**\chapter{Results and Evaluation}**

**\label{chap:results}**

In this chapter we evaluate FireInsight by applying it to our benchmark web application, JPS2.0. Our primary objective is to determine whether or not FireInsight can effectively model the UI-JavaScript mapping within JPS2.0. Our secondary goal is to investigate the interoperability of our tool with respect to JavaScript frameworks. In particular, we examine FireInsight’s effectiveness at mapping UI behavior to source code, when the application logic is integrated with a JavaScript framework. In the case of JPS2.0, the application logic integrates with the Dojo JavaScript framework.

Instead of using a quantitative measurement approach, we evaluate our tool by presenting multiple case studies. Each case study involves a hypothetical scenario where the developer wants to modify a specific piece of JavaScript behavior in the JPS2.0 user interface. For each scenario we show how our tool might improve developer understanding. We utilize a narrative approach when presenting each case study to illustrate the obstacles that a JavaScript developer would face when attempting to understand a piece of functionality. After presenting those obstacles we show how FireInsight can be used to overcome them with less development effort.

Following the evaluation section we close the chapter by discussing some drawbacks to our approach and specifically some technical issues in our implementation. This will lead into our final chapter where we provide conclusions and future work for our project.

**\section{FireInsight Evaluation}**

**\label{sec:resultsEvaluation}**

To provide a comprehensive evaluation of FireInsight, we present three case studies, each centered on a different piece of JavaScript behavior in the JPS2.0 user interface. For clarity, we denote these units of JavaScript functionality in the user interface as \textit{user interactions}. For each case, we begin the discussion by introducing a real-world scenario requiring the front-end developer to perform a software maintenance task.

We then discuss the program understanding challenges that the developer must overcome to complete the programming task. In Section \ref{sec:methodologyJPS2.0}, we looked at two major challenges. The first was determining which DOM elements are affected by the user interaction and locating the corresponding HTML code. The second was determining the correct JavaScript logic that mutates the DOM elements and understanding how the logic works. We saw that using Firebug’s HTML inspection mode resolves the first obstacle.

However, the second challenge remains unaddressed. Thus, for each case study we focus on illustrating the problem of mapping the DOM elements to the corresponding JavaScript logic and understanding the control-flow for the user interaction. We end each case study by explaining how FireInsight alleviates this challenge by generating the UI-JavaScript mapping for the developer.

**\subsection{Case Study: RSS News Feed}**

**\label{subsec:resultsRSSNewsFeed}**

We begin our evaluation by looking back at the RSS news feed scenario that was introduced in Section \ref{sec:methodologyJPS2.0}. The developer’s objective is again to slow down the refresh rate of the news feed. Having already explored the scenario, we know that one major obstacle to using Firebug is it cannot assist the developer with locating the JavaScript logic driving the user interaction and visually modeling how the code works.

Using FireInsight, the developer will begin by visually inspecting the page in browser view to locate the RSS news feed area. Since our tool integrates with Firebug, our page inspection mechanism is the same as Firebug’s HTML inspection mode. Once the developer has selected the news feed area in the browser view, Firebug will show the corresponding HTML code for the DOM element. At this point the developer can open the FireInsight side-panel from within Firebug. This will show FireInsight’s Attribute view, which lists the attributes of the currently selected DOM element that have a corresponding DOM mutator in the application code.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-1}

\caption[FireInsight Attribute View: RSS News Feed]

{The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area on the JPS2.0 Home page. The FireInsight panel is showing the Attribute view.}

\label{fig:Evaluation-RSS-Example-1}

\end{figure}

Figure \ref{fig:Evaluation-RSS-Example-1} illustrates the current scenario and the interplay between (1) the DOM element selected in browser view, (2) the HTML code in Firebug, and (3) the execution context in FireInsight’s Attribute view. We can see that there is only a single attribute that was found by FireInsight and it is called locator. This is actually not a standard HTML attribute. It is a specific value that is inserted by FireInsight’s analysis code to record when a DOM element is first created. In this scenario, a news headline item corresponds to an \verb!<a>! DOM element, which is first created at line 105 of rssbar.js. More importantly, looking down the call-stack, we can see the calling context for this DOM mutator. We can see the calling context was initiated on line 37 in rssbar.js. It would be useful to view the actual source code.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-2}

\caption[FireInsight Source Code View: RSS News Feed]

{The above screenshot shows an example of the FireInsight Source Code view. This is a pop-up window that appears when the developer clicks on a specific entry in an execution call-stack. The Source Code view displays the source code corresponding to the call-stack entry, with the exact JavaScript statement highlighted and the line number shown at the bottom of the window.}

\label{fig:Evaluation-RSS-Example-2}

\end{figure}

If the developer clicks on one of the entries in the execution call-stack, FireInsight opens the Source Code view that is a pop-up window containing the source code. Figure \ref{fig:Evaluation-RSS-Example-2} shows an example of the Source Code view where the developer has clicked on the call-stack entry for line 105 of source file rssbar.js. FireInsight highlights the exact JavaScript statement and shows the line number at the bottom of the pop-up window. Using this feature, the developer can quickly pin point the code behind a specific DOM mutation within the browser view. Looking at the source code in Figure \ref{fig:Evaluation-RSS-Example-2}, we can see that the \verb!createElement()! function is used to create the \verb!<a>! element, representing the news headline. We can see that the \verb!href! and \verb!target! attributes are set for the \verb!<a>! element. A text node, containing the actual headline text, is also created and appended as a child to the \verb!<a>!. These mutations are not shown in the Attribute view, because our current version of FireInsight does not look for \verb!setAttribute()! and \verb!appendChild()!\footnote{Since we had to modify the Rhino parser source code directly, we found some DOM API methods easier to capture than others. Namely, createElement() was straightforward while appendChild() and setAttribute() were not.}. Therefore, our HTTP proxy does not add analysis code to record those mutations.

If the developer views the source code on line 37 it will become clear that the calling context is part of an initialization procedure called \verb!getRssInJson()!. Looking at the HTML page containing the RSS news feed DOM element (i.e. banner.jsp) we can see a block of JavaScript code containing a call to Dojo to register \verb!getRssInJson()! as an event handler for the on load event. This means \verb!getRssInJson()! will get executed through Dojo when the page first loads. Thus, the developer gains a deeper understanding of the application logic by viewing the execution context.

However, the developer still needs to figure out the control-flow in cases when a user interaction involves multiple DOM mutations. In this example, the initial invocation of the DOM mutator is not actually the calling context that we want, because it is executed once and then never again. There is in fact a second calling context that is executed to handle updating the news feed area once the page has fully loaded. The reason we cannot see it is because it is invoked by a separate event handler.

For this reason FireInsight also models the control-flow for the event handlers attached to the page. Once the developer has inspected a DOM element on the page, FireInsight will locate all of the event handlers that affect the selected DOM element. A listing of the event handlers that affect the currently selected DOM element will appear in FireInsight’s Event Handler view.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-3}

\caption[FireInsight Event Handler View and DMG View: RSS News Feed]

{The above screenshot shows an example of the FireInsight Event Handler and DMG views. (1) The Event Handler view shows a listing of event handler functions that affect the currently selected DOM element. (2) Clicking on one of the event handlers will take the developer to the DMG view, which will display a graphical model containing all DOM mutators that are invoked by the event handler.}

\label{fig:Evaluation-RSS-Example-3}

\end{figure}

Figure \ref{fig:Evaluation-RSS-Example-3} illustrates what the developer would see in FireInsight’s Event Handler view for the RSS news feed example. In this case, there is only one event handler, an anonymous function declared on line 36 of rssbar.js. If the developer clicks on this event handler, FireInsight will switch from Event Handler view to DMG view, as shown in Figure \ref{fig:Evaluation-RSS-Example-3}. In the DMG view, we can see a graphical representation of the control-flow for this event handler. Note that there is always an empty node in the graph, because it is used to indicate the starting point of the control flow. In this example, there are two distinct DOM mutator nodes, both of which create \verb!<a>! elements.

The developer can right-click on a node and see its execution context, just like in the Attribute view. The developer can then click on an entry in the call-stack, and FireInsight will show the source code, again just like in the Attribute view. In this example, if the developer explores the execution context for both nodes in the DMG, it will become apparent that only one of them affects the headline section of the RSS news feed area. Figure \ref{fig:Evaluation-RSS-Example-4} shows the two sections of the RSS news feed area and how they are related to the DOM mutators represented in the DMG. Specifically, we are interested only in the ``rss-item’‘ section, which contains the news headline, and not the ``rss-channel’‘ section.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-4}

\caption[Browser View and FireInsight DMG View: RSS News Feed]

{The above screenshot illustrates how the DMG for the news feed relates to the sections on the page.}

\label{fig:Evaluation-RSS-Example-4}

\end{figure}

The first node in the DMG mutates the ``rss-channel’‘ section, and the second node affects the ``rss-item’‘ section. However, when the developer jumps into the JavaScript code to examine the DOM mutator for the ``rss-channel’‘ section it will become clear that it is not the section of code that we want. This is because the code does not contain any logic to periodically refresh the news headline. We know from observing the browser view of the page that the news headline is supposed to get updated. This means that neither DOM mutator node is actually what we are looking for. At first glance, it may appear that FireInsight missed a DOM mutator during the instrumentation of the application code, when in fact it did not.

The second DOM mutator for the ``rss-item’‘ section belongs to a separate event handler. That is, the second execution context is part of another anonymous event handler. If the developer waits for the news feed area to refresh in real-time on the page, and then inspects the same area again, this time, FireInsight will show a different call-stack for the locater. Recall that DOM mutators are recorded in real-time as JavaScript behavior is executing on the page. The RSS news feed cycles through a set of headlines without requiring user input. This is a subtle difference compared to more common user interactions that execute based on user events. As the news feed automatically refreshes. FireInsight records the corresponding DOM mutations. After the initial DOM mutation, all subsequent DOM mutations of the ``rss-item’‘ area will be grouped under the second event handler.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-5}

\caption[FireInsight Attribute View: RSS News Feed Part 2]

{The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area for a second time. The Attribute view shows a different execution context this time.}

\label{fig:Evaluation-RSS-Example-5}

\end{figure}

Figure \ref{fig:Evaluation-RSS-Example-5} shows the call-stack for the locator value after the developer has inspected the ``rss-item’‘ section again. This time clicking the bottom level entry in the call-stack shows the second anonymous event handler. Viewing the source code for the event handler on line 137 of rssbar.js shows the default refresh rate is 500. From here the developer can increase the value to slow down the refresh rate.

Similarly, the DMG has changed as well, as shown in Figure \ref{fig:Evaluation-RSS-Example-6}. The graphical model now contains a single node, instead of two as we saw in Figure \ref{fig:Evaluation-RSS-Example-4}. The single node has a self-directed edge, indicating that the control-flow entails invoking the same DOM mutator repeatedly, without any transitions to other DOM mutators. This DMG is consistent with the conceptual model we presented for the RSS news feed in Section \ref{sec:methodologyJPS2.0}, namely Figure \ref{fig:RSS-Example-4}.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-RSS-Example-6}

\caption[FireInsight DMG View: RSS News Feed Part 2]

{The above screenshot shows the FireInsight inspection mode being applied to the RSS news feed area for a second time. The DMG view shows a different event handler this time.}

\label{fig:Evaluation-RSS-Example-6}

\end{figure}

**\subsection{Case Study: Catalog Browser Info Pane}**

**\label{subsec:resultsCatalogBrowserInfoPane}**

Our next case study involves a complex user interaction on the JPS2.0 Catalog Browser page. This page allows the user to peruse pet listings stored in the JPS2.0 database. On the left-hand side, the navigation menu displays the categories of animal types (e.g. cats) and sub-types (e.g. hairy cats). On the right-hand side, the main panel allows the user to explore pet listings within the current category sub-type. A gallery of thumbnail images located at the bottom of the main panel allows the user to scroll through pet listings. The user can view more details for each pet by clicking the thumbnail image. This action loads the selected pet’s information into the main panel area, directly above the image gallery. Two items are loaded into the main panel area: (1) a large snapshot image of the pet, (2) summary information about the pet. The summary information is loaded into an Info Pane area, which is located above the image gallery but in front of the snapshot image.

\begin{figure}[tbp!]

\centering

\includegraphics[height=1\columnwidth]{Evaluation-Catalog-Info-Pane-Example-1}

\caption[Case Study: Catalog Browser Info Pane]

{The above screenshot shows the JPS2.0 Catalog Browser page. In particular it shows the two states of the Info Pane area: (1) minimized, and (2) maximized.}

\label{fig:Evaluation-Catalog-Info-Pane-Example-1}

\end{figure}

The Info Pane area can be in a minimized state or a maximized state, as shown in Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-1}. When the Info Pane is minimized, it is partly hidden behind the image gallery. This puts emphasis on the large snapshot image of the pet. An arrow image on the right-hand side of the Info Pane is pointing upward, indicating that the pane can be expanded. When the Info Pane is maximized it slides up and obscures the large snapshot image. This allows the user to read the summary information and learn more about the selected pet. In this state, the arrow on the right-hand side is pointing downward, indicating that the pane can be closed.

When transitioning between minimized and maximized states, the Info Pane is animated to slide upward and downward. The user can control the state of the Info Pane by clicking on the arrow image. This toggles the Info Pane from minimized to maximized, and vice versa. Thus, the arrow image is actually a button. For this reason, we denote the arrow as the toggle button.

In our scenario, the developer is assigned to modify the user interaction so that the Info Pane expands to cover the entire snapshot image. In other words, we want to increase the height of the Info Pane when it is maximized. Perhaps some of the pet summaries have long descriptions that require the Info Pane area to be larger when it is maximized.

Using Firebug, the developer can inspect the Info Pane to discover that the corresponding DOM element is a \verb!<div>! tag on the page with an id attribute value of ``infopane’’. Although without FireInsight, the developer again faces the challenge of manually mapping the DOM element to the JavaScript source code. Examining all the included JavaScript source files on the page (i.e. catalog.jsp) reveals that there are 1,542 lines of code. This number excludes the Dojo source code. Manually searching the source code for references to ``infopane’’ in order to understand the logic will be very time consuming.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Catalog-Info-Pane-Example-2}

\caption[FireInsight Attribute View: Catalog Browser Info Pane]

{The above screenshot shows the Info Pane area being selected using the FireInsight inspection mode. For the Info Pane, the developer can see the association between: (1) the DOM element in the browser view, (2) the HTML code in the HTML view, (3) the element attributes that have been mutated in the Attribute view.}

\label{fig:Evaluation-Catalog-Info-Pane-Example-2}

\end{figure}

If the developer uses FireInsight’s inspection mode, it becomes immediately apparent which of the attributes for the Info Pane \verb!<div>! are being mutated. Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-2} shows the list of attributes that are affected by DOM mutators. Specifically, \verb!style.width!, \verb!style.height!, \verb!style.top!, \verb!style.left!, and \verb!style.clip! each get mutated as part of the user interaction. We can see in the figure that the call-stack for \verb!style.width! has been expanded. Examining the call-stacks reveals that all of the DOM mutators occur within the function \verb!createInfoPane()!, which is located in scroller.js. However, this function does not appear to be responsible for the toggle animation between the Info Pane’s minimized and maximized state. Similar to the situation we saw with the RSS news feed, there appears to be one event handler responsible for the initialization of the DOM element, and another is responsible for animating it.

Since the Info Pane transitions between two different states the sequence of DOM mutators that get invoked during the user interaction will depend on its current state (i.e. minimized or maximized). Due to the complexity of this behavior, we need to examine the DMG(s) for better program understanding. Since FireInsight records a history of the DOM mutators that have executed so far, the DMG will change in real-time accordingly.

\begin{figure}[tbp!]

\centering

\includegraphics[width=0.8\columnwidth]{Evaluation-Catalog-Info-Pane-Example-3}

\caption[FireInsight Event Handler and DMG Views Part 1: Catalog Browser Info Pane]

{The above diagram shows how FireInsight’s Event Handler and DMG views evolve in real-time as the user interacts with the Info Pane. (1) The initial state of the Info Pane is minimized; the Event Handler view does not show any function related to animating the sliding of the Info Pane and there is no corresponding DMG. (2) The user toggles the Info Pane to its maximized state; the Event Handler now shows the function changeInfoPane(). A corresponding DMG has been generated.}

\label{fig:Evaluation-Catalog-Info-Pane-Example-3}

\end{figure}

\begin{figure}[tbp!]

\centering

\includegraphics[width=0.9\columnwidth]{Evaluation-Catalog-Info-Pane-Example-4}

\caption[FireInsight Event Handler and DMG Views Part 2: Catalog Browser Info Pane]

{The above diagram shows how FireInsight’s Event Handler and DMG views evolve in real-time as the user interacts with the Info Pane. (3) The user toggles the Info Pane back to its minimized state; the corresponding DMG has evolved to show additional DOM mutator nodes. (4) The user toggles the Info Pane back to its maximized state; the corresponding DMG has been fully generated.}

\label{fig:Evaluation-Catalog-Info-Pane-Example-4}

\end{figure}

We illustrate how the DMG for the Info Pane behavior evolves over time in Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-3} and Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-4}. When the Catalog Browser page first loads in the browser, the Info Pane is minimized and no DOM mutators related to the Info Pane behavior have been executed yet (Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-3} – Step 1). Note in the initial state, FireInsight’s Event Handler view does not display the event handler function that we are looking for. We can verify this by exploring the DMG for \verb!initCatalog()! and observing that the code is only responsible for rendering the initial view of the Info Pane. In contrast, we are interested in how the Info Pane is animated from minimized state to maximized state and vice versa.

If we now click on the toggle button in the browser view, the Info Pane will expand and slide upward, causing DOM mutators to execute in order to animate the sliding effect (Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-3} – Step 2). Looking at the Event Handler view, we can see an event handler function called \verb!changeInfoPane()!. If we examine its DMG we will see that it is responsible for animating the maximization of the Info Pane. The DMG shown in Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-3} – Step 2 contains a cycle between the nodes \verb!style.clip!, \verb!style.height! and \verb!style.top!. This corresponds to an execution sequence that mutates the three aforementioned attribute repeatedly in a loop. That is how the JavaScript behavior animates the Info Pane sliding upward. The animation ends with the \verb!title! being set to the string ``Show Less Details’’. The \verb!title! attribute corresponds to text that appears when the user hovers the mouse over the toggle button, which is shown in Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-3} – Step 2.

If we now click the toggle button again, the Info Pane will compress and slide downward. Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-4} – Step 3 shows that the DMG has evolved to contain a new set of DOM mutator nodes. It now has double the number of nodes, excluding the empty start node. The new nodes represent the opposite animation effect, namely to minimize the Info Pane. Notice that this new group of nodes also has a cycle as well, which is the looping of DOM mutators to animate the Info Pane sliding downwards. Looking at the DMG, it is straightforward to see how the control-flow moves from one DOM mutator to the next and understand how this corresponds with the actual user interaction in the browser view. After the second button click, the Info Pane has essentially completed a full cycle of its animation but has not returned back to its initial state yet.

If we click the toggle button for a third time, the event handler will execute the first five mutator nodes in the DMG again, representing an animation of the Info Pane sliding upwards. In order to visit those nodes again we need an additional link in order to transition from the final node in the graph (a DOM mutator to change the \verb!title! attribute) to the beginning of the graph (a DOM mutator to alter the \verb!style.clip! attribute). Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-4} – Step 4 shows this final edge has been added to the DMG after the toggle button is clicked for the third time.

Armed with this new understanding of how the Info Pane user interaction works, our developer can now easily modify the behavior so that the Info Pane covers the entire snapshot image when it is maximized. Specifically, looking at the first half of the DMG for \verb!changeInfoPane()! we can view the source code, by right-clicking any of the graph nodes and then clicking the top level entry in the call-stack. This will show the function that maximizes the Info Pane is called \verb!maxmizeInfoPane()!\footnote{Note that the function name is misspelled. This is unfortunately a mistake within the JSP2.0 codebase.}. The function animates the Info Pane by incrementing the values for \verb!style.clip!, \verb!style.height!, and \verb!style.top!. Making a delayed call to \verb!changeInfoPane()!, which when executed invokes \verb!maxmizeInfoPane()! again. This cycle only stops when the Info Pane has reached a predefined maximum height, as specified by the variable \verb!INFOPANE\_EXPAND\_HEIGHT!. Therefore, the developer simply needs to increase the value of \verb!INFOPANE\_EXPAND\_HEIGHT! to achieve the objective for this scenario. Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-5} shows the final result after the developer has increased the value of \verb!INFOPANE\_EXPAND\_HEIGHT!.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Catalog-Info-Pane-Example-5}

\caption[Case Study Final Result: Catalog Browser Info Pane]

{The above screenshot shows the final result after the developer has finished modifying the maximum allowable height for the Info Pane.}

\label{fig:Evaluation-Catalog-Info-Pane-Example-5}

\end{figure}

It is important to mention that our Info Pane case study represents the same scenario Li and Wohlstadter used for evaluating their Script Insight tool. Comparing our full DMG (Figure \ref{fig:Evaluation-Catalog-Info-Pane-Example-4} – Step 4) with theirs (Figure 6 of \cite{lw09:insight}) we can see that our graphs are not identical. The first difference is the ordering of nodes within the cycle that is responsible for animating the sliding effect. In our DMG the cycle has the order \verb!style.clip! - \verb!style.height! - \verb!style.top!, whereas in Figure 6 of \cite{lw09:insight} the order is \verb!style.height! - \verb!style.top! - \verb!style.clip!. Examining the code for the event handler \verb!changeInfoPane()!, we can see that two auxiliary functions are used to animate the Info Pane. The first is the aforementioned \verb!maxmizeInfoPane()!, which slides the pane upwards. The second is a function called \verb!minimizeInfoPane()!, which slides the pane downwards. The source code for both functions clearly shows that the ordering of the DOM mutator statements is \verb!style.clip! - \verb!style.height! - \verb!style.top!. Therefore, our ordering of the nodes is correct.

The second difference is our DMG includes an additional state which represents the DOM mutator for the \verb!title! attribute. This is missing from the DMG in Figure 6 of \cite{lw09:insight}. Once again examining the code verifies that the \verb!title! is indeed mutated, thus our version of the DMG is correct.

**\subsection{Case Study: Pet Info Overlay Pop-up}**

**\label{subsec:resultsPetInfoOverlayPopup}**

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Overlay-Example-1}

\caption[Case Study: Pet Info Overlay Pop-up]

{The Overlay Pop-up behavior is present on both the Search and Tags pages. The behavior produces a pop-up window over the existing page, which displays additional information regarding a pet listing.}

\label{fig:Evaluation-Overlay-Example-1}

\end{figure}

For our final evaluation we explore a case study involving Overlay Pop-ups, which are used to display additional details for a pet listing. The Overlay Pop-up behavior is found on both the Tags page (tag.jsp) and the Search page (search.jsp), as shown in Figure \ref{fig:Evaluation-Overlay-Example-1}. The Tags page displays a listing of results when the user performs a tag query, which involves selecting a tag value from a set of preexisting tags. For example, clicking on the ``excellent’’ tag will return all pet listings associated with that tag descriptor. Similarly, the Search page displays a listing of results when the user performs a keyword query. In each case, the returned results represent pet listings. Both pages display the name, description, associated tags (if any), and price for each pet listing. The name field contains a URL link which directs the user to the Catalog Browser page and automatically loads the particular pet that the user clicked. All other fields for a result are static text.

The overlay interaction is triggered by a user event. Specifically, when the mouse pointer is moved over the name field for a pet listing an overlay window pops up in front of the search results. The purpose of the pop-up is to display some additional information about the pet, although some fields are repeated. The overlay window displays the pet name, address of the seller, the price, and most importantly an image of the pet. It is important to indicate that the overlay behavior is triggered by a mouse over event not a mouse click event, i.e. the mouse is placed over the name field but does not click the name. If the user clicks the name field, the user will be hyperlinked to the Catalog Browser page and the browser reloads the page.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Overlay-Example-2}

\caption[JavaScript Bug: Pet Info Overlay Pop-up]

{There are two possible types of overlay windows: (A) An overlay with a downward pointing arrow, (B) An overlay with an upward pointing arrow. The JavaScript bug only occurs for type (B).}

\label{fig:Evaluation-Overlay-Example-2}

\end{figure}

In our scenario, the developer is assigned to fix a bug that has been discovered in the existing overlay behavior. The problem is the Overlay Pop-up contains a misplaced arrow image, which overlaps the pop-up and obscures the pet listing details. An additional obstacle in this scenario is the bug occurs only in a specific situation. Figure \ref{fig:Evaluation-Overlay-Example-2} shows there are two types of overlay windows. Type (A) has a downward pointing arrow, while type (B) has an upward pointing arrow. We can see from the figure that it is type (B) that contains the logic bug.

We assume our developer is not familiar with the code base or at least does not have an understanding of the entire overlay behavior. Since the overlay behavior is present on both the Search and Tags pages, we can use either one to reproduce the bug and investigate the issue. The Search page has 866 lines of JavaScript code, while the Tags page has 839 lines of code; neither line count includes the Dojo source code. Since both are relatively similar in terms of code size, we make an arbitrary choice and pick the Search page to use for debugging.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Overlay-Example-3}

\caption[Inspection Mode: Pet Info Overlay Pop-up]

{Since the Overlay Pop-up behavior is triggered by a mouse over event and not a mouse click event, we cannot select the overlay DOM elements using inspection mode in the standard way. Instead of using the standard point and click to select the overlay area, we must use Firebug’s inspection highlighting feature. We can see in the above screenshot the mutated DOM elements are highlighted in yellow.}

\label{fig:Evaluation-Overlay-Example-3}

\end{figure}

Because the overlay behavior occurs only on a mouse over event and not a mouse click event, we cannot use the inspection mode from Firebug/FireInsight to directly select the name field to see the corresponding code. In fact, since the overlay simulates a pop-up window, it has its own DOM elements that exist separately from the name field\footnote{We say that the overlay simulates a pop-up window, because it is not actually a separate browser window. The overlay simply displays a piece of HTML on top of the existing elements on the page.}. Thankfully, there is a solution to this problem using Firebug. As shown in Figure \ref{fig:Evaluation-Overlay-Example-3}, we can use Firebug’s inspection mode highlighting to spot the DOM elements that are related to the overlay behavior. When a DOM element is mutated on the page Firebug automatically highlights the corresponding HTML elements in the HTML code view. This allows our developer to observe which section of the HTML code is mapped to the overlay in the browser view. That said, again there is no mapping back to the JavaScript code.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Overlay-Example-4}

\caption[Event Handlers: Pet Info Overlay Pop-up]

{The Overlay Pop-up behavior has two event handlers. The first is a function (bpui.popup.hide) is registered to handle mouse out events. The second function (bpui.popup.showInternal) is registered to handle mouse over events.}

\label{fig:Evaluation-Overlay-Example-4}

\end{figure}

Using FireInsight, the developer can select the DOM elements corresponding to the overlay and view the generated DMG(s). There are two event handlers of interest, both of which are anonymous functions. The first function occurs on line 7 of source file popup.js and is responsible for hiding the overlay window. The function is assigned to the variable \verb!bpui.popup.hide!. The second function occurs on line 51 of source file popup.js and is responsible for showing the overlay. The function is assigned to \verb!bpui.popup.showInternal!. We can map these two functions back to the HTML code; more specifically to the name field DOM element. Figure \ref{fig:Evaluation-Overlay-Example-4} shows how both of the functions are registered for event handling on the name field element; \verb!bpui.popup.hide! is registered to handle the mouse out event and \verb!bpui.popup.showInternal! is registered to handle the mouse over event.

For this scenario, we know that the bug occurs when the overlay is displayed, not when it is hidden. Consequently, we are interested in \verb!bpui.popup.showInternal! exclusively. But there are two different versions of the Overlay Pop-up and only type (B) exhibits the incorrect behavior, we must ensure that we interact with the type (B) overlay. Otherwise, FireInsight will capture execution context data for the type (A) overlay and generate a DMG model that contains DOM mutator nodes that we are not interested in.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Evaluation-Overlay-Example-5}

\caption[DMG View: Pet Info Overlay Pop-up]

{Since the type (B) overlay is the only one that contains incorrect logic, we are only interested in inspecting that particular behavior. The above diagram shows the generated DMG for the type (B) overlay. The first four DOM mutator nodes have been mapped back to the exact DOM mutator statements in the source file popup.js. For illustration purposes we have marked the statement that caused the bug.}

\label{fig:Evaluation-Overlay-Example-5}

\end{figure}

Figure \ref{fig:Evaluation-Overlay-Example-5} shows the generated DMG for the type (B) overlay behavior. For illustration purposes we have mapped the first four nodes directly to their corresponding mutator statements in the source code (popup.js). The other nodes in the DMG are related to attributes which affect positioning but not visibility. Therefore our developer should be able to deduce that the bug is related to one of the first four nodes in the DMG. Through some testing by making changes to the four statements and seeing the resulting behavior in browser view, our developer should be able to discover which of the four nodes produces the error. The developer should easily discover that setting an object’s \verb!style.display! attribute to ``inline’’ causes it to become visible. Setting it to ``none’’ causes the object to become hidden. We can see from Figure \ref{fig:Evaluation-Overlay-Example-5} that the second DOM mutator is the erroneous statement.

**\section{Challenges and Drawbacks}**

**\label{sec:resultsDrawbacks}**

**\subsection{JavaScript Frameworks}**

**\label{subsec:resultsDrawbacksFrameworks}**

One significant challenge we faced when identifying event handler functions in JPS2.0 was dealing with Dojo function calls. JavaScript frameworks have become common place in web development. In this regard, JPS2.0 is an accurate representation of the complexity found in user interfaces for real-world web applications. The difficulty with modeling control-flow in applications that integrate with JavaScript frameworks is isolating event handlers specific to the application code. Our technique for detecting event handlers involves looking at the bottom of the call-stack. The bottom function represents the initial entry point into the current execution context and is commonly the event handler function we are interested in modeling.

However, this technique breaks down when a framework such as Dojo is added to the application. This is because Dojo provides its own system for registering application specific event handlers to listen for events through the browser. Dojo provides a layer of abstraction between the application’s event handler functions and the browser. Dojo assumes responsibility for wiring the application code together and invoking application logic. Therefore, the bottom of the call-stack is always a Dojo function. This can obscure the event handlers that we are interested in.

\begin{figure}[tbp!]

\centering

\includegraphics[width=1\columnwidth]{Drawback-1}

\caption[Handling JavaScript Frameworks]

{The above call-stack represents an execution context related to animating the info pane for the Catalog Browser. Some of the entries in the call-stack have been omitted for illustration purposes. The call-stack is divided into two portions: (1) Application function calls, and (2) Dojo function calls. The Dojo function calls all involve dojo.js and are located at the bottom of the stack. This indicates that Dojo is responsible for initiating this execution context.}

\label{fig:Drawback-1}

\end{figure}

Figure \ref{fig:Drawback-1} shows an example call-stack where the bottom level function is a Dojo function. In fact, a significant portion of the call-stack involves Dojo function calls (i.e. references to dojo.js). We can see there is a clear separation between the Dojo function calls, and the application specific function calls. At some point in the execution context, control is handed over from Dojo to the application. However, Dojo is responsible for initiating the execution.

To model application specific event handlers, we would need to ignore the Dojo function calls and isolate the application specific portion. Although, to accomplish this we would need a generic algorithm for determining the division between application function calls and framework function calls in the call-stack. We currently do not have a generic strategy. As mentioned in Chapter \ref{chap:implementation}, our best workaround is to ignore the Dojo source code during JavaScript instrumentation. We do this by configuring an ignore list of JavaScript source files to explicitly ignore. The source file dojo.js is currently on that list. The problem with this workaround is that it prevents the developer from seeing the entire call-stack. There may be times when it is important to examine Dojo function calls. By ignoring dojo.js during instrumentation we effectively avoid analyzing the Dojo source code.

**\subsection{JavaScript Instrumentation}**

**\label{sec:resultsDrawbacksInstrumentation}**

There are a number of DOM mutator types which are unaccounted for in our current implementation. Unfortunately, our implementation for JavaScript instrumentation required modifying the Rhino JavaScript parser directly. This technique was error prone and time consuming. As a result we were unable to get the Rhino parser to capture all the DOM API methods that we had hoped to cover.

Besides \verb!createElement()! there are many additional API methods that mutate the DOM, such as \verb!appendChild()! and \verb!setAttribute()!. While the function \verb!createElement()! is attached to the global \verb!document! object and therefore easier to replace with our own method, both \verb!appendChild()! and \verb!setAttribute()! are invoked at the individual DOM-element level. Consequently, finding a generic method to capture all calls to \verb!appendChild()! And \verb!setAttribute()! proved to be quite difficult.

Modifying our JavaScript parser to take these additional implementation details into consideration will increase the number of DOM mutators that FireInsight discovers during JavaScript execution. An improved detection rate for DOM mutators will increase the accuracy of FireInsight’s DMGs. It would be very interesting to see how many additional DOM mutator nodes are added to the DMGs as a result of capturing more DOM mutator types. If there are too many additional nodes in the graph, then we also lose some clarity because of the increased complexity in understanding the control-flow represented by the DMG.