Perceiving the GUISE: Graphical User Interface Specification Extraction

Leo Meyerovich[*] , Raluca Sauciuc
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776
{lmeyerov, sauciuc}@cs.berkeley.edu

ABSTRACT

We present a dynamic control-flow analysis and state classifier for graphical user interfaces. Search engines, end-user programming interfaces, and automated testers exploit such information, but are challenged by clientside and serverside scripts obscuring it: our analysis succeeds on popular web applications that contain both. We further motivate such analyses. First, we introduce a new type of browser extension: a natural-language interface to third-party applications. Second, we begin to address the problem of updates to a website changing how a third-party application must use it: by extending our analysis to yield change-impact information, meta-applications can automatically repair their broken interactions.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Enhancement, Restructuring, reverse engineering, and reengineering*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Elicitation methods, Tools*

General Terms

Web, User interfaces, Abstract interpretation, Data mining, Specification extraction, Programming-by-demonstration, Metaprogramming

Keywords

Graphical user interfaces, Model extraction, Interaction models, Control-flow analysis, Change-impact analysis, Macro repair, Natural language interfaces

[*]This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1. INTRODUCTION

Many web programs *generically* operate over user interfaces of other applications. For example, some browser extensions detect password fields and automate filling them in, and others restructure a user-interface to match user and device constraints. These are typically built by only assuming input websites employ the Document Object Model (DOM), which is a tree representation to describe the document contents a user may see and interact with at a particular instant. However, the DOM only represents the *current* contents of a page: what if meta-applications could query the *full structure* of a web application, not just the current state?

Meta-applications, such as search engines, are increasingly extracting and then exploiting the structure of website user interfaces (UIs). Furthermore, new application domains are arising that more deeply exploit UI models. As examples of using variants of this knowledge, automated testers[20] test GUI-driven software, mashup generators[9] help end-users compose web applications, and screen scrapers[6] help end-users extract application content. Third-party meta-applications that generically exploit UI models are emerging as an effective way to finally achieve a variety of application compositions (Section 2).

Model extractors enable sophisticated program manipulations, but traditional ones for the web should be considered (static) document analyses: they fail on applications. For example, Flash and JavaScript applications typically break the assumption that the content visible when loading a URI is all of the content visible from that URI, and opaque server-side programs might use state or multiple URIs to represent different sortings of the same data. Furthermore, websites frequently undergo structural changes[7], so analyses must adapt to these. Finally, large-scale spidering is often infeasible: fully crawling an application has demanding trust and resource implications. More traditional program analyses are insufficient as well: server-side code is generally unavailable, the dynamic nature of client-side code makes it difficult to analyze[3], and it is unclear how to match program models extracted by these techniques to user interactions.

Our approach is to exercise a GUI and learn a model from the observed transitions, but we ignore traditional hints like page boundaries. The challenge is to find abstractions fine-grained enough to yield a useful model (*precision*), but coarse-enough to abstract over a sufficient portion of the true program (*recall*). Consider an email client that shows a list of emails and a panel to interact with an email. Including every email directly in the model blows up the state space[20]: precision is high, but, as every email must be observed, coverage is low. Abstracting states with similar textual content as being the same reduces the state space [22], but discards most user actions (e.g., that a 'reply' button transitions the state from previewing to replying) and still requires all data to be seen: models are smaller, but precision suffers and coverage does not improve.

We recast the problem as extracting a control-flow model of the user interface, highlighting that we should abstract away user and application data. Actions are often parameterized by data: our approach is to abstract them, such as by attempting to equate a link to one email with a link to another. [1] Our core analysis consumes snapshots of transitions between DOM states and outputs 1) a graph describing transitions between GUI states and 2) a classifier to label an arbitrary page as a state in this graph. For example, a meta-application can classify the state of a page and use the model to predict where clicking on a widget will lead. We quantitatively evaluate our model extractor (Section 5) on large, popular web applications: it is precise, lightweight, and quickly achieves high coverage.

We qualitatively evaluate the versatility and appropriateness of our approach by exploiting it in otherwise challenging meta-applications. First, we describe our novel browser extension(Section 2.1) that interprets natural-language commands for *arbitrary* applications. Previous attempts interpret sloppy[23] commands for directly visible screens like `click menu; click photo; ...`: ours interprets the more natural command `upload private photo`. Second, we extend our analysis to yield change information (Section 2.2) because updates to a website often break any third-party applications that manipulates it. An application can use this in a change-impact analysis, detecting which interactions with the website must be repaired and, if so, how.

In summary, we contribute:

1. A control-flow analysis for web application GUIs.
2. Motivation for exposing and manipulating UI models.
3. A natural-language interface to web applications.
4. A change-impact analysis for GUI manipulations.

2. UI MODEL-DRIVEN APPLICATIONS

User interfaces are an exploitable abstraction layer. They represent what a user may perform, eliding often unnecessary details like lower-level or unused program functionality. Even exposing as simple a representation of a user interface as a finite state machine where actions are abstract DOM tree configurations and edges are clickable DOM nodes, as learned by our analysis, is already useful. A meta-application writer, to be productive, any hook into such a simple model of a website, only considering the exposed website-specific language of user interactions. Decisions are now based on high-level information about what sequences of commands take the user interface to which states, while low-level details about the underlying code that interface commands invoke are filtered away.

[1]Abstracting away data does not mean we lose sight of it: this helps isolate it, so downstream processes are better equipped to analyze data.

Programming-by-Demonstration (PBD) Tools are instructed by a user to perform tasks that would otherwise be performed manually. Screenscrapers[6] can be shown how to extract data from websites, and mashup generators can be shown how to manipulate them[9]. In an earlier version of this work, we considered how to achieve PBD tools that 1) infer loops based off of a user's actions and 2) detect demonstrations that, when synthesized, yield potentially underspecified program, and query users about these relevant scenarios.

Mixed Initiative Interfaces augment direct user manipulations with automated actions. For example, when using a complex image editing program, a mixed initiative interface might suggest effects that others have performed in analogous program states[5]. Extracting the usage data for this is difficult, yet it can be phrased as a simple probabilistic extension of finite state machines where edges have probability distributions.

Automated testing benefits as well: bug finders that explore models are driven by such tools[18, 20, 24].

Indexing applications for search engines is one of our long-term motivations. Recent work examines randomly changing parameters in URLs [12] or clicking [22] to extract data from applications. However, just as directed methods are useful in automated testing [24], we believe an analogous notion can be applied to indexing data in programs, and are pursuing this.

Analytics help gauge if and how features are used. Again, our tool can be adapted to describing common flows at a legible abstraction level by adding edge probabilities.

There are many other uses. Next, we attempt to interpret natural language commands by a user and make meta-applications more robust against changes to a website's interface. Meta-applications that manipulate UI models are increasingly helping us achieve otherwise difficult programs.

2.1 A Natural-Language Program Interface

Suppose we visit the home page of the `flickr.com` application and want to quickly upload a photo, but do not yet want to make it public. No single action available on the start page can perform this. Similar to how we use a search engine, we'd like to simply issue the command "upload private photo." We introduce our analysis by showing how we use it in a Firefox extension that, without any customization for Flickr, recognizes this command.

Our analysis constructs a control-flow model for the user interface of Flickr in the form of a finite state machine (FSM). Figure 1 shows part of the one generated for Flickr. Directed edges represent legal user actions such as clicking on buttons and following links. Only the actual labels of these elements

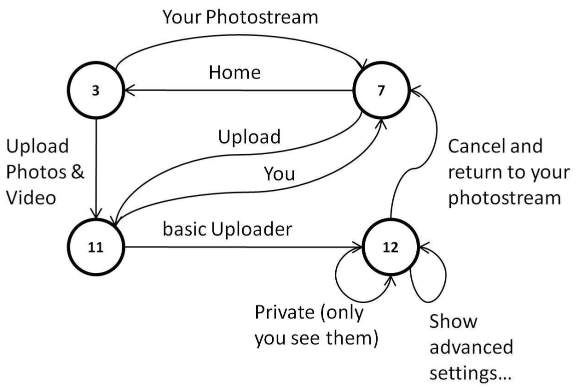

Your Photostream
Home
3
7
Upload Photos & Video
Upload
You
Cancel and return to your photostream
11
basic Uploader
12
Private (only you see them)
Show advanced settings...

Figure 1: Snippet of the Flickr.com FSM

are presented here, to simplify the exposition. Nodes represent states of the application that share the same control structure, namely the same set of outgoing actions.

We recast the problem in terms of directed graphs: given an initial node s and a set of keywords $Q = \{q_i | i = 1..n\}$, we wish to find a path in this graph starting at s and ending at some other node t, such that *i)* the labels of the edges on this path match the maximum number of keywords and *ii)* the path is the shortest possible. We do not require exact word matches, opting instead for word senses, and in lieu of a grammar model, accept commands in *any* order.

Our extension constructs the FSM and extracts the adjacency matrix. It then perform a transitive closure over the matrix to compute reachability information and collect at each step the unordered set of labels on each path. Let $labels(s_1, s_2)^k$ denote the set of labels on all paths from s_1 to s_2 of length up to k. The function $\texttt{match}(A, B)$ counts the number of query keywords from set A that match keywords from set B. In essence, our algorithm computes:

$$max\ min_k\ \texttt{match}(\texttt{labels}(s, s')^k, Q)$$

For the Flickr example, we indeed find the shortest path from state 3 to state 12 going through state 11, as it maximizes the number of matched keywords (3). Our model extractor opens up a variety of new possibilities.

2.2 Interacting with Applications that Change

Programs written to manipulate user interfaces are sensitive to changes to interfaces. In a survey of programming-by-demonstration tools that operate over other websites (including but not limited to [6, 15, 23, 9]), most archived user-generated programs did not work. We assume some worked at some point and updates to the PBD tools were backwards-compatible. Thus, the chief cause of disruption is likely external. Innocuous updates to web sites are frequent and include structural interfaces changes[7] that cause PBD-generated programs to break. We show that such promising tools can be made *autonomic* by utilizing information of how a user interface changes to repair themselves.

First, we adapted our model extractor into a change analysis, revealing what abstract UI states are added and removed. We examined the open source customer relationship management tool SugarCRM, consisting of 443,001 lines of PHP code, for examples of structural changes when updating one major minor version (from 5.0.0h to 5.1.0b). We first randomly clicked around four sub-components of SugarCRM 5.0, and then randomly used them again in 5.1, recording 448 and 219 non-spurious clicks, respectively. We ran our model extractor on all of these, yielding the basic transition structure partially shown in Figure 2. By coloring every abstract state based on whether it was demonstrated in version 5.0, 5.1, or both, we estimate how the application changed.

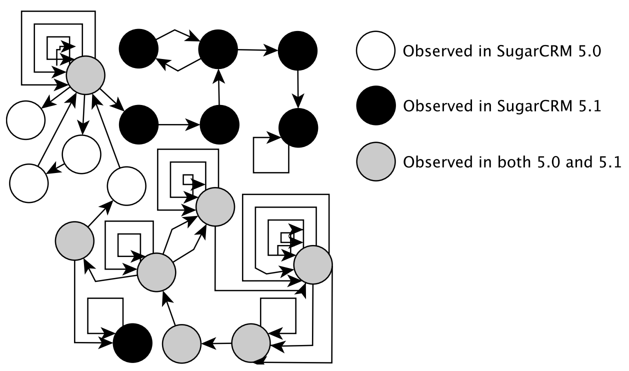


Figure 2: Fragment of a change analysis for upgrading SugarCRM from 5.0 to 5.1. Circles are states and edges are actions.

In unsupervised deployments, a subtlety is of how to determine which concrete recordings belong to which program versions. In our case, we knew that URLs involving version number "5.0.0h" were from one epoch and "5.1.0b" from another. As will be clearer in the evaluation, there is an expected error rate for predictions made by an extracted model: an indicator for a new epoch is that a model is suddenly making worse predictions than usual. Domain knowledge also helps, such as tracking when third-party programs break. Techniques like the former are safer in that they operate without new interactions with the application.

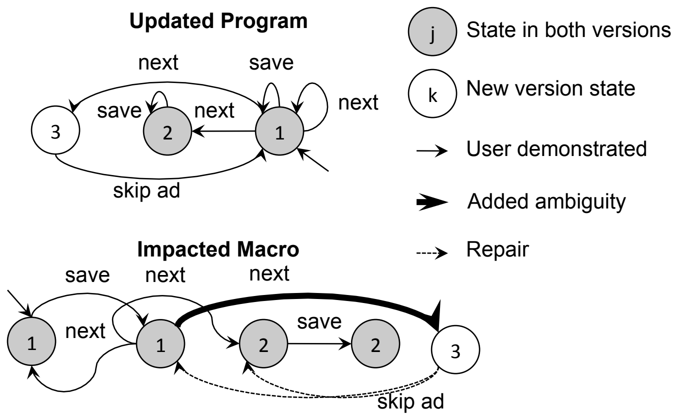


Figure 3: A photoviewing application that is updated to show advertisements and an impacted third-party program to download all photos.

We adapt our change analysis for websites into a change-impact analysis for third-party programs. Figure 3 describes part of a model photo viewing application with `next` and `save` buttons. Consider a third-party program to follow `next` buttons and `save` every photo, such as one authored by an end-user with a PBD tool or a programmer writing Python. If the website is updated to occasionally show an intermediate advertising page with an explicit `skip ad` button, the third-party program will stop prematurely on that state (Updated Program in Figure 3).

Our initial solution is to simply label the third-party program states with respect to the website state they are acting upon. This implicitly colors states as well. A program is broken if it utilizes a black colored state. Program transitions to gray edges that, non-deterministically, may also lead to white (new) states, may also break. Once these bad programs are detected, a tool might query a user for more information or try to automatically reroute the program. For example, in the case of ads, a user model would show that the common course of action is to press 'skip ad' and realign with the original program (Impacted Macro in Figure 3).

Coloring is insufficient. It helps when abstract states are added and removed. However, actions also change, such as if `save` is renamed to `download`[2]. Our analysis is still useful: the task is now akin to that of machine translation[4].

In summary, model extractors like ours are a viable route to change-related challenges like autonomic application macros.

3. MINING INTERACTION MODELS

$$\begin{tikzcd}
\mathit{GUI\ model} \arrow[r, mapsto, "Properties", "Semantics"'] & \mathit{Semantic\ Queries} \\
\mathit{Application\ Source} \arrow[u] \arrow[r, mapsto, "Properties", "Semantics"'] & \mathit{Program\ Queries} \arrow[u, dotted]
\end{tikzcd}$$

Figure 4: Two choices for extracting semantic information from an application.

We want to export a high-level model of an application to support semantic queries directly upon it. Standard program analyses operate on a low-level view of the application, closer to the program source than to the end-user's conceptual model, as in Figure 4. They offer program querying functionality, but have a hard time converting a low-level query answer into a high-level one (e.g., in terms of a UI model – the dotted arrow). Users typically do not care about source code or the internal state of the application, as their mental model is the navigation through the user interface. Rich meta-applications, such as our natural language extension, require semantic queries over more abstract states. We introduce a technique for extracting GUI models to enable such semantic queries. It is provably sound and can be automated, drawing directly from the general abstract interpretation framework for stacking abstractions.

3.1 Abstraction Stack for GUI Models

We start by representing the true user interface model of the application as a transition system. A program state, represented by a node in a graph, encapsulates the visual representation and any associated data such as cookies or persistent data. Transitions, corresponding to edges, are defined by actions that affect the application state – synchronous user interactions such as clicks, or GUI changes triggered asynchronously by I/O or network activity, timeouts, etc. For the synchronous case (the most common), transitions are labeled by the visual element that triggered the update – which button, link, etc.

We assume that action labels are informative enough to guide the user through the interface, such as through a button label, link text, etc. We also assume that the current state provides all the necessary information to the user and thus determines all future interactions, hence our state machine representation. These assumptions rely on commonly practiced UI design principles.

Concretely, let S be the set of interaction states of the application, A be the set of all actions and Δ a transition relation, $\Delta \subseteq S \times (A \cup \epsilon) \times S$. The ϵ action models asynchronous transitions. We do not perform a static analysis over the source code (it is not tractable[3]) nor instrument it (it may be hidden by a server), so we do not know the set of states S a priori nor the set of actions A. Just taking observed and recorded triples (s, a, s') and defining them as the approximation of the set Δ is insufficient: we only observe a tiny portion of the true set. Instead, we stack three abstractions that ultimately yield the computable finite state machine approximation $(S_{\widehat{A}}, \widehat{A}, \Delta_{\widehat{A}})$.

$$S \xrightarrow{\beta_{S\to DOM}} S_{DOM} \xrightarrow{\beta_{DOM\to A}} S_A \xrightarrow{\beta_{A\to\widehat{A}}} S_{\widehat{A}} \qquad (1)$$

Loosely, this corresponds to abstracting to the visual DOM state(Section 3.3), and then (Section 3.4) the item a user interacted with and detecting data within the manipulated item. Every inferred triple $(s_{\widehat{A}}, \widehat{a}, s'_{\widehat{A}})$ corresponds to potentially many (s, a, s') triples.

3.2 Algorithm Intuition and Composing Abstractions

Every abstraction is driven by two heuristics: a function β that abstracts states, and an equivalence class $\sim$ over actions, with a corresponding projection function π mapping actions to their equivalence class. We do not know the bins defined by $\sim$ a priori, so we learn it by grouping similar observed actions using heuristics. For the first abstraction, we define β, but in the subsequent abstractions, we define $\beta(s)$ as the set of abstract actions enabled in s. On every pass we can use these to compute a more abstract transition relation Δ. This over-approximates the previous, more concrete transition relation: for every transition (s_1, a, s_2) in the old state machine, we add a transition $(\beta(s_1), \pi(a), \beta(s_2))$ in the new state machine. This preserves all real transitions and possibly introduces spurious ones when the heuristics are too loose. Finally, instead of making multiple passes, we can compose the action abstractions into one action abstractor $\pi_{\widehat{A}}$ and, more efficiently, make one large pass. We now proceed to describe each step in detail.

3.3 DOM Actions

[2]We label actions with node paths: switching HTML tags is tantamount to renaming in our models.

Our first abstraction function $\beta_{S\to DOM}$ maps concrete visual states into DOM trees (S_{DOM}). This discards internal state such as cookies and client-side and server-side persistent state. The projection function π_{DOM} plays a similar role for actions. Synchronous user actions are declared equivalent if they occurred on the same target element in the DOM tree of the page (the user clicked the same button, link, etc.); the projection function maps concrete actions to exact paths in the DOM tree. This discards side-effects, such as performing a POST request. The state machine at this step has transitions Δ_{DOM}, safely mirroring the transitions from Δ.

$$\beta_{S\to DOM}(s) = DOM(s), \forall s \in S \tag{2}$$
$$\pi_{DOM}(a) = DOM\_PATH(a), \forall a \in A \tag{3}$$

The semantics we described so far is uncomputable whenever the state machine has an infinite number of states. The reason is that we retain too much information in the abstraction – the whole DOM tree. Much of this content is data-dependent and varies from user to user. The intuition is that we want to define a page in terms of what the user can do with it, namely the set of enabled actions. We say a state s has action a enabled if there exists $(s, a, s') \in \Delta_{DOM}$ for some $s' \in S_{DOM}$. This means the element (button, link, etc.) represented by a is present in the DOM tree of the page.

Let $en(s)$ denote the set of enabled actions for state s, $en : S_{DOM} \to \mathcal{P}(A)$. Our next abstraction $\beta_{DOM\to A}$ groups states that have the same set of enabled actions and builds the state machine from S_A and Δ_A accordingly:

$$\beta_{DOM\to A}(s_1) = \beta_{DOM\to A}(s_2) \iff en(s_1) = en(s_2) \tag{4}$$

3.4 Program vs. Data Actions

The previous abstraction step might still yield an infinite state machine. The set A of actions, while appearing finite for a snapshot of the application at a particular moment in time, for a particular user, is not. Many actions are labeled with data-dependent values: for example, in a web-mail application some links are labeled with the subjects of the emails. The insight is that each application has a control-flow skeleton that is independent of data. We call actions in A which do not depend on data *program actions*, and all others *data actions*. From a developer's point of view, program actions have labels that don't depend on inputs and are fixed in number, while data actions have dynamically-generated labels and their number can vary.

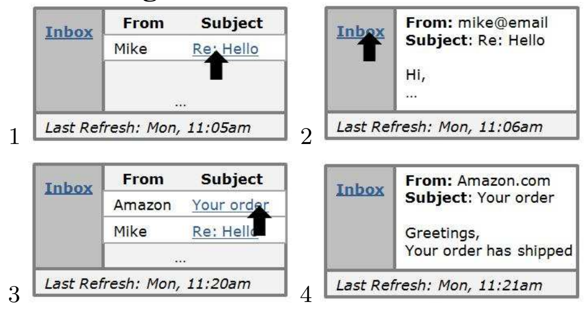
The crucial step is to define an equivalence relation $\sim$ on actions and construct the finite set $\widehat{A}$ based on the equivalence classes induced. Two data actions a_1 and a_2 constructed in the same way and playing the same role in the UI will be equivalent $a_1 \sim a_2$, so they will have a unique representative $\widehat{a_1} = \widehat{a_2} \in \widehat{A}$. Since the number of such representatives is finite, the set $\widehat{A}$ is finite.

The final abstraction step defines $\beta_{A\to\widehat{A}}$:

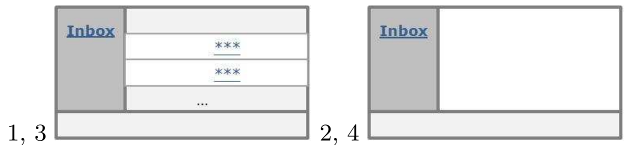
$$\sim \quad \subseteq A \times A \tag{5}$$
$$\widehat{a} = \{a' \in A \mid a \sim a'\} \tag{6}$$
$$\pi_{\widehat{A}}(a) = \widehat{a} \tag{7}$$
$$\widehat{en} : S_A \to \mathcal{P}(\widehat{A}) \tag{8}$$
$$\widehat{en}(s) =^{def} \{\widehat{a} \mid a \in en(s)\} \tag{9}$$
$$\beta_{A\to\widehat{A}}(s_1) = \beta_{A\to\widehat{A}}(s_2) \iff \widehat{en}(s_1) = \widehat{en}(s_2) \tag{10}$$

Note that ϵ-transitions are preserved. The resulting finite state machine combines the three layers of abstraction and over-approximates safely the original state machine. (Proofs for safety are easily constructed: from the abstraction β one can build the Galois connection between domains S and $S_{\widehat{A}}$, and then prove safe simulation).

3.5 The Algorithm



(b) Trace of three user actions



(c) State set $S_{\widehat{A}}$

Figure 5: Running example for Algorithm 1

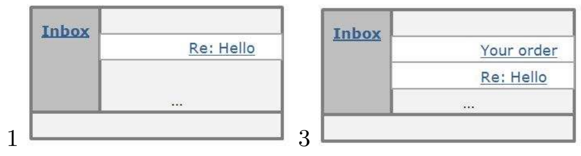


Figure 6: State abstraction without data actions

The input to our core analysis is a set of observed program transitions $\{(Page_i, a_i, Page_{i+1})\}$. We gather ours by breaking apart trace trees we observed with a Firefox extension (Section 4). The algorithm iteratively stitches these back together into a finite state machine representation.

We now illustrate how the model extraction algorithm works with with Figure 5. To simplify the notation, $\beta_{\widehat{A}}$ denotes the composition $\beta_{A\to A}(\beta_{DOM\to A}(\beta_{S\to DOM}(\cdot)))$, and $\pi_{\widehat{A}} : A \to \widehat{A}$ denotes the projection function based on the $\sim$ equivalence relation.

Let's consider a hypothetical web-mail application and the recordings of three user interactions as depicted in Figure

Algorithm 1 Model Extraction From Traces

```
Input: Set of recordings $\{(Page_i, a_i, Page_{i+1}) | i = 1..n\}$
Output: FSM $(S_{\widehat{A}}, \Delta_{\widehat{A}})$
 1: $A \leftarrow \{a_i | i = 1..n\}$
 2: $S \leftarrow \{Page_i | i = 1..n+1\}$
 3: $\widehat{A} \leftarrow \{\pi_{\widehat{A}}(a) | a \in A\}$
 4: for all states $s \in S$, $s = Page_i$ do
 5:   for all actions $\widehat{a} \in \widehat{A}$ do
 6:     if action $\widehat{a}$ matches $s$ then
 7:       $en(s) \leftarrow en(s) \cup \widehat{a}$
 8:     end if
 9:   end for
10: end for
11: $\widehat{S} \leftarrow \{\beta_{\widehat{A}}(s) | s \in S\})$
12: $\Delta_{\widehat{A}} \leftarrow \emptyset$
13: for all recordings $(Page_i, a_i, Page_{i+1})$ do
14:   $\Delta_{\widehat{A}} \leftarrow \Delta_{\widehat{A}} \cup (\beta_{\widehat{A}}(Page_i), \pi_{\widehat{A}}(a_i), \beta_{\widehat{A}}(Page_{i+1}))$
15: end for
```

5(b): the user clicks to view the first email message, returns to the Inbox and then clicks again to view a new email message (that wasn't previously available). Lines 1–2 in the algorithm identify the set of actions A and the set of states S: in our case, the three actions labelled `Re: Hello`, `Inbox` and `Your order`, and the four states shown in the figure.

Line 3 builds the set $\widehat{A}$ of data actions, using the projection function $\pi_{\widehat{A}}$. In our example, `Your order` and `Re: Hello` are instances of the same data action, namely viewing the content of a message. They will be merged together by the projection function, so $\widehat{A}$ will contain the program action labelled `Inbox` and the data action for viewing a message (labelled generically "***"). This data action matches any message, anywhere in the list. Each action is represented by its path in the DOM tree.

Lines 4–10 identify the set of enabled actions for each state, and line 11 builds the abstraction $\beta_{\widehat{A}}$, grouping together all states that have the same set of enabled actions. In our example, the program action `Inbox` matches all four states, while the data action matches only the first and third states. We therefore obtain two abstract states, as sketched in Figure 5(c). This abstraction ignores any textual differences between states, such as the "Last Refresh" line or the message body, and views a page as just a DOM skeleton containing the paths of the enabled actions.

Lines 12–15 build the transition relation for the finite state machine, by lifting each transition observed in the trace through the abstraction functions. In the example, the first user click yields a transition from the first abstract state to the second, for the data action labelled "***" matching any message subject in the list. The second click yields a transition back to the first state, for the program action `Inbox`. The third click yields the same transition as the first.

Finally, to give an intuition for the importance of the data action abstraction, let's consider what would happen without it. Our three user actions would be treated as program actions. As such, `Re: Hello` would only match the first state (since in the third state the message appears in a different position, with a different path in the DOM tree), and `Your order` would only match the third state. The single abstract state 1,3 from Figure 5(c) would be split in the two states in Figure 6. Furthermore, note that any state with a different list of email messages would not be abstracted into either one of these states; as the user receives new messages and opens them, we record program actions with different labels, causing a state explosion in the algorithm. Only the data action abstraction can merge all of them.

4. RECORDING TRACES

We created a browser extension that records a user's transitions between DOM states and feeds this training corpus into our analysis to yield a UI model. Other aggregation mechanisms are possible, such as monitoring unit testing frameworks or gathering traces from different users; our next project is to adapt directed automated random testing[24]. Our current implementation, while simple, has some subtleties. First, we must be careful in how we represent the user interface's DOM state. Second, recordings of web interactions[16] such as using a back button should not be modeled as typical, consecutive actions. Finally, we must model asynchronous function calls.

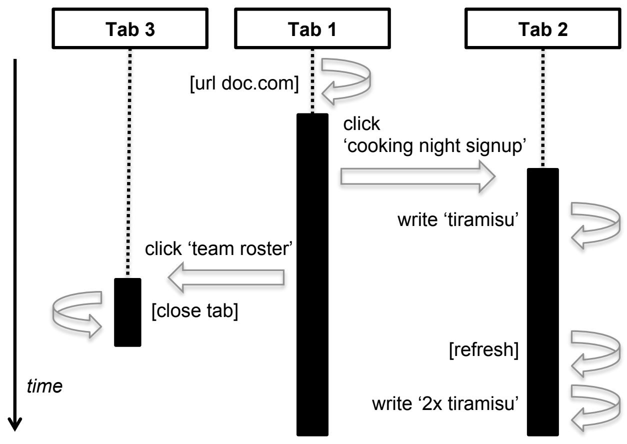
4.1 Recording a Window

Recording the state of an interface is subtle. We are interested in encoding the visual state of a window and detecting when the user interacts with it. In a survey of web PBD tools, we found most struggled to consistently record such data. The DOM tree conveniently encodes data and actions that can be acted upon, which we record, but some nodes may not be visible to the user due to sophisticated styling. We try to detect and prune such nodes; APIs do not support this so we still miss some cases. Furthermore, pages may be nested in a window using `frame` and `iframe` tags, so we stitch them together. Finally, our click handlers must not interfere with that of the application; we achieve this for programs that are not unusually sensitive to the time to handle a click.

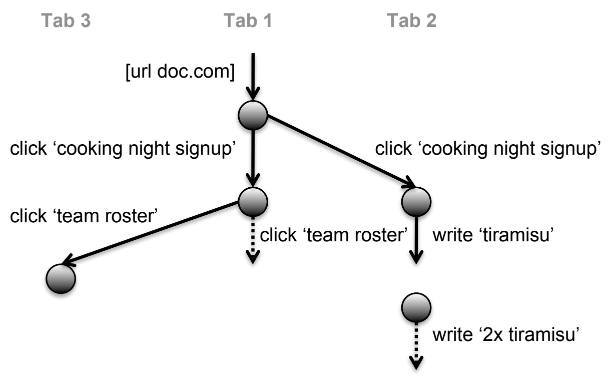
4.2 Web Interaction Traces as Trees

Web applications feature user interactions so subtle that they have even prompted the invention of many new types of continuations [19] to properly characterize them. Consider the message sequence diagram in 7(a). A user opens a list of documents, adds tiramisu to a 'cooking night signup' sheet, inspects a team roster in another tab, and then decides that more tiramisu is necessary. User interactions with the browser are denoted with [brackets], and the rest are explicit user interactions with the actual application. To emit the trace seen in 7(b), we see several interactions that are not found in typical desktop applications:

- Actions in one tab may effect another, such as opening the signup sheet document in a new window. Opening a new tab is typically more like a continuation of the full application. Thus, we model an action taking effect in a new tab as a fork in a trace, creating a trace tree, unlike others that use a string[25].
- Applications typically support browser interactions like refreshing a page or using the Back or Forward buttons. These interactions imply that there is an edge at



(a) Sequence diagram



(b) Forest of trace trees

Figure 7: Example of user interactions with the browser and the associated trace trees.

every state going back to a temporally preceding state in which a URL changed [16], so we must record all URL changes to allow implicit encoding of back, forward, and refresh buttons. In practice, there are also many spurious URL redirections, so we batch them together temporally.

- Similarly, users may manually enter in a new URL. This essentially starts a new application, even if potentially at an intermediate state. We start a new trace when this occurs.

4.3 Modeling Asynchronous Events

A final subtlety on modeling web interactions that we investigated is how to handle AJAX events. State-changing asynchronous events mostly occur during animations spread over an event loop or when communicating with a server. The former rarely impacts our notion of enabled and disabled actions, just changing visual formatting, but the latter requires more reasoning.

Consider a mail application in which a user may hit the `Next` button to view the next message and some messages are cached on the client. If the user clicks `Next` and the message is cached locally, the UI will synchronously update. However, if the message is not in the cache, an asynchronous request will be sent to the server and the user can continue interacting with the application. If the email is returned before the user interacts with the application again, it is still like a synchronous event. However, the server response can be at any time, and thus during any subsequent UI state for the same page session. An FSM model must conservatively include an ϵ-transition on all future states where the subframe that made the AJAX request maintains the same URL. Interestingly, after months recording, we never encountered a spurious user interaction between a server request and its response; AJAX usage is synchronous with respect to the UI model.

5. EVALUATION

Section 2 qualitatively shows our analysis is useful; we now quantitatively evaluate performance and effectiveness.

5.1 Performance

We separately examine our trace aggregation mechanism and the analysis itself. Our library is a standard Firefox 3.0 extension: it uses JavaScript (without JavaScript tracing optimizations) and SQLite 3.1, and we ran it on a 2.4Ghz MacbookPro laptop with 2GB of RAM and OS X 10.5.5.

5.1.1 Aggregation Complexity

Recording time is on the order of 10ms. It is dominated by the tasks of accessing a database and serializing data. These are optimizable and half the work can be taken off of a user's critical path.

Over a period of 5 months, space usage, including table indices that cause duplication of data, amounted to 1.9MB of compressed recordings per day. Over 20% of these recordings are exact duplicates, most recordings include data we do not utilize, few recordings are of large websites, and our models label many as duplicate abstract states (e.g., repeatedly viewing a news page): unoptimized, space is linear in the number of page recordings with a high constant. While not a concern for our prototype, it can be made a function of the model size ($<$ 200 states in our tests) by removing duplicates with respect to it. Alternative aggregation mechanisms, like randomly sampling different users or methodically exploring an application, are further common ways to assuage recording concerns. Recording is cheap.

5.1.2 Analysis Complexity

Our analysis is dominated by two tasks: marshalling data from our database and determining the enabled set of actions *en* on a page. Despite the high interpretation overhead of JavaScript, the rest of the steps were on the order of 1-100ms. Retrieving recordings can be largely circumvented by better storage practices and avoided in online usage (e.g., *directed* spidering[24]). Labeling actions in a recording is akin to matching CSS selectors to nodes in an HTML page. Matching one action against a page is fast (0.01-1ms), but matching against all actions ($|\widehat{A}|$) is slow because our abstraction still misses some common idioms: to be discussed next, $|\widehat{A}|$ is $0.2n$, where n is the amount of recordings to train over. Detecting badly abstracted actions, even without better abstracting them, would eliminate the dependency on n. Finally, using native code to match selectors would be faster, and in concurrent work, we are achieving 10-40x speedups over Firefox's optimized CSS selector engine[14].

In expected usage scenarios, time spent interacting with an application should dominate recording and analysis time.

5.2 Effectiveness

We analyze our blackbox analysis with the typical metrics of precision and recall. Our task is to learn abstractions over states and actions and a model of transtions over them:

$$\beta_{\widehat{A}} : S \to S_{\widehat{A}} \tag{11}$$

$$\pi_{\widehat{A}} : A \to A_{\widehat{A}} \tag{12}$$

$$\Delta_{\widehat{A}} : S_{\widehat{A}} \times A_{\widehat{A}} \times S_{\widehat{A}} \tag{13}$$

In particular, we want the following relationship to hold for most program transitions (s, a, s'):

$$(\beta_{\widehat{A}}(s), \pi_{\widehat{A}}(a), \beta_{\widehat{A}}(s)) \in \Delta_{\widehat{A}} \iff (s, a, s') \in \Delta \tag{14}$$

In the right direction, if our model predicts the transition, we want the probability that this guess is correct to be high (*precision*). Conversely, if it is a valid transition, the probability that the model predicts it should be high (*recall*).

Table 1 describes these values as well as the sizes of the learned sets of abstract states and actions for popular publicly accessible websites. A trial of a given training set size uses an untrained test set of the same size from a temporally similarly recording span, and for true negatives and false positives, recordings from randomly chosen websites.

We have two goals. First, to achieve exact precision, so it is *safe* to perform predicted sequence of actions, and high recall, so predictions are made. A large set of abstract actions is useful, which led to the "rich" heuristics: our second goal is to have as high a precision and recall rate for it as with the loose abstraction, but to infer more abstract actions.

5.2.1 *Precision*

Precision measures, if the model states a transition is possible, how often it really is. For example, if the model predicts clicking a button will show a list of emails, high precision implies clicking the button will indeed show emails. We first define true and false positive and negatives:

$$TP = \{(s, a, s') \in \Delta \mid (\beta_{\widehat{A}}(s), \pi_{\widehat{A}}(a), \beta_{\widehat{A}}(s')) \in \Delta_{\widehat{A}}\}$$

$$FP = \{(s, a, s') \notin \Delta \mid (\beta_{\widehat{A}}(s), \pi_{\widehat{A}}(a), \beta_{\widehat{A}}(s')) \in \Delta_{\widehat{A}}\}$$

$$TN = \{(s, a, s') \notin \Delta \mid \neg((\beta_{\widehat{A}}(s), \pi_{\widehat{A}}(a), \beta_{\widehat{A}}(s')) \notin \Delta_{\widehat{A}})\}$$

$$FN = \{(s, a, s') \in \Delta \mid \neg((\beta_{\widehat{A}}(s), \pi_{\widehat{A}}(a), \beta_{\widehat{A}}(s')) \in \Delta_{\widehat{A}})\}$$

Precision is simply $|TP|/(|TP| + |FP|)$. We cannot iterate over all program transitions Δ when calculating these, so we instead approximate it with observed transitions $\Delta_{DOM_\rho} \subseteq \Delta_{DOM}$ and use recordings from other sites for the set complement. Furthermore, Δ_{DOM_ρ} contains recordings we did *not* train on: our near-perfect precision would otherwise inflate all results by 10-25%.

Program actions are too exact. Precisions for them were often either 0% and 100% because few guesses were made. We slightly redefine precision in Table 1 as $|TP|/(1+|TP|+|FP|)$ to help illustrate this by penalizing rarely acting. Finally, for data actions, even when our loose abstraction was refined with more sophisticated heuristics, we maintain our extreme precision. We achieve our precision goals: transition predictions by our analysis are almost aways correct.

5.2.2 *Recall*

Recall measures, for a random valid transition, whether the model predicts it. For example, if clicking a button shows a list of emails, high recall implies the model predicts this. Formally, recall is defined as $|TP|/(|TP| + |FN|)$. An abstraction's ability to correctly predict possible transitions that were not demonstrated raises $|TP|$. Its ability to weed out impossible transitions decreases $|FN|$: coupled with model size, this suggests model predictions are non-trivial.

Just using program actions[20, 22] leads to low recall. In most cases (Table 1), we saw at most 5% recall: many applications, like calendars and email programs, are highly dynamic with respect to data. Excessive retraining over time (instead of sticking with a learned model) might alleviate this in part, but this is costly. Applications need not be data-heavy, but every one we tested was.

Our data action abstractions fared much better. As the size of a training set increases (Figure 8(a)), the number of states first quickly rises, but then only increases at a slow, constant rate. An intuition is that we cover much of the program in the rising phase, but, once the program is largely covered, there are still some actions that our abstractions do not recognize: the gradual incline is the proportion of how often these are encountered[3]. For Google Calendars, once the recall rate stabilizes (Figure 8(b)), we do not account for application idioms that occur 20% of the time, as was also the case for GMail.

We thus achieve our goals of exact precision and high enough recall to be useful and to show the viability of our approach.

6. RELATED WORK

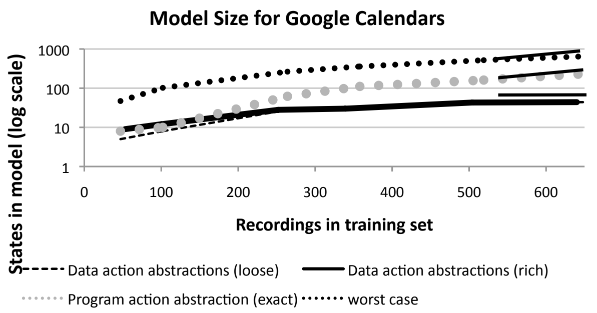
We organize our discussion first by general approaches and then by particular projects.

The dynamic analysis community has long considered variants of our problem. Learning automata from examples is an old problem [10], but doing so for complex software has only recently been gaining traction. For example, examining execution traces [1] to infer temporal specifications is successful even on operating system kernels [8]. But such approaches struggle to find exportable abstractions appropriate for downstream consumption of their analysis [2]: traces are instead used to automatically find low-level bugs. We find a high-level abstraction and are able to incorporate abstract states alongside abstract actions into our analysis; these approaches generally only use one. Finally, such approaches have only been discussed with respect to probability; while doing so would be simple for us, we more strongly motivate our analysis by making the connection to abstract interpretation when picking abstractions.

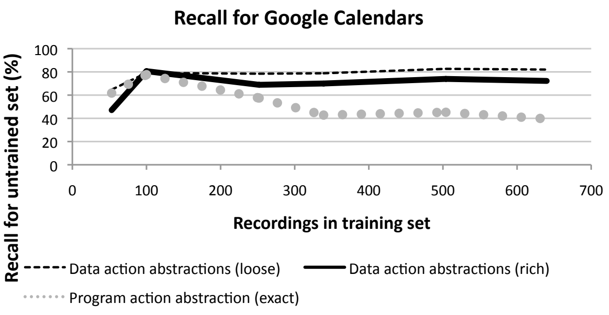
[3]E.g., over time, rare unseen features become rarer

Website	Training set	Program Actions (exact)				Data Actions (rich)				Data Actions (loose)			
		precision	recall	states	actions	p	r	s	a	p	r	s	a
`Google Documents`	492	92%	3%	303	349	99%	40%	43	146	99%	49%	47	71
	656	96%	5%	392	470	100%	48%	62	206	100%	54%	56	83
`Facebook`	251	88%	4%	172	190	92%	7%	100	138	97%	22%	87	83
	380	83%	2%	268	267	96%	10%	163	190	97%	18%	162	109
`Google Calendars`	47	95%	61%	8	17	94%	47%	9	22	95%	64%	5	10
	264	98%	57%	62	70	100%	68%	28	53	100%	78%	28	37
	358	99%	42%	111	89	100%	70%	30	77	100%	78%	34	38
`Google Search`	478	88%	2%	380	388	99%	58%	35	66	100%	74%	21	27

Table 1: Quantitative results.



(a) State explosion only with program actions



(b) Future work: 20% of user clicks are not modeled

Figure 8: Inferring a model for Google Calendar

The HCI community has long been interested in programming-by-demonstration, which again involves learning a model based on observations. One typical key difference is what we focus on extracting: we are interested in extracting the entire program model, not just the popular subset that forms a PBD specification, the latter of which we view as an interesting subproblem. Many tools even assume application developers expose such information, despite claims of genericity or of supporting the web. We isolated a useful abstraction level and showed how to automatically abstract information relevant to it: while original tools[5] that did attempt to extract models use abstractions similar to program actions, none suggest further ones like data actions. Finally, to respect web interactions, we cast traces as trees, not strings[15].

The model checking field is focused on defining program models and verifying their conformance to generic properties (the absence of entire classes of bugs such as deadlocks, race conditions, memory violation errors, etc) or user-defined properties. Abstraction is a well established technique for reducing the infinite state space that the model checker has to explore, and the latest techniques [13] are able to infer the necessary abstraction from the properties that have to be checked. We could also adjust our abstraction if downstream applications had a formal way of specifying their queries.

Crawljax [22] is a dynamic analysis tool for inferring state machines of rich, AJAX applications. It automatically explores the state space randomly clicking on pages and abstracts pages based off of the tree-edit distance between DOM trees. This abstraction is infinite in the data: it inherently fails on applications because it performs a bounded exploration of an infinite space. Furthermore, it hides small program transitions like advancing through an interactive form due to its similarity metric: this is unacceptable for tools like our natural-language command-line.

State-based testing of AJAX applications has been concurrently proposed in [18]. As with Crawljax, their abstractions do not yield rich models. Three differences in our approach are that we abstract over both states and actions, validate our abstractions over multiple popular state-of-the-art web applications, and extract models grounded in end-user semantics that we have shown to be useful. The first and last differences are fundamental: this tool works at an inappropriate abstraction level for our purposes, exporting results that are difficult to use and do not exploit important domain notions.

State-based GUI testing of desktop applications has been similarly proposed[20, 18]. When recast in our terms, it only uses program action abstractions, suffering as described in our evaluation. Repairing GUI tests (e.g., interaction sequences generated by a record-and-replay meta-application) when GUIs change is a large concern: state-based approaches have been used to detect tests broken for a particular state [11]. These lead to randomly evolving new tests to "repair" the number of tests run in a testsuite[21]. Applied to testing, our approach could repair individual tests during the common case of structural GUI changes; we are the first to model how the GUI structure transitions.

7. FUTURE WORK

First, there is room for further abstractions, such as for records[17]. Next, we want to detect hierarchical models. Also, while we can detect data, we also want to label data dependencies. Finally, to drive our work, we want to investigate further testing, HCI, and search-related applications.

8. CONCLUSION

In summary, we have shown how meta-applications may exploit *existing* user-interface through model extractors. This includes discussion of particular application domains, a prototype for a new one (getting closer to achieving natural-language interfaces), and, to make such programs robust, a precise change analysis for websites and derived change-impact analyses to help such programs repair themselves.

Our biggest technical contribution is to show that by phrasing the core model extraction problem as a control-flow analysis over user interfaces, we see that we should abstract over data, with user-manipulated objects being a key target. When picking abstractions, we show how to exploit the assumption that the frontend is an abstraction of an underlying program (a layer of abstract interpretation). Thus, while we still miss some idioms, our basic approach quickly achieves high recall on non-trivial models with 99-100% precision: we have contributed a fundamental tool to power new meta-applications and make them more robust.

9. ACKNOWLEDGEMENTS

We thank Koushik Sen for motivation and Fabian Wauthier and Lester Mackey for discussions on machine learning. Ras Bodik, Rob Ennals, Beth Trushkowsky, Benjamin Hindman and Chris Jones provided useful critiques.

10. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02*, pages 4–16, New York, NY, USA, 2002. ACM.

[2] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *PLDI '03*, pages 182–195, New York, NY, USA, 2003. ACM.

[3] T. J. Arjun Guha, Shriram Krishnamurthi. Using static analysis for ajax intrusion detection, 2009. to appear.

[4] P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A statistical approach to machine translation. *Comput. Linguist.*, 16(2):79–85, 1990.

[5] A. Cypher. Eager: programming repetitive tasks by demonstration. *Watch what I do: programming by demonstration*, pages 205–217, 1993.

[6] dapper. http://www.dappit.com.

[7] M. Dontcheva. Changes in webpage structure over time. Technical Report TR2007-04-02, UW CSE, April 2007.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[9] R. J. Ennals and M. N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*, pages 1116–1118, New York, NY, USA, 2007. ACM.

[10] J. Feldman and A. Biermann. On the synthesis of finite-state machines from samples of their behavior. In *IEEE Transactions on Computers*, pages 592–596. IEEE Computer Society, 1972.

[11] C. Fu, M. Grechanik, and Q. Xie. Inferring types of references to gui objects in test scripts. *2nd International Conference on Software Testing.* to appear.

[12] A. Halevy. http://googlewebmastercentral.blogspot.com/2008/04/crawling-through-html-forms.html.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70, New York, NY, USA, 2002. ACM.

[14] C. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser, 2009. to appear.

[15] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *CHI '08*, pages 1719–1728, New York, NY, USA, 2008. ACM.

[16] D. R. Licata and S. Krishnamurthi. Verifying interactive web programs. In *ASE '04*, pages 164–173, Washington, DC, USA, 2004. IEEE Computer Society.

[17] B. Liu, R. Grossman, and Y. Zhai. Mining data records in web pages. In *KDD '03*, pages 601–606, New York, NY, USA, 2003. ACM.

[18] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130, April 2008.

[19] J. McCarthy and S. Krishnamurthi. Interaction-safe state for the web. In *Scheme and Functional Programming, 2006*, September 2006.

[20] A. M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[21] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–36, 2008.

[22] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. *Web Engineering, International Conference on*, 0:122–134, 2008.

[23] R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. V. Kleek, D. Karger, and mc schraefel. Inky: a sloppy command line for the web with rich visual feedback. In *UIST '08*, pages 131–140, New York, NY, USA, 2008. ACM.

[24] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.

[25] A. Sugiura and Y. Koseki. Simplifying macro definition in programming by demonstration. In *UIST '96*, pages 173–182, New York, NY, USA, 1996. ACM.