Using Indexed Sequence Diagrams to Recover the Behaviour of AJAX Applications

Shane McIntosh, Bram Adams, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{*mcintosh, bram, ahmed*}*@cs.queensu.ca*

Ying Zou
Dept. of Electrical and Computer Engineering (ECE)
Queen's University, Canada
ying.zou@queensu.ca

***Abstract*—AJAX is an asynchronous client-side technology that enables feature-rich, interactive Web 2.0 applications. AJAX applications and technologies are very complex compared to classic web applications, having to cope with asynchronous communication over (unstable) network connections. Yet, AJAX developers still rely on the ad hoc development processes and techniques of the early '00s. To determine how the inherent complexity of AJAX impacts the design and maintenance of AJAX applications, this paper studies the amount of code reuse across the different features of an AJAX application. Furthermore, we analyze how the design of existing AJAX systems deal with AJAX-specific crosscutting concerns, such as handling the loss of network connectivity. We use dynamic analysis to recover the run-time behaviour of AJAX applications in the form of sequence diagrams that are indexed by the different asynchronous communication states that the application can be in. Exploratory case studies on three AJAX applications show that (1) a majority (60–90%) of the run-time behaviour is shared, theoretically simplifying maintenance, and (2) that the studied projects seem unprepared for loss of network connectivity, often presenting the user with an incorrect view of the application state.**

I. Introduction

Web applications are ubiquitous on the world wide web, driving many popular web sites, such as Amazon, GMail, and eBay. "Web 2.0" [1] is a blanket term used to describe the revolutionary change over the last seven years from static and document-centric web applications to dynamic and user-centric ones. Web 2.0 enables the design of highly interactive User Interfaces (UIs) for web applications.

At the heart of "Web 2.0" are Asynchronous Javascript and XML (AJAX) [2], a web technology used to decouple the user interface from web server processing. Traditional server communication forces users to wait until a request has been delivered and processed, whereas AJAX requests are passed to the web server asynchronously, without interrupting the user experience.

Despite the explosive growth of AJAX applications, their development is still plagued by the same fundamental problems that hamper classic web application development. Web applications are known for being developed in a rapidly changing environment that does not lend itself to the use of established software engineering practices [3]. Development cycles are usually short with a high employee turnover rate [4]. AJAX applications are suspected to suffer from the same shortcomings [5]. Consider the plight of a novice developer who must maintain an application that is sparsely documented and lacks senior personnel.

In addition, AJAX introduces some important challenges of its own. Developers have difficulty understanding the complex flow of AJAX applications, because of the mixture of web technologies like JavaScript, CSS and XML [5], and the crosscutting nature of asynchronous communication. Recent research recognizes that the current AJAX development tools are not sufficient for maintaining AJAX applications. Kiciman *et al.* [6] develop AjaxScope, a policy-based Javascript instrumentation proxy for monitoring the client-side behaviour of AJAX applications. Schrock discusses the rather limited debugging options available for AJAX applications in production environments [7]. Mesbah *et al.* use CrawlJAX to infer the flow graph of AJAX applications [8] and automate the testing of functionality [9]. Matthijssen *et al.* use FireDetective to visualize AJAX trace data and aid developers in understanding AJAX applications [5].

As AJAX applications slowly replace desktop applications, and more and more users begin to rely on them, the maintenance of AJAX applications becomes critical. Hence, we need to study the quality of the design of current AJAX applications. For example, classic web applications are known to have ad hoc designs, without much component reuse (except for the browser and web server). Although in the AJAX world, there seems to be a movement towards standardization of libraries, it is not clear how AJAX systems are internally structured. Similarly, asynchronous communication plays an important role in the design of AJAX applications. Asynchronous communication is a crosscutting concern, since connections can be initiated from several different locations in an application. Loss of connection is even more critical, since connection failure might impact the whole system, not only the initiator sites. The way in which AJAX applications deal with these crosscutting concerns may have a large impact on future maintenance.

To analyze code reuse and crosscutting in the design of AJAX applications, we present an approach that recovers the run-time behaviour of an AJAX application as a set of indexed sequence diagrams. An indexed sequence diagram for a particular feature annotates each function invocation in the application with the asynchronous communication state

in which the invocation was made. This allows us to analyze the distribution of functionality across communication states, and to compare this distribution between different features or even between different scenarios of the same feature (e.g., with and without server connectivity). We perform exploratory case studies on three AJAX applications to address two research questions:

RQ1) How much behaviour is shared amongst features of AJAX applications?

Motivation – Recovered architecture diagrams have been shown to aid developers in understanding their systems [3]. However, prior work suggests that static analysis of (AJAX) web applications may not suffice due to dynamic code generated at run-time [5, 10]. Hence, we are interested in using dynamic analysis to recover the run-time behaviour of features in AJAX applications.

Results – 60–90% of the functions in the studied projects are shared amongst features. In addition to basic asynchronous communication logic, the studied projects have dedicated framework components for batching of server requests and prevention of request flooding.

RQ2) How do AJAX applications cope with the network connectivity crosscutting concern?

Motivation – The current focus of AJAX research assumes that the connectivity between the client and server is available. As AJAX clients become more capable, handling a (temporary) lack of server connectivity becomes a crosscutting concern.

Results – Without network connectivity, the client component of an AJAX application cannot communicate with the server. In this disconnected state, the client component of all studied AJAX applications became out-of-sync with the server state, and presented users with an incorrect view of the state of the system.

This paper provides the following contributions:

- A semi-automatic approach to recover the framework and feature-specific behaviour of AJAX applications;
- A comparative study of the design of the Oceans, Tudu Lists, and Reddit AJAX applications;
- An analysis of the connectivity crosscutting concern.

The paper is organized as follows. Section II covers key AJAX application concepts. Section III presents our approach for recovering and comparing the behaviour of features in AJAX applications. Section IV presents the setup and results of exploratory case studies on three AJAX systems to address the two research questions. Section V discusses the threats to the validity of our work. Section VI covers related work. Finally, Section VII draws conclusions.

II. Background

This section provides an overview of AJAX technology and terminology.

A. Overview of AJAX Concepts

Figure 1 and 2 compare the flow of events in AJAX and classic web applications. Under the classic web application model, a user points a web browser at a Uniform Resource Locator (URL), which sends an HyperText Transfer Protocol (HTTP) request to a web server. The web server processes the request by executing server-side scripts and potentially contacting back-end databases. The server constructs an HTML response and sends it to the client. The user's browser is then reloaded with the new HTML document. The user may not simultaneously interact with the web page while the request is being sent or processed, since the web page in view will be discarded once the response is received, and subsequent interaction will simply override previous interactions.

The AJAX design reduces idle user time by allowing the user to interact with a dynamic web page while new content is being retrieved and processed. In the AJAX model (Figure 2), the user triggers a Javascript event by interacting with web page elements. The Javascript code sends an asynchronous HTTP request to the web server. The web server processes the request using a procedure similar to the classic server model, with the exception that the response is sent in XML. Once the response is entirely received, the AJAX engine will trigger a Javascript callback function that may dynamically update the browser by manipulating the data structure representing the HTML elements of the current web page (i.e., the DOM). The user may continue to interact with the page in view while the DOM is being updated, since the page will not be discarded.

B. AJAX Application Components

As shown in Figure 2, a typical AJAX application is composed of two client side components [11], i.e., the user interface and AJAX engine. These components communicate asynchronously with the web server component.

The *User Interface* (UI) is made up of components that are laid out in HTML and decorated using CSS. The *AJAX engine* is a Javascript component that handles the user-triggered events and sends asynchronous requests to the web server. The asynchronous request transmission details and callback function are encoded using the XMLHttpRequest object (XHR) specified by the W3C [12]. The XHR has five possible states with the following values:

1) The initial state (UNSENT),
2) The open connection state (OPENED),
3) The state indicating that the server response headers have been received (HEADERS_RECEIVED),
4) The state that indicates that the response body is being retrieved (LOADING), and
5) The state that indicates that the server response has been received (DONE).

The *web server* component consists of server-side request-processing scripts. The server component may communicate

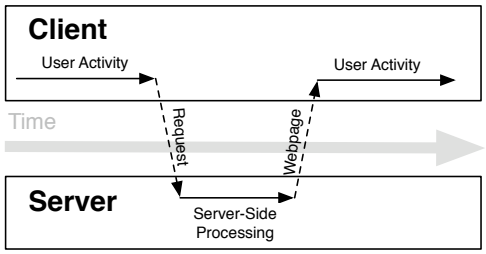


Figure 1: Event Flow in Classic Web Applications [2].

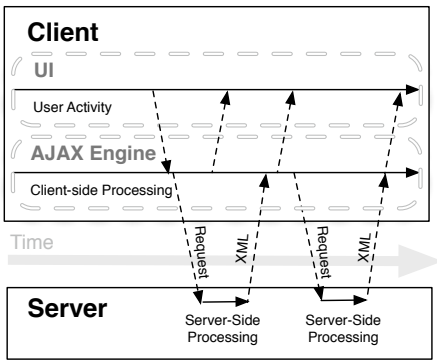


Figure 2: Event Flow in AJAX Web Applications [2].

with back-end databases to fulfil a request and subsequently respond to the client, triggering a callback function in the AJAX engine to update the UI. This paper focuses on the AJAX engine components.

III. Recovering the Behaviour of AJAX Applications

This section presents our approach for recovering the runtime behaviour of AJAX applications. Our approach recovers execution traces for a set of features in an AJAX application, then abstracts these traces into indexed sequence diagrams that have been annotated with the XHR states active during each function call. By comparing the diagrams of different features per XHR state, shared logic (i.e., framework logic) can be distinguished from feature-specific logic. By comparing the diagrams of one feature under different conditions (such as with and without network connectivity) per XHR state, the impact of these conditions on the application's design can be identified.

Figure 3 shows an overview of our approach, which is composed of six subtasks: (1) Instrument the web browser, (2) Drive the AJAX application through the user actions that compose a feature, (3) Semi-automatically filter the trace data, (4) Group function calls per XHR state, (5) Compare the recovered feature traces to separate framework and feature-specific functionality, and (6) Visualize the results.

In the following subsections, we elaborate on the above subtasks in detail.

A. Browser Instrumentation

In order to produce a trace of Javascript function calls, we equipped the Mozilla Firefox web browser with DTrace functionality and developed a DTrace instrumentation script.

DTrace [13] is a dynamic tracing framework developed by Oracle. It promotes "software observability" [14] by allowing DTrace developers to bind program logic to locations of interest ("probes") embedded in a DTrace-compliant program. The Mozilla suite (Firefox included) is DTrace-compliant, although the DTrace probes are turned off in public releases. We built Firefox 3.6 with the appropriate feature flag to enable DTrace (--enable-dtrace), and wrote a DTrace instrumentation script to track Javascript function entry and exit probes inside Firefox. Our DTrace script prints the file name, function name, and the line number for each function entry and exit. As Javascript functions may be unnamed or anonymous, we use the file name and line number to assign an identifier to each function call. Similar to Matthijssen *et al.* [5], we use the variable identifiers that unnamed functions are assigned to to uniquely identify them in our trace data.

B. Exercising a Feature

We drive through the set of user actions comprising a selected feature using our DTrace-enabled browser. Figure 4 shows an example of such a feature. After navigating to the Google web page, the user will: (1) change focus to the search text field by clicking on it or tabbing through page elements; then (2) begin typing "toronto blue jays". As a result, the user sees a drop down menu with similar Google queries (left column) and the approximate number of results they yield (right column).

We produce a trace log using our DTrace script while exercising a feature. This log contains all of the Javascript function calls listed in the order of execution. This process is repeated for each feature in the AJAX application.

C. Semi-automatic Trace Filtering

The raw trace data gathered in the previous subtask is too noisy to derive behavioural information and must be processed to extract meaningful Javascript interactions.

Similar to Di Lucca *et al.* [15, 16], our data filtering process is semi-automatic. The process is divided into two passes. During the first pass, internal browser function calls are filtered automatically. For example, code responsible for updating web page history is discarded. After this first pass, only the application code and third-party library function calls remain. The second filtering pass is initially manual, and removes the function calls within third-party library functions, since an application developer typically is not concerned with the functions supporting a library function call. If the library calls are known in advance, the second

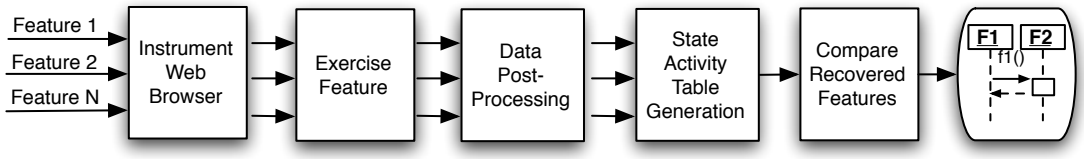


Figure 3: An overview of our approach for identifying features in AJAX applications.

filtering pass can be automated as well. After completing the two filtering phases, what remains is the trace of the executed application code and third-party API function calls.

D. State Activity Table Generation

Since each AJAX request cycles through five XHR states and each feature centers around (an) AJAX request(s), we group the filtered function trace by the state of the relevant XHR object at the time of invocation. This allows us to record which functions are executed in the context of each XHR state. Each feature can be represented by a *state activity table*, whose columns contain a list of function names associated with a particular state.

E. Comparing the Recovered Features

Each of the preceding steps are repeated for each application feature, producing a set of feature-specific state activity tables. We derive the common framework behaviour of an AJAX system (shared by all its features) by calculating the intersection of the set of function calls in all feature behaviour sets. The calls in the intersection are considered to be framework behaviour, the other function calls are considered to be feature-specific behaviour. The output of this phase is a state activity table for the framework behaviour as well as a set of feature-specific tables.

F. Visualization

We visualize the framework and feature behaviour of each AJAX feature as a sequence diagram annotated with XHR state changes. We call these diagrams indexed sequence diagrams, since the limited set of states provide an index into the potentially large sequence diagrams. In order to produce these diagrams, the filtered trace data must be translated into the right format. In our study, we use the TraceModeler tool [17] for sequence diagram visualization. TraceModeler uses its own text-based file format (.tmt) that we convert our data into. In the generated sequence diagrams, Javascript files are represented as sequence diagram *lifelines* and the function calls are represented as *messages* passed between the files. States are delimited by means of horizontal lines across the diagrams, and labeled with their name.

IV. Exploratory Case Studies

We present the results of our exploratory case studies in this section.

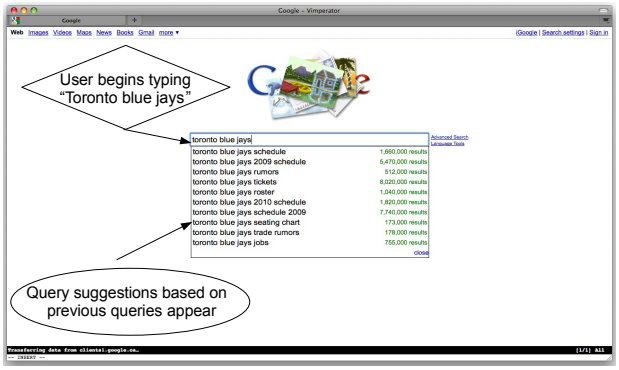


Figure 4: An example of AJAX functionality (Google Similar).

Studied Projects

To address our two research questions, we selected three AJAX applications of different sizes and domain. Oceans is an example of a use of AJAX-for-website-decoration, Tudu Lists is a "fat" AJAX application (i.e., an application with significant client side functionality), and Reddit is a collaborative crowd-sourcing tool.

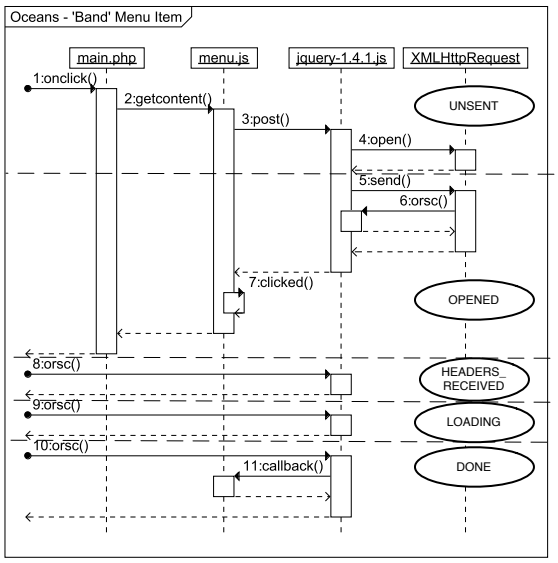
Table I shows the characteristics of each studied project. The LOC counts are raw line counts of the Javascript files in the projects. An asterisk is placed next to Tudu Lists, since the Javascript files are generated from Java sources. In this case, we counted the lines in the Java files using the SLOCCount utility [18].

Table I also shows the characteristics of the feature traces explored in each research question. Each feature trace was repeated three times to ensure the validity of the trace log. Each line in the trace was categorized as either part of Firefox (FF), a third-party library (Lib), or application code (App). We use the Signal to Noise Ratio to measure how much of the traced code is App code (SNR $= \frac{\text{App}}{\text{Total}}$). Overall, the measure is quite low, indicating that the raw data is very noisy.

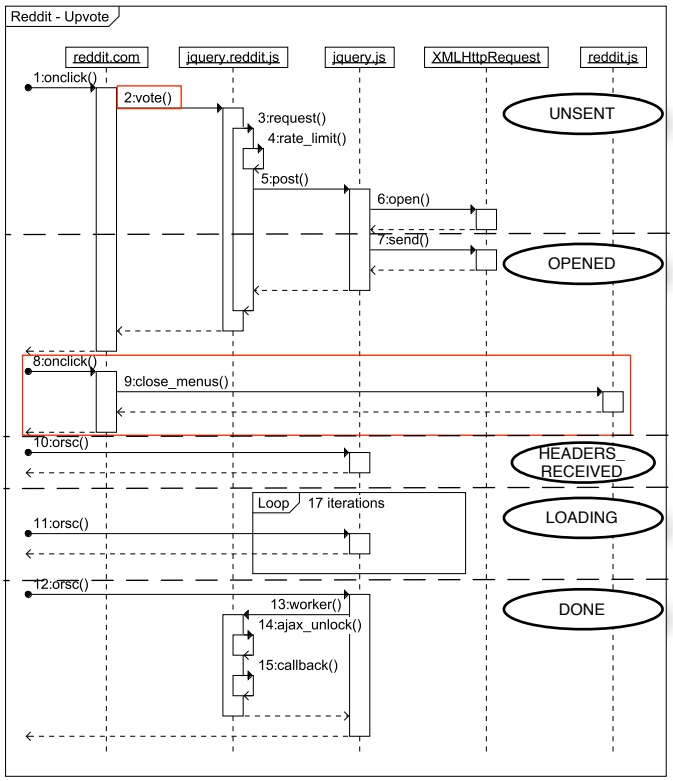
In our exploratory case studies, we attached meaning to the functions in the recovered feature behaviour by manually inspecting the executed application code.

RQ1) How much behaviour is shared amongst features of AJAX applications?

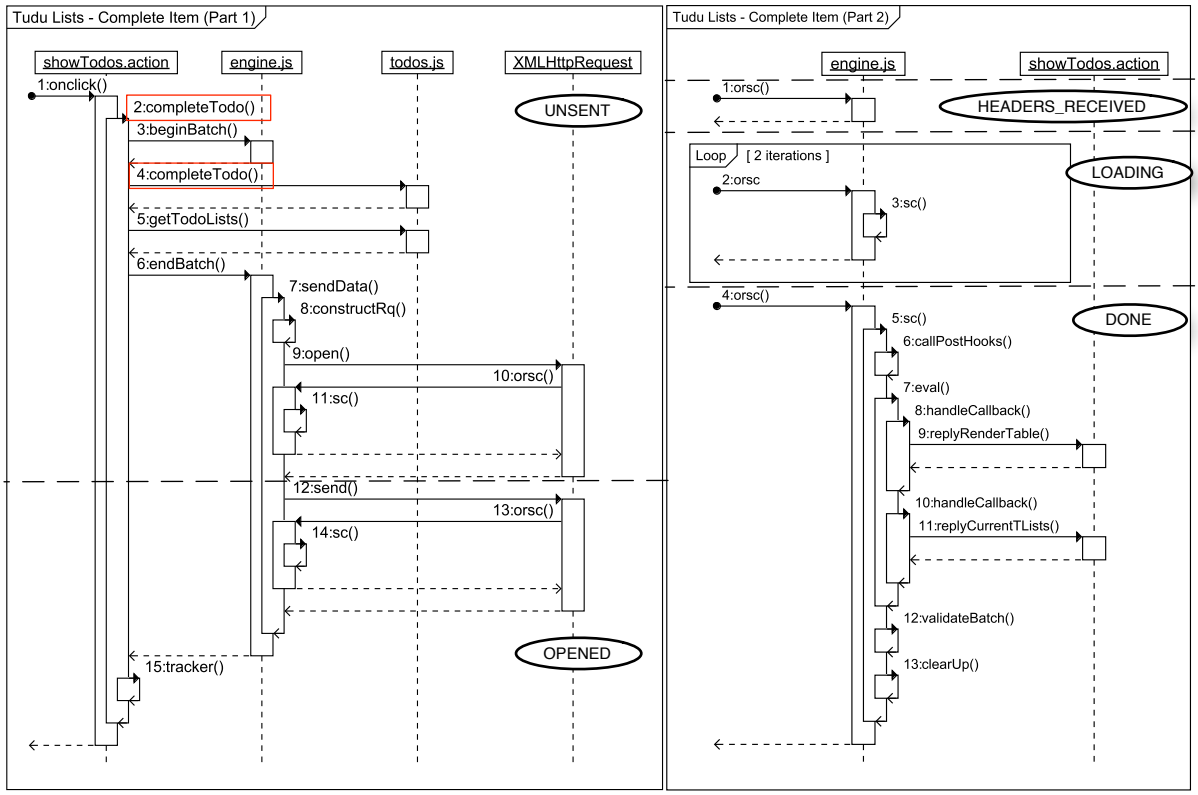
We recover and discuss the framework- and feature-specific behaviour in each AJAX application's design to



(a) Oceans



(b) Reddit



(c) Tudu Lists

Figure 5: Recovered framework and feature-specific behaviour.

Table I: Characteristics of the Studied Projects.

	Oceans					Tudu Lists					Reddit				
Domain	Menu Bar					Time Management					Social News				
LOC	6,744					9,380*					11,368				
	FF	Lib	App	Total	SNR	FF	Lib	App	Total	SNR	FF	Lib	App	Total	SNR
RQ1 Trace	7,193	584	11	7,788	0.0014	4,039	0	75	4,114	0.018	5,068	1,569	36	6,673	0.0054
RQ2 Trace	2,035	85	11	2,131	0.0052	2,488	0	42	2,530	0.017	860	534	13	1,407	0.0092

address this question. To save space and improve the clarity of our explanation, we do not show framework- and feature-specific sequence diagrams separately, instead we overlay an application's framework sequence diagram with the sequence diagram of a representative feature. In Figure 5, feature-specific calls are shown in boxes, and state changes are denoted with dashed lines and labelled ellipses.

To measure the proportion of framework and feature-specific behaviour in an AJAX application, we divide the number of functions in a particular state of the framework behaviour by the size of the union of all functions executed by any feature for that state, and report this proportion as a percentage. A high framework coverage percentage is desirable, as it indicates that there is high code reuse among features. A low percentage is undesirable as it indicates little code reuse amongst features. Table III reports the framework coverage percentages for each studied project.

Oceans: The Oceans web application only has one AJAX feature. This feature is responsible for menu bar navigation. Using the menu bar, a user selects a menu item to explore. As a sanity test for our approach, we recovered the behaviour for multiple features by clicking different menu items. Note that this does not exercise different features, but rather exercises a different instance of the same feature.

As indicated by row 1 of Table III, there were no differences between the two recovered sequence diagrams. This indicates that the behaviour supporting the retrieval of content for each menu item is implemented using the same functions, executed in the same order. Code inspection guided by the recovered indexed sequence diagram reveals that the content to be retrieved is controlled by a parameter passed to the getcontent() function.

The framework sequence diagram flows from state to state as shown in Figure 5a. We briefly discuss key parts of the flow below. Function calls 2 and 3 process the event and call functions to prepare an XHR that will be sent to the server. Function calls 4 and 5 establish a new connection to the server and send the request. Function call 7 updates the DOM to reflect that a new menu item has been selected, while the server processes the request for the new menu item's content. Functions 8 and 9 respond to XHR state changes that are not configured to be handled by the XHR callback. The final XHR state change at function 10 from LOADING to DONE indicates that the server response has been completely retrieved. Finally, function 11 updates the DOM with the new content.

The Oceans framework behaviour is summarized in Table II. It comprises the following subfeatures: Function calls 1-4 and 5-7 are responsible for preparing and sending a "new content" request to the server, while function calls 8 and 9 prepare for and receive the server response. Function calls 10 and 11 are responsible for updating the user interface with the retrieved content.

Tudu Lists: We examined the task completion, list creation/removal, and task creation/removal features of the Tudu Lists application. We use the "Task completion" feature to illustrate the Tudu Lists framework behaviour. Figure 5c shows the recovered indexed sequence diagram, split into two parts.

The Tudu Lists indexed sequence diagram in Figure 5c is composed of three components: (1) prepare and send the request (Part 1, function calls 1-14); (2) retrieve response packets (Part 2, function calls 1-3); and (3) handle the response (Part 2, functions calls 4-12). Interestingly, part 1 of the Tudu Lists application uses a job batching mechanism. Function call 3 creates a batch job and places a "Loading..." element on the UI as a placeholder until the server responds and the task is complete, function call 4 populates the job with the necessary details (i.e., the task that has to be completed), and function call 6 completes the batch job by sending it to the server by issuing function call 7. The XHR is then constructed with function call 8, the connection is initiated in function calls 9-11, and the request is sent in function calls 12-14. In Figure 5c Part 2, function calls 5-11 update the DOM with the retrieved content. Finally, function call 12 checks that a response has been received for every item in the batch job and function call 13 cleans up the user interface, removing a "Loading..." placeholder that was added during part 1, function call 3.

The majority of the task item completion feature is a part of the framework behaviour. The batch job framework that is used to create, populate and send the XHR, hides the details of the request being sent behind a layer of abstraction.

Conversely, the feature-specific behaviour consists of only two function calls. Figure 5c Part 1 shows the feature-specific functions 2 and 4, which occur in XHR state UNSENT and refer to feature-specific functions such as addTodo() for the task creation feature or removeTodoList() for the list removal feature. Part 2 only differs in the number of loop iterations required to retrieve the server response.

As a result, the Tudu Lists application has framework behaviour coverage values $\geq 80.5\%$ (Table III). This high level of behavioural reuse amongst features may make Tudu Lists easier to maintain than a system with little reuse.

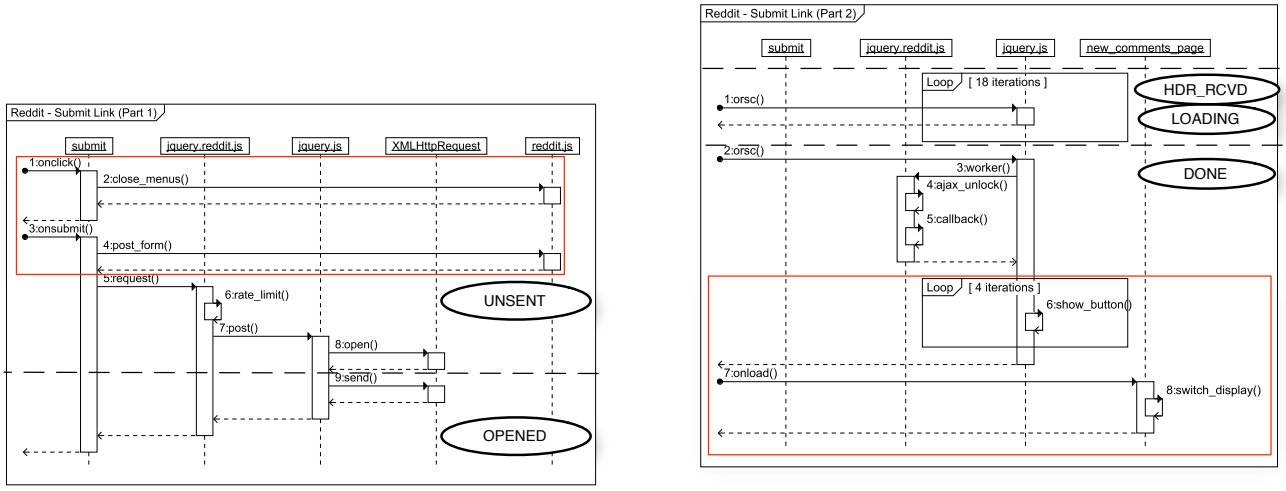


Figure 6: Reddit 'Submit Link' Feature

Table II: Framework State Activity Table - Oceans.

UNSENT	OPENED	HDR_RECV	LOADING	DONE
1. onclick	5. send	8. orsc	9. orsc	10. orsc
2. getcontent	6. orsc			11. callback
3. post	7. clicked			
4. open				

Table III: Framework Coverage of Three AJAX Systems.

	UNSENT	OPENED	HDR_RECV	LOADING	DONE
O	100%	100%	100%	100%	100%
T	80.5%	100%	100%	100%	100%
R	50.0%	58.3%	100%	100%	62.5%

Reddit: Reddit is a social news website where users may share links to content in designated discussion areas or "reddits" about politics, world news, science, programming, and many more topics. Users share their thoughts on the content in comment areas that are created for each link. Users also influence the ranking of links on the Reddit web page and of comments in designated comment areas by voting them up or down.

We extract the Reddit framework behaviour by comparing the behaviour of the "upvote", "downvote", link submission, and withdrawal features. Table III shows that the framework behaviour has lower coverage than that of Oceans or Tudu Lists. We use Figures 5b and 6 to show the key areas of differing functionality in the behaviour of Reddit features.

The recovered upvote feature behaviour consists of four components, as follows. Function calls 1-7 compose the AJAX request, function calls 8-9 are responsible for updating the user interface. Functions 10 and 11 are repeated 18 times while the client retrieves the server response. Finally, function calls 12-15 handle the server response.

The framework behaviour of Reddit consists of three components: (1) the request submission, (2) retrieval of the server response, and (3) handling of the server response. Function call 4 in Figure 5b shows a flood prevention component of the Reddit application. To protect the Reddit web server(s) from being flooded by a potential attacker, the rate_limit() function blocks rapid submissions before they are sent to the server. Rates are limited according to the action, i.e., one vote every 333 milliseconds and one comment every 5 seconds.

There is no difference between the behaviour of the upvoting and downvoting features. However, the differences emerge when comparing the voting features to the other features. Figure 6 emphasizes the differences in the link submission feature compared to the voting feature in Figure 5b. Structurally, the close_menus and XHR preparation behaviour components are in reversed order due to the different implementation details of each feature (function call 9 in Fig. 5b and 2 in Fig. 6). In the link submission feature, the user is redirected to the comments area for the newly submitted link after the server response has been received. The show_button() function (function call 6 in part 2) is called four times to display the user interface buttons on the new page. An additional onload() event (function call 7 in part 2) fires after the browser finishes loading the new page to switch the display context to the appropriate window.

As shown in row 3 of Table III, Reddit often has coverage values less than 65% with the exception of the HEADERS_RECEIVED and LOADING states, which contain no application code and offer little opportunity for deviation. Due to the low level of behaviour reuse among features, Reddit may require more effort to maintain than a system with high behaviour reuse.

We found that AJAX projects may have high framework behaviour coverage, an appealing quality for maintainers. Yet, some AJAX projects may have a low framework behaviour coverage, which may be less appealing for maintainers.

RQ2) How do AJAX applications cope with the network connectivity crosscutting concern?

To analyze the AJAX client's reaction to loss of network connectivity, we simulated disruptions to network connectivity. Using our comparison approach, the behaviour of each application under normal conditions is compared against the behaviour without connectivity to identify the areas of the design that were specific to handling connectivity issues.

Oceans: Without a connection to the server, the AJAX client cannot retrieve new content when the user requests it. As such, the best response for the Oceans application would be to inform the user of the connection issue.

Without network connectivity, the Oceans web application simply cleared the content area, without producing a warning, or notifying the user. We investigated further by using our comparison approach to compare the sequence diagrams both with connectivity and without, and were surprised to find that the diagrams were not different at all. Using the indexed sequence diagrams to drive a manual code inspection, we found that the response handling function (function call 10 in Figure 5a) did not check whether or not the server had correctly responded to an asynchronous call. Instead, the response handler populated the content area with the empty contents of the failed response object. While connectivity is not a direct concern of the Oceans application, its unpreparedness for this case is unsettling.

Tudu Lists: We expect Tudu Lists to have a more rigorous response to loss of server connectivity, as a user may make many edits to task lists before noticing an issue. Loss of those edits would be frustrating for a user.

After logging into the Tudu Lists system, we disabled the network connection. Without connectivity, we selected a task item and clicked the "completed" check box. Immediately, the application behaviour was noticeably different. The "Loading..." element that appeared for the duration of the AJAX request in RQ1 merely flashed on the screen for a moment before disappearing. The "completed" checkbox remained marked, which may lead users to believe that the request had been completed. The system offered no indication of failure to the user. After reconnecting the machine to the internet, we logged out and back into the Tudu Lists account to find that the checkbox we marked was now unmarked. When testing the task addition and removal features, the same "Loading..." issue occurred, however the user interface was left in the correct, unmodified state.

Comparing the recovered indexed sequence diagram to the one produced in RQ1 revealed that a new execution path was undertaken. Figure 5c part 1 was exactly the same in both cases. This was expected, since the two XHR states traversed (UNSENT and OPENED) were simply responsible for sending the request. In Part 2, the LOADING state is skipped meaning function calls 2 and 3 no longer exist. The new execution path begins in the stateChange() handler, which immediately calls the handleWarning() function. This handleWarning() function calls a defaultWarningHandler() to handle the warning, but this function simply logs a debugging message to the browser console. Eventually, clean-up functions are called.

The debug response indicates that while the Tudu Lists system is prepared for the occurrence of extreme conditions, it is not prepared to handle the failed connection case for the task completion feature. While it may be argued that presenting a user with an error message seems abrupt, presenting the user with an out-of-sync user interface, i.e., leaving the "completed" box checked, is much worse.

Reddit: When we examined the voting features in an offline state, we could not detect a difference in the Reddit user interface. The arrow that was clicked while offline changed colour and the vote count was changed just as it did when a connection was successfully made.

Comparing the offline indexed sequence diagram to the online one from RQ1, we see that Reddit is structured to handle errors much like Tudu Lists is. As expected, the two functions that are invoked before the request is sent to the server are identical to the online diagram, but the response retrieval and handling functions differ. The response retrieval only iterates once through the loop shown in function 10 of Figure 5b, since there is no server response. The final state change at function 11 calls a handleError() function that triggers an 'ajaxError' signal. Once again, an error handling skeleton seems to be in place but a real error recovery implementation seems to be lacking. The ajaxError signal is caught by a default error handling function that does not produce a user notification.

Similar to the result from Tudu Lists, the user interface is left out-of-sync with the server state. The arrow that was clicked was left coloured and the vote count was modified. The user was not notified of the communication error, which may lead them to believe that their vote has been counted.

In the disconnected state, the user interface of the studied AJAX applications were typically left out of sync with the server state.

Discussion: The studied applications seem unprepared to operate correctly in offline mode. Goncalves finds that identifying the correct behaviour for offline web applications is a non-trivial problem [19, 20]. Offline web application behaviour requires explicit API calls to offline frameworks (e.g., Google Gears [21]). Google recommends that users are made aware when an online-to-offline switch happens, so that they can adjust their usage accordingly. This behaviour was not observed with any of the studied applications.

V. Threats to Validity

Our feature recovery is based on being able to reconstruct a single thread of sequential execution. Both threading and multicore features of the Firefox browser were turned off at compile time. As a result, our analysis may be skewed by function calls that were flattened into one thread, but usually occur simultaneously in a multi-threaded environment.

We chose AJAX applications from three different domains to help balance our exploratory case study. These applications do however share a common thread of openness and availability of source code. It is possible that the results may not be reflective of all AJAX applications.

In our study, we ignore the server side of AJAX applications in lieu of a more thorough investigation of the client side behaviour. Current web application recovery approaches [3] or more dedicated AJAX approaches [5] can be used to analyze the server side interaction.

In practice, Javascript code is often obfuscated to protect the authors' intellectual property, by replacing meaningful function and variable identifiers with random strings. Since multiple functions may reside on one line, our approach cannot be used to extract the behaviour of obfuscated AJAX systems accurately.

Our dynamic analysis is limited to the code executed for each analyzed feature, and as such may not cover the entirety of an AJAX application. Furthermore, our selection (and exclusion) of features for analysis may also bias our results. However, we collected traces for a variety of features from each application to combat biases such as these.

VI. Related Work

Our work was initially inspired by Hassan *et al.* [3], who present an approach to recover the architecture of web applications. However, static design recovery approaches may not scale to AJAX applications, since AJAX interaction may be dynamically generated. We propose a feature identification approach for AJAX applications to bridge the gap between high-level web application data recovery and high-level AJAX application data recovery.

Second, our approach was influenced by Antoniol *et al.* [10], who discuss an approach to feature identification in large, multi-threaded applications. Their approach uses processor emulation, knowledge filtering, and probabilistic ranking to collect and analyze the dynamic data generated by highly complex applications. In our study, we do not address the concurrency problem directly, but rather force our web browser to operate using one thread of execution. Furthermore, Antoniol *et al.* show that recovered feature architectures (i.e., micro-architectures) that conform to a common meta-model can be programmatically compared. We used UML sequence diagrams as a meta-model that our dynamic data conforms to.

As mentioned above, we use annotated UML sequence diagrams to assist in explaining recovered framework and feature-specific logic. The concept of sequence diagram recovery was inspired by Briand *et al.* [22, 23]. They present an approach to reverse-engineer sequence diagrams from object-oriented systems using code instrumentation. We use a similar approach to recover sequence diagrams from Javascript components of AJAX applications, but rather than using code instrumentation, we use the DTrace dynamic tracing framework to provide tracing information. This frees our approach from having to make source code modifications or special compilations.

Mesbah *et al.* present CrawlJAX [24], a tool for crawling and automatically testing AJAX applications. CrawlJAX infers a state flow graph of an AJAX application by automatically interacting with web page elements and detecting DOM modifications [8, 9]. Unfortunately, due to platform incompatibilities of CrawlJAX (Windows/IE) and DTrace (Solaris or Mac OS X/Firefox), we could not leverage the CrawlJAX tool.

Amalfitano *et al.* present DynaRIA [25], a dynamic analysis tool designed for AJAX application analysis. They illustrate the utility of DynaRIA through analysis of the run-time behaviour of real AJAX applications. We also analyze the run-time behaviour of AJAX applications through dynamic analysis, however we focus on extracting the framework behaviour shared amongst features, as well as analyzing the offline behaviour of AJAX applications.

Matthijssen *et al.* discuss FireDetective, a tool for visualizing an AJAX application workflow to assist programmers in understanding AJAX applications [5]. They focus on improving program comprehension by visualizing a single feature at a time, while we recover a framework core and feature-specific extensions to conceptualize the whole program design.

VII. Conclusions

AJAX web applications inherit development challenges of classic web applications, and present new ones. We investigated the impact of these challenges on the design and maintenance of AJAX systems. Our analysis is based on the recovery of the run-time behaviour of AJAX systems, and visualization of this behaviour as state-indexed sequence diagrams. By comparing the diagrams across features and across working conditions, we are able to derive the framework behaviour shared by all features and the feature-specific behaviour of an AJAX system. Also, we use our approach to study the impact of the network connectivity crosscutting concern on the design of an AJAX system.

Through an exploratory case study on three AJAX applications, we made the following observations:

- Large amounts of behavioural reuse observed in the Tudu Lists and Oceans applications may make them easier to maintain.
- Client connectivity issues are generally not handled in an appropriate manner. In the studied projects, the client

component was either completely unprepared for such a response (i.e., Oceans) or more often, the user interface was out-of-sync with the server state (i.e., Tudu Lists and Reddit).

We are actively investigating other techniques for identifying feature-specific behaviour using Formal Concept Analysis (FCA), as inspired by Eisenbarth *et al.* [26].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their fruitful suggestions on earlier revisions of this paper.

REFERENCES

[1] D. DiNucci, "Fragmented Future," *Decision Processes*, vol. 50, no. 2, pp. 179–211, 1999.

[2] J. Garrett, "Ajax: A New Approach to Web Applications," http://adaptivepath.com/ideas/essays/archives/000385.php, viewed on: 30-Jul-2011.

[3] A. E. Hassan and R. C. Holt, "Architecture Recovery of Web Applications," in *Proc. of the 24th Int'l Conf. on Software Engineering (ICSE)*. ACM, 2002, pp. 349–362.

[4] R. Konrad, "Tech employees jumping jobs faster," http://news.cnet.com/2100-1017-241914.html, viewed on: 30-Jul-2011.

[5] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. van Deursen, "Connecting Traces: Understanding Client-Server Interactions in Ajax Applications," in *Proc. of the 18th Int'l Conf. on Program Comprehension (ICPC)*. IEEE, 2010, pp. 216–225.

[6] E. Kiciman and B. Livshits, "AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications," in *Proc. of the 21st Symposium on Operating Systems Principles (SOSP)*. ACM, 2007, pp. 17–30.

[7] E. Schrock, "Debugging AJAX in production," *Communications of the ACM*, vol. 52, no. 5, pp. 57–60, 2009.

[8] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling Ajax by inferring user interface state changes," in *Proc. of the 8th Int'l Conf. on Web Engineering (ICWE)*. IEEE, 2008.

[9] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of Ajax user interfaces," in *Proc. of the 2009 31st Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 210–220.

[10] G. Antoniol and Y. Guéhéneuc, "Feature identification: a novel approach and a case study," in *Proc. of the 21st Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2005, pp. 357–366.

[11] A. Mesbah and A. van Deursen, "A Component- and Push-based Architectural Style for AJAX Applications," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2194–2209, 2008.

[12] W3C, "Xmlhttprequest," http://www.w3.org/TR/XMLHttpRequest/, viewed on: 30-Jul-2011.

[13] Oracle Corp., "The DTrace Framework," http://www.oracle.com/technetwork/systems/dtrace/dtrace/index-jsp-137532.html, viewed on: 30-Jul-2011.

[14] B. Cantrill, "Hidden in Plain Sight," *ACM Queue*, vol. 4, no. 1, pp. 26–36, 2006.

[15] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Reverse engineering web applications: the ware approach," *Journal of Software Maintenance and Evolution*, vol. 16, pp. 71–101, January 2004.

[16] G. A. Di Lucca and M. Di Penta, "Integrating Static and Dynamic Analysis to Improve the Comprehension of Existing Web Applications," in *Proc. of the 7th Int'l Symposium on Web Site Evolution (WSE)*. IEEE, 2005, pp. 87–94.

[17] Y. Inghelbrecht, "Tracemodeler," http://www.tracemodeler.com/, viewed on: 30-Jul-2011.

[18] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/, viewed on: 30-Jul-2011.

[19] E. E. M. Gonçalves, "Current approaches to add offline work in web applications," Technical University of Lisbon, Lisbon, Portugal, Tech. Rep. 2542/D, 2008.

[20] E. E. M. Gonçalves and A. M. Leitão, "Implementing offline work in web applications for rich domains," in *Proc. of the 11th Int'l Symposium on Web Systems Evolution (WSE)*. IEEE, 2009, pp. 79–82.

[21] Google, "Google gears," http://gears.google.com/, last viewed: 30-Jul-2011.

[22] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams," in *Proc. of the 10th Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2003, p. 57.

[23] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 642–663, September 2006.

[24] Crawljax, "Crawljax: Automate ajax crawling and testing," http://crawljax.com/, viewed on: 30-Jul-2011.

[25] D. Amalfitano, A. Fasolino, A. Polcaro, and P. Tramontana, "Comprehending Ajax Web Applications by the DynaRIA Tool," in *Proc. of the 7th Int'l Conf. on Quality of Information and Communications Technology (QUATIC)*, 2010, pp. 122–131.

[26] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *Transactions on Software Engineering (TSE)*, vol. 29, no. 3, pp. 195–209, March 2003.