Web Engineering, pp: 13-23 (2008)
17. Michail, A. Browsing and searching source code of applications written using a GUI framework. In: Proc. of the International Conference on Software Engineering (2002)