Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection

Joshua Hibschman
Northwestern University
Evanston, IL USA
jh@u.northwestern.edu

Haoqi Zhang
Northwestern University
Evanston, IL USA
hq@northwestern.edu

ABSTRACT

Professional websites with complex UI features provide real world examples for developers to learn from. Yet despite the availability of source code, it is still difficult to understand how these features are implemented. Existing tools such as the Chrome Developer Tools and Firebug offer debugging and inspection, but reverse engineering is still a time consuming task. We thus present Unravel, an extension of the Chrome Developer Tools for quickly tracking and visualizing HTML changes, JavaScript method calls, and JavaScript libraries. Unravel injects an observation agent into websites to monitor DOM interactions in real-time without functional interference or external dependencies. To manage potentially large observations of events, the Unravel UI provides affordances to reduce, sort, and scope observations. Testing Unravel with 13 web developers on 5 large-scale websites, we found a 53% decrease in time to discovering the first key source behind a UI feature and a 32% decrease in time to understanding how to fully recreate a feature.

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces

Author Keywords

Unravel; Reverse Engineering; Inspecting; Web Applications; Tracing; Recording

INTRODUCTION

Developers can learn from professional websites, but the barriers to understanding unfamiliar code [11] hinder the potential for authentic learning [20]. Without documentation for UI features of complex websites, one must search for curated examples or attempt to reverse engineer the website to discover how a feature works. Examples may not be available for unique features or may only provide partial solutions. Professional websites combine many web technologies to present unified interfaces that are not straightforward to disassemble. Reverse engineering UI components such as a photo carousel, search autocomplete, or table filter is difficult, because it involves cyclical HTML inspections to follow element changes and find-all queries in JavaScript files for references to DOM elements. JavaScript methods have short runtimes, making it difficult to identify which lines of JavaScript to start examining [1].

UIST'15, November 08 - 11, 2015, Charlotte, NC, USA.

ACM 978-1-4503-3779-3/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2807442.2807468

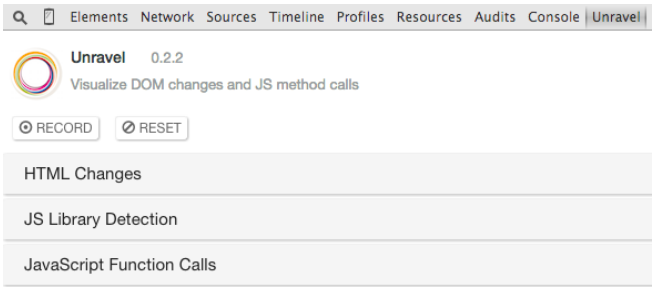

Figure 1. The Unravel Chrome Developer Tools extension detects HTML changes, JavaScript libraries, and JavaScript function calls while recording.

Current approaches including record & replay in the DOM and JavaScript tracing have inspired this paper (e.g. [2, 5, 7, 9, 19]) as they showed that recording and tracing changes in-context gives developers a better understanding of what's happening in the source code [6, 12]. With some context clues about where to begin looking, junior developers are more likely to overcome barriers that would otherwise prevent them from beginning a first attempt at reverse engineering [15]. But beyond source exposition, existing tools lack affordances to show the most relevant lines of source code. Complex features may consist of hundreds of recorded source code traces; without additional affordances, the inefficient process of searching, inspecting, and debugging to gain understanding is tedious and time-consuming.

Unravel aids the reverse engineering of websites by providing comprehensive yet targeted views of JavaScript invocations, HTML changes, and included libraries (see Figure 1). Unravel enhances Chrome's existing developer toolkit by linking all HTML and JavaScript components to their corresponding inspection panes for quick examination. Unravel works on all websites without interfering with existing functionality. For example, a developer can navigate to a landing page, record a parallax effect, and watch Unravel identify which lines of

JavaScript were executed, which DOM elements were modified, and which attributes were modified per each element.

The main conceptual contribution of this work is the idea of *tracing, identifying, and organizing the most relevant functions and DOM elements manipulated to support reverse engineering and understanding interactions on complex professional websites*. Unravel aggregates changes monitored from within a website and provides affordances to reduce, scope, and sort observations. Relevant sources become obvious choices for the user to examine. Complex UI features can invoke an enormous number of method calls and HTML changes. Navigating unstructured lists of change events for inspection is counterproductive. Unravel aggregates similar JavaScript call-stacks and HTML changes, increasing counts with each occurrence. Unravel's change panels are continually sorted by highest counts first, bubbling the most changed element or most called trace to the top. Affordances are provided to constrain observation scope to specific DOM subtrees and ignore event floods from SVG element changes.

The fundamental technical contribution of this work is *an observation agent that deploys an API harness for observing and recording UI interactions from within a website*. The API harness is an approach for monitoring an application's interaction with an API through a removable recording adapter placed between the application and the API. Unravel's observation agent publishes HTML changes and uses the API harness to monitor calls to the document API. While previous work was able to record and replay events, these solutions depended on access to a remote debugging API. Unravel's observation architecture only depends on native JavaScript and HTML, widening its application domain to other UI inspection toolkits.

In the rest of this paper, we review related work in discovering and extracting relevant source code. We then introduce Unravel and its main components for tracking HTML changes, tracing JavaScript method calls, and identifying libraries. We detail the observation agent and techniques for organizing and presenting trace information; evaluate the benefits of reverse engineering with Unravel; and conclude with a discussion of design principles and the limitations of our approach.

RELATED WORK

With Unravel, we are exploring strategies, interfaces, and methods for developers to quickly and easily overcome barriers in learning how interesting UI features of websites are constructed. In doing so, we aim to promote authentic learning that is personally meaningful and exists in a real-world context [20, 3] for web developers. The event-based and asynchronous nature of JavaScript poses a unique difficulty that can discourage those trying to learn [1]. Developers trying to overcome this difficulty turn to web foraging for speed and ease in finding help [4]. Unravel seeks to present JavaScript and HTML tracing in a way that decreases the need for web foraging and provides authentic learning for developers curious about features on professional websites.

Relevant Source Code Discovery

Unravel draws inspirations from previous work that aims to help users discover relevant sources behind a UI interaction. This includes animating a transition between the rendered UI and its source with Gliimpse [9], logging analytics on recordings of UI interactions with Mimic [5], extracting a REST API from a web application [22], providing breakpoints for DOM changes [2], and visualizing JavaScript traces during runtime with Theseus [16]. Extending this body of work, Unravel contributes a lightweight approach to linking UI features to their relevant sources in context within the browser.

Chrome Developer Tools and Firebug

Chrome Developer Tools (CDT) and Firebug both provide a rich suite for debugging, inspecting, and live-editing methods. DOM breakpoints in CDT[1] and Firebug[2] trigger JavaScript breakpoints when elements in the DOM are changed. This enables bottom-up inspection of a UI feature, whereas Unravel enables top-down inspection of sources recorded for a feature. Unravel captures and aggregates JavaScript traces and HTML changes during a recording and reduces its findings into a sortable view. The most used sources and most changed elements bubble to the top. Unravel integrates with CDT by linking lines of HTML and JavaScript to their corresponding inspection panes.

FireCrystal, WebCrystal, and Source Extraction

Unravel builds upon a body of previous work including FireCrystal [19], Maras et al [18], and WebCrystal [8] by providing in-context aggregation of the most relevant sources supporting a UI interaction with a study analyzing its effectiveness.

Maras et al created a strategy for extracting the minimum amount of sources relevant to a feature by communicating with a JavaScript debugger. Unravel implements a different strategy for finding relevant sources by using an API harness and observation agent, which require no external dependencies. WebCrystal caters to novice learners by allowing them to ask questions about HTML and CSS features of a website. Unravel provides HTML selectors and attribute changes, but focuses on the dynamic relationship between JavaScript and HTML, rather than the structural and stylistic features of HTML and CSS.

Most relevant to our work, FireCrystal [19] allows users to record an interaction with the DOM and to replay the interaction with highlighting over sources that are active at each point in time. We address with Unravel three main limitations of this previous work for reverse engineering professional web applications. First, rich UI features can involve thousands of lines of JavaScript across multiple call frames; FireCrystal's interface requires users to replay through interactions to discover relevant sources via linear search, which becomes tedious and time-consuming. In Unravel, users are

[1]Using DOM Breakpoints in Chrome Developer Tools: https://developer.chrome.com/devtools/docs/javascript-debugging#breakpoints-mutation-events.

[2]Using DOM Breakpoints in Firebug: https://getfirebug.com/doc/breakpoints/demo.html#dom.

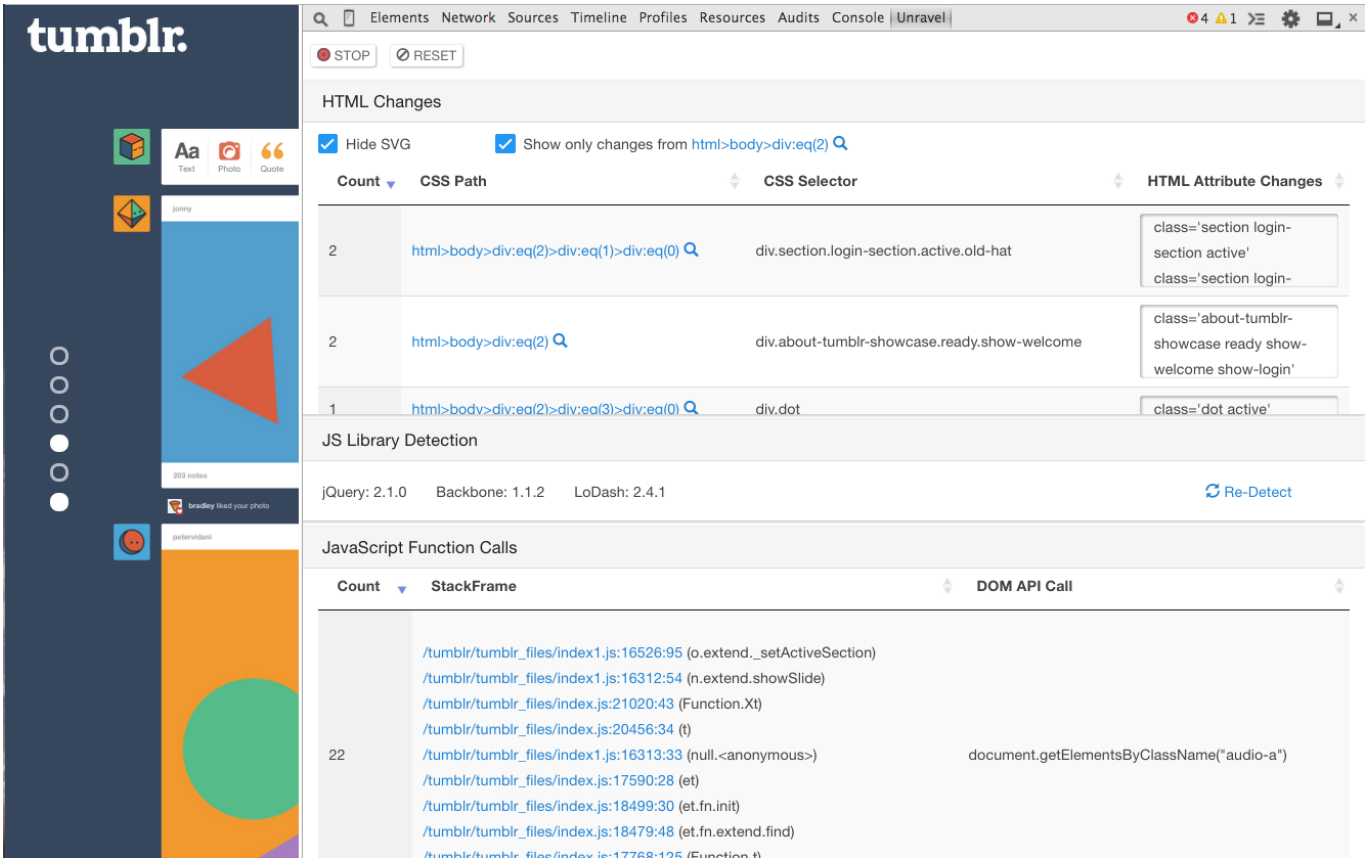

Figure 2. The Unravel recording interface consists of the HTML Changes pane (top), the JS Library Detection pane (middle), and the JavaScript Function Calls pane (bottom). Unravel has two controls for recording/stopping and resetting change detection (top). The HTML Changes pane shows the total count, CSS Path, CSS Selector, and HTML Attribute Changes for each element change. The JS Library Detection pane shows libraries detected. The JavaScript Function Calls pane shows the total count, stack frames, and DOM API call for each JavaScript call-stack recorded.

able to scope which elements to observe, and the system automatically reduces observations by aggregating and displaying sources with the most activity first. Second, FireCrystal relies on the Firefox debugging API; the authors noted slow-downs with large volumes of JavaScript changes. In contrast, Unravel scales well with observing complex UI features and uses an observation architecture that can be reused in other web UI toolkits. Finally, while FireCrystal's effectiveness was not studied, our user study showed significant reductions in time to understanding how to recreate UI features on professional websites with Unravel as well as differences in reverse engineering strategies.

Overcoming Barriers in Programming

Programmers often experience barriers in learning new programming concepts [15], and Unravel aims to provide in-context affordances to overcoming barriers such as not knowing where to begin. Front-end languages are delivered to client browsers with all sources available to be inspected like sample code. However, if a developer is not familiar with a programming API, they struggle with understanding sample code [12] and finding good examples to learn from [14]. The nature of unfamiliar client code promotes unusual barriers [11], but these barriers can be overcome by displaying relevant starting points and scoped sets of changes through Unravel.

UNRAVEL

Unravel is a Chrome Developer Tools extension that provides affordances for discovering and navigating relevant UI source code through three main activities: recording source code activity triggered by a user's interaction with a web page, refining the scope of source code under observation, and linking lines of source code to corresponding inspection and debugging panes for further analysis (see Figure 2).

Unravel Feature Design

To inform the design of Unravel, we conducted a small exploratory study that observed the existing approaches for reverse engineering web pages. The study consisted of two senior and two junior developers for 20 minutes each, who were asked to reconstruct an animated feature from Tumblr on their own page. We observed participants repeating the animation frequently while inspecting the HTML to see changes. We watched participants slowly scan through numerous JavaScript files to discover source code causing the animation. One participant said, "I just want to know how

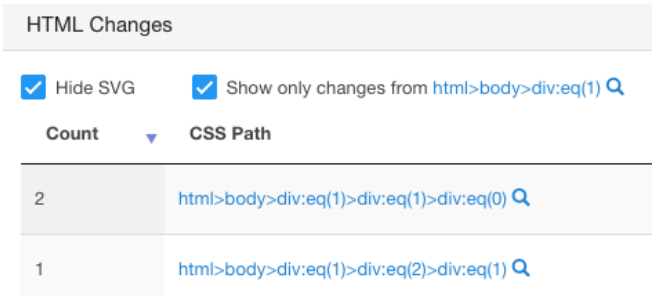

Figure 3. Within the Unravel HTML Changes pane, users can opt into hiding changes from SVG elements (top left). Users can constrain Unravel's observation scope by selecting an HTML element to observe (top right).

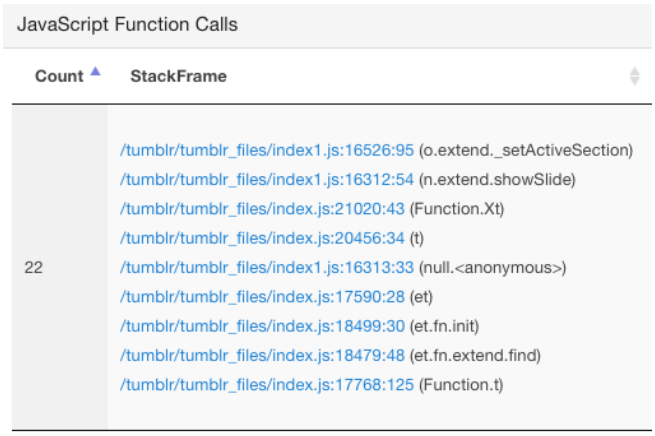

Figure 4. The Unravel JavaScript Function Calls pane has captured a call-stack that was executed 22 times. A stack frame with a method called _setActiveSection on object o.extend initiated the call-stack (top), which arrived at a document query for elements with class "audio-a" (shown in Figure 2 bottom right).

they achieved the effect, but it's not entirely clear from the web inspector."

Unravel's features were designed to help address frustrations and inefficiencies expressed by the test participants. The Unravel HTML Changes feature was designed to record and present modifications to lessen repeat behavior (see Figure 3). The JS Function Calls feature was designed to capture JavaScript traces with links to executed line numbers in JavaScript files, making it easier to skim active source code (see Figure 4). While no inefficiencies were observed related to JavaScript libraries, we noticed many non-native functions appearing in JavaScript traces. We decided to add library detection to inform the user about the presence of frameworks, polyfills, shims, or syntactic sugar (see Figure 5). Unravel's three views are presented as one inspection interface to highlight relevant source code supporting a feature.

Tracking HTML & CSS Changes

Without Unravel, current methods for detecting changes in HTML elements involve setting DOM breakpoints or watching for changes in element inspectors. Stepping through hundreds of attribute changes and looking through a DOM tree becomes time consuming. Unravel aims to streamline searches by providing a list of changes instead.

The Unravel extension begins to track HTML changes that occur in the website as a user starts a new recording. With each user interaction in the website, changes are streamed into the Unravel console under the HTML Changes section (see Figure 3). A DOM element's attribute and sub-tree modifications are then viewable in list form with direct links to structural and CSS inspection in the CDT elements pane (see Figure 2). While unravel does not capture preloaded CSS or CSS pseudo-classes like `:hover`, it monitors CSS class and style changes in HTML attributes such as changing opacity, toggling a class, or modifying WebKit attributes.

An Unravel HTML Change Record

Each record in the HTML Changes in the Unravel tool contains:

- Change Count: how many changes were recorded for the HTML element
- CSS Path: a unique CSS selector based on the element's DOM tree location that links to the corresponding node in the CDT Elements Pane
- CSS Selector: a CSS selector based on common query patterns including id, class, and name
- HTML Attribute Changes: a list of changes that occurred in the element's attributes in chronological order

An example user Alice wishes to discover how a modal window is hidden after clicking an × icon. She clicks *record* in Unravel and watches for changes while clicking the ×. Alice stops the recording and looks at the changes listed in the HTML Changes Pane of Unravel. She notices that the list is presorted by highest count of changes first. The first record shows a div with CSS selector `div#modal`. She clicks on the record to see what it is referencing in the actual website and elements panel. Chrome highlights the element in the panel and in the website. Alice confirms it is her element and examines the attribute changes, listed as `class="modal-front"` followed by `class="modal-hidden"`. Alice learns that removing class modal-front and adding class modal-hidden caused the desired effect.

Tracing JavaScript Method Calls

The bottom panel of Unravel lists JavaScript call-stacks captured while recording (see Figure 4). Unravel listens for calls to `window.document` and reports JavaScript traces involved in querying and manipulating the DOM. Every stack frame of each call-stack is linked to its corresponding file and line number in the CDT JavaScript inspector.

Each record in the Unravel JavaScript Changes pane contains:

- Call Count: how many times a call-stack was invoked
- Stack Frame(s): the call-stack leading to a document query
- DOM API Call: which document API method was invoked

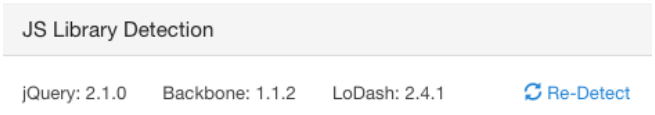

Figure 5. The Unravel JS Library Detection pane requests detection for libraries when Unravel starts and as users select re-detect (right). Re-detection is an affordance provided for libraries added after the initial page load. In this figure, jQuery, Backbone, and LoDash were detected.

An example user Carol wishes to better understand how a web application's card-flip effect revels new data when scrolling down in the interface. Carol initiates a new recording session in Unravel and begins to see stack frames captured in the JavaScript changes pane. Carol stops the recording and notices a call-stack was captured. Carol clicks the first frame in the call-stack and is linked to the CDT JavaScript inspector for `index1.js` at line `16526:95`. She immediately notices a function `_setActiveSection` that contains logic to change the `translate3d` style attribute of a div element. With the first clue, Carol returns to Unravel to search for how data is loaded. Carol skims the methods and arguments of additional stack frames and finds a method called `fetchCard`. She clicks the stack frame and discovers an XHR request contained a callback that triggered `_setActiveSection`.

Identifying JavaScript Libraries Used

As a precursor to examining source, a list of libraries active in a website prepares the user to understand source in context with the libraries. This may help them to reproduce code for their own use without the frustration of missing libraries. Further, this provides users with clues to how features are implemented using the libraries.

Unravel detects JavaScript Libraries immediately upon launch and lists the libraries with their corresponding versions (see Figure 5). An option to re-detect libraries is provided for websites that use a lazy-loading strategy for installing libraries into the application scope.

An example user Bob wishes to discover how a stock-ticker web application easily reformats numbers in many variations. He opens Unravel and finds many sources using a `numeral()` function. If Bob tried to invoke `numeral` in his own application, he would discover that it is not included in native JavaScript. Using Unravel's library detection, Bob sees that Numeral.js version 1.5.3 is present in the stock-ticker web application. Bob includes the numeral library in his application and is now able to use the same numeral conversion methods as the stock-ticker application.

ORGANIZING AND TRACING RELEVANT SOURCE CODE

Organizing Large Volumes of Trace Information

Complex UI features can generate large volumes of HTML changes and JavaScript traces. Navigating through long lists of changes and traces fails to resolve the *Information Learning Barrier* [15], because the program's internal behavior may remain unclear despite a wealth of information. This section discusses four strategies Unravel provides to counter information overload: DOM Tree Scoping, CSS Path Aggregation, SVG Hiding, and Call-Stack Aggregation.

DOM Tree Scoping

Without affordances to reduce observation events, simultaneous UI effects can cause confusion. As a user records an interaction, other dynamic behaviors in the application could highlight source code not relevant to the user's interests. After selecting an HTML element to observe, users can opt for Unravel to scope recordings to a single element and its subtree (see Figure 3). With the focus option selected, changes outside the scope of selection will be ignored.

CSS Paths and Selectors

HTML changes are recorded and reduced in real-time to the unique DOM tree path of an element. Continuous changes to one element's attributes are rolled up under a single record in Unravel's HTML change pane (see Figure 3). Elements with the most changes bubble to the top. While DOM tree paths can be queried, they can become quite long and difficult to read. Unravel provides simpler selectors by combining the elements tag, id, class, and name if present (see top middle of Figure 2).

SVG Hiding

During a preliminary study with an Unravel prototype, we discovered that users were being shown too many irrelevant HTML changes for pages that made use of SVG animations. The users weren't interested in the SVG animation itself, but rather DOM elements and interactions surrounding the SVG elements. In the HTML Changes pane, users can select an option to hide superfluous SVG changes (see Figure 3).

Call-Stack Aggregation

Similar to the HTML Changes feature, JavaScript traces are recorded and reduced by unique call-stack. Continuous calls through the same set of methods are logged by increasing the call-stack count. During our pilot study, users sorted lists by highest count first with stack frames ordered top-down. This became the default and our measure for source code relevance. All of Unravel's columns are sortable, allowing users to quickly navigate through different perspectives of their recordings.

Tracing UI Features to Relevant Source Code

In building Unravel, we sought to improve upon architecture from related systems to provide a scalable and portable implementation. Systems like FireCrystal and that of Maras et al depended on the Firefox Debugging API to query for sources involved behind UI feature [18, 19]. Both the scalability and portability of this strategy are limited to the constraints of the Firefox Debugging API. Theseus proposed a global method-wrapping policy for monitoring JavaScript traces that depended on a third-party server to alter sources [16]. We strived to build Unravel without any dependencies on external servers or environmental APIs so that it could scale to handling larger UI changes and share a reusable architecture for implementations in other UI toolkits.

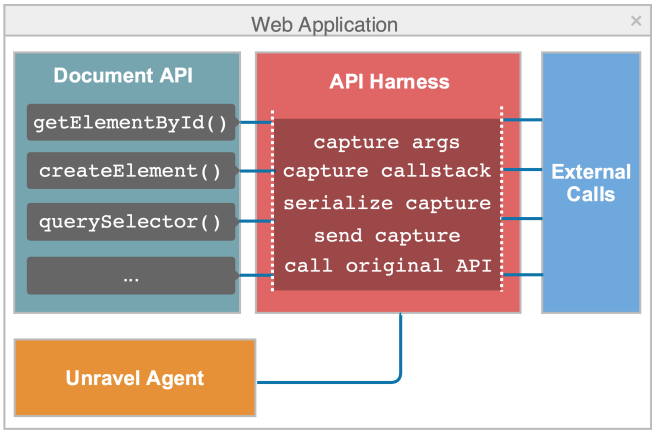

Figure 6. An API harness is placed on the document API. Unravel captures and serializes call-stacks and arguments made to the API. Normal interaction with the API resumes after the details of a method call are broadcast.

API Harness

We introduce an API harness as a novel method for monitoring all interactions with an API by placing a removable recording adapter on top of the API. Unravel's agent applies the API harness to monitor call-stacks and arguments to `window.document` when the user begins a recording and removes it when the user stops a recording. By monitoring the document API, we can see the execution route and arguments of functions asking to query and change the DOM (see Figure 6). Data from the harness is sorted and reduced prior to appearing in the Unravel's JavaScript Function Calls view. Technical details are discussed in the next section.

IMPLEMENTATION

API Harness

The API harness is a removable device installed during runtime that captures JavaScript method traces and arguments (see Figure 6). The harness implementation is straightforward: for each method in the API, save a reference to the original method and temporarily replace it with a new method that implements the following:

1. Capture the call-stack invoking an API method.
2. Capture arguments passed to the API method.
3. Serialize the captures for transport.
4. Propagate the capture to subscribers.
5. Call the original API method with the incoming arguments.

Captures are broadcast from the harness without modification as method calls are made to the API, giving subscribers flexibility in processing the data. Unravel's API harness call-stack captures implement the JavaScript `Error` interface. As each method call is made to the document API, an error object containing a snapshot of the call-stack is thrown and caught. This snapshot captures comprehensive execution traces from event handlers down to document queries. Unravel reduces and sorts its captures to simplify inspection for the user (discussed earlier). When a recording is finished, the API harness is removed by restoring the original methods to their respective endpoints in the API.

Alternative approaches to implementing an API harness either require external dependencies or aren't designed to monitor program execution. The Mozilla Remote Debugging Protocol[3] allows developers to access JavaScript threads and observe their execution but it is only available to extensions of Firefox. Lieber et al's Fondue wraps all functions in the JavaScript source to monitor execution, but exists as a separate proxy server that modifies a web page's JavaScript as it passes through [16]. Eagan et al's Scotty enables modification to non-extensible components during runtime, but it does not monitor interactions with those components [10].

Engineering trade-offs limit the capabilities of the API harness but give portability to its implementation. The harness must be able to modify public methods of the original API, it must be able to store references to the original method implementations, and it must be able to access callers and arguments. For example, an API harness would not be able to monitor an API reference that was closured in a private variable, because the harness requires public access to API methods. Despite these limitations, the API harness inspects program activity from within a program and operates without external dependencies. With minimal performance overhead, the API harness scales with API demand without causing interference.

HTML Observer and Library Detection

Unravel's HTML observation implements the JavaScript MutationObserver interface. When the observation scope is changed in the Unravel UI, new MutationObservers are created to monitor the corresponding subsections of the DOM tree. As the observers notice events, they are propagated to Unravel's sorting and reduction implementation. When each observation is received, its element's CSS path is calculated by determining the DOM tree location relative to parent and sibling nodes.

The JavaScript libraries are detected by a simple interface detection strategy: for each known library, the Unravel agent tries to invoke published interface methods from the library. We began with Hidayat's try-catch detection strategy [13], but extended it as we discovered libraries with identical identifiers and overlapping interface methods such as Underscore.js and lodash.js, both of whom have array methods like `_.reduce()`. If the test is successful, the agent detects the library version and returns the name and version number. There are many JavaScript libraries available, yet there is no published standard for declaring the library name and version from within the library. To detect all JavaScript libraries and display information about them is beyond the scope of Unravel, so we tested Unravel with support for the top 20 JavaScript libraries[4].

[3]Mozilla Remote Debugging Protocol https://wiki.mozilla.org/Remote_Debugging_Protocol

[4]These libraries are tracked at http://bower.io/stats

Website	UI Feature	Trigger
Tumblr	Card Flip View Change	Scroll, Click
Apple iPad	iPad Cover Change	Click
Flickr	Effect Sync to Video	Scroll, Click
Amazon	Product Carousel	Interval, Click
Kickstarter	Photo Carousel	Interval, Click

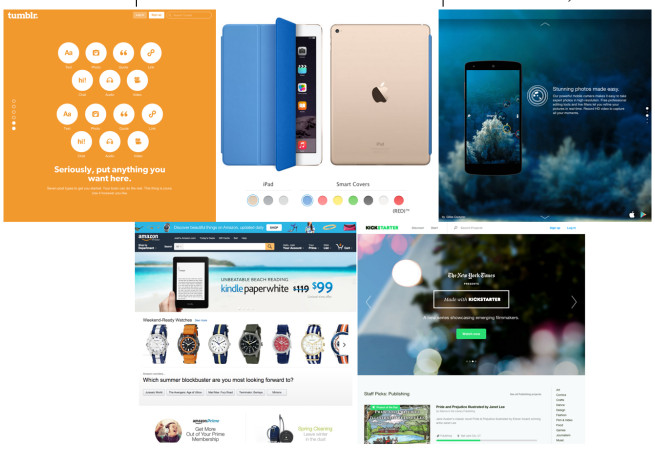

Figure 7. Participants reverse engineered 2 UI features from a set of 5. The top table lists each website with its corresponding feature and trigger under inspection. The screen-shots are of Tumblr, iPad, Flickr, Amazon, and Kickstarter (mid left to bottom right).

UNRAVEL USER STUDY

Method

Our study aims to answer the following research questions:

RQ1 How does a user's strategy for reverse engineering a web application UI feature differ with and without Unravel?

RQ2 How does Unravel affect the amount of time it takes a user to reverse engineer a web application UI feature?

RQ3 Which features in Unravel are the most effective while reverse engineering a web application UI feature?

RQ4 How do junior developers' use of Unravel and reverse engineering strategies differ from senior developers?

The target users of our study are junior web developers with less than one year of professional experience and senior web developers with greater than five years of professional experience. The study is a within-subjects design, where each user was asked to reverse engineer a UI feature in each of two websites from a pool of five, one website with CDT and one with CDT + Unravel (see Figure 7). CDT as the control requires no training or installation, and our initial study showed that both junior and senior developers could discover key sources using just CDT. 13 web developers, both junior and senior, participated in study sessions lasting 45 minutes. Time was limited to 15 minutes each for each reverse engineering task with a 15-minute follow-up discussion. Each participant was compensated $20. The assignment of websites to participants was randomized, and the order of using Unravel first was reversed for half of the participants.

We chose UI features from five popular professional websites: Tumblr, Apple, Flickr, Amazon, and Kickstarter. While widely used, each contains a clever implementation. When scrolling down on Tumblr's homepage, a card flip effect peels away each page view. Selecting different iPad covers on Apple's product page fades through user choices without changing the iPad image. Flickr's mobile demo synchronizes changes on its virtual phone screen with background fades. Amazon animates its product carousel with easing transitions based on user selection. Kickstarter flips through its banner carousel with fades during pre-programmed intervals. Though not obvious, functionality in these features consists of changing CSS classes, modifying HTML positioning attributes, and loading media in subtle ways.

We taught users about the tool, verified their background, and recorded their tests to ensure result accuracy. Before starting the test, participants were asked to watch a two-minute demo to help them become familiar with how to use Unravel. While participants were recruited by the experience on their CV, they were asked to confirm their amount of professional engineering experience before starting the experiment. Each participant provided a screen recording with audio and click history for the entire experiment.

We tracked three key milestones for reverse engineering. The milestones correspond to events happening at certain times, but participants were encouraged to proceed at their own pace throughout the tests.

M1. Time to finding the first key source.

M2. Time to finding the second key source.

M3. Time to fully understanding how to recreate a feature.

These milestones were tracked via each participant's screen recording to assess understanding. A key source is defined as a high-level code snippet that provides critical-path functionality for a behavior such as a click handler that adjusts the opacity of a div. Some participants had enough experience to describe a solution without reverse engineering, but they were required to find sources to support their claims. Prior to performing the study, the test set of five UI features were fully reverse engineered to identify significant methods, line numbers, classes and variable names in JavaScript, CSS, and HTML. For each solution, two key sources were identified that users must find for each UI feature in order to fully defend how the behavior is functioning. Timestamps for M1 and M2 were logged if a user displayed a key source in view for three or more seconds. M3 was logged when a participant gave notice of complete understanding.

Study pre-tests revealed inconsistency between web applications caused by source minification and obfuscation. Some users knew of the Chrome Dev Tools "Pretty Print" feature that reformats JavaScript source to be readable, while others were confused by large undecipherable blobs of JavaScript. To remedy source minification, we cloned versions of the popular web applications, manually unminified their sources, and hosted them on a private mirror. Subsequent tests showed that mirroring unminified versions resolved the testing inconsistency.

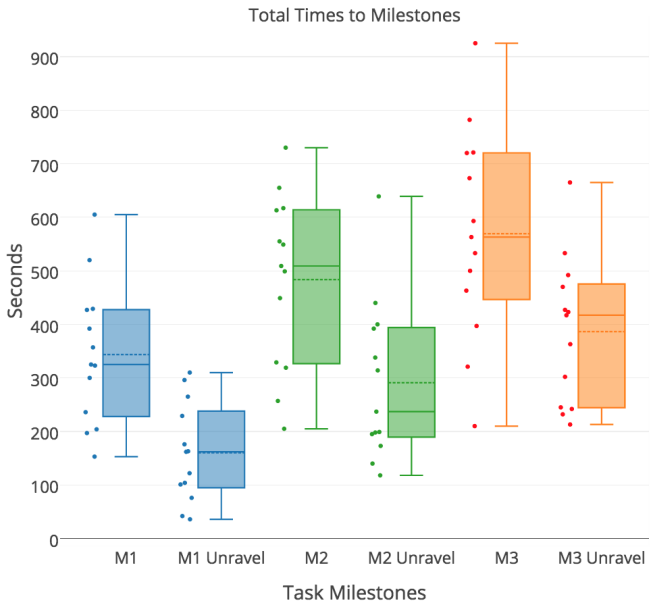

Figure 8. Results of the users study are compared in total times to milestones. Boxes indicate inner quartile range. Means are shown as dotted lines and medians are solid lines. The box whiskers indicate range including outliers. There is a significant difference in each of total times for M1, M2, M3.

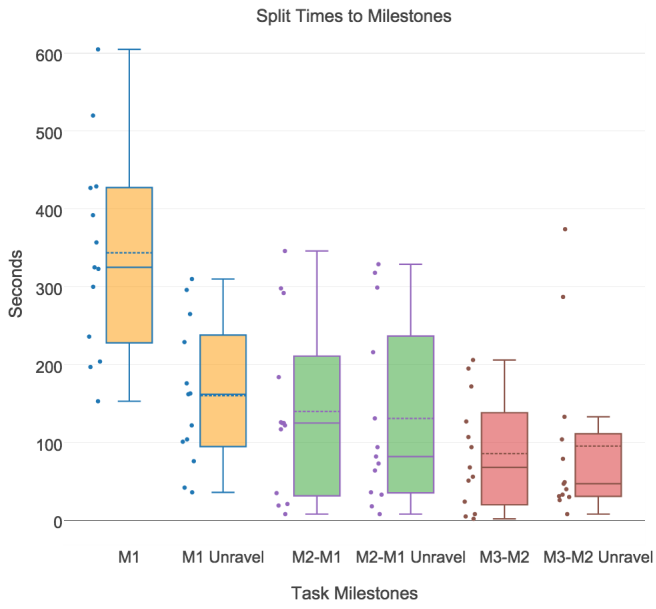

Figure 9. Results of the users study are compared in split times between milestones. There is a statistically significant difference between M1 and M1 with Unravel. However, there was no significant difference for the M2 or M3 split times. This means that Unravel was most effective for decreasing the time to first key source.

Participants were given a short follow-up discussion to assess how using Unravel altered their strategy and understanding of web application engineering. Questions about specific features of Unravel were included to assess their qualitative value and provide opportunities for feedback on feature usability. Survey results were compiled into four categories: useful features, improvements, learning, and strategies.

Data recordings from each participant were analyzed for statistics on 25 distinct user activities in CDT and Unravel and the time signatures of the major milestones. User activities include actions with similar complexity to switching an inspector pane, inspecting an event handler, or setting a breakpoint. Paired t-tests for with-Unravel vs without-Unravel were performed across all the coded data in the screen recordings to check for significant differences. Distributions were analyzed on an aggregate to determine average milestone times and activity counts.

STUDY RESULTS

How did Unravel affect task completion times?

Unravel significantly decreased time to all three milestones (see Figure 8). Developers achieved milestone I, finding their first key source responsible for the UI interaction 53.4% faster with Unravel ($t(13) = 4.2, p = 0.0012, \mu_1 = 184s, \mu_2 = 344s$) where μ_1 is CDT + Unravel. Developers achieved milestone II, finding their second key source 39.8% faster with Unravel ($t(13) = 4.533, p = 0.0007, \mu_1 = 291s, \mu_2 = 484s$). Developers achieved milestone III, reaching full understanding 32.1% faster with Unravel ($t(13) = 3.81, p = 0.0025, \mu_1 = 386s, \mu_2 = 569s$).

No significant difference was found in the split times between M1, M2, and M3 (see Figure 9). Developers had no significant difference between M1 and M2 ($t(13) = -0.24, p = 0.81, \mu_1 = 131, \mu_2 = 140s$) where μ_1 is CDT + Unravel. Developers had no significant difference between M2 and M3 ($t(13) = 0.33, p = 0.75, \mu_1 = 95s, \mu_2 = 86s$).

Differences in milestone times with and without Unravel are explained by variations in user interactions. Developers noted difficulty in finding a starting point during tests without Unravel, which increased their time to M1. Without significant differences in M2-M1 and M3-M2, some participants may have altered their strategy to depend on existing developer tools to find related sources. Total times to M2 and M3 show that no major inefficiencies affected overall time savings by using Unravel.

How did Unravel affect reverse engineering strategy?

Unravel significantly altered the reverse engineering strategy developers used when completing their tasks. Developers browsed an average of 2 JavaScript source files with Unravel compared to 10 without Unravel ($t(13) = 2.84, p = 0.015$). Developers searched for text in sources an average of 1 time with Unravel compared to an average of 9 times without Unravel ($t(13) = 5.6, p = 0.0001$). Developers focused on an element for inspection an average of 1 time with Unravel compared to 10 times without Unravel ($t(13) = 4.67, p = 0.0005$). Developers recreated the UI interactions an average of 5 times with Unravel compared to 11 times without Unravel ($t(13) = 3.45, p = 0.0048$).

How do junior developers compare to senior developers? Junior developers reached M1 without Unravel faster than senior developers and had no significant difference in other areas ($t(13) = 2.24, p = 0.05, \mu_1 - \mu_2 = 141s$), where μ_1 is for senior developers. Differences with less statistical significance include: senior developers set more breakpoints ($t(13) = 1.99, p = 0.09, \mu_1 - \mu_2 = 4$), senior developers were more likely to inspect network ($t(13) = 2.29, p = 0.06, \mu_1 = 1, \mu_2 = 0$), senior developers inspected more elements ($t(13) = 1.8, p = 0.11, \mu_1 - \mu_2 = 6$), and senior developers inspected more event handlers ($t(13) = 1.84, p = 0.11, \mu_1 - \mu_2 = 3$). While senior developers used different CDT interface controls to reverse engineer, they desired a broader understanding of the UI feature in the context of the website. A senior developer stated, "I had an idea of how the feature worked before I started, so I wanted to see how the feature was situated in the application first."

Which features of Unravel were the most effective? In the follow-up discussion, all 13 participants were interviewed for their opinion on Unravel's features, Unravel's weaknesses, reverse engineering strategies, and concepts learned. 10 out of 13 developers stated the JavaScript method traces were most helpful for them to understand a solution. The remaining 3 out of 13 developers stated the HTML changes pane was most helpful. 5 out of 13 developers found the library detection pane useful, while the other 8 stated it did not provide any help. 4 out of 13 participants noted that Unravel could be improved by integrating library detection into the JavaScript stack-traces to highlight the difference between library vs non-library source. 7 out of 13 participants stated a new programming concept they learned while reverse engineering with Unravel. 5 out of 13 participants found constraining the scope of observation useful.

DISCUSSION

Having demonstrated the effectiveness of Unravel for helping web developers reverse engineer professional websites quickly, we revisit techniques that contribute to Unravel's effectiveness.

Organizing and Presenting Large Volumes of Traces

Compared to the performance and interfaces of other source-tracking systems, Unravel is distinguished by its abilities to reduce, scope, and filter large amounts of source detection information in a way that highlights relevant data for the user. A participant stated, "Unravel was way easier to locate specifically where and when in the files the code was executed." We observed through the study that participants found relevant sources by looking at the top items in the HTML JS tracking panels in Unravel. A different participant stated, "Without a doubt I prefer Unravel over sifting through element changes in the Chrome Inspector."

Tracing UI Features to Relevant Sources

Advancing related work [19, 18, 8, 5, 7, 12, 22], Unravel introduces a reusable architecture that is both portable and scalable. Unravel serves as a recorder and reducer of meaningful information, with detailed inspection delegated externally. The implementation for this paper was in CDT, but a participant asked, "Could we have this for Node.js?" While there isn't a DOM to observe, Unravel's JavaScript source tracing and library detection would work in Node.js. For example, an API harness placed on the HTTP API could capture meaningful traces supporting a GET or POST request. The API harness and application agent allow Unravel's architecture to be reused in any JavaScript environment. The scalable nature of Unravel's architecture allows it to accommodate long recordings of complex features. A participant stated, "I don't even need to inspect, I just hit record and it goes. That by itself is great."

LIMITATIONS

Unravel only provides recordings of client-side traces and execution. Server-side source code typically isn't made available for external inspections, but there is an effort to study how to expose API endpoint and behaviors from front-end source [18]. Further, professional websites typically use source-code minification techniques to decrease the size of their files an average of 20% [21]. For our user tests, sources were manually unminified for participants. This feature can be added to Unravel with the use of JavaScript libraries like js-beautify [17], where sources would be parsed and reloaded in unminified form.

Our study did not attempt to identify UI features for which Unravel is not able to provide meaningful information. In our preliminary study, we discovered shortcomings from SVG transitions where elements had hundreds of positioning attribute changes each second. This flood of changes carries a risk of burying relevant sources. Pseudo-elements and CSS pseudo-classes are outside Unravel's scope of observation but can be easily discovered with existing inspection tools.

In-memory state storage techniques are outside the observation scope of Unravel. Unravel's API harness will not be able to monitor communication with privately closured references to an API. If a web application is designed to preload DOM API queries into memory on page load, Unravel will not capture the query in its API harness if it was not actively recording at page load. A potential workaround is to detect these behaviors and inject the API harness and observation agents prior to page load.

FUTURE WORK

Our future work seeks to understand the needs of developers as they face reverse engineering and learning challenges in their own projects and while assisting others. We wish to provide new methods and tools for authentic learning through real-world examples that support novice learners wanting to become professional developers. We are currently exploring methods for providing *readily available learning experiences* that scaffold the learning process for recreating and adapting complex examples. One such application is to automatically extract relevant source code for a feature to generate a learning example. By providing developers with tools like Unravel to quickly expose and disassemble the hidden complexities of web applications, we hope to increase interest and lower the resistance in learning from professional examples.

REFERENCES

1. Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 367–377.
2. John J Barton and Jan Odvarko. 2010. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*. ACM, 81–90.
3. Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences* 22, 4 (2013), 564–599.
4. Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
5. Simon Breslav, Azam Khan, and Kasper Hornbæk. 2014. Mimic: visual analytics of online micro-interactions. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*. ACM, 245–252.
6. Brian Burg. 2013. Answering program understanding questions on demand with task-specific runtime information. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 167–168.
7. Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 473–484.
8. Kerry Shih-Ping Chang and Brad A Myers. 2012. WebCrystal: understanding and reusing examples in web authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3205–3214.
9. Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from markup code to rendered documents and vice versa. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 257–262.
10. James R Eagan, Michel Beaudouin-Lafon, and Wendy E Mackay. 2011. Cracking the cocoa nut: user interface programming at runtime. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 225–234.
11. Paul Gross and Caitlin Kelleher. 2010. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing* 21, 5 (2010), 263–276.
12. Paul Gross, Jennifer Yang, and Caitlin Kelleher. 2011. Dinah: An interface to assist non-programmers with selecting program code causing graphical output. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3397–3400.
13. Ariya Hidayat. 2015. Detecting JavaScript Libraries and Versions. Don't Code Today What You Can't Debug Tomorrow. (2015). http://ariya.ofilabs.com/2013/07/detecting-js-libraries-versions.html
14. Raphael Hoffmann, James Fogarty, and Daniel S Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 13–22.
15. Andrew Jensen Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.
16. Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, 2481–2490.
17. Einar Lielmanis. 2015. Beautify-web/js-beautify. (2015). https://github.com/beautify-web/js-beautify
18. Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting client-side web application code. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 819–828.
19. Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*. IEEE, 105–108.
20. David Williamson Shaffer and Mitchel Resnick. 1999. " Thick" Authenticity: New Media and Authentic Learning. *Journal of interactive learning research* 10, 2 (1999), 195–215.
21. Steve Souders. 2008. High-performance web sites. *Commun. ACM* 51, 12 (2008), 36–41.
22. Bipin Upadhyaya, Foutse Khomh, and Ying Zou. 2012. Extracting RESTful services from Web applications.. In *SOCA*. 1–4.