Bridging the gap between JavaScript analysis and web analysis

Ben Spencer

Balliol College
Department of Computer Science
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity Term 2018

Abstract

This thesis presents my work towards web interface analysis and web data extraction by making connections between high-level user actions on web pages and the corresponding low-level implementation details.

I have developed a concolic testing platform for web-based JavaScript code, which can be used to analyse JavaScript running as part of live web pages without support from the developers. This platform is used as the basis for ArtForm, a tool which analyses web forms to infer their validation constraints. This involves simulating which actions and input values are available to a real user at the interface. ArtForm includes a suite of manual analysis tools which demonstrate the concolic testing platform and allow manual tracing of JavaScript execution for low-level debugging. The concolic platform has also been applied to the problem of event delegation, where it can be used to determine which elements on a page will respond to user actions, and should therefore be considered interactive. Finally, ArtForm can be connected to third-party analysis tools to provide advice on which values and actions are likely to lead to new code paths being explored, thus extending the client tool with low-level analysis results about the targeted site.

I also present work on optimising data extraction wrappers. By analysing the low-level HTTP calls corresponding to the user-level actions made by typical browser-based wrappers, it is possible to synthesise new, equivalent wrappers which can run without a browser. These optimised wrappers can provide a drop-in replacement for many data extraction tasks, with a huge performance benefit.

Acknowledgements

My first thanks are to my supervisor, Michael Benedikt. Michael has been a valuable mentor throughout my studies, and has always devoted a great deal of his time and attention to our discussions, both theoretical and technical, both high- and low-level. It is thanks to him that I have been able to pursue such interesting work over the past few years.

I also thank Anders Møller and Casper Svenning Jensen, who first suggested concolic testing as an avenue to explore, and helped enormously in getting the project started. Casper's collaboration was invaluable for much of the low-level implementation work on ArtForm and the vision for our concolic testing tools.

I am grateful for the generous advice and expertise from the whole of the DIADEM group, particularly from Tim Furche and Xiaonan Guo, who I worked with closely. I extend the same gratitude to the VADA group, who have also all been generous with their time and experience. In particular, I have had a great deal of help and attention from Emanuel Sallinger and Ruslan Fayzrakhmanov. Georg Gottlob has provided valuable advice, and a stream of interesting problems to work on.

I had the pleasure of working closely with Franck van Breugel, who has been a great source of enthusiastic insights and helpful discussions. Pierre Senellart not only helped to set up the project, but has provided great help and advice along the way. Both Franck and Pierre were involved in testing the tools and helping with their debugging and implementation.

Thanks also to Microsoft Research and Matthew Parkinson, who helped to set up the project and provided my initial funding, and to the VADA project and Georg Gottlob for further funding.

Thanks to my two thesis examiners, Fabian Suchanek and Dan Olteanu for the interesting discussions and helpful suggestions.

I owe my start in computer science to Tom Melham, my undergraduate tutor. Tom helped me to understand what computer science was about, and how to study effectively. He has remained a valuable mentor to this day.

Finally, thanks to all the family and friends who have supported and encouraged me along the way. Being too many to name, you can remain anonymous.

Contents

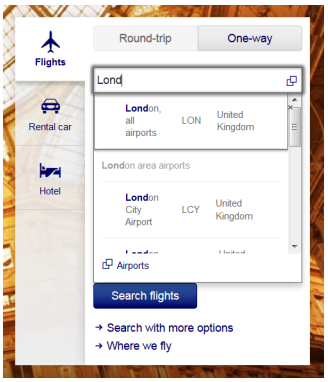
1 Introduction

Finding data on the web is important for searching, information extraction and aggregation. The massive scale of the web and the data itself means these tasks must necessarily be automated. Data is found by *web crawlers*, automated tools which scan pages looking for interesting data. This interesting data is typically accessed via human-accessible user interfaces which hinder direct access by automated crawlers. For example, airline or property websites provide free access to their data, but only via search forms; there is no public API or standard format for accessing this information. This "hidden" data makes up the *deep web* [43].

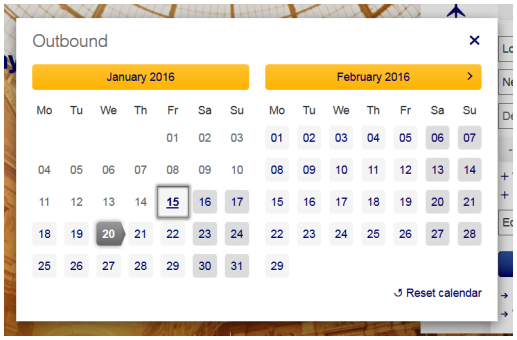
An automated tool accessing deep web data must find some sequence of user actions (for example clicking buttons or filling in input fields) which leads to a *result page* showing some data. The example search form shown in Figure 1.1 uses complex interactive user-interface elements (including drop-down lists, date pickers and tabs) and restrictions on the entered values (including the selected locations being from a certain pre-defined set, the dates being correctly formatted and numeric inputs being within a certain range). Further restrictions govern which parts of the form are mandatory and which are optional, and when different parts are shown or hidden from the user. These business logic restrictions are usually enforced in the browser directly with client-side JavaScript (for usability) and again on the server when the query is received (for security). Human-focused interfaces and input validation rules make it difficult for automated tools to correctly fill and submit the forms. *Interface understanding* is the problem of analysing the human-focused interface of a website to produce a machine-readable representation of the interface which can be used by automated tools to access the site's hidden data.

The final goal of creating these machine-readable representations is building a *wrapper*. A wrapper is a script which extracts data by performing simulated user actions on the page. Wrappers often take certain field values as arguments and extract results matching those values. For example, a wrapper for extracting flight times between certain locations (given as arguments) from an airline website would fill out the search form with the provided locations and suitable dates, then identify and extract the corresponding flight times from the result page. Thus a wrapper can be thought of as transforming the human-readable web page into a machine-readable API for accessing the site's data. The problem of creating these wrappers automatically is called *wrapper induction*.

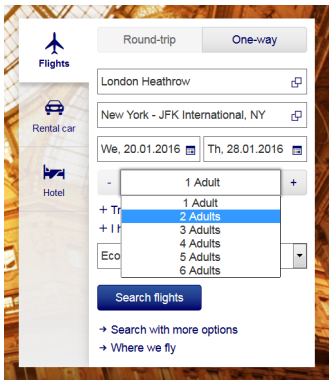
Wrapper induction applies a variety of interface understanding techniques targeting different aspects of a web page, and combines the results to create a full wrapper. As



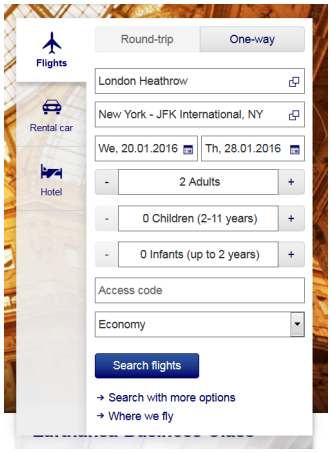
(a) An autocomplete field allows a choice of valid airports.



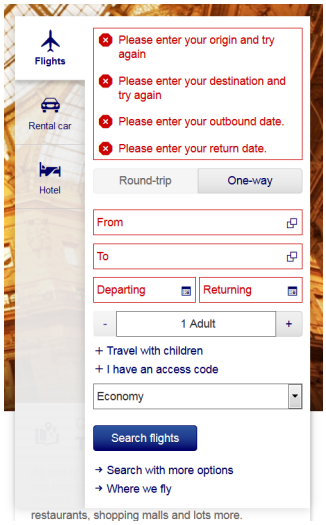
(b) A date picker is used to choose valid combinations of outbound and return dates.



(c) Customised drop-down list with added +/- buttons.



(d) Extra fields are added after certain actions.



(e) JavaScript code is used to prevent invalid submissions.

Figure 1.1: Examples of difficult input fields and validation rules on Lufthansa.com. The *Round-trip*/*One-way* toggle will change the visible fields, and the tabs on the left and the *Search with more options* link at the bottom lead to new forms. The autocomplete-based *From* and *To* fields, the date pickers, and even the fields for the numbers of passengers are customised to have different behaviour to the standard form input fields.

an example of a specific technique providing a specific type of information, Chapter 4 presents a tool which produces a symbolic description of form validation rules, showing which values will lead to a valid submission and which will be rejected. This type of analysis leads to a better understanding of how the page is expected to be used and therefore contributes to wrapper induction by showing how the generated wrapper should behave when encountering forms. In this thesis we will both look at techniques that produce these intermediate representations and also how representations can be combined for use in data-extraction tools.

The interface analysis problem is difficult because of the complex user-interface elements and input restrictions used by modern websites. Some standard interface elements, or *widgets*, have well-known behaviour and can be identified and modelled relatively easily (for example text boxes and simple links). Others are custom-built by the web developers, which is a problem because their behaviour is defined by application-specific code. An analysis tool must understand or model this code in order to understand the widget itself. Figure 1.1 shows some examples of custom user-interface widgets. The same problem applies to the inference of form validation rules; some rules are standard, such as a checkbox only permitting Boolean input values, while others rely on application-specific code, such as checking for valid airport locations in a text field. Customised interfaces are simple for human users to understand but can be very difficult for automated tools to deal with.

Interface analysis tools typically load a web page and perform simulated user actions such as clicking links and filling in forms, looking for sequences of actions which lead to interesting data or new page states. In order to do this the tool requires a model of which page elements are interactive, and how they are expected to be used. Custom widgets and input restrictions cause a problem because the automated analysis tool must discover how to interact with the page dynamically, rather than relying on the well-known behaviour of standard widgets. Even identifying which page elements represent user-interface widgets in the first place can already be difficult.

To find a sequence of user actions leading to an interesting result, an analysis tool must search a potentially huge space of possible actions and input values. For example, to choose a date as input to a web form typically requires a sequence of multiple clicks (at minimum, identifying and clicking the button to open the date picker, then identifying the numbered date buttons and clicking one). Any possible sequence of clicks anywhere on a page could potentially form part of an interesting user action filling input into such a field. Similarly, finding form input values which pass the validation rules and lead to a successful form submission requires searching the space of all possible combinations of input values to find a certain subset of "interesting" inputs. Some interactions, such as partially filling a text field and selecting an auto-complete suggestion, require choosing interesting input values and actions together, so the two problems are related and can never be fully separated.

Different tools take different approaches to web interface understanding. Many consider the user-level view: they load a web page and perform actions such as clicking links and filling in forms as a user would, looking for action sequences which lead to interesting data or new page states. A second view is the programmatic, or implementation-level view, which considers the HTML DOM, the client-side JavaScript code, the DOM events that are fired, and the HTTP requests and responses generated during an interaction. *The goal of this work is to create interface analysis tools which can span both views.* Combining information about the user-level interface and events with an analysis of the underlying implementation allows a deeper understanding of the web interface as a whole. This approach can be used, among other things, to create more effective data extraction wrappers.

1.1 JavaScript

The majority of this thesis concerns JavaScript analysis, since JavaScript represents a large portion of the implementation-level view of modern websites. Information about JavaScript is less often applied to web interface understanding than information about other aspects of the page's implementation, such as the HTML DOM, for example.

While HTML defines the structure of a web page, client-side scripting with JavaScript is used to add interactive functionality and to communicate with the web server dynamically, updating the page with new information. JavaScript is used on almost all modern websites [135] and the JavaScript executed during interaction with a page is an obvious target for web page analysis. The customised interface widgets discussed earlier are a typical example of how JavaScript is used on many sites.

Despite its importance, JavaScript is typically considered a "black box" by existing web analysis tools, which do not analyse the JavaScript code directly. Existing approaches look at the structure and contents of the web page, either via the implementation-level HTML structure or by considering the user-level visual hierarchy and position of the page elements. Test actions are executed (for example clicking buttons and entering values into form fields) and their effect on the page's structure and contents is observed. Actions with JavaScript implementations, such as button clicks, are treated as atomic transitions between different page states. These approaches only model what each action does by looking at the resulting changes to the page state.

Ignoring the implementation of page actions is a limitation because the decisions made by JavaScript about the page state are not exposed. For example, when filling a form with validation constraints enforced by JavaScript code, a black-box tool can only know that some input values were rejected and some accepted, without any *understanding* of the decision and therefore no ability to predict which new values may be accepted or rejected. JavaScript actions which cause no immediate visible change to the page state are also difficult for such tools to model. If a button click causes a change

in application state whose effect is not seen until a later action, a black-box tool will find it hard to decide whether the click had any effect, and whether the resulting behaviour in the second action was related to the first click in any way. Understanding of the interface must be supported by background knowledge and heuristics. For example a JavaScript-unaware tool might use hard-coded rules for identifying and using date pickers. The more JavaScript is used to control application state, and the more varied its effects on the page, the more limiting it becomes to model these behaviours indirectly.

By analysing JavaScript we can build a more accurate model of how a page is affected by user actions. Identifying the JavaScript code enforcing form validation, for example, could allow a JavaScript-aware tool to infer the constraints on the input values and generate new valid form submissions instead of using heuristics or random guesses. Analysing JavaScript reduces the amount of domain knowledge a tool must use to explore a page. Instead of hard-coded models of certain interface elements (such as date pickers) a JavaScript-aware tool can infer some information about how these widgets are expected to be used by looking at their implementation. Accurate models of page behaviour are useful for creating more effective wrappers, but also for assisting the analysis itself. The number of "boring" states to be searched can be reduced, and new interesting actions can be suggested which might otherwise have been missed.

As an example of a reduction in the state space to be searched, our analysis of forms (described in Chapter 4) can generate form field values which reach interesting states in the validation code, aiming for successful form submissions in particular, without requiring expensive randomised or heuristic testing. Conversely, the state space can be expanded by adding new actions which a JavaScript-unaware analysis could not find without an exhaustive search. Our analysis of event delegation (described in Section 4.8) identifies new interesting elements to be clicked which do not have event handlers registered (but nonetheless respond to a click), meaning they would not have been identified as clickable by a simpler analysis.

Analysing JavaScript code is difficult because of its many highly-dynamic features. It is hard to prove even simple properties of JavaScript programs by looking statically at their source code. A *dynamic analysis*, which executes the code being tested, can help with some of these problems. But even dynamic analysis is not straightforward; the analysis must still model JavaScript's built-in functions and objects, and normally requires a large number of test executions to find interesting behaviour.

Moreover, analysing JavaScript alone is not enough to analyse the web. When running in the context of a web page, JavaScript uses the browser and web-page APIs to read and modify the page structure, and to receive input. For example, looking up an element from JavaScript introduces a dependence on the HTML structure of that particular page, which must be understood in order for the analysis to know whether the element is found and what its properties are. Browser-based JavaScript also uses an event-driven interaction model to respond to page events and user input. These events, and their relationship with the underlying page, must be modelled in order

to know when JavaScript-based behaviour might occur. Interaction with the other languages used to implement the web page and the event-driven execution model mean that web-based JavaScript behaves significantly differently to standalone JavaScript, in ways which increase its complexity. This new complexity must be supported by any analysis tool hoping to analyse JavaScript on the web.

The ways JavaScript is used on real-world web pages causes further issues. Minimised or intentionally obfuscated code is very common, for example. JavaScript can be included on a page in multiple ways and dynamically loaded at run-time, making it difficult even to determine ahead-of-time which code will be executed. Additionally, the JavaScript found on real-world web pages commonly uses the most difficult language features for analysis tools to handle, even when they are not strictly necessary [114, 115]. Care must be taken to realistically model the real-world behaviour for interpreting JavaScript, parsing HTML and manipulating the page structure, simulating user-level and JavaScript events, and checking and modifying run-time properties such as layout information, as these behaviours in browsers do not always follow any standard. To analyse web-based JavaScript any tool needs to model the behaviour of a production web browser. Many web pages include invalid or browser-specific code which the browsers make a best-effort to interpret (Section 2.3 has more details). This leniency of the browsers is expected by web developers, so an analysis tool is forced to model a browser—as opposed to the JavaScript specification and web standards—in order to get an accurate picture of the page as it appears to a normal user.

In spite of all these difficulties, JavaScript analysis is a critical component in web interface understanding, and understanding of web forms in particular. By looking at the JavaScript code directly, instead of treating it as a black box, it is possible to build more accurate and useful models of how the page is expected to be used by a human visitor. Improved understanding of the human-focused interface is useful for creating wrappers and thus for automated data extraction.

1.2 Contributions

My work aims to bridge the gap between traditional web analysis tools and existing analysis techniques for JavaScript. Traditional web analysis tools do not consider JavaScript or the low-level implementation view of a website. On the other hand, most existing JavaScript analyses do not model the interactions between JavaScript and the web page itself, and therefore cannot be used to analyse real-world websites without significant developer support. Combining these two approaches leads to a more accurate analysis which models how a web page is expected to be used more closely and can provide more useful results for wrapper generation than either approach alone.

The first contribution of thesis is an analysis platform I have created for web-based JavaScript (detailed in Chapter 3). This platform allows the exploration of JavaScript-

based web applications in order to discover new data or functionality. It uses a dynamic analysis technique called concolic testing, along with web page and browser modelling to enable it to achieve useful results on real-world pages. The problems encountered, and the adaptations required, highlight the importance of this modelling and the pitfalls that standard JavaScript analysis tools can fall into when attempting to analyse web-based JavaScript. The concolic testing platform then forms the base of a set of more specialised analysis tools.

The next contribution is ArtForm, a tool which analyses web forms to extract the input validation constraints applied by client-side JavaScript code. It is described in Chapter 4. The concolic testing platform is used to generate new input values for web forms. This is done by analysing previous executions of form-related code, and extending the platform to model form input fields and interactions with the DOM. After introducing the core functionality of ArtForm, I present some applications and extensions of it:

- Artform was used to implement some manual web page analysis tools, described in Section 4.6. These were used to investigate which JavaScript features and patterns were common on real sites and to aid the development of the automated analysis. They are also useful in their own right for investigating and debugging a web page's implementation.

- ArtForm's concolic testing of web forms is made available to third-party tools via a *concolic advice mode*, described in Section 4.7. This allows ArtForm's analysis results to be shared with other tools, and could lead to combined tools uniting high- and low- level analysis information for a more effective analysis overall.

- The platform's concolic testing is also applied to detecting new events available on a web page, addressing a pattern known as *event delegation* (Section 4.8). We also model element visibility to determine which events a user can or cannot trigger, and thus which sequences of events the analysis tool should be allowed to explore.

The final contribution applies the same broad idea as the previous ones—the combination of information about user-level actions and implementation-level details—to a different problem in web interface analysis, that of optimising existing data extraction wrappers. I present the FastWrap system, which is able to analyse the HTTP requests made by traditional browser-based wrappers and generate an equivalent wrapper which does not use a browser and makes the required HTTP requests directly. This results in more than an order of magnitude improvement in run-time performance when executing the wrappers.

1.3 Thesis structure

Chapter 2 covers the setting for my work and provides a brief introduction to modern website implementation.

Chapter 3 introduces concolic testing, and presents my concolic testing engine for JavaScript, along with the adaptations required to analyse web-based JavaScript. Our platform is compared with Jalangi, another JavaScript concolic testing tool.

Chapter 4 describes ArtForm, a tool which analyses the validation code attached to web forms by making test submissions and generating new input values, aiming for a successful submission. The result is a symbolic representation of the input validation rules enforced on the form. Several extensions of ArtForm which apply its symbolic tracing and concolic testing infrastructure to other problems are described: in Section 4.6, the manual page analysis tools based on ArtForm; in Section 4.7, a concolic advice server providing analysis results to third-party tools; and in Section 4.8, an analysis of user-realisable events, encompassing event delegation and element visibility. Finally, ArtForm is compared with two other web application analysis tools, Artemis and Crawljax, and tested on a selection of real airline search forms.

Chapter 5 demonstrates a different aspect of web information extraction, that of optimising data extraction wrappers by analysing the low-level HTTP interactions which the browser uses to communicate with—and receive data from—the web server.

My work on ArtForm was presented at the International Conference on Web Engineering (ICWE) 2018 [125]. A demonstration paper showing ArtForm's symbolic tracing and concolic testing modes was presented at the International Symposium on Software Testing and Analysis (ISSTA) 2017 [124]. A screencast of the demo is available at:

- http://www.cs.ox.ac.uk/projects/ArtForm/demo/

ArtForm is also open source. More information is available at:

- http://www.cs.ox.ac.uk/projects/ArtForm/
- https://github.com/cs-au-dk/Artemis/blob/master/ArtForm.md

The work on FastWrap and its automatic wrapper optimisation was presented at The Web Conference (WWW) 2018 [41].

2 Background

This chapter covers the background information necessary to understand the technical parts of the thesis. First, we will give an outline of JavaScript itself and how modern websites are built, with a focus on the features which are important to my interface analysis work. Second, we will provide the background in program analysis required to discuss the analysis techniques used in my work.

2.1 The JavaScript language

For this thesis it is not necessary to have a detailed understanding of JavaScript as a whole, but some background is required to explain the examples and the particular features relating to my work. Some relevant examples are shown in the next section.

JavaScript is a dynamic, weakly-typed language used primarily for scripting on the web, but also as a general-purpose language elsewhere. It includes imperative, object-oriented and functional features and is often thought of as "multi-paradigm". Because of its history as a browser scripting language, JavaScript itself does not include certain core features, such as input and output, which are instead provided by its environment.

JavaScript's data types are objects, strings, Booleans, and numbers,[1] as well as the special values `null` and `undefined`. Arrays and functions are both considered objects, but have some special syntax and behaviour. Wrapper objects are also created on-the-fly so that primitive values can appear to have properties: `"Hello world".length` is valid and returns 11. JavaScript will silently convert between types as needed, although the rules are sometimes unintuitive. For example `"My age is "+20` produces a string "My age is 20" and `"5"+"2"` produces "52", but for `"5"*"2"` both strings are converted to integers and the result is the number 10, and `!"Hello"` converts the string to Boolean **true** and therefore results in **false**. This type of coercion is very common in JavaScript, and "truthy" values such as arrays, objects and strings are often used in `if` statement conditions as shorthand for checking if the values are defined and non-empty.

Objects are unordered collections of *properties*, a mapping between property names (which are strings) and values (which can be of any type). An object's methods are simply properties whose value is a function object. Property values can be updated at runtime by assigning to them, and properties themselves can be added or removed. Accessing undefined properties is not an error, the special value `undefined` is returned.

[1] All numeric values are represented as IEEE 754 64-bit floats.

Functions are first-class values in JavaScript. They are simply a special type of object, and so can be stored in variables, passed to other functions as arguments, and so on. Functions are either defined by name (with the function keyword, similarly to many other languages), or given as function literals. The built-in eval function and Function objects can also be used to execute strings containing JavaScript code.

Regular expressions are a built-in part of the language, provided by the RegExp class. They can be used for both matching and replacement. As in many languages, JavaScript's regular expressions are not limited to matching regular languages. References allow matches of repeated patterns and make the RegExp language much more powerful. This extra complexity can be problematic when solving constraints involving regular expressions. For this thesis regular expressions will always refer to the RegExp implementation in JavaScript and not theoretical regular languages.

JavaScript is increasingly used outside of client-side browser scripting as well. Server-side programming with JavaScript is becoming common as it allows the same developers to work on both parts of a web application and share related code. JavaScript, HTML and CSS is one of the three supported platforms, along with .NET and C++, for Windows 8 and Windows 10 modern application development, so many desktop applications are now written using these technologies. As such, even outside of the traditional web setting, understanding JavaScript code is still important.

2.2 Modern website implementation

Before discussing web analysis, it is useful to explain something about how modern websites, and web forms in particular, are implemented. I will show how simple static pages are created, how users can send data back to the web server using web forms, how JavaScript is used to make the client-side pages more interactive and dynamic, and then how it is applied to web forms. Finally I describe how dynamic two-way communication with the server is implemented, which allows the creation of modern, highly-interactive web applications.

Static pages using HTML and CSS

Content on a web page is structured using *HTML* (hyper-text markup language) [62]. HTML provides the structure of the page, by enclosing the content in different HTML *elements*. For example <p> denotes a paragraph of text, <h1> a heading, <li> a list item, and so on. For example, <h1>My Web Page</h1> represents a level-1 heading. The elements are nested in a tree structure, ideally reflecting the logical structure of the content, with the <html> element at the root. There are also generic container elements and with no particular meaning which are used to group related content together, or to identify certain content to apply style or behaviour changes later.

Elements can be assigned attributes which either change their behaviour or provide a way to identify the element when making layout or functionality changes later. An example of a behaviour-changing attribute is `href` for a (link) elements, which sets the target of the link. The `id` and `class` attributes are valid on any elements and are used to identify them in CSS or JavaScript. An `id` uniquely identifies a single element, whereas a `class` can be used on multiple elements to identify them all at once.

CSS (cascading style sheets) is a declarative language specifying how an HTML document should look when rendered in a web browser [10]. A stylesheet is simply a list of *CSS rules*, included either in the HTML file via a `style` element or more commonly via a `link` element which includes a standalone CSS file. The rules are given as a *selector* which matches certain elements on the page and a list of *declarations* [25]. The declarations specify which properties of the selected element should be changed, and the value they should be set to. For example, the rule `h1,p {color:#002147;}` sets the color property, which controls the text colour, of all `h1` and `p` elements to the value `#002147`, a dark blue. The selector is `h1,p` and matches all `h1` and `p` elements in the document. The values of the elements' CSS properties are used by the browser to render the page.

Selectors can filter elements by name, by their attributes, or by their position in the tree of elements. For example `p.important` selects all `p` elements whose `class` attribute includes the class "important", and `a[hrefˆ="https://"]` selects all link elements which link to secure pages (by matching `href` attributes beginning with the string "https://"). Selectors which are more specific,[2] or appear later in the style-sheet can override earlier definitions.

Communication with the server using web forms

When a web page is accessed, data (i.e. the page's HTML and other page assets) is sent one-way from the server to the client web browser. Two-way communication, where data is also sent from the client back to the server, is implemented in HTML using *forms*. Sending data back to the server with forms enables interactive web pages with dynamic content which can be updated by the website's users. Forms consist of an outer `form` element and a set of form input elements. The input elements are shown by the browser as interactive controls: text boxes, drop-down lists, and so on. Figures 2.1 and 2.2 show how a simple HTML form works, and how it is handled by the server, respectively. The most common form input elements are listed in Table 2.1.

When the form is filled in, the input controls are updated to show their new values, but nothing is sent to the server until the form is submitted. Once the form is submitted (typically by clicking the submit button), the name and value of each form field is sent to the server at the URL given in the `form` element's `action` attribute. The data format is controlled by the form's `method` attribute, which can be set to "GET" or "POST". For

[2] CSS defines particular *specificity* rules, but broadly a selector which adds more restrictions on the elements it will match is considered more specific.

```
<!doctype html>
<html>
<head>
  <title>Simple Form Example</title>
  <link rel="stylesheet" type="text/css" href="simple.css" >
</head>
<body>
  <h1>Simple Form Example</h1>
  <form action="simple-form-handler.php" method="POST" id="simple-form" >
    <label for="name" >Name:</label>
    <input type="text" id="name" name="name" ><br>

    <label for="age" >Age:</label>
    <input type="text" id="age" name="age" ><br>

    <label for="department" >Department:</label>
    <select id="department" name="department" >
      <option value="cs" >Computer Science</option>
      <option value="eng" >Engineering</option>
      <option value="maths" >Mathematics</option>
    </select><br>

    <label for="terms" >Accept terms:</label>
    <input type="checkbox" id="terms" name="terms" ><br>

    <input type="submit" value="Submit" id="button" >
  </form>
</body>
</html>
```

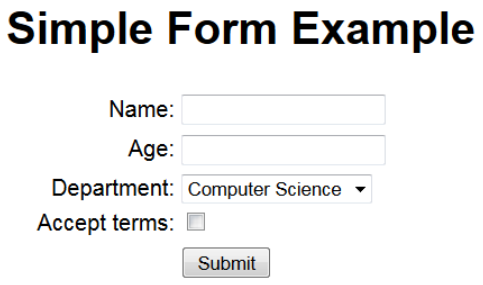
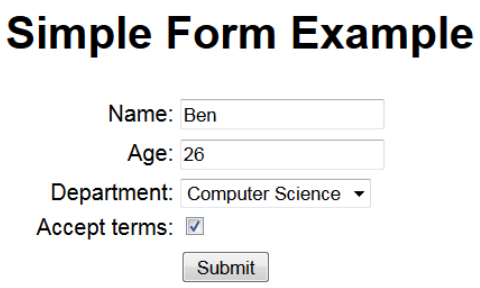


Figure 2.1: A simple HTML form, consisting of four input fields and a submit button. When the button is clicked, the browser sends the form data to the URL simple-form-handler.php. This server-side form handler is shown in Figure 2.2.

simple-form-handler.php

```
<!doctype html>
<html>
<head>
  <title>Form Response</title>
  <link rel="stylesheet" type="text/css" href="simple.css" >
</head>
<body>
  <h1>Form Response</h1>
  <?php
    if (count($_POST) > 0) {
      print "<p>Your submitted data is as follows:</p>\n";
      print "<table cellpadding='0' cellspacing='0' >\n";
      foreach ($_POST as $key => $value) {
        print "<tr><td>" . $key . "</td><td>" . $value . "</td></tr>\n";
      }
      print "</table>\n";
    } else {
      print "<p>No POST data submitted.</p>\n";
    }
  ?>
</body>
</html>
```

The form page before submission:



The result page after submission:

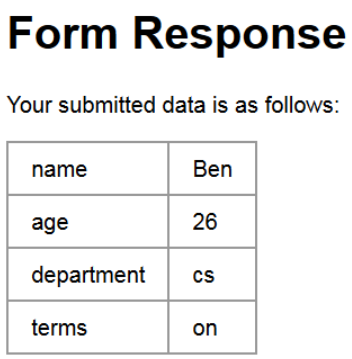


Figure 2.2: When the form from Figure 2.1 is submitted, the form data is sent to the server as a POST request. This example shows an HTML template with PHP code to receive and display this POST data from our sample form. In a real application, the server would most likely perform some validation on the received data and save it to a database, or use it to retrieve search results from the database.

Table 2.1: Examples of the most commonly used form input elements available in HTML. There are many more input types and form-related elements available [102, 103].

Element	Description	Example
<input type="text">	A simple text box for a single line of text.	Name:
<input type="checkbox">	An on/off tick-box.	Red Green Blue
<input type="radio">	A group of radio buttons where exactly one must be selected.	Large Medium Small
<select> and <option>	A drop-down list allowing a single item to be selected.[3]	Department: Computer Science Computer Science Engineering Mathematics
<textarea>	A larger text box for multi-line text input.	Comments:
<input type="submit">, <input type="button">, and <button>	Buttons whose default behaviour is to submit the form.	Submit
<input type="hidden">	No input field is shown on the page. Hidden inputs are used to include extra data when the form is submitted to the server.	

[3]The multiple attribute can be set to allow selecting multiple values from a single field, but this is rare in practice.

a GET request, which is typically used for search and query forms, the data is added to the URL of the page being requested. For example, if the form from Figure 2.1 used method="GET", then the browser would load the following URL:

simple-form-handler.php?name=Ben&age=26&department=cs&terms=on

The server parses this URL to retrieve the key/value pairs of the submitted data, and shows an appropriate result page. With a POST request—typically used for more private information or forms which will modify the server state—the browser would load the URL simple-form-handler.php and include a separate field in the request for the POST data. This includes the same set of key/value pairs, but can accommodate much more data than can be encoded into a URL (which is useful for file uploads, for example). In this case, the server parses the POST data field to retrieve the key/value pairs for the submitted data, as shown in the server-side PHP script in Figure 2.2.

With HTML forms, a web page is able to receive content from its visitors; the communication is no longer restricted to one-way downloads of static pages. Dynamic websites, where the content can be queried, modified or created by their users, are enabled by this type of two-way communication.

Dynamic and interactive pages using client-side JavaScript

Just as HTML and CSS define the structure and appearance of elements in a web page, JavaScript is used to specify their behaviour [83]. JavaScript (formally specified as ECMAScript) is a high-level, dynamic scripting language which runs in the client-side web browser. It is included, similarly to CSS stylesheets, either directly in a script element or loaded from a standalone JavaScript file.

JavaScript interacts with the web page via the DOM (document object model) interface [77]. The DOM allows access and modification of the tree hierarchy of HTML elements (represented as DOM nodes), their attributes (via object properties of the nodes) and other properties (such as style and content).

HTML element attributes and DOM node properties correspond closely (and in fact the terms are often used interchangeably). The id property for example, contains the value of the id attribute of the corresponding element. Some DOM properties do not represent HTML attributes, such as tagName which returns the element name (e.g. “DIV" for a div element) although there is no such attribute in the HTML code itself. There are even a few properties which have a subtly different meaning to the corresponding attribute. A form input field has a value attribute, specifying the *initial* value to be pre-filled in the field. However, the value property of the corresponding DOM node represents the *current* value of the input field and will change as the user types input.

JavaScript runs in the browser using an *event-driven* model. All JavaScript execution is in response to some event on the page—a user interaction, a timer, other content

being loaded, and so on. JavaScript functions called *event handlers* are associated with these browser events using the DOM API call addEventListener.[4] The events can include clicks, mouse movements, key presses, form input or submission, scrolling, and many more. When the specified event occurs, the browser executes the corresponding JavaScript event handler function. Many events are triggered by the user's actions (such as those listed above), but some are triggered automatically by the browser, for instance timers or page-load events. JavaScript's asynchronous, event-driven execution model allows developers to create dynamic, interactive web pages which respond to user and external actions.

Figure 2.3 shows how simple events are registered. In this example, two pop-up messages (called *alerts*) are shown; one when the page has finished loading, and one when a button is clicked. There are two event handlers: one for the load event on the window object, triggered when the page has finished loading; and one for the click event on the button element. The script will begin executing as soon as it is loaded, which may be before the rest of the page has finished loading. Therefore the click handler must be registered from *within* the page-load event handler, once we know the button which the event is supposed to attach to will be loaded. If getElementById(*"button"*) was called immediately, without the page-load handler, the button is not guaranteed to be in the DOM and the click event may not be registered correctly.

Interactive forms using JavaScript

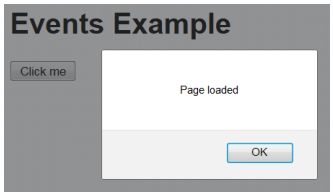
JavaScript is often used together with web forms to create an interactive user-interface for a website. Data entered into forms or other input controls is typically checked against input validation rules to prevent invalid or malicious queries or database updates being made. To prevent abuse the rules must be enforced by the server, but they are typically also checked by client-side JavaScript as well to provide immediate feedback to the user and a smoother user-experience. To implement client-side validation checks, a validation function is registered as the click event handler for the form's submit button, the submit event handler for the form itself, or the change event handler for a specific field. When the user attempts to submit the form, the validation function can check that the given input values match the validation rules and prevent submission if not, usually showing an error message.

Figure 2.4 shows how the simple form shown earlier can be extended with a validation function to check the user-supplied inputs. In this case, the validate function will be called when the submit button is clicked. It uses the DOM API to look up the current values of the form fields, checks their validity, and adds messages to the page to report any errors. The error handler adds the classes error and error-msg to certain elements, which are matched in CSS to highlight those elements. The call to evt.preventDefault()

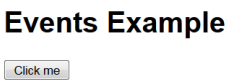
[4]Event handlers can also be registered via HTML attributes, such as onclick, or the attachEvent method in Internet Explorer.

```
1:  <!doctype html>
2:  <html>
3:  <head>
4:    <title>Events Example</title>
5:    <link rel="stylesheet" type="text/css" href="simple.css" >
6:    <script type="text/javascript" >
7:      window.addEventListener("load", function(e) {
8:        alert("Page loaded");
9:        var btn = document.getElementById("button");
10:       btn.addEventListener("click", function(e) {
11:         alert("Button clicked");
12:       });
13:     });
14:   </script>
15: </head>
16: <body>
17:   <h1>Events Example</h1>
18:   <button id="button" >Click me</button>
19: </body>
20: </html>
```

The alert once the page has loaded:



The page before clicking the button:



The alert after clicking the button:

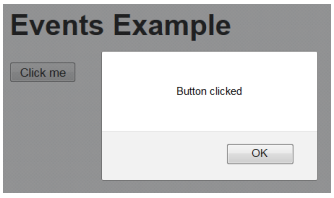


Figure 2.3: This example shows how event listeners are registered in JavaScript. An event listener is added for the page-load event and for clicks on the button.

cancels the default form submission action which would usually be triggered by clicking the submit button.

Many web developers customise their forms using JavaScript in order to change their default functionality or appearance. Standard HTML elements are re-purposed with appropriate JavaScript event handlers to create custom-made form controls. Custom fields are used either because HTML does not provide standard controls for those types of input or because they allow more control to the developer about exactly how the control looks or works.

A common example of a custom-made input field is a date-picker. Many forms use links and images to create a calendar widget where the user can enter a date. Another common example is adding an autocomplete list to a text input field which suggests possible values to enter. Even simple text boxes are sometimes replaced with custom versions, when the developer wishes to significantly alter their design. Some examples of custom form fields are shown in Figure 1.1, on page 8.

Dynamic client-server communication using AJAX

Traditional static HTML forms can send data back to the web server, but this requires form fields to be filled out and a fresh page-load. More dynamic and interactive communication between the client and server can be implemented in JavaScript instead using *AJAX*.[5] AJAX refers to JavaScript running on a web page communicating directly with a remote server to display dynamically changing content. This is done using an API called XMLHttpRequest[6] which sends a request and triggers a callback function once the server returns a response. Figure 2.5 shows an example of XMLHttpRequest being used to fetch data dynamically. The request is asynchronous: it is set up in the button click handler (and the "Loading…" message is displayed) and then control returns to the browser. Once the request is completed, the onload method of the XMLHttpRequest object is called which displays the returned data. Another more recent, but as-yet much less common, method for dynamic communication with the server is *WebSockets*, which can even provide persistent two-way connections.

Using AJAX to dynamically load and update content allows the creation of modern *web applications*. These web pages use live-updated content from the remote server to behave like a traditional desktop application, with some processing done client-side by JavaScript in the browser and some done server-side (in particular fetching data from the application's database). Web applications—such as Gmail, Google Docs, Facebook, and many other smaller examples—are becoming very common (as evidenced by the increasing use of web application frameworks such as Angular and React [134]) and are an obvious motivation for analysing client-side JavaScript.

[5] AJAX stands for Asynchronous JavaScript And XML but today refers to any data fetched from a server dynamically by JavaScript, however it is formatted.

[6] Again, the name is historical; an XMLHttpRequest call doesn't necessarily send or receive XML data.

simple-form-dom.js

```
function validate(evt) {
  // Clear any existing error messages.
  var messages = document.getElementsByClassName("error-msg");
  while (messages.length > 0) { messages[0].remove(); }
  var errors = document.getElementsByClassName("error");
  while (errors.length > 0) { errors[0].classList.remove("error"); }

  var name = document.getElementById("name");
  var age = document.getElementById("age");
  var terms = document.getElementById("terms");

  // Check name is not too short
  if (name.value.length < 3) {
    error(name, "Name must be at least 3 characters.");
    evt.preventDefault();
  }
  // Check the age is an integer in the correct range.
  if ((parseInt(age.value, 10) || 0) < 18) {
    error(age, "Age must be a number and at least 18.");
    evt.preventDefault();
  }
  // Check the terms checkbox is ticked.
  if (!terms.checked) {
    error(terms, "The terms must be accepted.");
    evt.preventDefault();
  }
}
function error(element, message) {
  var errorElt = document.createElement("span");
  errorElt.className = "error-msg";
  errorElt.textContent = message;
  element.parentNode.insertBefore(errorElt, element.nextSibling);
  element.classList.add("error");
}
window.addEventListener("load", function() {
  var button = document.getElementById("button");
  button.addEventListener("click", validate);
});
```

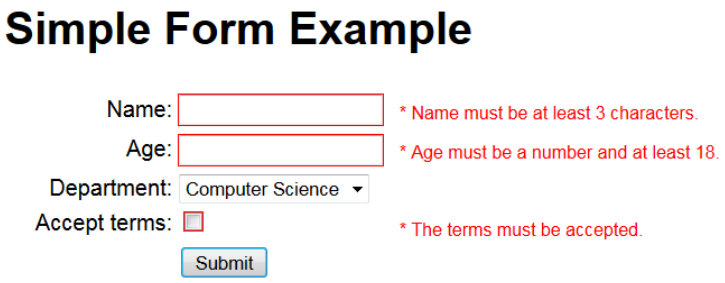


Figure 2.4: The simple HTML form shown earlier is extended with JavaScript to validate the input before the form can be submitted.

```
<!doctype html>
<html>
<head>
  <title>XHR Example</title>
  <link rel="stylesheet" type="text/css" href="simple.css" >
  <script type="text/javascript" >
    window.addEventListener("load", function(e) {
      var btn = document.getElementById("button");
      var result = document.getElementById("returned-data");
      btn.addEventListener("click", function(e) {
        result.textContent = "Loading...";
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "http://httpbin.org/user-agent", true);
        xhr.onload = function() {
          result.textContent = this.responseText;
        };
        var xhr_connection_fail_reported = false
        xhr.onreadystatechange = function() {
          if (this.readyState == XMLHttpRequest.DONE &&
              this.status == 0 && !xhr_connection_fail_reported) {
            xhr_connection_fail_reported = true;
            result.textContent = "XHR connection failed.";
          }
        };
        xhr.send();
      });
    });
  </script>
</head>
<body>
  <h1>XHR Example</h1>
  <button id="button" >Send Request</button>
  <p>Returned data:</p>
  <pre id="returned-data"></pre>
</body>
</html>
```

Before the button has been clicked:

While the AJAX request executes:

After the result has been returned:

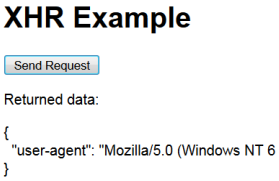


Figure 2.5: An example of XMLHttpRequest being used to load data from an external server and display it.

2.3 The problems with real-world websites

So far, I have outlined how websites are implemented with HTML, CSS and JavaScript. In the real world, websites are much more complex. This makes it more difficult—both for humans and automated tools—to analyse how they work. The reasons why analysis is more difficult for real sites can be grouped into the following categories:

- *Scale and complexity.* Modern websites, and web applications in particular, use a lot of JavaScript. It is difficult to track which JavaScript events correspond to which DOM elements, which actions could trigger a certain piece of code, or conversely which code may be executed when a certain action is triggered. Dynamic loading of JavaScript from different sources makes it hard to even enumerate the JavaScript files involved in an application.

- *Non-standard code.* JavaScript and the other web technologies have evolved over many years, while the browser vendors and web developers have each tried to maintain compatibility. This means the browsers do not implement any strict standards, and the websites often use outdated or invalid code to work around browser bugs or inconsistencies. Both sides of this issue must be addressed by any real-world web analysis tool.

- *Frameworks.* Large CSS and JavaScript frameworks are often used to mitigate browser differences and provide additional functionality. This introduces a large body of highly-optimised, browser-dependent code, even on otherwise simple sites. Due to the wide use of these libraries, JavaScript's most complex and difficult to analyse language features are commonly encountered on real sites.

- *Generated code.* Many sites are developed in tools which auto-generate the HTML, CSS and JavaScript, or translate business logic written in one language into JavaScript code which can be embedded in a website. This makes it difficult to exploit common patterns which human programmers might use.

Each of these issues is discussed in more detail below.

Large and complex web applications

In the past, web pages used static HTML for most of their structure and functionality, with a small amount of JavaScript and dynamic content to add interactivity. Today, almost all websites use JavaScript, either for their own implementation or from third-party additions such as advertising or analytics packages, and more often for both. According to a survey of around 10 million websites from W3Techs, 94.9% of websites use JavaScript [135]. With the rise of web applications, the majority of a site's functionality

is often provided dynamically via JavaScript, with dynamically generated HTML only used to display the results in the browser.

This trend means that a lot of code is used in the implementation of even apparently simple sites, and the code is often complex. Code used for page structure, managing content and layout, and communicating with the server contains no interesting information about the functionality of the website itself, making it irrelevant noise for many web analysis tasks. The large (and increasing) amount of this "uninteresting" but complex JavaScript makes identifying the interesting parts for analysis very difficult.

Changing standards, invalid code and lenient browsers

As the standards for HTML and the web have changed over time, browsers supporting the new features have tried to maintain backwards compatibility with older standards and conventions. Browsers do not try to differentiate different versions of the HTML, CSS and JavaScript standards as a compiler might for other languages (and in the early days these technologies were not standardised at all, the only consensus was by convention), they have a single browser engine which attempts to handle any website. As new standards emerge, they are supported partially and incrementally by browsers, without necessarily supporting all the new features, or even the same ones as other browsers. Because of changing standards, backwards compatibility and the attempt to make a "best effort" to parse whatever code they find, modern web browsers are extremely lenient in what they accept as valid code [133].

Because the browsers are so lenient, there is little incentive for web developers to strictly follow the standards. In fact, they are often forced to write non-standard or invalid code to get the desired effect from the browser. Patching over the differences between different browsers is one of the main reasons for this, as discussed below. The prevalence of non-standard code is illustrated by a survey of 832 URLs chosen at random from the Common Crawl index [30] and tested against the W3C validity checker [110]. Of these sites, only 1.8% were found to have no validation errors, and only 0.6% had neither errors nor warnings. Although this metric is unnecessarily strict, it does illustrate the scale of the divergence between real websites and the formal standards.

Invalid code without strict standards is difficult to analyse. When looking at a web page which uses invalid code or non-standard features, the language standards obviously cannot define its behaviour. In these cases it is necessary to model the behaviour of a real production web browser, or even use one directly, to know what the code does. Even accurately modelling a real browser does not completely solve the problem, as different browsers can behave in different ways on non-standard (and even perfectly valid) pages.

Browser differences, browser detection and hacks

One of the main reasons for non-standard or invalid code, and for the complexity of some library code, is to work around browser inconsistencies. A web page has visitors using many different web browsers, each using a different HTML and CSS rendering engine and JavaScript interpreter. Either intentionally or due to bugs, each browser behaves slightly differently and displays the same page in a slightly different way. This was especially true in older browsers, but is still the case to a lesser extent today. These differences affect the JavaScript APIs available and how they work, as well as the CSS rules supported and how they are applied to lay out the page.

To get around these inconsistencies web developers either restrict themselves to the common subset of matching features (which may be difficult) or detect the client browser and provide extra code or rules for each browser to make sure they all behave the same way. *Feature detection* is used to detect different browsers by checking whether a certain feature exists or behaves in a way which is known to be unique to a certain subset of browsers.

For example in Internet Explorer before version 9, the addEventListener function for registering events in JavaScript was not present and those browsers used attachEvent to get a similar (but not identical) result. To register events which work in any browser requires checking in JavaScript which method is defined:

```
if (element.addEventListener) {
  element.addEventListener('click', myClickHandler); // IE >= 9 and others
} else {
  element.attachEvent('onclick', myClickHandler); // IE <= 8
}
```

In CSS, these browser detection tricks are known as *hacks* because they exploit differences in the CSS support in each browser to apply different rules to each browser. The most famous example is known as the "holly hack" to apply certain CSS rules only for Internet Explorer 6 and below. Those browsers add an extra element enclosing the html element which can be matched in CSS. As all other browsers consider html to be the root of the DOM tree, a selector which relies on html having a parent element can only ever match in IE6.

```
h1 {
  color: blue; /* IE >= 7 and others */
}
* html h1 {
  color: red; /* IE6 */
}
```

The most common way to mitigate browser differences today is by using a third-party library like jQuery which uses these tricks internally to abstract the browser inconsistencies away from the developer, as discussed below. Alternatively, the browser tests are bundled in special-purpose libraries such as Modernizr.[7]

Browser-specific code is problematic for analysis for three reasons. First, the code is complicated in a way which does not represent any interesting fact about the behaviour of the website. The JavaScript example above introduces an interesting-looking branch in event-handing code which is in fact just a workaround for browser compatibility and is completely boring for someone analysing the code. Second, the code used for feature detection is often not valid according to any standard. This makes it necessary to model a real web browser instead of the language specification in order to interpret ambiguous or invalid code. This relates to the final issue, which is that any analysis must pick a certain interpretation for this ambiguous code, so it will miss behaviour designed for other browsers.

Rozzle is a JavaScript malware detection tool which avoids this issue in a novel way. JavaScript malware often uses browser- or plugin-specific tricks to target specific browsers; when it encounters a browser check, Rozzle is able to execute both branches of the code, in order to explore all possible behaviour under different browsers and environments, at the cost of a less precise analysis [79].

Frameworks and libraries

JavaScript and CSS frameworks are used to simplify the development of complex, browser-independent websites.

The CSS frameworks, such as Bootstrap,[8] Foundation[9] or 960gs,[10] consist of pre-defined CSS rules which simplify laying out a website. The developer adds appropriate class attributes to their HTML which the framework's rules use to format the content, often within a grid system. The frameworks also provide pre-styled elements such as buttons, tables, forms, and so on.

JavaScript DOM libraries, such as jQuery,[11] Prototype[12] or MooTools,[13] provide an abstraction of the DOM API which is simpler to use, more consistent, or more browser-independent. CSS selectors are used to retrieve certain sets of elements for processing, which is a much simpler method than navigating the DOM tree manually with JavaScript alone. For example to attach a click handler to all button elements with the class clickable becomes very simple using jQuery:

```
$("button.clickable").on("click", function(e) { $(this).text("Clicked!"); });
```

[7] https://modernizr.com/
[8] http://getbootstrap.com/
[9] http://foundation.zurb.com/
[10] http://960.gs/
[11] https://jquery.com/
[12] http://prototypejs.org/
[13] http://mootools.net/

Other JavaScript libraries focus on user-interface widgets, for example jQuery UI[14] or DOJO.[15] These "widget toolkits" include interactive page elements such as draggable or resizable boxes, tabbed pages, accordion containers (which show a single block of content at a time from a list) and buttons. Customised form input controls are also common, such as autocomplete text-boxes, date pickers, or sliders.

There are also frameworks focused on web application infrastructure and dynamic page updates with data from the server. These include AngularJS,[16] Backbone[17] and Ember.[18] They focus on the problem of keeping the DOM elements and contents synchronised with the internal application state and updating the DOM in response to changes in that state.

It should be noted that there is a lot of overlap between the different types of frameworks. For example jQuery includes some widgets, most CSS frameworks include JavaScript components, and the web-application frameworks include DOM manipulation methods.

The HTML "scaffolding" required for the CSS and JavaScript frameworks to hook into means that even simple pages end up using complex DOM structures. Apparently simple pages can use complex custom widgets and have dynamically modified contents. Because the libraries include browser-independence hacks, seemingly straightforward application-level code which calls the library can result in complex JavaScript being executed under the hood. Analysis tools must either handle the libraries themselves, or add specific modelling for each library API. The prevalence of JavaScript frameworks means that almost all real-world sites are complex in these ways. JavaScript libraries and frameworks are used on 76% of sites, with jQuery in particular in use on 73%, often alongside other frameworks [136]. The JavaScript used on the web routinely uses many of the language's dynamic and difficult to analyse features [114, 115]. Many of these difficult features can be attributed to libraries' internal implementation.

Generated and obfuscated code

As well as DOM structure generated on-the-fly by JavaScript frameworks, most modern sites also use server-side code to generate their pages. A server-side language such as PHP, C#, Java, Python, or Ruby is used to combine the page contents (e.g. loaded form a database) with a *template* which defines how that content should be rendered as an HTML page. In many cases even the JavaScript code is auto-generated by the server-side framework, sometimes by translation from the server-side language itself. This is done so the developers can define the site behaviour in a single place and a single language, and their tools will translate this into the HTML, CSS and JavaScript required to interface with their server-side code.

[14] https://jqueryui.com/
[15] https://dojotoolkit.org/
[16] https://angularjs.org/
[17] http://backbonejs.org/
[18] http://emberjs.com/

The problem with generated code is that it does not follow standard patterns for JavaScript. Because it is not designed to be human readable, the code can be arranged in any way convenient to the generator. For example it is common to see all the application state lumped together into a single giant object in the global scope, which removes any context about what that state refers to. As it does not aim to be readable, auto-generated JavaScript code can freely use any "difficult" language features which make the translation simpler, such as eval.

Code is often also *intentionally* obfuscated to discourage reverse-engineering. The site's source JavaScript is automatically "mangled" to make it more difficult for humans to read, usually introducing tricky language features which are difficult for analysis tools to reason about in the process.

2.4 Program analysis

Program analysis is the process of automatically determining facts about the behaviour of a program. This can be done by looking at a program's source code without executing it, known as *static analysis*; by generating and executing tests, known as *dynamic analysis*; or by a combination of the two. The goal of program analysis can be to verify correctness, detect bugs, or suggest performance improvements. This section gives a brief outline of static analysis, in particular symbolic execution; and dynamic test generation. These two analysis techniques form the background required to describe *concolic testing* in the next chapter, which combines aspects of both and is the basis of much of my work.

Static analysis

Static analysis is the analysis of a program's source code, object code or bytecode *statically*, that is, without executing it, to prove properties of the program's run-time behaviour. There are many static analysis techniques available, depending on the goal of the specific tool [108, 119]. For example *data-flow analysis* can be used to derive facts about how data flows through the program, such as which values each variable may have at different program points. *Type systems* are used to disallow certain unsafe operations and make sure appropriate arguments are given to functions. *Model checking* is used to confirm that a software system or hardware design matches its formal specification. *Abstract interpretation* simulates the execution of a program using an abstraction of the real program state, in order to simplify reasoning about that state, which can generate many types of analysis results. Static analysis is often useful to find corner-case issues which are difficult to discover or reproduce via testing. For example, certain race conditions or instances of undefined behaviour may appear correct when tested, and these bugs are difficult to discover without source code analysis.

Static analysis is a difficult problem, both in theory and in practice. Analyses must focus on detecting certain classes of errors and most can only return approximate results. Even when it is possible to fully analyse a program, this can be very expensive. The practical performance problem can be reduced by further approximation of the analysis results. Despite these issues, approximated or simplified static analysis is still a powerful tool for ensuring software quality and security.

Symbolic execution

Symbolic execution is a static program analysis technique—a particular type of abstract interpretation—where the program's execution is simulated and the program state at each point is represented symbolically. Each variable's value is given by a symbolic expression (a logical formula) over input values and constants from the program. This allows the analysis to determine which program states are possible to reach, and which input values can reach them.

Example. In the following program the returned value is always at least 5, and the assertion is always satisfied. To check this with symbolic execution, the program's execution is simulated from the beginning.

```
1:  function f(x) {
2:      var r;
3:      var y = 10 * x;
4:      var z = x + y + 5;
5:      if (z >= 60) {
6:          r = x;
7:      } else {
8:          r = 5;
9:      }
10:     assert(r >= 5);
11:     return r;
12: }
```

When the function is first entered on line 1, the symbolic state is simply that variable x has the value of the input: $\{x = x_{\text{input}}\}$. As each line is simulated, the symbolic state is updated with whatever new information is known about each variable. When a variable is used on the right hand side of an assignment, it's symbolic value is used to update the symbolic value of the newly assigned variable. For example on line 3 the symbolic state is updated to

$$\{x = x_{\text{input}}, r = \text{undefined}, y = 10 \cdot x_{\text{input}}\}$$

and at line 4 it becomes

$$\{x = x_{\text{input}}, r = \text{undefined}, y = 10 \cdot x_{\text{input}}, z = x_{\text{input}} + 10 \cdot x_{\text{input}} + 5\}$$

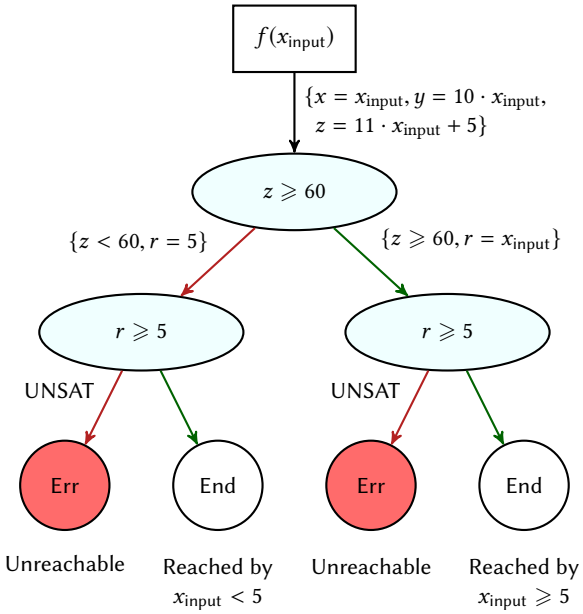


Figure 2.6: The path tree implicitly explored by the symbolic execution.

where the value for z can be simplified to $z = 11 \cdot x_{\text{input}} + 5$.

At the branch on line 5, the branch condition z >= 60 can be evaluated symbolically as $11 \cdot x_{\text{input}} + 5 \geqslant 60$ using the value of z from the symbolic state. This constraint simplifies to just $x_{\text{input}} \geqslant 5$. As this condition could be satisfied or not depending on the input, the symbolic analysis must consider both paths.

On the true branch, we know the branch condition was satisfied, so $x_{\text{input}} \geqslant 5$ can be added to the symbolic state. Line 6 then updates the symbolic state to

$$\{x = x_{\text{input}}, r = x_{\text{input}}, y = 10 \cdot x_{\text{input}}, z = 11 \cdot x_{\text{input}} + 5, x_{\text{input}} \geqslant 5\}$$

When we reach the assertion on line 10, the symbolic state can be used to show that $r = x_{\text{input}} \geqslant 5$ so the condition r >= 5 is satisfied.

On the false branch, we know the branch condition was not satisfied, so $x_{\text{input}} < 5$ is added to the symbolic state. At line 8 the state is updated to

$$\{x = x_{\text{input}}, r = 5, y = 10 \cdot x_{\text{input}}, z = 11 \cdot x_{\text{input}} + 5, x_{\text{input}} \geqslant 5\}$$

Again, at line 10 the symbolic state proves that the assertion is satisfied.

Figure 2.6 shows the tree of program paths which are checked during this symbolic execution, and shows how the assertion is proved to always be satisfied. □

By simulating the program like this and checking down all possible paths the symbolic execution can prove which program states can possibly be reached and provide the conditions on the input required to reach them. These conditions can be

used to generate concrete inputs which reach the given state. For example, for the above function to return 5, the analysis can prove that the input x must either be 5 (in the case where the branch condition is satisfied) or be less than 5 (in the case where the branch condition fails).

Symbolic execution is limited by the path explosion problem. At each branch in the program, the analysis itself needs to branch and check what happens if the branch condition is satisfied, and what happens if it is not, continuing execution in each case. This means an exponential number of paths (compared to the number of branch points in the program) must be checked. Some techniques exist to merge the analysis of identical or similar paths, which reduces the size of the problem but can introduce imprecision if the paths are not completely equivalent [11]. When analysing input-dependent loops the procedure described above will try to test every possible number of iterations of the loop and will never terminate; an analysis must detect these cases and merge the equivalent paths.

During the analysis symbolic expressions must be checked for satisfiability (when deciding which branches are feasible or whether an assertion is satisfied) and used to generate input values which satisfy them (to generate example inputs leading to a bug). These constraints are passed to a *constraint solver* which can check satisfiability and return a satisfying assignment to the variables when one exists. The constraint solving is another limitation of symbolic execution; the constraints generated by the analysis may be too complex to effectively solve. If a constraint cannot be solved it may be simplified, but the analysis will again lose precision.

Automated testing and dynamic analysis

Automated testing is the process of generating test inputs to a program, executing the program with these inputs and checking for any problems. The problems might be crashes, failed assertions, reading or writing the wrong areas of memory, or even incorrect results. A dynamic analysis (where real tests are run with the code) has the advantage over static analysis that any discovered bug is easily reproducible. There is no approximation and there can be no false positives.

The simplest technique to automatically generate new tests is randomised testing, or *fuzzing*. The disadvantage of generating inputs at random is the difficulty of finding *interesting* inputs. In particular, complex data structures are extremely unlikely to be generated randomly, making it difficult to find bugs in functions which process them.

To generate interesting test cases, fuzzing tools are normally given some knowledge about the expected formats of the inputs they are supposed to generate (an input specification). These *specification-based* tools can then generate a variety of valid, invalid and edge-case inputs, while maintaining the benefits of randomisation [49].

As automated testing is not expected to cover all possible paths in a program, some metric is needed to evaluate the success of the testing. Common metrics include line or statement coverage, branch coverage, or the full path coverage. Line coverage refers

to the proportion of source-code lines which were executed during testing. Branch coverage is a stronger requirement, checking that both the true and false conditions of every branch are exercised at least once. Path coverage is the strictest coverage metric, requiring that every unique path through the program is tested. This is the most comprehensive, and most expensive, type of testing.

2.5 SMT solving

Constraint solving is a key component of many types of program analysis. It can be used to check the reachability of certain parts of the program, or to generate new values to be tested. The program property to be tested is expressed as a logical formula, and the constraint solver determines whether the formula has a satisfying assignment. The most common way to express these formulae is as *SMT constraints*.

SMT (satisfiability modulo theories) is the problem of checking whether a given SMT formula is satisfiable or not. An SMT formula is a logical combination of predicates, which are Boolean assertions over variables of certain types (such as Booleans, integers, strings, and so on). For example $x \leqslant 5$, $x = y + 1$ and $y \geqslant 2$ are predicates over the integer variables x and y. Then $x \leqslant 5 \wedge x = y + 1 \wedge y \geqslant 2$ is an SMT formula making use of these predicates. In this case it is satisfiable; it is satisfied by any of the assignments $x = 3, y = 2$; $x = 4, y = 3$; or $x = 5, y = 4$. In contrast, the formula $x \leqslant 5 \wedge x = y + 1 \wedge y \geqslant 10$ is not satisfiable; there is no assignment of integer values to x and y for which the formula evaluates to *true*.

SMT solvers are software tools which solve the SMT problem. Given an SMT formula as input, they check whether it is satisfiable or not. If it is, they return "SAT" along with a satisfying assignment; an assignment of values of the appropriate types to the free variables in the formula such that evaluating the formula with those values leads to *true*. If the formula is not satisfiable, then the solver returns "UNSAT". SMT solvers are a standard way to solve constraints in many areas of program analysis, verification, and testing [24, 82].

Each solver supports a certain set of *theories* which defines the types of constraints it can solve. Each theory allows certain types of variables and predicates to be used in the input formula. For example the formulae given above use the theory of integers, which means the predicates are expressions using integer variables, integer constants and symbols including $+$, $-$, $\times$, $=$, $<$, $>$, $\leqslant$ and $\geqslant$. Other commonly supported theories include Booleans, strings, arrays, real numbers, uninterpreted functions and bit-vectors. For example, the theory of strings would allow the formula $\text{concat}(x, y) = \textit{"Hello world"} \wedge \text{len}(x) = 6$, which is satisfied by the assignment $x = \textit{"Hello "}, y = \textit{"world"}$, among many others. Support for these expressive theories allows many problems to be written very naturally and directly as SMT instances and makes them simple to solve.

Some solvers support constraints which span theories. For example a constraint might use the length of a string as part of an integer expression, or even coercions between types from different theories. This is especially important in the analysis of dynamic languages, such as JavaScript, where coercions between types are common. For example a constraint may require coercion between strings and integers: str-to-int(x) $\geqslant$ 10 is satisfied by x being *"15"* or *"17"* but not *"3"* or *"hello"*. This requires specialist support from the solver.

To provide a standard interface, most modern SMT solvers support the SMT-LIB language for their input and output [17]. This standard language means that tools like mine can output a single set of constraints independently of the solver. However, as we use theories which are not yet standardised by SMT-LIB (most importantly, strings) we use the solver-specific extensions to SMT-LIB as well, meaning that some work is required to switch solvers.

SMT solving is an important problem in verification and program analysis, and there has been a lot of work on the efficiency and effectiveness of SMT solving. Thus, the available solvers are extremely efficient—even for very large problem instances.

Solver implementation

Techniques for SMT solving are based on the related problem of SAT solving. In the SAT (Boolean satisfiability) problem, only propositional logic formulae are allowed as input (that is, formulae consisting of logical combinations of Boolean variables, such as $(a \wedge \neg b) \vee (a \wedge c)$, which is satisfiable by, for example, $a = \mathit{true}, b = \mathit{false}, c = \mathit{false}$). A SAT solver takes a propositional formula and returns "SAT" if it is satisfiable, along with a satisfying assignment, and "UNSAT" otherwise.

The standard approach to SAT solving is known as CDCL (conflict-driven clause learning) [82]. CDCL proceeds by making a provisional assignment to one of the propositional variables, and checking what implications this assignment has on the rest of the formula. If this assignment implies that some other variables must take particular values, these new assignments are assumed and included in the propagation as well. This process of guessing an assignment and propagating its implications is repeated until the formula is satisfied, in which case a satisfying assignment is found, or a conflict is discovered. Once a conflict is found, the assumptions which lead to this conflict are determined (by backwards search in the graph of implications) and their negation is added as a new clause to the formula. It is known that this new clause *must* be satisfied in any satisfying assignment to the original formula, and adding the new clause simplifies the discovery of new conflicts later. The search then backtracks, removing the assumptions which lead to the conflict, and continues.

SMT solvers use a central CDCL SAT solver and a set of theory solvers [82]. Each theory constraint in the input formula is replaced by a Boolean variable to create a SAT formula encoding the structure of the original SMT instance. Then as the SAT

solver works, it can make queries to the theory solvers about which combinations of propositions are satisfiable. The theory solver only needs to be able to check satisfiability of conjunctions of the individual theory predicates in the original formula, and to generate values. In more advanced solvers, the central SAT solver and theory solvers can all share information between each other to improve performance.

This general procedure is sound and complete, and guaranteed to terminate, as long as the individual theory solvers are also sound, complete, and terminating. This means the solver will always return an answer, and that answer is guaranteed to be correct. However, many useful theories cannot be solved efficiently, and some are even undecidable. Many theory solvers must restrict themselves to efficiently solvable subsets of the theories, or return "unknown" for cases which cannot be proved satisfiable or unsatisfiable within a certain time limit. In practice, this is still a beneficial trade-off for being able to use such expressive input formulae.

3 Concolic testing for web JavaScript

A first step towards our goal of applying information from the low-level implementation view and the high-level user-event view of a web application is to have a means of understanding the JavaScript that runs on a website. This chapter describes a platform for understanding web JavaScript and real-world JavaScript-based web applications, based on a common technique called concolic testing. This provides the infrastructure required for concolic testing of web-based JavaScript, as a base which different analysis tools to attack specific problems can be built on. This chapter describes the platform, and the customisations required to handle web-based JavaScript. Then the different analysis tools built on top of the platform are described in the next chapter.

Example. To demonstrate concolic testing for web-based JavaScript, we will use the following running example: an airline search form, which includes fields From and To (with id attributes "from" and "to", respectively) whose values are chosen from a drop-down list, along with a field Date (with identifier "date"), populated by a date picker. The event handler for To performs client-side validation using the function validate_to shown in Listing 1. The event handler for Date uses the function validate_date, which is simplified for our example to assume that all dates are in the current year.

Note that the validation code involves restrictions on both the values of the fields, and on the order in which they are filled. Filling To before From causes an error at line 9, for example. In addition to the constraints explicitly enforced by the event handlers, there are a number of implicit constraints on the fields: for example, the values of From and To should come from their respective drop-down lists. A form-filling or wrapper-generation tool must find values that satisfy all of these constraints. □

3.1 Concolic testing

Concolic testing, or directed automated random testing, is a testing technique which uses concrete executions of a program to drive a symbolic analysis [50, 122]. It can be thought of as a combination of symbolic execution and automated testing. The symbolic analysis is able to guide the automated tester and suggest specific inputs which reach new parts of the program which would be very difficult for traditional test-generation approaches to discover. Conversely, the concrete executions allow exploration through parts of the program which are only partially understood, or otherwise can't be reached by the symbolic analysis alone.

Listing 1 The form validation code for the simple airline example.

```
1:  function validate_to() {
2:    var from = document.getElementById("from").value;
3:    var to = document.getElementById("to").value;
4:    return validate_aux(from, to);
5:  }
6:  function validate_aux(from, to) {
7:    // Check departure airport is set
8:    if (from.length == 0) {
9:      alert("Error: Departure airport must be set");
10:     return false;
11:   }
12:   // Check the departure and destination are different
13:   if (from === to) {
14:     alert("Error: Departure must differ from Destination");
15:     return false;
16:   }
17:   return true;
18: }
19: function validate_date() {
20:   var to = document.getElementById("from").value;
21:   var date = document.getElementById("date").value;
22:   // Check destination airport is set
23:   if (to.length == 0) {
24:     alert("Error: Destination Airport must be set");
25:     return false;
26:   }
27:   // Check that date is after today
28:   var today = new Date();
29:   var day = parseInt(date.substr(0, 2), 10);
30:   var month = parseInt(date.substr(3, 5), 10);
31:   var valid_date = (month >= today.getMonth()+1 &&
32:         (month != today.getMonth()+1 || day >= today.getDate()));
33:   if (!valid_date) {
34:     alert("Error: date cannot be before today");
35:     return false;
36:   }
37:   return true;
38: }
```

Algorithm 1 The high-level algorithm for concolic testing.

```
1:  procedure CONCOLIC-TESTING(program)
2:      values ← CHOOSE-INITIAL-VALUES(program)
3:      trace ← EXECUTE-AND-RECORD(program, values)
4:      path-tree ← INIT-TREE(trace)
5:      while path-tree is not fully explored do
6:          target-path ← SEARCH(path-tree)
7:          values ← SOLVE-PATH-CONSTRAINT(target-path)
8:          if target-path was successfully solved then
9:              trace ← EXECUTE-AND-RECORD(program, values)
10:             EXTEND-TREE(path-tree, trace)
11:         else
12:             Mark target-path as unreachable in path-tree
13:         end if
14:     end while
15: end procedure
```

A generic (that is, not JavaScript or web specific) concolic testing algorithm is given in Algorithm 1. Concolic testing begins by choosing some default starting values for the variables (represented by CHOOSE-INITIAL-VALUES on line 2). The program being tested is then executed concretely (that is, using a real interpreter) and some symbolic information is recorded about how the inputs are modified and when they occur in branch conditions. Thus each trace is associated with a *path constraint*: the set of individual branch conditions (expressed in terms of the input values) which must be satisfied for the program's execution to follow that particular path. Each path constraint is a logical formula describing an equivalence class of input values, with equivalent values resulting in the same execution path in the program. Thus the state of the exploration can be characterised by the set of path constraints of explored traces, which form a tree. The testing algorithm proceeds in an execute-and-analyse loop. After executing a trace (via EXECUTE-AND-RECORD), the new information gathered during that trace is added to the tree, either by INIT-TREE on line 4 for the initial trace, or by EXTEND-TREE on line 10 for subsequent traces. In order to find set a of input values that reaches a new path, the algorithm chooses a sequence of branch conditions that has not yet been explored, and generates the corresponding path constraint using the SEARCH method on line 6. This path constraint is sent to a constraint solver (by the call to SOLVE-PATH-CONSTRAINT on line 7). If the solver can satisfy the constraint, a solution is chosen as the next set of input values. If the solver cannot, the path is marked as unreachable, and it will not be chosen again by SEARCH in subsequent iterations.

Table 3.1: The tests executed during concolic testing of validate_aux(from, to).

Run	from	to	Conditions hit	Result	Next goal
1	“”	“”	length(from) = 0	Error	¬(length(from) = 0)
2	“a”	“a”	¬(length(from) = 0), from = to	Error	¬(length(from) = 0) ∧ ¬(from = to)
3	“a”	“b”	¬(length(from) = 0), ¬(from = to)	Success	Finished

Example. Consider the function validate_aux(from, to) used as part of the validation for the To field in our example form. Concolic testing would first execute the function using default or random values for the arguments from and to; say empty strings for both. This would lead the function's execution to follow a path which terminates after the first alert on line 9. While the function is executed concretely, symbolic information is recorded about how the inputs are used and when they are used in branch conditions. In this initial execution, the only branch condition depending on a symbolic input (called a *symbolic branch*) was length(from) = 0, generated at the first if-statement. The recorded trace of symbolic branches (in this case just the single branch, plus some meta-data about the concrete execution) becomes the first trace added to the tree (line 4 in Algorithm 1). The search procedure on line 6 will isolate the path consisting of the single constraint ¬(length(from) = 0) as an unexplored path in the code, and this will be sent to the constraint solver (line 7). The solver will return values for from and to which satisfy this constraint: for example, the solver might return the values from = “a” and to = “a”. In the next iteration of the loop, the function being tested is executed again, with these new values as input, which drive the execution to the second alert at line 14 of the example code. This second execution is symbolically traced, and generates the path ¬(length(from) = 0) ∧ from = to. Note that this is the conjunction of two separate branch conditions: taking the false branch at the first if statement, and the true branch at the second. The search procedure on line 6 will now return the path constraint ¬(length(from) = 0) ∧ ¬(from = to) as the next target to explore. Solving this constraint gives values which pass this validation function and avoid each alert.

The path tree generated during this testing is shown in Figure 3.1, and the list of tests executed is shown in Table 3.1. Traces are marked as errors or successes depending on whether an error message was shown by a call to alert. □

The constraint solver is a critical component in any concolic testing tool. Effective concolic testing requires high-performance solvers which support the constraint types encountered in common programming languages: Booleans, integers, reals, bit-vectors, and arrays.

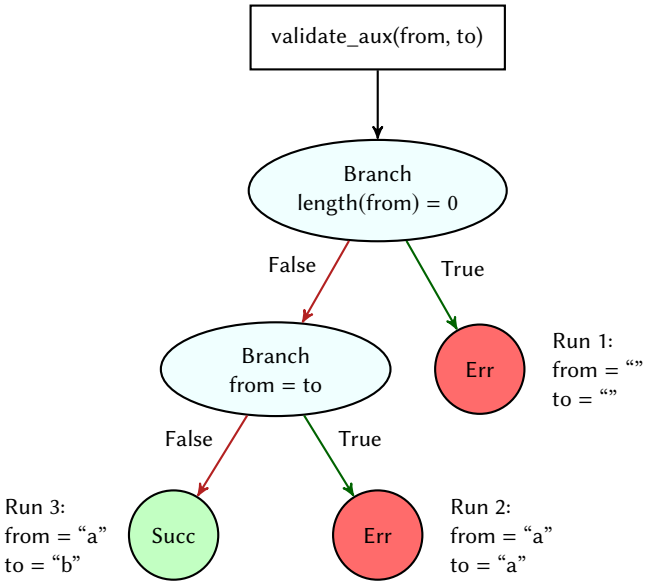


Figure 3.1: The symbolic path tree for validate_aux(from, to) which is explored during the concolic testing.

Concolic testing normally includes *classification* of a trace. In the context of testing, this means determining whether an error occurred during a given test. In Algorithm 1 this is assumed to be done within Execute-and-Record. When applied to form exploration, the classifier determines whether the result of a trace is a successful submission (leading to a new page) or not. We can detect unsuccessful submissions by flagging the execution of certain commands such as alert, or by inspecting the DOM (for example checking for the string "Error" being added).

The choice of which branch in the partial execution tree to explore next is made by the *search procedure* (line 6 in the algorithm listing). Given a partially explored tree of symbolic path traces, the search procedure selects an unexplored branch as the next target. The order in which branches are selected is important in cases where the whole tree cannot feasibly be explored, which are common when analysing real code. For example, when searching for crashes, a search procedure which could explore a crashing path more quickly will provide useful results faster.

Our concolic testing platform supports several search procedures: depth-first search with iterative deepening, random-order search, a procedure to avoid useless explorations, or a customisable round-robin combination of other procedures.

An extension to this algorithm, which is commonly used in traditional concolic testing, is to use concrete values to simplify the constraints to be solved. Because the code under test is executed concretely, the analysis has access to concrete information, such as the exact values of variables in each iteration, which the symbolic part cannot

determine itself. This is helpful because these values can be substituted into constraints which are otherwise too difficult to solve. For example a step could be added between lines 6 and 7 which uses concrete values to simplify *target-path* if necessary. By doing this a concolic analysis can explore more of a program than a purely symbolic approach. In our implementation, we apply concrete information to simplify the constraints in a slightly different way, which will be described later.

Using concrete information to simplify constraints or avoid analysing certain "black box" areas of the code gives concolic testing an advantage compared to pure symbolic execution. Where a purely symbolic analysis would be blocked by an intractable constraint, a concolic analysis can use concrete information from the program to make progress—with some loss of precision—and continue exploration.

Compared to other forms of automated testing, concolic testing requires many fewer tests. A tool with symbolic understanding of the code being tested can intentionally drive execution towards certain interesting areas of the code, and can also avoid wasting time executing many redundant tests which end up taking the same execution paths.

However, the number of paths which may need to be explored is still a significant limitation. Each symbolic branch in the code provides a choice of paths, so the state space is exponential in the number of symbolic branches in the code being tested. Modern tools have implemented various techniques to reduce this path explosion, for example by generating symbolic summaries of certain functions and re-using them instead of re-analysing the function each time it is called [48].

3.2 System architecture

To apply concolic testing to web JavaScript, our platform requires two components: an instrumented web browser, and a web-specific analysis engine to analyse the pages visited by that browser. The browser part allows loading pages and interpreting their HTML, CSS and JavaScript, inspecting the DOM, clicking on buttons, logging JavaScript event handlers which are registered on the page, triggering those events, and so on. The analysis part records traces of symbolic information based on the actions performed in the browser, analyses these symbolic traces, and suggests new input values to be fed back into the browser on subsequent iterations.

Although the central components of concolic testing are mostly generic and are not specific to the web setting, analysing web-based JavaScript imposes some extra requirements on top of those for traditional concolic testing of standalone JavaScript:

- The symbolic model must account for the whole web page: HTML, CSS, and the browser and DOM APIs; not just JavaScript;
- Invalid and browser-specific code on real websites needs to be handled;
- As web-based JavaScript is event-driven, a sequence of user-level events and inputs must be chosen and simulated for the concolic analysis to test; and

- Nondeterminism (whether real or apparent) of the remote server, the client-side code, and asynchronous events must be accounted for.

Concrete execution and symbolic analysis of web-based JavaScript also poses significant extra implementation challenges. First, one needs to control the browser, simulate user actions faithfully, and ensure that the browser behaves deterministically from one test to the next. Frameworks such as Selenium WebDriver [120] are commonly used; they provide high-level control of a browser, but give limited control over certain low-level events such as timers and AJAX requests. Second, we need to get information about JavaScript execution in order to analyse the executed code. There are also frameworks which allow instrumentation and monitoring of JavaScript, with the goal of recording useful information for analysis [121]. However, it is problematic to apply these to third-party JavaScript on the web since they require some instrumentation of the JavaScript source, which is not typically available for third-party sites without developer support.

Our approach is to work not at the level of JavaScript source, but directly with a browser engine. We build on top of WebKit, an open-source production web browser engine, which is used in Apple Safari, with a variant also used in Google Chrome. The browser engine includes page fetching, HTML and CSS rendering, and a JavaScript interpreter (called JSC or JavaScriptCore); but excludes the user-interface of the browser (the URL bar, buttons, bookmarks, etc.). It can be run with a GUI, for testing and debugging, or "headless" for automated analysis.

Using a production browser engine is important when analysing real-world websites; it avoids the problems of modelling invalid or browser-specific code, and includes implementations of the DOM APIs built-in. JavaScript interpreters which are not designed for web-based JavaScript, such as Rhino [104], cannot handle the wide variety of JavaScript found on the web, and are therefore not suitable.

We use WebKit's API to control the browser, as well as some custom instrumentation, and the interpreter itself is modified to track and propagate symbolic values. The symbolic interpreter instrumentation is added at the *bytecode level*, rather than the JavaScript source level, so the operations we instrument are those from JSC's internal bytecode language.

WebKit's JSC JavaScript interpreter runs over an internal bytecode language called *JSC bytecode*. When JavaScript is loaded from a web page it is first compiled into bytecode and this bytecode is then executed by the interpreter. Thus, our symbolic execution infrastructure is built on these bytecode instructions and runs at the bytecode level. This simplifies the language which must be supported symbolically. For example there is no eval or dynamic code loading in the bytecode language; the code strings to be evaluated are compiled to bytecode instructions like any other code before being executed. This in turn simplifies the constraints which must be solved, making exploration easier. Working at the bytecode level abstracts away many of the dynamic

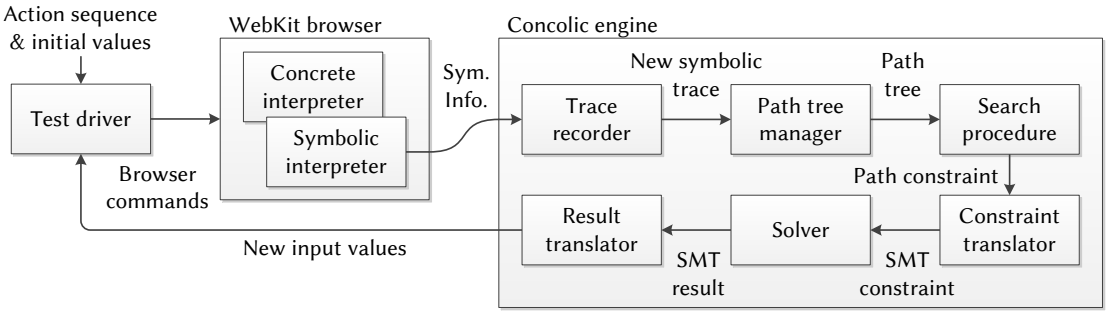


Figure 3.2: The architecture of the concolic testing platform.

features which make JavaScript analysis particularly difficult; those features are handled concretely by the browser engine and can be safely ignored by the analysis.

Figure 3.2 shows the key components of the analysis platform. It uses many of the same components as traditional concolic testing, but there are several new web-specific components and many web-specific issues and implementation details.

To begin an analysis, a sequence of browser actions is provided. When the platform is used for standalone JavaScript, this is simply a JavaScript file to execute; in the form analysis setting, this will be a URL to load, along with a sequence of form fields to fill and a submit button to click. The platform first uses a set of default values and executes the initial trace in the instrumented browser.

The actions and their associated code are executed concretely by WebKit. As this execution proceeds, our symbolic interpreter tracks how the input values are used and reports this information to the trace recorder. The trace recorder links symbolic information from the interpreter together with concrete information about the actions being executed to form a symbolic trace. A trace describes a single test execution of the code under test and the path which was exercised by that test.

The standard concolic testing procedure shown in Algorithm 1 now takes over. Once a trace is recorded, it is merged into a path tree, and the search procedure chooses a new path to explore, generating a target path constraint. This path constraint is translated to a form that our third-party solver can understand, and solved. The new input values generated by the solver are decoded and matched-up with their origins. For standalone JavaScript analysis, symbolic inputs originate from special-purpose symbolic input functions, and in the form validation setting they originate from form input fields. The browser is reset and the action sequence is replayed by the test driver with the new input values. This new execution is expected to reach the search procedure's target path in the execution tree. This process is repeated until either the tree is fully explored, or some exploration budget is reached.

We now highlight three distinctions from traditional concolic testing. The first distinction is about resetting the environment. Between each test with a new set of

input values, the browser must be reset. This ensures that each trace begins from a clean starting point and the browser environment is as similar as possible for each iteration. At the start of each iteration, the test page is reloaded. This resets the DOM and JavaScript state, and undoes any page loads which may have occurred during the previous iteration. As well as reloading the page, the platform must account for cookies, which allow state to be saved between page loads.

Cookies allow a web server to maintain a persistent state with a web browser between requests. This is a problem for analysis, because subsequent traces may receive different content, or behave differently to the initial test. Our browser is modified to ignore cookies, ensuring that each page load appears as a fresh request from a "clean" browser. When the server or client-side code sets a cookie, the browser reports success, but does not actually save it, so it is not used in future requests.

Cookies are not the only way for a browser to save persistent state locally. Other technologies include Local Shared Objects (normally known as Flash cookies), HTML5's local storage APIs, and Java applets' persistent storage service. The evercookie project is a demonstration of many such alternative ways to store data via a web browser [75]. As these are primarily used for tracking and analytics—not user-facing functionality—it has not been necessary to support these cases in our browser. Other alternatives to cookies, such as browser fingerprinting [37], are implemented on the server-side and can never be directly controlled by the browser.

The second distinction from traditional concolic testing, already alluded to above, is that the symbolic inputs may not be simply the arguments for a function, but rather something that has to be provided by the platform on-the-fly during the code's execution. This is dealt with in Chapter 4.

The final distinction involves what code is being executed. Traditional concolic testing tests the behaviour of a single function or program. Web pages, in contrast, are event-driven and interactive. There is no single function to test, but a set of possible actions (which may change during the course of a test) which may be triggered. Some actions have associated input values; in the form validation setting for example, a form-filling action is parametrised by the value to be filled into the form field. Therefore, instead of testing a specific function, the test driver executes a certain sequence of actions. Running the sequence of actions together is important; actions may have side-effects which affect future actions, so testing each one individually is not enough.

Once an action sequence is chosen, the "top level function" which a traditional concolic analysis would use can be defined. The function takes as input the values for each of the form fields, and when called, executes the actions one at a time, using its arguments as the parameters to each action. This decomposition from the top-level function (which is never actually materialised) down to the browser-level implementation is shown in Figure 3.3. This demonstrates the connection between the high-level user-action view and the low-level implementation view discussed in Chapter 1. Note

that not only does the analysis need to execute multiple pieces of code, but it must also track the propagation of symbolic inputs across these executions.

3.3 Symbolic execution and constraint generation

Our symbolic interpreter runs alongside WebKit's existing concrete JavaScript interpreter JSC and computes the symbolic values used in the analysis. Each concrete value is optionally tagged with a corresponding symbolic value. The symbolic value is a description of how that value was derived from the inputs. Initially all values in the interpreter are concrete; they have no symbolic value. When an input value is accessed (from a form field in the form validation setting, or via special-purpose functions when analysing standalone JavaScript), that value is tagged with a corresponding symbolic variable name. For example, when

```
var from = document.getElementById("from").value;
```

is executed, the value is fetched by a WebKit-internal getter and returned. This getter is instrumented so that the values returned have a symbolic tag showing where they originated. For this example the symbolic tag says that this value originates from a form field's value property and that the form input element was called from. This represents a new symbolic variable.

The interpreter propagates symbolic information as the JavaScript code is executed (in a similar way to a taint propagation). Continuing the above example, when from.length is evaluated the resulting value also has a symbolic tag, this time representing the length of the From field's value. Each time a branch instruction (e.g., an if statement or a loop condition check) is executed the interpreter checks whether it is a symbolic branch and if so records the corresponding symbolic branch condition. A branch is called symbolic if its branch condition uses any symbolic value; that is, if the branch decision depends on an input value.

The symbolic interpreter is implemented as an extension of the existing concrete interpreter. When the concrete interpreter receives an instruction it executes it as usual, but also calls the corresponding symbolic instruction in the symbolic interpreter. The symbolic interpreter then has the opportunity to update any symbolic values required. Thus the interpreters are kept in perfect lock-step and every JSC bytecode instruction can be handled symbolically. Built-in methods must also be instrumented separately from the interpreter. WebKit implements JavaScript's built-in functions with C++ methods internally, which receive and return interpreter values. When the concrete interpreter reaches a call to a built-in, it calls the relevant C++ code in WebKit (external to the main interpreter) which implements this call, and uses the returned value as the result of that call. The WebKit-internal implementations are instrumented in ArtForm so they return symbolic values correctly when required which can then be

Top-level function	Action sequence	JavaScript events	JavaScript functions
$f(x, y, z)$	A_1: Fill field From with x	focus change blur	validate_from
	A_2: Fill field To with y	focus change blur	validate_to ↳ validate_aux
	A_3: Fill field Date with z	focus change blur	validate_date
	A_4: Click Submit	mouseover mousemove mousedown focus mouseup click mousemove mouseout blur	validate_btn

Figure 3.3: The connection between the high-level actions and the actual JavaScript code which is executed. The exact JavaScript events triggered by each action depend on the level of event simulation used in the platform. The JavaScript functions which are called in response to each event are determined by the web page's code.

Table 3.2: JavaScript level, bytecode-level, and symbolic execution.

JavaScript code	Concrete bytecode	Concrete state change	Symbolic state change
var from = document. getElementById(*"from"*).value;	**op_call** r1 *"getElementById" "from"* **op_get_by_id** r2 r1 *"value"*	r1 := DOM node with ID *"from"* r2 := "" [value of From field]	(none) r2 := **SymStr**(*"from"*)
if (from.length == 0)	**op_get_by_id** r3 r2 *"length"* **op_eq** r4 r3 0 **op_jfalse** r4 else_label	r3 := 0 [length of field value] r4 := **true** [length is 0] [r4 is **true**, so we do not jump]	r3 := **StrLen**(r2) r4 := **IntOp**(r3, **INT_EQ**, **ConstInt**(0)) tmp := **BoolOp**(r4, **BOOL_EQ**, **ConstBool**(false))

used by both interpreters as usual. Not all built-in functions are instrumented; we have chosen to only instrument the functions which were most commonly used on the sites we were analysing. This is partly because of the implementation effort of individually instrumenting each built-in, and partly to simplify the generated constraints. Dropping certain difficult conditions is a way to make the analysis more concrete, in order to allow it to continue past certain patterns which cannot be analysed (such as built-ins with complex behaviour which can't be encoded as an SMT constraint).

Note that functions which are not JavaScript built-ins (such as JavaScript library code) are implemented in JavaScript, so they appear in the interpreter along with all other JavaScript code. They do not require any special handling by the interpreter or the analysis. Built-ins require special handling because they are not implemented in JavaScript, or in the interpreter, but via native-code methods in the browser.

Example. We will explain the symbolic value propagation using this JavaScript snippet, adapted from the running airline search example:

```
var from = document.getElementById("from").value;
if (from.length == 0) { ... }
```

Table 3.2 shows the JavaScript code being executed, the corresponding bytecode generated by WebKit, the state change that results from this code at the register level, and the changes in the symbolic state made by the symbolic interpreter. For example, the first line of the table shows that the first line of JavaScript generates two bytecode instructions:

op_call r1 *"getElementById" "from"*

calling the getElementById function on the From field and storing the result in the register r1, followed by

op_get_by_id r2 r1 *"value"*

which fetches the value property of the object in r1 and storing it in register r2. This results in a concrete state change setting r1 equal to the DOM node for the From field, and then r2 equal to the empty string (the value of the From field). The corresponding symbolic instrumentation will set r2 to the symbolic value of the value property lookup,

which will be **SymStr**(*"from"*), representing a symbolic input originating from a field with identifier "from".

By the time the branch instruction **op_jfalse** is reached, the symbolic interpreter has built up the following symbolic value in register r4:

IntOp(**StrLen**(**SymStr**(*"from"*)), **INT_EQ**, **ConstInt**(0))

That is, symbolically, r4 is known to contain a Boolean value representing whether or not the length of the value in the From field is zero. If this value is *false*, then the jump will be taken (in JavaScript source code terms, skipping over the bytecode representing the *if* block and jumping to the *else* block). Thus the final symbolic condition which is calculated at the branch instruction is the negation of r4:

BoolOp(**IntOp**(**StrLen**(**SymStr**(*"from"*)), **INT_EQ**, **ConstInt**(0)),
BOOL_EQ, **ConstBool**(false))

This is the symbolic branch condition which is reported from the symbolic interpreter to the trace recorder, to be added to the trace as a new symbolic branch.

Note that because the symbolic interpreter works at the level of *values*, not source-code variables or bytecode-level registers, these conditions are "inlined". The branch conditions recorded in the symbolic traces refer only to the input values, and not to any of the intermediate values or registers used to calculate them. □

The internal constraint language

The symbolic interpreter builds symbolic values, and thus the symbolic branch conditions, in our own internal symbolic constraint language. This internal constraint language is designed to mirror the actual operations in the real JavaScript program as closely as possible. It is necessary to reduce precision in some places so that the constraints are solvable, but this is done at a later stage, when they are written out in the solver's input language. The main concession at the level of the symbolic constraint language is that we use integer constraints to represent numeric values, which are in fact floating point values in JavaScript.

The terms of the internal constraint language are shown in Figure 3.4. Note that for brevity, I have replaced the full names with abbreviations (such as **SymInt** for **SymbolicInteger**) when writing out symbolic values in the examples above.

The terms **SymbolicString**, **SymbolicInteger**, and **SymbolicBoolean** represent user inputs and are created by the special-purpose concolic testing input methods provided by the platform. The other terms are all produced by the symbolic interpreter as it propagates symbolic information in response to concrete operations involving existing symbolic values, as the above example showed.

This propagation of symbolic values via the symbolic interpreter means the values expressed in our internal constraint language have a very close correspondence with the

⟨Expression⟩ ⊨ ⟨StringExpression⟩ | ⟨IntegerExpression⟩ | ⟨BooleanExpression⟩
| ⟨ObjectExpression⟩ | ⟨StringRegexSubmatchArray⟩
⟨StringExpression⟩ ⊨ ⟨SymbolicString⟩ | ⟨ConstantString⟩ | ⟨StringBinaryOperation⟩
| ⟨StringCoercion⟩ | ⟨StringSubstring⟩ | ⟨StringToLowerCase⟩
| ⟨StringToUpperCase⟩ | ⟨StringCharAt⟩ | ⟨StringReplace⟩
| ⟨StringRegexReplace⟩ | ⟨StringRegexSubmatchArrayAt⟩
| ⟨SymbolicObjectPropertyString⟩
⟨IntegerExpression⟩ ⊨ ⟨SymbolicInteger⟩ | ⟨ConstantInteger⟩ | ⟨IntegerBinaryOperation⟩
| ⟨IntegerCoercion⟩ | ⟨IntegerMaxMin⟩ | ⟨StringLength⟩ | ⟨StringIndexOf⟩
| ⟨StringRegexSubmatchIndex⟩ | ⟨ObjectArrayIndexOf⟩
⟨BooleanExpression⟩ ⊨ ⟨SymbolicBoolean⟩ | ⟨ConstantBoolean⟩ | ⟨BooleanCoercion⟩
| ⟨BooleanBinaryOperation⟩ | ⟨StringRegexSubmatch⟩
| ⟨ObjectBinaryOperation⟩
⟨ObjectExpression⟩ ⊨ ⟨SymbolicObject⟩ | ⟨ConstantObject⟩ | ⟨ObjectBinaryOperation⟩
| ⟨StringRegexSubmatchArrayMatch⟩

⟨SymbolicString⟩ ⊨ SymbolicString(⟨SYMBOLIC_SOURCE⟩)
⟨ConstantString⟩ ⊨ ConstantString(⟨STR_VAL⟩)
⟨StringBinaryOperation⟩ ⊨ StringBinaryOperation(⟨StringExpression⟩ , ⟨STR_OP⟩ , ⟨StringExpression⟩)
⟨STR_OP⟩ ⊨ CONCAT | STR_EQ | STR_NEQ | STR_LT | STR_LEQ | STR_GT
| STR_GEQ | STR_SEQ | STR_SNEQ
⟨StringCoercion⟩ ⊨ StringCoercion(⟨Expression⟩)
⟨StringLength⟩ ⊨ StringLength(⟨StringExpression⟩)
⟨StringSubstring⟩ ⊨ StringSubstring(⟨StringExpression⟩ , ⟨INT_VAL⟩ , ⟨INT_VAL⟩)
⟨StringToLowerCase⟩ ⊨ StringToLowerCase(⟨StringExpression⟩)
⟨StringToUpperCase⟩ ⊨ StringToUpperCase(⟨StringExpression⟩)
⟨StringIndexOf⟩ ⊨ StringIndexOf(⟨StringExpression⟩ , ⟨StringExpression⟩ , ⟨IntegerExpression⟩)
⟨StringCharAt⟩ ⊨ StringCharAt(⟨StringExpression⟩ , ⟨INT_VAL⟩)
⟨StringReplace⟩ ⊨ StringReplace(⟨StringExpression⟩ , ⟨STR_VAL⟩ , ⟨STR_VAL⟩)
⟨StringRegexReplace⟩ ⊨ StringRegexReplace(⟨StringExpression⟩ , ⟨STR_VAL⟩ , ⟨STR_VAL⟩)
⟨StringRegexSubmatch⟩ ⊨ StringRegexSubmatch(⟨StringExpression⟩ , ⟨STR_VAL⟩)
⟨StringRegexSubmatchIndex⟩ ⊨ StringRegexSubmatchIndex(⟨StringExpression⟩ , ⟨STR_VAL⟩)
⟨StringRegexSubmatchArray⟩ ⊨ StringRegexSubmatchArray(⟨StringExpression⟩ , ⟨STR_VAL⟩)
⟨StringRegexSubmatchArrayAt⟩ ⊨ StringRegexSubmatchArrayAt(⟨StringRegexSubmatchArray⟩ , ⟨INT_VAL⟩)
⟨StringRegexSubmatchArrayMatch⟩ ⊨ StringRegexSubmatchArrayMatch(⟨StringRegexSubmatchArray⟩)

Figure 3.4: The grammar for our internal symbolic constraint language.

⟨SymbolicInteger⟩ ⊨ SymbolicInteger(⟨SYMBOLIC_SOURCE⟩)
⟨ConstantInteger⟩ ⊨ ConstantInteger(⟨INT_VAL⟩)
⟨IntegerBinaryOperation⟩ ⊨ IntegerBinaryOperation(⟨IntegerExpression⟩ , ⟨INT_OP⟩ , ⟨IntegerExpression⟩)
⟨INT_OP⟩ ⊨ INT_ADD | INT_SUBTRACT | INT_MULTIPLY | INT_DIVIDE | INT_EQ | INT_NEQ | INT_LEQ | INT_LT | INT_GEQ | INT_GT | INT_MODULO | INT_SEQ | INT_SNEQ
⟨IntegerCoercion⟩ ⊨ IntegerCoercion(⟨Expression⟩)
⟨IntegerMaxMin⟩ ⊨ IntegerMaxMin(⟨INT_EXPR_LIST⟩ , ⟨BOOL_VAL⟩)
⟨INT_EXPR_LIST⟩ ⊨ ⟨IntegerExpression⟩ | ⟨IntegerExpression⟩ , ⟨INT_EXPR_LIST⟩

⟨SymbolicBoolean⟩ ⊨ SymbolicBoolean(⟨SYMBOLIC_SOURCE⟩)
⟨ConstantBoolean⟩ ⊨ ConstantBoolean(⟨BOOL_VAL⟩)
⟨BooleanCoercion⟩ ⊨ BooleanCoercion(⟨Expression⟩)
⟨BooleanBinaryOperation⟩ ⊨ BooleanBinaryOperation(⟨BooleanExpression⟩ , ⟨BOOL_OP⟩ , ⟨BooleanExpression⟩)
⟨BOOL_OP⟩ ⊨ BOOL_EQ | BOOL_NEQ | BOOL_SEQ | BOOL_SNEQ

⟨SymbolicObject⟩ ⊨ SymbolicObject(⟨SYMBOLIC_SOURCE⟩)
⟨ConstantObject⟩ ⊨ ConstantObject(⟨INT_VAL⟩)
⟨SymbolicObjectPropertyString⟩ ⊨ SymbolicObjectPropertyString(⟨SymbolicObject⟩ , ⟨STR_VAL⟩)
⟨ObjectArrayIndexOf⟩ ⊨ ObjectArrayIndexOf(⟨EXPR_LIST⟩ , ⟨Expression⟩)
⟨EXPR_LIST⟩ ⊨ ⟨Expression⟩ | ⟨Expression⟩ , ⟨EXPR_LIST⟩
⟨ObjectBinaryOperation⟩ ⊨ ObjectBinaryOperation(⟨ObjectExpression⟩ , ⟨OBJ_OP⟩ , ⟨ObjectExpression⟩)
⟨OBJ_OP⟩ ⊨ OBJ_EQ | OBJ_NEQ

⟨SYMBOLIC_SOURCE⟩ ⊨ SymbolicSource(⟨SOURCE_TYPE⟩ , ⟨SOURCE_IDENT_METHOD⟩ , ⟨STR_VAL⟩)
⟨SOURCE_TYPE⟩ ⊨ TEXT | SELECT | SELECT_INDEX | RADIO | CHECKBOX | EVENT_TARGET | DIRECT_ACCESS | UNKNOWN
⟨SOURCE_IDENT_METHOD⟩ ⊨ INPUT_NAME | ELEMENT_ID | EVENT_TARGET_IDENT | DIRECT_ACCESS_IDENT
⟨INT_VAL⟩ ⊨ *(An integer value)*
⟨STR_VAL⟩ ⊨ *(A string value)*
⟨BOOL_VAL⟩ ⊨ *(A Boolean value)*

Figure 3.4: (continued)

original JavaScript execution. This correspondence is only broken when the interpreter executes a built-in function which is not instrumented, and therefore not supported by our symbolic infrastructure.

The platform supports constraints over Boolean, integer and string inputs, using the standard operations on those types. For example, the term **IntegerBinaryOperation** supports the operations *add*, *subtract*, *multiply*, *divide*, *equals*, *not-equals*, *less-than*, *less-than-or-equal*, *greater-than*, *greater-than-or-equal*, *modulo*, *strict-equals* and *strict-not-equals*.[19] Certain specific types of constraints over arrays and objects can be expressed, but these are included as special-case support for specific features. For example, the regular expression function String.match returns an array of matched subexpressions, or null if there was no match. To support common uses of String.match symbolically, the internal constraint language includes five **StringRegexSubmatch** terms which can encode accesses to this array of matches to extract the matched string. Other than these special cases, there is no general support for symbolic arrays or objects, and it is not necessary. The platform does not permit these types as inputs, so it is not required to model them directly; symbolically modelling the primitive values they contain is enough. Most of the remaining terms in the internal constraint language represent the different JavaScript built-ins which are supported.

Note that he grammar rules given in Figure 3.4 do not give the types of expressions. For example, **StringBinaryOperation** is generated from the term **StringExpression**, even though it encompasses operations which return strings, such as **CONCAT**, and operations which return Booleans, such as **STR_EQ**. The symbolic interpreter also enforces some consistency rules of its own by the symbolic expressions it generates, which are not represented by the grammar. For example, a **StringInput** may only have a symbolic source with source type **TEXT** or **SELECT**; the other source types correspond to other symbolic input types.

3.4 Constraint solving

The generated symbolic conditions are recorded into symbolic traces, and saved into the path tree. From there, the search procedure chooses a new path to explore and generates a target path constraint, which is solved to find the next set of input values to test. As is common in concolic testing, we use a third-party SMT solver. Our path constraints are expressed as conjunctions of terms in the internal constraint language, and therefore must first be translated into the solver's input language. We use CVC4 [16] as our solver, and the standard SMT-LIB language [17] to express the constraints.

[19] JavaScript uses the concept of strict equality (=== and !==) to test for equality of values and their types together, without the normal implicit type coercions. For example "7" == 7 under normal equality, but "7" !== 7 under strict equality. In our language, the normal type coercions are made explicit, so these operators are treated the same as the standard *equals* and *not-equals*.

CVC4 was chosen because of its support for a wide variety of theories compared with other solvers [29], its strong string support [90], and especially its support for constraints involving coercions between multiple theories. JavaScript analysis, especially in the web setting, involves many complex string constraints, and transformations between strings and other data types are common in real-world code, so these features are important. It is a great benefit if the solver can support them directly.

Each constraint can be thought of as passing through four stages on its way to being solved. Initially, they are encoded in the web page's JavaScript source code. WebKit translates the original source code into its own JSC bytecode, which encodes the same constraints. The symbolic interpreter tracks the execution of the bytecode instructions, and builds symbolic constraints using our own internal constraint language. Finally, these constraints are translated to the solver's SMT-LIB input language, ready for solving. The first two transformations—from JavaScript source, to bytecode, to symbolic constraints—have already been described. The final transformation—from internal constraints to the solver's input—is the subject of this section.

Constraint translation

The constraint writer is the component of our system which rewrites our internal constraints for the solver. Its input is a path constraint, and any auxiliary constraints to be included (such as those describing the possible values for a drop-down select box for example). As output, it produces a string describing the same constraint in the solver's input language: SMT-LIB with CVC4-specific extensions. It is not always possible or feasible to provide a perfect translation of the constraints, and approximation is necessary in some cases.

The structure of the path constraints is simple: a list of assertions on the input variables. There is a single assertion generated for each symbolic branch on the target path, and each branch can be translated independently. In some cases the constraint writer modifies the translated constraint to make it simpler for the solver. This is separate to the necessary approximations mentioned above; in these cases a straightforward translation is possible, but some rewriting within the platform can significantly improve the solver's performance.

The constraint writer is split into two parts: a general-purpose core and solver-specific extensions. The general-purpose part handles the parts of the constraints which can be expressed in the SMT-LIB standard language, and as such are suitable for all SMT-LIB based solvers. However, the core SMT-LIB language does not include syntax for all the constraints solvers can handle, so each solver augments the core language with their own extensions. For example, SMT-LIB does not include any string constraints; these are all solver-specific. The solver-specific part of the constraint writer translates the symbolic constraints which rely on these solver-specific SMT-LIB extensions. This architecture allows us to add support for new solvers and new solver features relatively

easily, and choose which solver to use with a run-time flag, as the constraint translator only needs to create the appropriate constraint writer object and call the corresponding solver binary. We have implemented solver-specific constraint writers for CVC4, Z3-str and Kaluza. As the CVC4 constraint writer is the most mature, and all our experiments use CVC4, the remainder of this section will describe the constraint translation as it is done by the generic and CVC4-specific translators together.

Inputs to the JavaScript code being tested are represented as "symbolic sources" in the internal constraint language. A symbolic source includes information about the origin of that symbolic value. For example, if the value was the result of reading from a form field, the symbolic source includes that fact, along with the identifier of the field. In the output constraints, each input is represented by an SMT variable. These variables are named using the symbolic source identifiers, so that when the solution is read back, the solved values can easily be matched-up with the corresponding sources of those values, which is where they will need to be re-injected on the next iteration. This naming scheme also makes the SMT constraints much easier to read manually, compared to using meaningless variable identifiers in the constraint and a lookup table of symbolic sources and variable names in the constraint writer. Finally, the variable names also include the type of the symbolic source. This allows us to distinguish multiple variables which may come from the same symbolic source. For example, the value of a drop-down select field called "myselect" would have the variable name SYM_IN_myselect, whereas the selectedIndex property of the same field would be represented by the variable SYM_IN_INT_myselect. Boolean fields—checkboxes and radio buttons—have variable names of the form SYM_IN_BOOL_myfield.

Symbolic constraints are always represented in terms of the original inputs; there are no intermediate variables in the internal constraint language. In the output, we occasionally introduce intermediate variables as part of the translation, but this is rare; the vast majority of real-world constraints are fully unfolded and expressed directly in terms of the symbolic inputs. Normally, this is just to simplify the written constraint. For example when a **StringSubString** term is written where the substring runs from a certain index to the end of the string, then length of the string is used to calculate the substring's end index in a temporary variable which is used in the invocation of CVC4's **str.len** function. In some even rarer instances the intermediate variables are necessary due to the structure of the SMT-LIB constraints.

Example. We will demonstrate the constraint translation process by continuing with the example code from the airline form, discussed earlier:

```
var from = document.getElementById("from").value;
if (from.length == 0) { ... }
```

As we saw, this branch gives rise to the following symbolic condition:

BoolOp(**IntOp**(**StrLen**(**SymStr**("from")), **INT_EQ**, **ConstInt**(0)),
BOOL_EQ, **ConstBool**(false))

In the initial test, the default value of the empty string would have been used for From. Therefore, the first trace recorded will take the false branch at the bytecode jump instruction (recall that the bytecode translation uses **op_jfalse**, so the sense of the branch is reversed from that of the JavaScript source code). This means that the search procedure will select the true branch as the next target to explore, and the path constraint to reach this branch is exactly the constraint above. This constraint will be translated and solved. Note that because this is the topmost branch in the tree, this single condition makes up the whole path constraint; in this case there are no other conditions to be translated.

The symbolic constraint is translated by a depth-first traversal of its structure, beginning at the leaves of the expression. **SymStr**(*"from"*) is translated in two parts; a declaration of a new string variable

(**declare-const** SYM_IN_from **String**)

and the current translation for this sub-expression, which is just the variable name: SYM_IN_from. Similarly, the constants are translated directly: **ConstInt**(0) becomes 0 and **ConstBool**(false) becomes **false**.

StrLen(**SymStr**(*"from"*)) is the first non-leaf sub-expression to be translated. It uses the CVC4 built-in function **str.len** as follows: (**str.len** SYM_IN_from). The terms **IntOp**(x, **INT_EQ**, y) and **BoolOp**(x, **BOOL_EQ**, y) are both translated into the same pattern: (= x y). As such, the SMT expression for the above condition is:

(= (= (**str.len** SYM_IN_from) 0) **false**)

The path constraint includes the above condition positively, that is, not negated. An extra operation is added to say that the branch condition should be satisfied (strictly this is superfluous in the positive cases, but the extra operation is required when the branch condition is to be negated). Finally, the whole SMT expression is asserted, to tell the solver that these constraints are to be satisfied in any model it returns. Therefore the final emitted constraints are as follows:

(**declare-const** SYM_IN_from **String**)
(**assert** (= (= (= (**str.len** SYM_IN_from) 0) **false**) **true**))

As the translation proceeds, the constraint writer tracks the type of each term, ensuring they are always consistent and the result is a Boolean SMT expression. Finally, some header and footer information is added to set the solver's options and request an explicit model (rather than simply returning "SAT"). The resulting constraint file is sent to the solver, which in this case returns the model SYM_IN_from = "A". □

Most constraints are translated directly, but sometimes some re-writing is necessary. For example a *not-equal* constraint such as **BoolOp**(x, **BOOL_NEQ**, y) is translated as (= (= x y) **false**).

Table 3.3: The type conversions used by the constraint writer.

	Boolean	Integer	String
Boolean	—	if ⟨Bool⟩ then 1 else 0	if ⟨Bool⟩ then “true” else “false”
Integer	⟨Int⟩ ≠ 0	—	int.to.str
String	⟨String⟩ ≠ “”	str.to.int	—

Type coercions are another area where the SMT constraints we produce are not direct translations of the internal constraint language. The terms **BooleanCoercion**, **IntegerCoercion** and **StringCoercion** are translated according to the rules in Table 3.3. The coercions to and from Booleans are encoded as regular constraints which match the semantics of the corresponding JavaScript coercion. For example, a string coerces to true in JavaScript if and only if it is non-empty; thus, we simply convert the constraint **BooleanCoercion**(x) to (= x ″″) when x is a string. When converting between strings and integers, we use CVC4’s built-in functions **str.to.int** and **int.to.str**, which were added to CVC4 in response to our need to convert between these types with no simple way to encode the translation in other SMT constraints. Our internal constraint language has no general-purpose symbolic object inputs, so we avoid having to encode the complex coercion and comparison rules for non-primitive types.

Mismatches and approximations in constraint translation

There are several types of constraint which cannot be translated accurately into the solver’s SMT constraints. The problematic constraints can be broken down into two classes: constraints which cannot be expressed in the solver’s input language or cannot be successfully solved, and constraints which can be expressed but where the solver’s semantics do not precisely match the original JavaScript branches.

One example of constraints which cannot be translated at all are regular expression tests which include back- or forward-references. These features of JavaScript’s RegExp language make it non-regular, and so solvers supporting regular languages cannot handle them. The ExpoSE tool is able to rewrite difficult RegExp constraints into SMT expressions which can be solved in many cases [94]. Another example which occasionally comes up in practice is string inequalities. JavaScript supports tests such as *"hello" < "world"* which checks the lexicographic ordering between strings. There is no corresponding support for string ordering in CVC4’s string theory, so these constraints cannot be translated. Sometimes it is possible to approximate such constraints, and an example of this (for upper- and lower-case conversions) is given below.

There are also constraints which we do translate and successfully solve, but where the translation is not completely precise. For example, although CVC4 provides a **str.to.int** function to convert strings into integers, its semantics do not exactly match

those of JavaScript's conversions. In JavaScript, the empty string coerces to 0, whereas in CVC4, it coerces to -1. This can cause issues where the empty string is the default value in an otherwise numeric list, and all items are converted to integers for testing. There are other cases where CVC4 produces -1 for non-numeric strings, which JavaScript's conversions can handle. For example space-padded numbers, such as " 123 ", or numbers in non-decimal bases, such as "0x123" can be converted in JavaScript, but produce -1 by design in CVC4. There are similar corner cases (although not as severe) for other type conversions. It would be possible to attempt to encode the specifics of JavaScript conversions into our SMT constraints, which would make them much more complex, but improve precision. It is generally not feasible to write fully precise constraints for constraint types which the solver does not support directly.

To support constraints using upper and lower case conversions (which are not supported in the solver) we implemented a new relation, defined in the SMT formula itself, which matches upper- and lower-cased versions of strings. This could only be implemented by matching characters in a look-up table up to a fixed bound; strings longer than this bound are not properly constrained. Even for strings which are short enough, the solver's performance on these complex constraints is limited; they are much slower than functions which are natively supported.

In practice, it is important to be able to make a "best effort" at solving as many constraints as possible. It is worthwhile to solve many common cases, even at the cost of a theoretically imprecise translation. Continuing the example of **str.to.int** above, it is very rare for real-world web-based code to convert strings from non-decimal bases in any context which requires symbolic modelling. Lack of support for such corner cases is not a limitation. Other cases, such as converting the empty string or space-padded strings, are more common, but still rare enough to not cause a significant problem.

Optimisations during constraint translation

The constraints produced by the platform are not always easy for the solver to handle, even if it can solve them eventually. An obvious example is the coercion between strings and integers using **str.to.int** and **int.to.str**, which works correctly but can be very slow if there are non-trivial constraints applied to the converted result. Such constraints require careful co-ordination between the different theory solvers to be solved efficiently, and the search for a solution can easily get stuck in a brute-force search. The constraint writer can apply certain "optimisations" to the generated constraints, re-writing them into equivalent forms which are simpler to solve. These are performance improvements, which transform solvable but difficult constraints into equivalent, easier to solve ones.

The main optimisation applied is the *integer coercion optimisation*. When working with web-based JavaScript, much of the data being handled comes from the DOM. Thus, almost all input values in the code being analysed in a web setting are strings, even if they represent other types. A very common example of this is form

fields which take numeric values; these numbers are typed in as strings, and are available to the page's JavaScript code as strings. Typically, the strings will be immediately converted to integers after being read from the DOM (sometimes after a check for the empty string or white-space trimming), and the code which processes them will work with them as integers. Thus, we generate a lot of constraints of the form **IntOp**(**IntCoercion**(**SymStr**(*"x"*)), **INT_GT**, **ConstInt**(100)), which are "almost entirely" integer constraints, but which necessarily operate on string inputs and so involve coercions between theories. These are expensive for CVC4 to process, and impossible for most other solvers.

Where possible, the constraint writer rewrites these constraints to be based on integer inputs directly. This removes the coercions and makes the constraints much easier for the solver. For example, the above constraint is written as (> SYM_IN_x 100), where SYM_IN_x is an integer variable. When the solutions are read back, the platform converts the solver's integer solution back to a string which can be used as the actual input for the next test. Note that this optimisation requires a global analysis of the whole path constraint to check if a variable is *ever* used in both a string and an integer context; it cannot be applied locally to individual branch conditions.

The optimisation can only be applied when string inputs are always coerced to integers, and no non-trivial string constraints are applied beforehand. If a symbolic variable has non-trivial constraints in two different theories, a coercion is always required. It is very common, especially when libraries like jQuery are used, that all values read from form fields have some simple transformations applied, typically trimming white-space characters from the start and end of the input. To allow the integer coercion optimisation to still be used in these cases, we simultaneously run a *white-space filter optimisation*. This detects simple filters (string or regular expression replacements) which replace white-space characters by empty strings. These filters are ignored for the purposes of checking whether the integer coercion optimisation can be applied. If the integer coercion optimisation could be applied without the filters, then the filters are dropped from the constraint and the optimisation goes ahead. Similarly, code checking that an input is non-empty can be dropped in cases where an integer input could otherwise be used. These checks allow the integer coercion optimisation to be applied in many common situations.

Example. Consider the validate function in Listing 2, which reads its input from the DOM. The initial trace, with an empty input, reaches the alert on line 15. The first path selected for exploration is the one which satisfies both the conditions on lines 6 and 7, and therefore reaches line 8. The corresponding path constraint is:

$$\neg\textbf{IntOp}(\textbf{IntCoercion}(\textbf{SymStr}(\textit{"i"})), \textbf{INT\_GT}, \textbf{ConstInt}(4567))$$
$$\wedge\neg\textbf{IntOp}(\textbf{IntCoercion}(\textbf{SymStr}(\textit{"i"})), \textbf{INT\_LT}, \textbf{ConstInt}(123))$$

Listing 2 A function reading integer inputs from the DOM.

```
1:  var x = 123;
2:  var y = 4567;
3:  var z = 890;
4:  function validate() {
5:    var input = document.getElementById("i").value;
6:    if (input <= y) {
7:      if (input >= x) {
8:        if (input == z) {
9:          alert(input + " is forbidden.");
10:         return false;
11:       } else {
12:         return true;
13:       }
14:     } else {
15:       alert(input + " is too small.");
16:       return false;
17:     }
18:   } else {
19:     if (input <= 2 * y) {
20:       alert(input + " is a bit too large.");
21:     } else {
22:       alert(input + " is much too large.");
23:     }
24:     return false;
25:   }
26: }
```

Without the optimisation enabled, the constraint generated would be:

(**declare-const** SYM_IN_i **String**)
(**assert** (= (> (**str.to.int** SYM_IN_i) 4567) **false**))
(**assert** (= (< (**str.to.int** SYM_IN_i) 123) **false**))

With the optimisation, the following integer-based constraint is generated instead:

(**declare-const** SYM_IN_i **Int**)
(**assert** (= (> SYM_IN_i 4567) **false**))
(**assert** (= (< SYM_IN_i 123) **false**))

Using an integer variable and avoiding **str.to.int** gives a dramatic performance improvement. To fully explore the above example requires 5 tests, and 4 calls to the solver. These solver calls take a combined 7.00 seconds without the optimisation, or only 0.16 seconds with it enabled, a speed-up of over 40 times. In real world analysis the performance difference for solving can be massive. Other solvers do not implement **str.to.int**, so this optimisation would allow them to solve many common constraints using coercions in JavaScript which would be impossible otherwise. □

Handling branch conditions which cannot be solved

Because the internal constraint language aims to mirror the real JavaScript code as closely as possible, there are some symbolic constraints which cannot be successfully translated. For example, certain complex regular expressions cannot be translated because the features they use are not supported by the solver. In these cases, the platform attempts to simplify the constraint by removing the difficult clauses and retrying. This effectively makes the constraints more concrete, as is standard for concolic testing.

In our platform, the clauses in the path constraint come directly from the individual symbolic branches in the path tree which lead to the target path. The branch conditions for each branch are translated separately, and the full path constraint is the conjunction of these branch constraints. If an individual branch condition cannot be translated, it does not affect the other conditions. In this case, the branch node which cannot be translated is marked as "difficult" in the tree, and that constraint is dropped from the path constraint. When building new path constraints in future, the constraints at difficult branches are ignored, as we already know they cannot be translated.

Relaxing the constraints in this way allows us to attempt exploration through the difficult branch even though its branch condition cannot be solved. Difficult branches become, in effect, nondeterministic branches; the analysis knows they have some dependence on the input (otherwise they would never have been symbolic), but it cannot generate values to choose whether the left or right subtree is explored.

Exploration can still continue in the tree underneath the difficult branch. One branch near the top of the tree which cannot be translated does not necessarily prevent

a useful analysis of the rest of the code. The difficult branch itself is dropped, but other branch conditions on the target path, including those in the subtree underneath the difficult branch, are still translated and solved as usual.

3.5 Specifics of our concolic testing engine

This section presents some details of how our concolic testing platform is implemented, and how it addresses the particular requirements for testing web-based JavaScript.

The execution tree

The core data structure for concolic testing is the tree of all explored execution paths. The tree is built by merging symbolic traces as they are recorded. The traces (and thus, the tree) in our analysis include information which is useful for web interface analysis. As well as branches and their symbolic conditions, which must be included, we also include annotations for alert boxes, page loads, modifications of the DOM, console messages and JavaScript exceptions which the analysis can use to decide whether a certain trace was successful or not. We also include markers to show which unexplored branches have already been unsuccessfully attempted in the past; this allows the search procedure to ignore those branches in future. Table 3.4 lists the different node types and what each represents.

The different node types are organised in a hierarchy. This is useful so that any process which works on the traces or the tree (of which there are many in our system) is able to do so at an appropriate level of granularity. For example, all types of annotations to traces (alerts, page loads, and so on) are grouped under the general type TraceAnnotation. This means that the search procedure—which does not consider these nodes—only implements one case to handle all types of trace annotations at once. The trace classifier—which does depend on these annotations—is able to implement different behaviour for each type of annotation individually while ignoring the other node types. This makes the trace- and tree-processing code simpler and more flexible.

Search procedures

Once concolic testing is under-way, the search procedure is responsible for choosing unexplored branches in the tree as the next targets for exploration. Given an unlimited time budget, the entire tree will be explored, so the search order is not important. In practice however, we will only be able to explore a small part of the full tree, and will aim to explore as much of the application as possible within the budget. We want to choose a search order which leads to interesting application behaviours and improves the exploration metrics (which could be branch coverage, line coverage, function coverage, etc.) as soon as possible.

Table 3.4: The node types which make up our symbolic traces and execution tree.

Node type	Description
Symbolic branch	A branch instruction with a symbolic condition.
Difficult branch	A symbolic branch whose condition cannot be solved.
Concrete branch	A branch instruction without a symbolic condition.
Unexplored marker	An execution path which has not yet been explored.
UNSAT marker	An execution path which is unreachable because its path constraint is unsatisfiable.
Unsolvable marker	An execution path which is unreachable because its path constraint cannot be solved.
Missed marker	An execution path where exploration was attempted, but the execution took a different path.
Queued marker	An unexplored path which has been selected by the search procedure for exploration, but whose corresponding input values have not yet been tested.
Alert	A JavaScript alert box was shown.
Console message	A message was sent to the JavaScript console. This includes calls to console.log and console.error, but also exceptions in the executing code.
DOM modification	A marker added at the end of a trace to describe how the page's DOM was modified.
Page load	A new page was loaded (for example after clicking a button).
Action marker	Marks the start of each action in the action sequence.
Function call	A JavaScript function was called.
Concrete summary	The platform allows "boring" stretches of concrete execution—function calls and concrete branches—to be summarised into a single node, which makes the trees more compact and easier to read.
End marker	The end of each execution trace. The trace classifier can also add these markers into the trace as soon as it determines the trace is a success or a failure, to show that nothing after that point is interesting to the analysis.

The platform's default search procedure is a form of depth-first search with iterative deepening. The tree is searched depth-first down to some depth limit; when the search is exhausted, the depth limit can be increased and the search continues. This procedure is not "smart" in any way; it cannot drive execution towards the most interesting parts of the code, but it does provide a reasonably broad search to quickly cover different parts of the code being tested. The main downside is the focus on branches which occur early in the execution trace. Branches at the end of the trace are not explored at all until the earlier ones have been thoroughly analysed.

To combat this problem, we have introduced other search procedures. The platform includes a simple, flexible approach to defining new search procedures, so new procedures are simple to add and evaluate. So far, we have implemented the iterative deepening described above, a random-order search, a procedure which aims to avoid redundantly exploring many unsatisfiable branches, and a procedure which uses round-robin scheduling to combine multiple procedures together.

Avoiding repeated unsatisfiable branches is useful because of a specific pattern which we often encountered on real sites; the site would check whether an input was part of a certain set using a loop, checking the input against each element in turn. This generates a long chain of branches which are mutually exclusive: once one is satisfied, none of the others ever will be. This pattern leads to long chains of "boring" execution in the tree, where it is easy for the search procedure to get stuck searching hundreds of branches—all corresponding to the same source-code branch—which are almost all unsatisfiable. Our new search procedure uses information about the source-code branches which correspond to each symbolic branch node to determine which branches have often been seen to be unsatisfiable in the past. These "often unsatisfiable" branches are then de-prioritised in the search order in favour of branches which have not yet been explored or which have more often been satisfiable. This search procedure is called "avoid-UNSAT" and is useful to avoid wasting time in analysing long loops in the recorded traces and allows the testing to skip forward to the more interesting code which comes after the loop.

Keeping each iteration consistent

As well as ensuring a clean starting state for each iteration of the concolic analysis (by resetting cookies, and so on, as described earlier), we must ensure that all the recorded traces behave in a consistent way. The first step is obviously for the analysis to trigger a consistent sequence of events on each iteration. However, other sources of inconsistency between traces include: (a) race conditions between simulated user events, AJAX requests, or timers; (b) random numbers and the current date and time; (c) external environment information, such as the results of AJAX calls; and (d) server-side inconsistency between requests.

The platform keeps tight control over the browser's execution. By design, JavaScript events are synchronous—one event handler cannot interrupt another—and the platform

fires the events one at a time in a fixed, known order. The browser is also prevented from processing any events on the page while the platform's analysis code runs so that, for example, the page can never be updated while the DOM is being inspected, or constraints are being solved.

AJAX calls are handled synchronously in our browser. Normally when an AJAX request is made control returns immediately to the page, which continues processing other events. When the response eventually arrives, the event handling callback is added to the browser's event queue, ready to be executed. Our instrumented browser blocks execution when an AJAX request is sent until the response is returned and the response handler event is queued. This means the callback events will be triggered in a predictable, repeatable order.

Timers are handled similarly; they are queued until the browser is next idle, then fired in sequence. If the timer's callback itself registers new timers then this process is repeated for a certain number of iterations. Recurring or nested timers (often used to run animations, for example) are triggered a few times and then dropped, so that the main test execution can proceed.

Synchronous AJAX and timer handling together mean there are no outstanding asynchronous events after the platform executes an action. Events fire in a consistent, deterministic order, and the browser is always left in a clean state where no "unexpected" execution can take place and affect the symbolic recording of future actions.

JavaScript APIs which provide non-repeatable results, such as the Date.now() and Math.random() methods could be modified to return fixed, repeatable information, which is reset at the start of each iteration. However this has not proved to be necessary—these calls have not been problematic in our testing so far.

Some interactions cause state to be saved on the remote server which will affect future tests. For example, when a user registration form is filled, the new user is saved into the server's database. If a subsequent test fills the form in the same way, the otherwise valid submission may be rejected because a user with that name already exists. Although this change of behaviour is expected and deterministic, it is impossible for an analysis which only considers client-side behaviour to understand. Our tools are designed to work on reusable interface components, such as query forms, and not "single-shot" actions like registering a new user, so these issues are not tackled.

Servers which don't return consistent data on repeated requests, or remote APIs which provide other dynamically changing information from the environment can never be completely controlled without a pre-existing knowledge of the requests a JavaScript application can make to external resources. A partial solution, which turned out not to be feasible in practice, is to create static mirrors of the sites to be tested. Note that small or otherwise irrelevant changes, such as to a page's adverts, are not a problem for our analysis.

Handling nondeterminism and inconsistency in the recorded traces
Inconsistent traces cannot be entirely prevented, and in the web setting, a concolic testing platform must be robust enough to handle these cases. These mismatching traces appear to the analysis as nondeterminism in the code being analysed, whether they are caused by "real nondeterminism" or not. Even branches which deterministically depend on the inputs will appear to be unpredictable if the symbolic interpreter is missing some instrumentation required to assign the branch a symbolic condition. Some of these issues have been prevented or mitigated, for example by resetting the browser, ignoring cookies, and removing race conditions between asynchronous events. However, some sources of inconsistent traces are impractical or impossible to remove. For example, if AJAX requests were cached and replayed to avoid them returning unpredictable information, then the analysis would be changing the behaviour of the web page, maybe in important ways, which is not a useful trade-off.

There are two ways that a newly recorded trace can disagree with the existing partial execution tree. In the first case, the first branch where the new trace takes a different path to those already in the tree is concrete instead of symbolic. This represents an execution which has missed its intended target. The platform is never able to intentionally explore a concrete branch, seeing as it has no symbolic condition. In the second case, the new trace and the existing tree disagree on even what code was executed. For example if the tree contains a symbolic branch in a place where the new trace contains a function call, the trace is said to have *diverged* from the expected behaviour. These divergent traces represent different or unexpected code being executed by the browser, and cannot be merged sensibly for concolic testing. This can occur if there is something inconsistent about the action sequence, if the web server returns different content on different iterations, or from certain kinds of code which are not symbolically understood.

When any disagreement between traces is detected, both cases are saved into the tree as normal for later analysis. The search procedure can then decide how to handle these unexpected traces. In our implementation, concrete branches which are already explored on both sides can be explored by the search procedure (they just do not result in a branch condition being added to the path constraint), but divergences are ignored.

In real tests, true nondeterminism in the recorded traces was not common and did not cause a big problem for the analysis. Concrete branches which were explored on both sides were seen on many sites, but as the search procedure treats these as normal branches (just without symbolic conditions) they did not hold up the analysis. Because they are unpredictable, these branches result in many missed exploration targets in their subtrees, where the recorded trace takes the corresponding path but on the opposite branch path. Trace divergence was rare, although it was seen occasionally, and the root causes are unknown.

Concolic testing as a source of advice

The concolic testing engine is designed as an *advice provider*, and is not necessarily in direct control of the tests being run. That is, it is not the overall controller of the system's actions, it simply provides advice which the rest of the analysis system can use. This feature is very useful where a full analysis system relies on multiple sources of information—not only concolic testing—to determine the next steps.

Decoupling the concolic testing from the main analysis adds flexibility in two key places: (a) when a set of new input values are returned by the concolic testing, they do not need to be tested immediately, and (b) the main test driver is free to execute other tests, using values which were never suggested by the concolic testing. These features are common in other systems which combine fuzzing and concolic testing, for example.

This flexibility is implemented by introducing "queued" suggestions which have already been made but which have not yet been tested. When a target branch in the tree is chosen by the search procedure, the path constraint is solved to generate a new set of inputs to test that branch. The branch is marked as queued, and the new inputs are returned to the calling code, along with an identifier which the concolic engine can use later to look up the queued branch. Now the concolic testing engine is free to operate as normal, recording new traces and making new suggestions. When the calling code decides to test a suggestion made by the concolic analysis, it notifies the analysis engine to begin recording a new trace, and (optionally) provides the identifier of the queued branch back to the analysis so that the analysis can match up the recorded trace with the suggestion. In the traditional presentation of concolic analysis, this matching up is implicit—once a new set of values is generated it is immediately tested, so the target branch is always known.

Another feature which allows the concolic testing platform to be used flexibly is running multiple instances of the concolic engine simultaneously. This allows different parts of a web application—that is, different sequences of test actions—to be tested without needing to exhaustively test the first part before moving on. With separate analyses for each part, an analysis tool is free to use other heuristics to choose which tests to run, while still building up a useful base of concolic testing results.

Supporting multiple concolic states concurrently in the system is not especially difficult to implement. We simply make sure that all concolic state—the execution tree, the current progress, information about the DOM, and so on—is stored locally in a ConcolicAnalysis class which can be instantiated multiple times without each instance affecting the others. It is not required for the separate testing engines to actually be able to execute in parallel; only one analysis is actually used at a time. For example there is only a single instance of the solver, which is never used in parallel.

3.6 Experimental evaluation

Our concolic testing platform supports web-based JavaScript, but does not itself model any interactions with the DOM. Therefore, it is evaluated on standalone JavaScript programs, that is, scripts which take inputs directly and are not part of a web page. Thus, the evaluation of the platform is analogous to traditional concolic testing. For the experiments, we randomly generated a suite of 1000 synthetic JavaScript programs which are representative of the code encountered on real websites. The platform is compared with Jalangi, a framework for JavaScript testing that includes concolic testing for standalone JavaScript [121]. The aim of these experiments is to show that the core platform provides a competitive implementation concolic testing for web JavaScript, and is therefore a good base on which to build further web-focused analysis tools.

Generating synthetic examples

To generate small example programs to test, we use a context-free grammar for a small language of expressions. Sampling from this grammar produces expressions which depend on certain input variables. The expression is then translated into a random JavaScript program which implements it. The expression language is distinct from the internal constraint language used by the concolic engine, as they serve different purposes, although they do share some similar structure.

The grammar for expressions is constructed in two phases: first, a basic grammar defines the structure of expressions, and then this grammar is modified to enforce certain desirable properties of the expressions it allows. The base grammar is given in Figure 3.5. This grammar is programmatically modified to ensure that no sub-expression may be composed entirely of constants; that is, every sub-expression must either be a constant value itself, or contain an input. This prevents expressions such as

IntegerBinaryOperation(**IntegerConstant**(6), **INT_ADD**, **IntegerConstant**(9))

being generated. This is a purely mechanical transformation, resulting in a new grammar. The new grammar could be defined directly, but generating it in two phases avoids having to manually specify many duplicate terms.

Expressions are sampled from the grammar using a standard algorithm, which ensures different expressions are sampled uniformly [98]. This algorithm also allows the length of the sampled string to be fixed. Our generator chooses a length at random, normally distributed around a target length, giving some variation in program size, but disallowing very short or very long programs. Once each expression is generated, it is translated to JavaScript term-by-term. Each expression term has several possible JavaScript implementations, one of which is chosen at random. Combining the implementations for each term produces a JavaScript program implementing the entire expression. For example, the term **BooleanBinaryOperation**(x, **BOOL_AND**, y) can be implemented in seven different ways, illustrated in Table 3.5.

⟨START⟩ ⊨ ⟨BOOL_FUN⟩ | ⟨INT_FUN⟩ | ⟨STR_FUN⟩
⟨BOOL_FUN⟩ ⊨ BooleanFunction(⟨BOOL_ITE⟩)
⟨INT_FUN⟩ ⊨ IntegerFunction(⟨INT_ITE⟩)
⟨STR_FUN⟩ ⊨ StringFunction(⟨STR_ITE⟩)
⟨BOOL_ITE⟩ ⊨ BooleanITE(⟨BOOL_EXPR⟩ , ⟨BOOL_EXPR⟩ , ⟨BOOL_EXPR⟩)
⟨INT_ITE⟩ ⊨ IntegerITE(⟨BOOL_EXPR⟩ , ⟨INT_EXPR⟩ , ⟨INT_EXPR⟩)
⟨STR_ITE⟩ ⊨ StringITE(⟨BOOL_EXPR⟩ , ⟨STR_EXPR⟩ , ⟨STR_EXPR⟩)
⟨BOOL_EXPR⟩ ⊨ ⟨BOOL_INPUT⟩ | ⟨BOOL_ITE⟩ | ⟨BOOL_FUN⟩ | ⟨BOOL_CONST⟩
| ⟨BOOL_EVAL⟩ | BooleanNot(⟨BOOL_EXPR⟩)
| BooleanBinaryOperation(⟨BOOL_EXPR⟩ , ⟨BOOL_OP⟩ , ⟨BOOL_EXPR⟩)
| BooleanBinaryOperation(⟨INT_EXPR⟩ , ⟨INT_CMP⟩ , ⟨INT_EXPR⟩)
| BooleanBinaryOperation(⟨STR_EXPR⟩ , ⟨STR_CMP⟩ , ⟨STR_EXPR⟩)
| IntToBool(⟨INT_EXPR⟩) | StrToBool(⟨STR_EXPR⟩)
| RegexTest(⟨STR_EXPR⟩ , random_regex)
| ElementInSet(⟨INT_EXPR⟩ , random_integer_set)
| ElementInSet(⟨STR_EXPR⟩ , random_string_set)
⟨INT_EXPR⟩ ⊨ ⟨INT_INPUT⟩ | ⟨INT_ITE⟩ | ⟨INT_FUN⟩ | ⟨INT_CONST⟩ | ⟨INT_EVAL⟩
| IntegerBinaryOperation(⟨INT_EXPR⟩ , ⟨INT_OP⟩ , ⟨INT_EXPR⟩)
| StringLength(⟨STR_EXPR⟩)
| BoolToInt(⟨BOOL_EXPR⟩)
| StringToInt(⟨STR_EXPR⟩)
⟨STR_EXPR⟩ ⊨ ⟨STR_INPUT⟩ | ⟨STR_ITE⟩ | ⟨STR_FUN⟩ | ⟨STR_CONST⟩ | ⟨STR_EVAL⟩
| StringBinaryOperation(⟨STR_EXPR⟩ , ⟨STR_OP⟩ , ⟨STR_EXPR⟩)
| BoolToStr(⟨BOOL_EXPR⟩) | IntToStr(⟨INT_EXPR⟩)
| StringReplace(⟨STR_EXPR⟩ , random_string)
| StringRegexReplace(⟨STR_EXPR⟩ , random_regex)
⟨BOOL_OP⟩ ⊨ BOOL_EQ | BOOL_NEQ | BOOL_SEQ | BOOL_SNEQ | BOOL_AND | BOOL_OR
⟨INT_CMP⟩ ⊨ INT_EQ | INT_NEQ | INT_SEQ | INT_SNEQ | INT_LT | INT_LEQ
| INT_GT | INT_GEQ
⟨INT_OP⟩ ⊨ INT_ADD | INT_SUBTRACT | INT_MULTIPLY | INT_DIVIDE | INT_MODULO
⟨STR_CMP⟩ ⊨ STR_EQ | STR_NEQ | STR_SEQ | STR_SNEQ | STR_LT | STR_LEQ
| STR_GT | STR_GEQ
⟨STR_OP⟩ ⊨ STR_CONCAT
⟨BOOL_EVAL⟩ ⊨ BooleanEval(⟨BOOL_EXPR⟩)
⟨INT_EVAL⟩ ⊨ IntegerEval(⟨INT_EXPR⟩)
⟨STR_EVAL⟩ ⊨ StringEval(⟨STR_EXPR⟩)
⟨BOOL_INPUT⟩ ⊨ BooleanInput(⟨BOOL_VAR_NAME⟩)
⟨INT_INPUT⟩ ⊨ IntegerInput(⟨INT_VAR_NAME⟩)
⟨STR_INPUT⟩ ⊨ StringInput(⟨STR_VAR_NAME⟩)
⟨BOOL_VAR_NAME⟩ ⊨ “B1” | “B2” | “B3”
⟨INT_VAR_NAME⟩ ⊨ “I1” | “I2” | “I3”
⟨STR_VAR_NAME⟩ ⊨ “S1” | “S2” | “S3”
⟨BOOL_CONST⟩ ⊨ BooleanConstant(random_boolean)
⟨INT_CONST⟩ ⊨ IntegerConstant(random_integer)
⟨STR_CONST⟩ ⊨ StringConstant(random_string)

Figure 3.5: The expression grammar for the synthetic JavaScript program generator.

Table 3.5: The different ways that **BooleanBinaryOperation**(x, **BOOL_AND**, y) can be implemented by the JavaScript generator.

Pre-processing	Expression
(none)	(x && y)
var b1 = (x && y);	b1
var b1 = x; **var** b2 = y;	(b1 && b2)
var b1; **if**(x && y) { b1 = **true**; } **else** { b1 = **false**; }	b1
var b1 = **false**; **if**(x && y) { b1 = **true**; }	b1
var b1; **if** (x) { b1 = y; } **else** { b1 = **false**; }	b1
var b1; **if** (y) { b1 = x; } **else** { b1 = **false**; }	b1

The generated JavaScript programs are representative of real-world code, although they are clearly not fully realistic. The grammar for program expressions shown in Figure 3.5 is different to the internal symbolic language used for the platform's concolic testing. Constraints cannot "round-trip" perfectly from the expression grammar, via the synthetic JavaScript programs, to the internal symbolic expressions used in the analysis. Some terms from the expression grammar, such as functions and set containment checks, have no analogue in the symbolic constraint language. In particular, the generated JavaScript programs may include code which the platform cannot analyse.

The main limitation of the synthetic example generator is that it cannot generate complex loops. Because the programs are generated from functional expressions, it is difficult to introduce loops which perform any significant computation. Although this is an obvious departure from real code, the examples are still appropriate for concolic testing. Concolic testing relies on individual branches (whether loop conditions or if statements) and does not directly analyse higher-level structures like loops.

Example. To demonstrate the whole generation process, we will present the first example generated for our experiments. The expression sampled from the CFG (with the random constants filled in) is shown in Listing 3. It contains 79 expression terms and 7 distinct inputs. Then the JavaScript program shown in Listing 4 was randomly generated from this expression. □

Results

The experiments used 1000 randomly generated JavaScript programs. The programs ranged in size from 2 lines (where the code is simply one big eval call) up to 91 lines (with 25.4 lines on average), and took between 2 and 9 inputs (5.6 on average). Each example was tested in both our platform and Jalangi's concolic testing mode with no time or iteration budget; each tool was allowed to run until its analysis was complete.

Listing 3 An example expression generated from the CFG.

```
StringFunction( StringITE(
  BooleanITE(
    BooleanBinaryOperation(
      StrToBool( BoolToStr( IntToBool( IntegerInput( "I3" ) ) ) ),
      BOOL_AND,
      BooleanEval( BooleanITE(
        BooleanITE(
          BooleanConstant( false ),
          BooleanBinaryOperation(
            BooleanInput( "B2" ),
            BOOL_NEQ,
            BooleanBinaryOperation(
              BooleanConstant( true ),
              BOOL_SNEQ,
              IntToBool( StringLength( StringInput( "S1" ) ) ) ) ) ),
          BooleanITE(
            IntToBool( IntEval( IntEval( StringToInt( StringEval( StringITE(
              BooleanInput( "B2" ),
              StringConstant( "" ),
              StringInput( "S1" ) ) ) ) ) ) ),
            BooleanNot( BooleanBinaryOperation(
              BooleanInput( "B2" ),
              BOOL_EQ,
              BooleanEval( BooleanBinaryOperation(
                BoolToStr( BooleanInput( "B2" ) ),
                STR_GEQ,
                IntToStr( IntegerITE(
                  BooleanInput( "B3" ),
                  StrToInt( StringInput( "S1" ) ),
                  IntegerConstant( 54 ) ) ) ) ) ) ),
            IntToBool( IntegerInput( "I1" ) ) ) ),
        IntegerEval( IntToBool( IntegerInput( "I3" ) ) ),
        BooleanInput( "B2" ) ) ) ),
    BooleanNot( BooleanNot( BooleanInput( "B2" ) ) ),
    BooleanEval( BooleanInput( "B2" ) ) ),
  StringInput( "S3" ),
  StringInput( "S2" ) ) )
```

Listing 4 A sample JavaScript program generated from the expression in Listing 3.

```
1:  var fn56 = function (s54, b24, x1, b13, x35, s14, s53) {
2:    var b47 = b13; var b50 = b13; var x39 = x1;
3:    var s3 = ( !! (x1)).toString();
4:    var b4 = (s3) ? true : false;
5:    var b45;
6:    if (b4) {
7:      var b42 = b13; var b5 = b13; var s6 = s14;
8:      var b8 = !! ((s6).length);
9:      var b11;
10:     if (b5) { b11 = !(true !== b8); } else { b11 = (true !== b8); }
11:     var b21 = b13; var s25 = s14;
12:     var e19 = "eval(\"var x15; if (b13) { x15 = \\\"\\\"; } else { x15 = s14; } Number(eval(\\\"x15\\\"))\")";
13:     var b36 = Boolean(x35);
14:     var x37;
15:     if ((eval(e19)) ? true : false) {
16:       var b22 = b21; var x27;
17:       if (b24) { x27 = parseInt(s25, 10); } else { x27 = 54; }
18:       var b29 = String(b22) >= '' + (x27);
19:       var e31 = "b29";
20:       var b32 = b21;
21:       var b33 = eval(e31);
22:       var b34;
23:       if (b32 == b33) { b34 = false; } else { b34 = true; }
24:       x37 = b34;
25:     } else {
26:       x37 = b36;
27:     }
28:     var x38;
29:     if (false) { x38 = b11; } else { x38 = x37; }
30:     var e41 = "!!(x39)";
31:     var x43;
32:     if (x38) { x43 = eval(e41); } else { x43 = b42; }
33:     b45 = eval("x43");
34:   } else {
35:     b45 = false;
36:   }
37:   var x52;
38:   if (b45) {
39:     var b48;
40:     if (b47) { b48 = false; } else { b48 = true; }
41:     x52 = !(b48);
42:   } else {
43:     var e51 = "b50";
44:     x52 = eval(e51);
45:   }
46:   return (x52 ? s53 : s54);
47: }
48: fn56(input_S2, input_B3, input_I3, input_B2, input_I1, input_S1, input_S3);
```

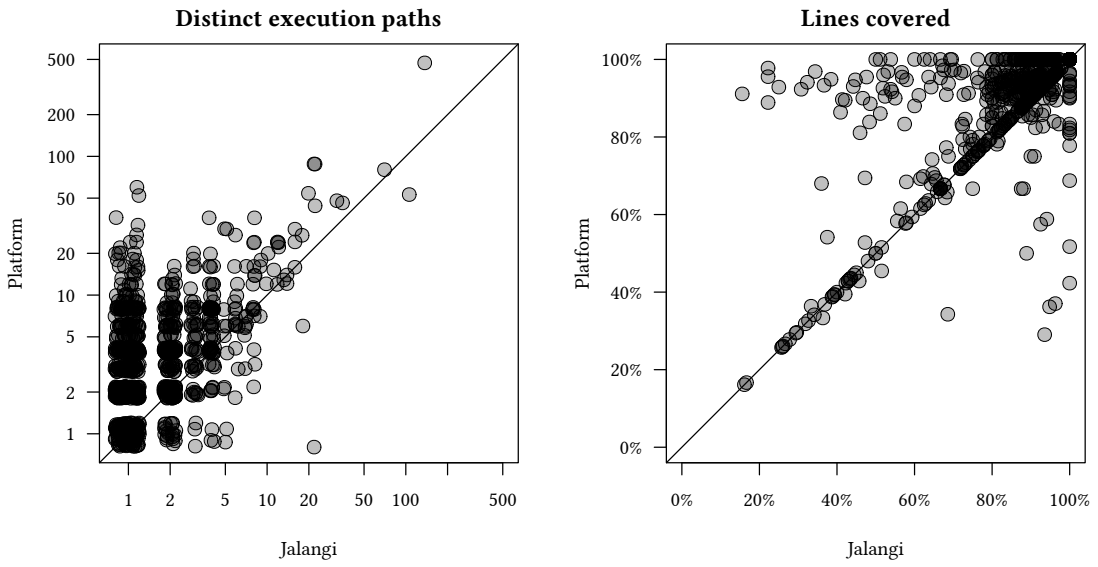


Figure 3.6: Paths explored and lines covered by our platform and Jalangi on 1000 synthetically generated JavaScript programs. In both cases, examples above the diagonal are explored more by our system, those below more by Jalangi.

Because the examples are relatively small, this is not unrealistic; the average number of iterations required was only 6.00 for our platform, and 5.24 for Jalangi.

The summarised results are shown in Table 3.6, with a breakdown by test-case shown in Figure 3.6. Our concolic testing explores many more paths than Jalangi in general. It benefits from more symbolic instrumentation of JavaScript built-ins, native string operations, and datatype coercions which are not supported (and may be difficult to support) in Jalangi. Part of this benefit comes from using CVC4, which is more advanced, especially for string solving, than the older version of CVC used by Jalangi. It seems that our platform is more often able to prove that certain branches are unreachable and therefore avoid attempting exploration in these cases. Jalangi's concolic testing generated values and attempted to explore a new path which it was then unable to reach in 16.4% of the examples, whereas our platform could prove that many of these cases were unreachable, and only made unsuccessful explorations in 0.6% of the examples.

Jalangi's test cases

As well as our own synthetically generated test programs, we also tested the platform on Jalangi's concolic test suite. These tests are mostly hand-written as unit tests, but they also include some examples taken from tutorials, which are somewhat representative of real-world code. In total, there are 45 test programs, using standalone JavaScript. We measured the number of distinct execution paths the platform managed to explore

Table 3.6: Comparison our platform with Jalangi on 1000 synthetic JavaScript programs.

	Platform	Jalangi
Avg. no. iterations	6.00	5.24
Avg. no. distinct paths explored	5.75	2.54
Examples with >1 path explored	81.63%	42.31%
Avg. line coverage	91.49%	87.69%
Examples with 100% line coverage	46.44%	37.05%

Table 3.7: Our platform's results on Jalangi's 45 concolic test cases.

Result	Tests	
Successfully solved	30	(66.7%)
No modelling of symbolic objects	7	(15.6%)
Non-instrumented built-ins	7	(15.6%)
Non-linear constraints	1	(2.2%)
Total	45	

in each program, as well as the number of lines covered during the testing. As before, there was no iteration limit, and the numbers of iterations required were low enough that this did not matter (the most required was 24, to test a small quicksort function). The results are shown in Table 3.7. Note that as these examples are Jalangi's own test cases, it can fully explore them all.

Our platform does not support general objects as inputs, only primitive values. As such, our internal constraint language cannot describe complex constraints on object values as they cannot be symbolic in our system. Jalangi does support such constraints, and thus several of their test cases cannot be solved in our platform.

The platform also fails some tests because of some un-instrumented built-ins. They are not instrumented simply because we have not encountered them on the web pages we have tested so far. In general they would not be problematic to add. For example, we support the indexOf function which searches a string for a given substring, but three of Jalangi's tests use lastIndexOf, which we do not have symbolic support for. Adding support requires adding a new term in the internal constraint language and translating it for the solver. CVC4 has a function **str.indexof** but not **str.lastindexof** or anything equivalent, so some rewriting would be required. Another example uses charCodeAt, which returns the character code of the character at a given index in a string. Our platform already supports charAt, so to support charCodeAt as well we would include a new relation mapping characters to their codes (effectively a look-up table) in our SMT constraint output.

One example uses a non-linear arithmetic constraint, which our platform does not solve. CVC4 can solve non-linear constraints, but at the cost of performance and correctness guarantees. As we have not encountered non-linear constraints on real sites, we have never enabled this option. For Jalangi's more general-purpose JavaScript analysis, the trade-off of allowing non-linear constraints is likely more beneficial.

3.7 Discussion

Our experiments show that our high-level approach, along with our bytecode-based implementation, has a lot of promise in practice. Our concolic testing engine is competitive, and the bytecode-level symbolic interpreter avoids many of the difficulties usually encountered in JavaScript analysis. Having the JavaScript pre-processed into a stream of bytecode instructions simplifies both the internal constraint language and constraint solving. There are, however, a number of limitations in the implementation which we will now discuss.

Being built on an instrumented web browser allows our platform to test code which other tools would find difficult. For example, Section 3.5 describes how asynchronous code using timers and AJAX callbacks can be fixed in a deterministic order so it can be tested consistently. The platform does not test all interleavings of asynchronous code—looking for race conditions, for example—it just executes such code deterministically so that different test runs can be directly compared.

Not every JavaScript built-in function is instrumented symbolically in our interpreter, or in some cases they are instrumented only for common or simple cases. Increasing the coverage of our symbolic instrumentation would reduce the incidences of symbolic information being "lost" in the interpreter (when a symbolic value is passed to an un-instrumented function and the result is returned with no symbolic tag). However, it would also increase the complexity of the constraints, so the new instrumentation is only useful when the solver is able to handle the new constraint types.

For example, the string methods toUpperCase and toLowerCase are instrumented, but there are no corresponding methods in CVC4. We have implemented SMT relations which match upper- and lower-cased versions of strings up to a limited length bound. This allows simple uses of toUpperCase and toLowerCase to be solved, but these functions still cannot be handled well in general.

Another area where the platform is currently limited by the constraint translation and the solver is regular expression constraints. There are two issues: first, non-regular expressions which the solver cannot handle, and second, the need to analyse complex objects returned by many regular-expression functions. As mentioned in Section 2.1, JavaScript's RegExp language includes many non-regular features. CVC4 supports regular expressions, but not these extensions. Thus, constraints which make use of

these extended expressions cannot be solved. Techniques exist to decompose such expressions into SMT constraints which can be solved [94], but these have not been applied in the platform. There are also some more subtle encoding restrictions which prevent the constraint translator from handling Unicode or negated character classes in regular expressions, some of which could be addressed with further engineering work, and some of which are imposed by the solver. The second issue relates to the JavaScript regular expression functions themselves. The simple case, RegExp.test, is handled correctly; it returns true if the expression matches and false otherwise. Other regular expression functions however, return more complex objects. For example, both RegExp.exec and String.match return an array of matches, or null if there are none. Our internal symbolic constraint language has no generally-applicable support for nulls, arrays or objects. We have added special-purpose support for the most common patterns observed when these regular expression functions are used (essentially, when they are used like RegExp.test), but more complex uses cannot be represented or solved.

There are other areas where the solver's support for certain constraint types does not exactly match JavaScript. For example, CVC4's integer type does not include any NaN (not-a-number) value. In JavaScript, if a non-numeric string is parsed into a number, the result is NaN. In CVC4 it is -1. As there is also no way to check for NaN in the SMT-LIB language (seeing as it is not a value at all), this means that certain constraints can be solved incorrectly, and the suggested values do not take the expected path when tested. The platform could in principle encode a model of JavaScript's NaN semantics into its SMT constraints. This would require tracking for each intermediate value whether it is NaN, and under what conditions, greatly increasing the complexity of the constraints. We instead take the opposite approach, using as direct a translation of the constraints as is feasible, and permitting certain corner cases to not be solved correctly. This "best effort" approach to constraint solving is a pragmatic middle-ground, which allows the vast majority of real-world constraints to be solved efficiently.

In general, the internal symbolic constraint language is tailored to its application of modelling inputs from forms and the DOM, and is not suited to fully-general JavaScript analysis. The input variable types are limited to strings, integers and Booleans, with no support for symbolic objects outside of special-purpose support for certain patterns, such as the regular expression matching discussed above. This is not a limitation for our application, and in fact simplifies the constraints (both internally and when sent to the solver), but it precludes reasoning about more general standalone JavaScript functions, which may operate on complex objects in ways our platform cannot model.

In certain cases, it would be beneficial to generate constraints at a higher level of abstraction. For example, when a page checks whether an input value is a member of a certain set, the bytecode-based interpreter generates a sequence of symbolic branches, checking whether the input is equal to each of the set members in turn (as discussed

earlier in relation to the platform's search procedures). This inflates both the size of the constraints and the analysis search space compared with generating a single set-containment constraint in a single symbolic branch. Such high-level constraint generation would be possible if the symbolic interpreter could recognise certain well-known library functions or patterns, and assign them a pre-defined symbolic representation. Such higher-level constraints would also mirror the developer's original intent more closely, making them more appropriate for modelling a website's user-interface rules.

Symbolic information can be lost if values leave the interpreter; for example if a symbolic value is saved into the DOM and later retrieved. The DOM contents are not modelled symbolically or instrumented, so any symbolic information saved into the DOM will lose its symbolic tag. In contrived situations it is *always* possible for symbolic information to be lost by leaving the control of the interpreter; for example an input value could be sent to a remote server via AJAX and returned unchanged, but no symbolic execution would be able to prove that the new value must always be identical to the original input without knowledge of the server-side processing. The DOM is relevant because some real web applications *do* store aspects of their state in the DOM, and it would be useful to be able to track these symbolically.

In all, our concolic testing platform is a useful and powerful base on which to build web interface analysis tools. A completely full-featured implementation of bytecode-based symbolic tracing in a state-of-the-art browser remains a major engineering challenge. With a bytecode-based approach, it becomes more difficult—though not impossible—to link analysis-level and source-code-level information, compared to a source-code-based approach as used by Jalangi. For example, if it is necessary to perform symbolic analysis of library code which cannot be analysed directly, then a symbolic model for the library can be included in place of the library calls much more simply and directly in a source-code-based approach. It may be useful to investigate variants of our approach which can perform the symbolic tracing at the JavaScript level, working with the code on-the-fly.

Our work on bytecode-based symbolic tracing within WebKit began before Jalangi's publication. If Jalangi had been available at the time, we would have strongly considered using it's symbolic tracing as the base for our analysis platform, rather than building our own symbolic interpreter. In this scenario, Jalangi would be used to instrument the web page's JavaScript code and record the symbolic traces. Those traces would be passed to our system and processed as normal, passing through a similar constraint translation and solving process as we have today. The advantage would be that the symbolic tracing would become browser-independent, and new browser features or APIs could be incorporated into the system much more easily. Such a system would inherit some of Jalangi's limitations, such as limited control over the browser (for example to issue GUI-level click events) or internal browser operations (such as keeping AJAX callback events deterministic). Communication between the running symbolic interpreter and other analysis features would also become more difficult. Even if some level of browser

instrumentation were still used, this approach would depend much less on specific browser versions, allowing platform to better keep up with browser updates. Although it is unclear which approach would be best overall, combining symbolic tracing from Jalangi with the rest of our system has some compelling advantages.

3.8 Related work

This section describes related work on standard (that is, not JavaScript- or web-based) concolic testing, and then discusses current approaches to JavaScript analysis, both standalone and on the web.

Concolic testing and constraint solving

Concolic testing was first introduced for C with the DART [50] and CUTE [122] tools, which developed the idea of combining symbolic execution with concrete information to drive the analysis forwards and through constraints which couldn't be solved by symbolic execution alone. More modern concolic testing tools use sophisticated techniques to reduce the search space of the analysis, such as specific handling for loops [118], which could be directly useful to my concolic testing tool (the core of which is not JavaScript or web-specific). Concolic testing has also been extended to test concurrent programs [40].

SAGE is an automated white-box fuzzing tool developed at Microsoft [51]. It is used for testing and bug-finding for Windows and Office and as such is the largest fuzzing system and the largest single application of SMT solving in the world [48]. SAGE has developed many state-of-the-art concolic analysis techniques which make analysis of such large programs feasible, such as dealing with input-dependent loops [52], and creating symbolic summaries of functions to avoid repeatedly analysing the same code again and again [3].

KLEE is another modern concolic analysis tool, which analyses LLVM bytecode (for example generated from a C program) to discover bugs [23]. Analysing LLVM bytecode means that many languages can be analysed in a language-agnostic way, and many compiler bugs or optimisations which cause problems will also be analysed.

Some similar techniques to ours are used in Apollo, a web application testing framework for server-side PHP code [9]. Apollo combines concrete and symbolic execution with model checking techniques to automatically generate test cases which may uncover bugs in the server-side of web applications.

Concolic testing requires high-performance SMT solvers to generate new test inputs. The most commonly used SMT theories are Booleans and bit-vectors, but constraints over integers, reals or arrays are also common. Some examples of solvers used for concolic testing or symbolic execution include Boolector [109], CVC4 [16], STP [46], Yices [36], and Z3 [32].

For our work, we require very strong support for the theory of strings. In particular, the more string functions the solver supports built-in, the easier our constraint translation becomes. When analysing web-based JavaScript, transformations between strings and other data types are common, so it is very useful if the solver supports these coercions directly.

Initially we tried Kaluza [116], the solver developed for use with Kudzu (discussed below) but we found it could not handle enough of the constraints generated by our tools. We also tested Z3-str [144], which is a plugin for the main Z3 solver which adds support for strings and string functions. Finally, we settled on CVC4. CVC4 supports a wide variety of theories compared with other solvers [29], strong string and regular expression support [90], as well as support for constraints which involve coercions between theories, which many other solvers have very little support for. It is worth noting that since we moved from Z3 to CVC4, the string support in Z3 has improved, and they have added support for regular expression solving, which were our two biggest motivations to move originally.

Standalone JavaScript analysis

As JavaScript has become more and more ubiquitous, there has been a lot of work on static analysis of JavaScript. Static analysis is difficult, because of JavaScript's highly dynamic nature, but progress has been made [6, 47, 58, 64, 67, 87, 128]. Much of this work focuses on certain simpler subsets of the language however, to avoid difficult combinations of features which can arise in full JavaScript. Studies on the real-world usage JavaScript show that these subsets are not enough to analyse real websites without developer support [115].

Some full tools for static analysis of JavaScript (as opposed to theory and techniques) include RATA, a type inference tool used to optimise JavaScript execution [93]; Flow, a type checker which can understand common JavaScript idioms [27]; WALA, a general-purpose static analysis framework including pointer and data-flow analysis which can be applied to JavaScript [130]; Actarus, a taint-analysis tool for detecting client-side security vulnerabilities [57]; and TAJS, a static analysis framework including DOM modelling, which is discussed below. JavaScript code checking tools, such as JSHint,[20] JSLint,[21] or ESLint,[22] are also based on static analysis. Many such tools are based on Esprima, a JavaScript parser implemented in JavaScript itself [39].

Other tools can be thought of a JavaScript re-compilers, and perform some analysis while translating JavaScript from one form to another. Google's Closure compiler is a JavaScript-to-JavaScript compiler which performs analysis to identify issues with the code [55]. It is used by web developers both as an optimiser (by minimising the code),

[20] http://jshint.com/about/
[21] http://www.jslint.com/
[22] https://eslint.org/

and as a code quality checker which can remove dead code, perform syntax- and type-checking, and warn about potentially dangerous patterns. TypeScript is not directly a JavaScript analysis tool, but a new language—a superset of JavaScript which adds type annotations to allow more powerful static analysis [132]. TypeScript's compiler translates TypeScript source code into standard JavaScript, so it can be used on the web with existing browsers, but the annotations allow much more sophisticated compile-time type checking and verification than would be possible with pure JavaScript.

Web-based JavaScript analysis

There are many existing tools which analyse JavaScript in a web setting. Studies have shown that, on the web at least, JavaScipt's most dynamic features (eval, dynamic code loading, and so on) are commonly used, even when they are not strictly necessary [114, 115]. New client-side APIs and browser features are developed rapidly, and these features are not always used correctly by developers [59]. This makes the analysis of web-based JavaScript particularly difficult.

The most similar work to ours is SymJS. SymJS also attempts concolic testing of web JavaScript, and is also based on an instrumented browser [88, 127]. The concolic testing applies many of the techniques described earlier to reduce its search space.

SymJS is based on the open source Rhino JavaScript engine. As mentioned below, only a small fraction of real-world websites' JavaScript can be correctly parsed and interpreted by Rhino. This is not a critical problem for SymJS, which is intended for developer-supported testing and so can be run in a more predictable and stable environment than a tool intended for testing live real-world websites.

TAJS is a static analysis tool for JavaScript, but includes a sophisticated model of the DOM API, which allows it to reason about JavaScript embedded in a web page, not only standalone JavaScript [71, 72]. TAJS has special support for detecting and removing certain cases of eval, which is normally difficult to reason about [70]. Recent updates have even allowed the tool to analyse JavaScript code which uses jQuery and similar libraries [7], which has previously been too complex for static analysis tools. Like SymJS, the main goal is developer supported analysis, so the tool is not intended—or suitable—for analysis of live, real-world web pages.

Jalangi is a framework for instrumentation and runtime monitoring of JavaScript code [121]. Certain instrumentation rules are defined, which are then applied to the code being tested at the source code level—each action in the original JavaScript code is mirrored by an analysis action in the instrumented version of the code which updates the analysis state. Jalangi's big advantage then is that this instrumented web page is loaded and tested in a real, unmodified web-browser. This allows testing with an up-to-date production web browser, as well as across multiple browsers, including mobile browsers. Separating the instrumentation from the execution makes Jalangi a very flexible analysis platform.

The framework has been used to implement several interesting JavaScript analyses for web applications [53, 54, 123]. Most relevant to my own work, it is possible to create a symbolic trace recorder, and therefore a concolic testing tool, by tracking marked input values as they pass through the program.

Jalangi's main limitation in the web setting is that it requires pre-processing of the JavaScript source. It is thus useful for testing websites with the support of developers, but is problematic for analysing third-party websites without developer support. Additionally, Jalangi's concolic analysis only supports a relatively small subset of string operations in their constraints, which are translated to Boolean and integer constraints before being sent to the solver (which is CVC3).

A new version of Jalangi with a slightly different focus is available which mitigates the source-code instrumentation issue [66, 69]. It is able to use mitmproxy [101] to intercept JavaScript files being downloaded and instrument them on the fly. This reduces the required developer support significantly. However, there is no known implementation of concolic testing using Jalangi 2.

Kudzu is an automated test-generation tool for JavaScript-based web applications, based on concolic testing [116]. As part of this work, a new constraint solver was developed called Kaluza, specifically tailored for concolic testing of web JavaScript. In our tests, Kaluza was not able to solve many of the constraints generated by our system, and did not support the theories necessary for a natural translation of constraints from the symbolic execution to the solver's language. Although designed to generate tests for web applications, Kudzu does not appear to include any modelling of the DOM, browser APIs, or user inputs. It seems that the tool analyses JavaScript in the setting of a web page, without analysing the interactions between the two.

ProFoUnd (Program-analysis-based Form Understanding) is a demonstration system that attempts to use static analysis of JavaScript to aid information extraction [20]. Because full static analysis of JavaScript is so difficult, ProFoUnd does not perform a sound analysis of the code. Instead it uses a pattern-matching approach. It identifies entry-points (JavaScript code attached to submit buttons and form handlers) by looking for a fixed set of common attachment patterns in a web page. It then looks statically at the page's JavaScript code and detects interceptions (alert boxes and error messages) by pattern matching. Given an interception, ProFoUnd attempts a static analysis to chain together conditions on the form elements that lead to the interception point. ProFoUnd relies on the Rhino JavaScript interpreter for parsing, and for the websites that Rhino can parse, ProFoUnd can detect many relevant constraints. However, Rhino is not a JavaScript interpreter for the web, and it can only process a minority of real-world sites.

Other tools which tackle web-based JavaScript analysis include FLAX, which uses taint-tracking to detect client-side security vulnerabilities in web applications [117]; JS-Slicer, a program slicing tool which can account for DOM dependencies (and which is built on Jalangi) [142]; Gulfstream, an incremental static analysis which can be applied

to “streaming” JavaScript programs as they are downloaded [92]; and Rozzle, which uses multi-execution to detect JavaScript malware which depends on certain browser features or environments [79]. There has also been work on formalising the DOM API to allow better type-checking of JavaScript programs which interact with the DOM [129].

As well as Jalangi, both SymJS and Kudzu would have been interesting to compare with our platform. Unfortunately neither tool is available and so we were not able to test them. While it is available and clearly related, TAJS uses static analysis, a very different approach, and is not designed to run on real-world websites. It is therefore difficult to directly compare with our system.

The evaluation data for the platform, including the synthetic program generator and the generated programs, is available on GitHub [42].

4 ArtForm: Exploring web forms

The previous chapter presented a platform for analysing JavaScript on the web. Analysis of web forms requires additional infrastructure on top of this. This chapter introduces ArtForm, a tool for exploring web forms which implements this additional infrastructure. It builds on the concolic testing platform from the previous chapter, but adds extra modelling to account for the connection between the user-level and implementation-level views of the page discussed in Chapter 1.

ArtForm's primary goal is, given the URL of a web page containing a form, to analyse the client-side validation code attached to the form and infer its validation constraints. This allows new valid form inputs to be generated, which can be used to explore further into the site or to extract data hidden behind the form.

Analysis of web form validation code involves a new set of challenges not encountered by standalone JavaScript analysis. These include:

- *Initial page analysis.* Before the main analysis can begin, the forms on the page, and in particular their submission buttons, must be identified. The set of user-level actions used to fill the form must be chosen.

- *Choosing an action sequence.* After identifying the actions to be tested, their ordering must be chosen. As actions can depend on each other, and on changes to the DOM as the form filling progresses, choosing an order is non-trivial. In fact, in many cases, multiple orderings must be tested to cover all the interesting behaviour of a single web form.

- *Simulating user actions.* To record a realistic symbolic trace which accurately represents the website's interface, it is important to simulate filling the form in a realistic way, as a real user would.

- *Modelling sources of symbolic information.* The symbolic interpreter propagates existing symbolic values, but does not *create* them. When analysing web forms, the input values originate in the DOM—from the form input elements themselves. ArtForm models the form inputs as sources of symbolic information and must choose how their different values and properties should be represented symbolically at each stage of the analysis.

- *Restricting the allowable input values.* ArtForm introduces extra SMT constraints which ensure the values generated by the concolic testing are suitable for the

particular form being analysed. For example, the inputs generated for a drop-down select box should only be chosen from among the possible values of that select box.

- *Trace classification.* After each trace is recorded, it is classified as either a successful form submission or a rejection before being merged into the path tree. The trace classifier analyses the annotations on each trace (the markers for page loads, alerts, and so on) to determine whether a trace represents a successful submission or not.

We address these problems by introducing new constraints into the standard concolic testing procedure. Constraints are added to restrict the values which an be chosen for different fields, and to determine which action orderings need to be tested to reach new paths. This new extended concolic testing algorithm, together with some other components tailored for form analysis (such as a form-specific trace classifier) forms the core of ArtForm.

4.1 Formalising the setting

Traditional concolic testing tests a given function or program, with given inputs and fixed behaviour. When analysing web forms, or any complex event-driven interface, it is not obvious how to choose the analogue of this function under test. If all the form related code is contained in the submit handler, then there is a clear mapping from a form to a single function. But in general the form's implementation is spread over the handlers for a variety of events; filling fields, clicking buttons, and so on. If a known, fixed sequence of such actions is to be tested, then a "top-level" function can be defined as the concatenation of those actions, provided with the appropriate inputs. However, for our purposes this model is still not powerful enough. It is desirable to have the action ordering be variable and under the control of the analysis.

This section presents a more realistic model for web forms than treating them as a single fixed function, which we use to develop our approach to web form analysis in the next section. We will use concolic testing to discover both the values to fill into form fields, and *also* the ordering of these actions that allows these values to submit the form. Although extremely idealised, this model is still useful for developing an effective analysis of real forms.

A form is a collection of individual input fields. Each field f is associated with an action Act_f. Actions in our model correspond to JavaScript event handlers. Each action Act_f takes an input value, writes it into a global variable v_f and executes some further code. Informally this input value is the input supplied by a user to fill the fields, v_f represents the field's value in the DOM, and the code is that of the event handler. A submit button can be modelled as a field whose input value is never read.

The code for the form actions manipulates a set of variables $\vec{v}$, including the distinguished variable v_f—the *input variable* of the form action. The variable v_f is associated with a *default value* D_f, which is the value it will take before the action Act_f is run. In our basic model, the program code for each action is written in a simple procedural language using the following grammar for expressions E:

$$x := \tau(\vec{y}) \qquad \mathsf{if}\ \varphi(\vec{y})\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \qquad E_1; E_2 \qquad \mathsf{Abort} \qquad \mathsf{do}_C$$

Here τ ranges over terms built up from set of atomic functions (e.g., $+, \times$) over variables and constants, while φ ranges over some set of conditions (e.g., Boolean combinations of atomic conditions $\tau_1\ \{\leqslant, \neq, =\}\ \tau_2$, where τ_i are terms). C in do_C ranges over some set of commands which do not impact form submission or control flow. Variables x on the left hand side of assignments can be from $\vec{v}$ (informally, the global variables), or freshly-named variables local to this action. Variables in $\vec{y}$ may be from $\vec{v}$ or any previously assigned local variable. A condition φ is *ground* if it contains no free variables. A variable is *assigned* in expression E if it occurs on the left side of some assignment statement and otherwise is *free*.

Example. The validate_to function from the previous chapter's running example (see page 40) can be expressed in this language as follows:

$$\begin{aligned}
&f := v_{\mathsf{from}};\\
&t := v_{\mathsf{to}};\\
&\mathsf{if}\ \mathsf{len}(f) = 0\ \mathsf{then}\ \mathsf{Abort}\ \mathsf{else}\ \mathsf{do}_{\mathsf{skip}};\\
&\mathsf{if}\ f = t\ \mathsf{then}\ \mathsf{Abort}\ \mathsf{else}\ \mathsf{do}_{\mathsf{skip}}
\end{aligned}$$

Here the function len is one of the atomic functions allowed in τ, and the action $\mathsf{do}_{\mathsf{skip}}$ is a no-op. Local variables f and t are assigned, whereas v_{from} and v_{to} are free. □

Although there are no loops in this simple language, this is not as much of a limitation as it seems. In particular, it reflects our platform's bytecode-based interpreter, which sees a stream of low-level instructions without high-level structure; the interpreter sees loops in their unrolled form. Infinite or input-dependent loops (where the number of iterations depends on the input) cannot be represented in this language, although they *can* be handled in our implementation, which does not need to know the program code for the actions statically and only explores it on-the-fly.

The semantics of the language are standard. Given an expression E and a binding σ for the free variables of E, the semantic function Eval returns the sequence of ground conditions and atomic actions Abort and do_{Ac} that are generated during an execution. For simplicity, we give the semantics in the absence of assignments, which can be inlined. Inlining is safe because nothing in our language except assignment itself can

cause side-effects. Then Eval is defined as follows, with $\oplus$ representing concatenation of sequences:

$$\begin{aligned}
\mathsf{Eval}(\sigma, \mathsf{Abort}) &= [\mathsf{Abort}] \\
\mathsf{Eval}(\sigma, \mathrm{do}_{Ac}) &= [\mathrm{do}_{Ac}] \\
\mathsf{Eval}(\sigma, \mathsf{if}\ \varphi(\vec{y})\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2) &= \begin{cases} [\varphi(\sigma(\vec{y}))] \oplus \mathsf{Eval}(\sigma, E_1) & \text{if } \varphi(\sigma(\vec{y})) \text{ holds} \\ [\varphi(\sigma(\vec{y}))] \oplus \mathsf{Eval}(\sigma, E_2) & \text{otherwise} \end{cases} \\
\mathsf{Eval}(\sigma, E_1; E_2) &= \mathsf{Eval}(\sigma, E_1) \oplus \mathsf{Eval}(\sigma, E_2)
\end{aligned}$$

For example, the result of Eval on the example validation program above, with the binding $\sigma = \{v_{\mathsf{from}} \mapsto \text{“LHR”}, v_{\mathsf{to}} \mapsto \text{“JFK”}\}$ would be:

$$[\mathsf{len}(\text{“LHR”}) = 0, \mathrm{do}_{\mathsf{skip}}, \text{“LHR”} = \text{“JFK”}, \mathrm{do}_{\mathsf{skip}}]$$

We will be particularly interested in whether conditions are satisfied and whether or not Abort is encountered. Given a condition φ and a binding σ for the variables in φ, the *value* of φ, denoted $\mathsf{Val}(\varphi, \sigma)$, is true if φ holds when the variables appearing in φ are replaced with their valuation in σ. Continuing the same example as before, $\mathsf{Val}(\mathsf{len}(v_{\mathsf{from}}) = 0, \{v_{\mathsf{from}} \mapsto \text{“LHR”}, v_{\mathsf{to}} \mapsto \text{“JFK”}\})$ is false. The *trace* of E on an assignment σ is the sequence of conditions encountered during the execution and their truth values under σ. If E executes Abort at any point when running on σ we say it *aborts on* σ, while otherwise we say it *terminates on* σ. The running example program terminates on the example binding σ with the following trace:

$$[(\mathsf{len}(v_{\mathsf{from}}) = 0, \mathsf{false}), (v_{\mathsf{from}} = v_{\mathsf{to}}, \mathsf{false})]$$

We will name our actions with numeric indices, so Act_i denotes the program associated with form filling action i, while v_i and D_i denote the input variable for Act_i and its default value, respectively. A set of such indexed actions $\mathrm{Act}_1 \ldots \mathrm{Act}_n$ has *restricted global state* if for every Act_i, each free variable v occurring in it is one of the input variables v_j and further no input variable is ever assigned. That is, Act_i is permitted to read any of the input variables $v_1 \ldots v_n$, but may not write to any global state, including those inputs. The only exception is the implicit assignment of v_i just before the program code attached to Act_i is executed.

The behaviour of a single expression is well-defined given a binding for all variables. When several actions are executed in sequence, it remains to explain how they interact. Intuitively, as actions are executed, the corresponding input values v_i are bound in turn, with the user input values replacing their previous default values.

A *bound form action* is a pairing of a form action Act_i with a value c_i for its input variable v_i, while a *form input* is a sequence of bound form actions. We define the *unfolding* of a form input $(\mathrm{Act}_1, c_1) \ldots (\mathrm{Act}_n, c_n)$ as the sequence: $(\sigma^1, E_1), \ldots, (\sigma^n, E_n)$,

where σ^i is the *order-modified assignment* which maps each v_j for $j \in \{1..n\}$ to the corresponding input value c_j if Act_j comes earlier than Act_i in the ordering, or its default value otherwise.

$$\sigma^i(v_j) = \begin{cases} c_j & \text{if } j \leqslant i \\ \mathsf{D}_j & \text{otherwise} \end{cases}$$

We can extend our semantics to form inputs via unfoldings. The *trace* of a form input is the concatenation of the traces in its unfolding. By extension, the result of Eval on a form input is the concatenation of the results of Eval on each term of its unfolding. A form input is said to abort if *any* of its action expressions E_i aborts on its corresponding σ_i; otherwise we say it terminates.

Intuitively, this model of web forms and their validation code allows each form field's validation code to process its own input value, possibly in conjunction with others which have already been filled. When the actions have restricted global state then the each field's associated code cannot affect the execution of other fields' code; the different field handlers only "communicate" via observing the values of the other form fields (which they are allowed to read).

The order modified assignments σ^i represent the state of the form at the time each action Act_i is invoked. For example, if our example form is filled in the order From, To, Date, filling the values "LHR", "JFK" and "17/09" respectively, then we will use the following assignments, which begin with the default values "", "", and "01/01" and include each input value as it is filled:

$$\begin{aligned} \sigma_{\mathsf{From}} &= \{v_{\mathsf{from}} \mapsto \text{“LHR”}, v_{\mathsf{to}} \mapsto \text{“”}, v_{\mathsf{date}} \mapsto \text{“01/01”}\} \\ \sigma_{\mathsf{To}} &= \{v_{\mathsf{from}} \mapsto \text{“LHR”}, v_{\mathsf{to}} \mapsto \text{“JFK”}, v_{\mathsf{date}} \mapsto \text{“01/01”}\} \\ \sigma_{\mathsf{Date}} &= \{v_{\mathsf{from}} \mapsto \text{“LHR”}, v_{\mathsf{to}} \mapsto \text{“JFK”}, v_{\mathsf{date}} \mapsto \text{“17/09”}\} \end{aligned}$$

4.2 Dynamic event reordering for concolic testing

This section presents a new algorithm which adapts concolic testing to our model of form validation code. The algorithm is shown in Algorithm 2. Its structure is similar to the classic concolic testing algorithm, with the key difference being that we are exploring a tree of paths for each action, instead of a single tree for the whole analysis. At each iteration we first choose an action with an unexplored path as the target, and solve for both a new value for each form action and an ordering for the actions. Testing the actions in the given order and with the given values should exercise the target path in the target action. The remainder of this section provides the intuition and formal details for the algorithm.

Reordering of events is required because the action code for a form field can depend on which actions have already been executed. For example, consider the functions validate_to and validate_aux from the running example (Listing 1; page 40). If

Algorithm 2 The high-level algorithm for form exploration.

```
1:  procedure FORM-CONCOLIC-TESTING(form)
2:      actions ← IDENTIFY-ACTIONS(form)
3:      initial-order, initial-values ← CHOOSE-INITIAL-ORDER-AND-VALUES(actions)
4:      trace ← EXECUTE-AND-RECORD(initial-order, initial-values)
5:      path-trees ← INIT-TREE(trace)
6:      if trace is terminating then
7:          Mark all local symbolic paths within trace as Known-Extendible
8:      else
9:          Mark local symbolic paths which aborted as Known-Unextendible
10:     end if
11:     while ∃Act Unresolved_Act(path-trees) ≠ ∅ do
12:         Choose target-path and Act where target-path ∈ Unresolved_Act(path-trees)
13:         order, values ← SOLVE-PATH-CONSTRAINT(OverApprox_Act(target-path))
14:         if the path constraint was successfully solved then
15:             trace ← EXECUTE-AND-RECORD(order, values)
16:             EXTEND-TREE(path-trees, trace)
17:             if trace is terminating then
18:                 Mark all local symbolic paths within trace as Known-Extendible
19:             else
20:                 Mark local symbolic paths which aborted as Known-Unextendible
21:             end if
22:         else
23:             Mark target as Known-Unextendible
24:         end if
25:     end while
26: end procedure
```

a fixed sequence of actions is tested where To is filled before From, then the check for from.length == 0 in validate_aux will *always* be satisfied. When validate_aux runs, the From field has not been filled yet, so it must always have its default value of the empty string. Under this ordering, the alert message on line 9 is always shown, and the remaining code can never be reached.

In our example, it is possible to pick an ordering which allows all the code to be tested (namely, filling From, To and then Date). However, this is not always possible in general. If we extended our example to include both departure date and return date fields, then the handlers for these fields would naturally depend on each other; both would check whether the other were set, and if so enforce that the departure date was before the return date. In this case, no single static ordering could be chosen which allows all the form validation code to be tested, and multiple orderings of the actions would be required in order to cover all the form's behaviour. Even if there is a suitable static ordering, a prior analysis would need to be performed to find it, and even given an ordering where the form can be submitted, some interesting code paths may still not be reachable.

The previous section defined concrete traces of action-related code. We now extend this notion to symbolic descriptions of these traces. We start by symbolically describing the "local behaviour" of a single form action, given a binding for all variables.

For a binding σ to the free variables of expression E, the *local symbolic path* of E executed under σ is a formula describing the values of conditions (with assignments inlined) in the trace of E on σ. That is, the local symbolic path is the conjunction

$$\bigwedge_{\mathsf{Val}(\varphi_i,\sigma)=\top} \varphi_i \wedge \bigwedge_{\mathsf{Val}(\varphi_i,\sigma)=\bot} \neg\varphi_i$$

where $\varphi_1 \ldots \varphi_k$ is the sequence of conditions encountered while executing E on σ.

In the previous section we saw the example concrete trace

$$[(\mathsf{len}(\text{“LHR”}) = 0, \mathsf{false}), (\text{“LHR”} = \text{“JFK”}, \mathsf{false})]$$

for the action corresponding to the form's From field with a binding σ. The corresponding local symbolic path for this trace is

$$\neg(\mathsf{len}(v_{\mathsf{from}}) = 0) \wedge \neg(v_{\mathsf{from}} = v_{\mathsf{to}})$$

This is the condition which the input variables must satisfy in order to cover the same execution path (and therefore generate the same concrete trace) as the original input binding σ did. In terms of concolic testing, this is the path constraint for that path, within this single action. This example trace only includes a single action, but in general a trace is for a whole form input, covering multiple actions in sequence.

We can now extend these definitions from symbolic paths emerging from concrete traces to formulae which represent new *possible* concrete paths which we would like to

explore. Given a trace t for an action, consisting of conditions $(\varphi_1, \tau_1) \ldots (\varphi_k, \tau_k)$, where τ_i is the truth value for condition φ_i, a *symbolic modification* is a local symbolic path corresponding to the sequence of condition/value pairs:

$$(\varphi_1, \tau_1) \ldots (\varphi_p, \tau_p), (\varphi_{p+1}, \neg\tau_{p+1})$$

where $p < k$ and where the negation of a truth value is defined in the usual way (negation of $\top$ is $\bot$ and vice versa). That is, we take a prefix of t, and then negate the last element in it. Given a set of traces T, a symbolic modification is *unexplored* if there is no trace already in T which subsumes it. Given set of traces T and action A, we define $\mathsf{Unexplored}_A(T)$ to be the set of symbolic modifications of traces in action A which are unexplored.

For example, if T consists of the single trace shown above, then $\mathsf{Unexplored}_A(T)$ consists of two symbolic modifications:

$$\{\mathsf{len}(v_{\mathsf{from}}) = 0, \neg(\mathsf{len}(v_{\mathsf{from}}) = 0) \wedge (v_{\mathsf{from}} = v_{\mathsf{to}})\}$$

We can lift the classification of traces as aborting or terminating to the symbolic level. Given a set of traces T and a form action A, let $T(A)$ be the restriction of the traces to A (that is, the set of the sub-sequences of each trace in T which corresponds to action A). Let $\mathsf{Abort}_A(T)$ be the traces in $T(A)$ that are aborting, and similarly let $\mathsf{Terminate}_A(T)$ be the set of traces that are terminating. Because a trace terminates if and only if it does not abort, $\mathsf{Abort}_A(T)$ and $\mathsf{Terminate}_A(T)$ form a partition of $T(A)$. Further, we denote the set of local symbolic paths of traces in T that are aborting in A by $\mathsf{Symbolic}(\mathsf{Abort}_A(T))$, and analogously the local symbolic paths which are terminating in A by $\mathsf{Symbolic}(\mathsf{Terminate}_A(T))$.

So far, we have used the conditions encountered to symbolically describe the behaviour of a single form action A. We can now extend this description to a sequence of form actions. In doing this we must account for the action ordering, which determines whether each action "sees" the default value or the user-supplied value for the other actions' input values. Since we are interested in finding paths which can be explored by a terminating trace (over all the actions), we have to track symbolically whether the other actions abort.

Given a local symbolic path $\psi(\vec{v})$ of a trace for a single form action Act_i, the *ordered version of* ψ, denoted $\mathsf{Ord}_i(\psi)$, is the same formula where each input variable v_j is replaced by v'_j. We also conjoin an extra constraint which sets v'_j to either v_j or D_j depending on a new ordering relation $\preceq$:

$$(j \preceq i \rightarrow v'_j = v_j) \wedge (j \succ i \rightarrow v'_j = \mathsf{D}_j).$$

Then $\mathsf{Ord}_i(\psi)$ and the relation $\preceq$ symbolically represent an ordering of the actions and a binding σ such that ψ holds on the order-modified assignment σ^i as defined earlier.

Proposition 1 *For any linear order $\preceq$ on actions $Act_1 \ldots Act_n$ and any set of input values for each action $c_1 \ldots c_n$, we have: $Ord_i(\psi)$ holds on the ordered action indices $j_1 \ldots j_n$ and variable values $c_1 \ldots c_n$ if and only if when evaluating the form input $(Act_{j_1}, c_{j_1}) \ldots (Act_{j_n}, c_{j_n})$, the trace of action Act_i satisfies ψ.*

Given a local symbolic path sympth for action Act_i, and a set of concrete traces T, we define $\mathsf{UnderApprox}^i(\mathsf{sympth})$ to be the formula

$$\mathsf{Ord}_i(\mathsf{sympth}) \wedge \mathsf{OrderAx} \wedge \bigwedge_{\substack{j \in \{1..n\}, \\ j \neq i}} \mathsf{MustTerminate}_j$$

where $\mathsf{MustTerminate}_j$ is

$$\bigvee_{\mathsf{sympth}'_j \in \mathsf{Symbolic}(\mathsf{Terminate}_{\mathsf{Act}_j}(T))} \mathsf{Ord}_j(\mathsf{sympth}'_j)$$

and OrderAx is a formula asserting that the relation $\preceq$ used within $\mathsf{Ord}_i(\mathsf{sympth})$ is a linear order on the indices of actions $1 \ldots n$. This formula represents orderings and values which will follow the path sympth in action Act_i while also forcing every other action to follow a known-terminating trace of T. Informally, $\mathsf{UnderApprox}^i(\mathsf{sympth})$ describes the form inputs and orderings that we are *sure* will explore sympth in Act_i without aborting in any action, based on what we already know about aborts in T.

We similarly define $\mathsf{OverApprox}^i(\mathsf{sympth})$ to be the formula

$$\mathsf{Ord}_i(\mathsf{sympth}) \wedge \mathsf{OrderAx} \wedge \bigwedge_{\substack{j \in \{1..n\}, \\ j \neq i}} \mathsf{MayTerminate}_j$$

where $\mathsf{MayTerminate}_j$ is

$$\bigwedge_{\mathsf{sympth}'_j \in \mathsf{Symbolic}(\mathsf{Abort}_{\mathsf{Act}_j}(T))} \neg \mathsf{Ord}_j(\mathsf{sympth}'_j)$$

That is, $\mathsf{OverApprox}^i(\mathsf{sympth})$ symbolically represents the orderings and values that will follow the path sympth in action Act_i and will not drive any other action to a known-aborting trace of T. Informally, this describes form inputs and orderings that will explore sympth and *may* not abort in any other action, based on what we currently know about aborts in T. That is, the selected paths are not known to abort in any other action.

The following lemmas give the critical properties of these symbolic descriptions. Intuitively, they simply state that $\mathsf{UnderApprox}^i(\mathsf{sympth})$ is an under-approximation of the terminating form inputs which satisfy sympth, and $\mathsf{OverApprox}^i(\mathsf{sympth})$ is an over-approximation of them.

Lemma 1 *Given a set of traces T and a path sympth from Act_i, let $UnderApprox^j(sympth)$ be formed based on T. If $UnderApprox^j(sympth)$ is satisfiable by input values $\vec{c}$, and ordering $\preceq_0$, then the form input $(Act_{j_1}, c_{j_1}) \ldots (Act_{j_n}, c_{j_n})$ results in a trace which satisfies sympth and terminates in every action other than Act_i.*

Proof: A satisfying assignment of $\mathsf{UnderApprox}^l(\mathsf{sympth})$ consists of values $c_1 \ldots c_n$ and an ordering $\preceq_0$. Since the formula $\mathsf{UnderApprox}^l(\mathsf{sympth})$ includes the order axioms, we know $\preceq_0$ is a linear order on the indices $1 \ldots n$, so we can rewrite the indices under this order as $j_1 \ldots j_n$.

Choose l' such that $j_{l'} = l$. Then action $\mathsf{Act}_{j_{l'}}$ is the one which must satisfy sympth. For any i, let σ^{j_i} be the order modified assignment that maps variable v_{j_k} to c_{j_k} when $k \leqslant i$, and otherwise maps it to its default value D_{j_k}.

We need to conclude that the form input $(\mathsf{Act}_{j_1}, c_{j_1}) \ldots (\mathsf{Act}_{j_n}, c_{j_n})$ extends sympth and does not abort outside $\mathsf{Act}_{j_{l'}}$. That is, we need to show that:

- When $\mathsf{Act}_{j_{l'}}$ is executed with binding $\sigma^{j_{l'}}$, the code path satisfies sympth, and
- When Act_{j_i} for $i \neq l'$ is executed with binding σ^{j_i} the trace does not abort.

Because $\mathsf{UnderApprox}^l(\mathsf{sympth})$ is satisfied, we have that $c_1 \ldots c_n$ and $\preceq_0$ satisfy $\mathsf{Ord}_{j_{l'}}(\mathsf{sympth})$, by the definition of $\mathsf{UnderApprox}^l$ and choice of l'. Then Proposition 1 shows that when $\mathsf{Act}_{j_{l'}}$ is executed with binding $\sigma^{j_{l'}}$, the code path satisfies sympth.

For j_i with $i \neq l'$ we know that $\preceq_0$ and $c_1 \ldots c_n$ satisfy $\mathsf{Ord}_{j_i}(\mathsf{sympth}')$ for some sympth' in $\mathsf{Symbolic}(\mathsf{Terminate}_{\mathsf{Act}_{j_i}}(T))$. This is by the definitions of $\mathsf{UnderApprox}^l$ and $\mathsf{MustTerminate}_l$. This means that σ^{j_i} satisfies sympth', and thus the trace of Act_{j_i} during the form input execution satisfies sympth', and is therefore terminating. □

Lemma 2 *Given a set of traces T, if sympth is a symbolic modification for a form action Act_l, and the form input $(Act_{j_1}, c_{j_1}) \ldots (Act_{j_n}, c_{j_n})$ generates a trace which extends sympth and terminates in every action other than Act_l, then $OverApprox^l(sympth)$ is satisfied.*

Proof: Let sympth be a symbolic path for action l and t a trace extending sympth, which does not generate an abort outside of action Act_l. Trace t is generated by a reordering of the actions $j_1 \ldots j_n$ and input values $c_{j_1} \ldots c_{j_n}$. For any i, let σ^{j_i} be the order modified assignment that maps variable v_{j_k} to c_{j_k} when $k \leqslant i$, and otherwise maps it to its default value D_{j_k}. Choose l' such that $j_{l'} = l$. From our assumptions about t we know that $\mathsf{Act}_{j_{l'}}$ executed on $\sigma^{j_{l'}}$ satisfies sympth, and $action_{j_i}$ for each $i \neq l'$ does not abort on σ^{j_i}.

Let $\preceq_0$ be the ordering relation corresponding to the reordered indices $j_1 \ldots j_n$. We need to show that $\preceq_0$ and $c_{j_1} \ldots c_{j_n}$ satisfy each conjunct of $\mathsf{OverApprox}^i(\mathsf{sympth})$.

- The order axioms satisfied by the premise that $j_1 \ldots j_n$ is a reordering of $1 \ldots n$.

- The fact that $\mathsf{Act}_{j_{l'}}$ executed on $\sigma^{j_{l'}}$ satisfies sympth implies that $\mathsf{Ord}_l(\mathsf{sympth})$ holds for ordering $\preceq_0$ and values $c_{j_1} \ldots c_{j_n}$.

- For $i \neq l'$, the fact that Act_{j_i} does not abort on σ^{j_i} implies that the symbolic trace corresponding to the execution of $action_{j_i}$ on σ^{j_i} is not in the list of known-aborting symbolic traces $\mathsf{Symbolic}(\mathsf{Abort}_{\mathsf{Act}_{j_i}}(T))$. Thus for each $i \neq l'$, we know that $\mathsf{MayTerminate}_{j_i}$ holds.

With each conjunct satisfied, $\mathsf{OverApprox}^i(\mathsf{sympth})$ itself is satisfied. □

Lemmas 1 and 2 form the basis of Algorithm 2. We maintain a set of traces, and from these we can form the set of symbolic paths and their symbolic modifications, which form a tree. We also classify the symbolic paths and their modifications. We distinguish the *known-extendible* paths—those that are known to have an extension that is terminating—as well as the *known-unextendible* ones, where it is known that there is no such extension. The paths which are neither known-extendible or known-unextendible are said to be *unresolved*. The set $\mathsf{Unresolved}_{\mathsf{Act}_i}(T)$ contains both the local symbolic paths from $T(\mathsf{Act}_i)$ which have not been marked as known-extendible or known-unextendible, as well as the symbolic modifications of local symbolic paths in $T(\mathsf{Act}_i)$ which have not yet been explored or considered by the algorithm.

At any step of the algorithm we choose an unresolved path sympth and check for satisfiability of $\mathsf{OverApprox}^i(\mathsf{sympth})$. If the formula is not satisfiable, we mark sympth as known-unextendible (that is, no terminating trace extends sympth). Otherwise we take a satisfying assignment consisting of values $c_1 \ldots c_n$ and ordering $j_1 \ldots j_n$, and use it in a new execution, giving trace t. We add t to our set of traces and iterate.

If t terminates, then it acts as a witness that each restriction of t to action Act_i can be extended by a terminating trace. Thus, we mark all the local symbolic paths in t as known-extendible.

If t aborts, then it either aborted in action Act_i or by Lemma 1 we know that $c_1 \ldots c_n$ and $j_1 \ldots j_n$ must not have satisfied $\mathsf{UnderApprox}^i(\mathsf{sympth})$. In the latter case, there must be some other action Act_j ($j \neq i$) for which t does not follow an explored branch, and where t aborts. That is, in Act_j, t follows a previously unexplored path and discovers an abort. Note that these two cases are not mutually exclusive; t may have aborted in multiple actions. In either case, we have at least one aborting action, and thus at least one local symbolic modification is resolved (even if it had not been a known modification until now).

It is possible for a newly recorded trace to give rise to *new* symbolic modifications, so the total number of unresolved traces does not necessarily decrease at each iteration. However, each action's program code has a finite number of branches, so there are a finite number of symbolic paths available to explore. Because at least one path is resolved in every iteration, $\bigcup_{i \in \{1..n\}} \mathsf{Unresolved}_{\mathsf{Act}_i}(T)$ must eventually become empty, and this guarantees termination.

Proposition 2 *Algorithm 2 is complete. Assuming completeness of the solver, on completion the path tree will have the property that for every local symbolic path sympth which has any extension which terminates overall, then at least one such extension is explored by the algorithm. In other words, every local symbolic path which is reachable on a terminating trace is explored.*

Proof: Suppose we have a local symbolic path sympth in action Act_i which has a terminating extension t, but no terminating extension has yet been explored. Consider sympth′, a symbolic path which is the maximal prefix of sympth that has been explored by the algorithm (which may be the whole of sympth). This path cannot be marked as known-unextendible by the algorithm, by the assumption that sympth is extendible (by t), so $\mathsf{OverApprox}_{\mathrm{Act}_i}(\mathsf{sympth}')$ must be satisfiable (by Lemma 2) and sympth itself cannot abort. If it is marked as known-extendible, then the algorithm has already explored some terminating extension, violating our assumption. So consider the final case that sympth′ is unresolved. Because the set of unresolved modifications is finite, the algorithm will eventually choose sympth′ as the symbolic modification to test at line 12. We know that $\mathsf{OverApprox}_{\mathrm{Act}_i}(\mathsf{sympth}')$ is satisfiable, so the algorithm generates new inputs and records a new trace t'. If t' terminates and follows sympth, then we have explored a terminating extension of sympth as required. If t' terminates but does not follow sympth, or if t' aborts, then after it is added to the set of explored traces T, there must *still* be an unresolved symbolic modification yet to be explored (again, by assuming the existence of t). In either case, the algorithm continues repeating this process, each time either finding a terminating extension of sympth or reducing the number of unresolved modifications available. As this set is finite, the algorithm must eventually resolve sympth by discovering a terminating extension. Note that the algorithm can never run out of unresolved paths to test (and thus terminate) *before* sympth is resolved, as sympth will always have some unresolved prefix which is known to the algorithm. □

The algorithm is complete, depending on some strong assumptions. This result requires completeness of the solver, and the ability of the analysis to track all conditions symbolically and solve the constraints with full precision. Further, it requires that the code conforms to the simple structure in which form actions set their input variables but otherwise do not update any global state shared by other actions. Realistic action code does not obey these assumptions, but we can still apply Algorithm 2 on arbitrary real-world code, tracking only the input fields symbolically across actions, while dropping the completeness guarantees. Although no longer guaranteed to be complete—it is possible that some paths will not be explored—this algorithm is still effective and useful on real-world web forms.

Example. We will illustrate Algorithm 2 on the running example airline form (introduced on page 40 and expressed in our formalism in the previous section). Assume that

we have identified the form actions as $\mathsf{Act}_{\mathsf{From}}$, $\mathsf{Act}_{\mathsf{To}}$, and $\mathsf{Act}_{\mathsf{Date}}$ for entering values into the departure airport, arrival airport, and departure date fields. Algorithm 2 will choose an arbitrary default initial order and values for these fields: for example the order From $\prec$ To $\prec$ Date with empty strings for the airports and "01/01" for the date. The code is executed on these values, and reaches the first alert message in validate_aux, and the first alert in validate_date, both classified as aborts. The corresponding local symbolic path for $\mathsf{Act}_{\mathsf{To}}$ consists of the single constraint from.length $= 0$, and for $\mathsf{Act}_{\mathsf{Date}}$ it is to.length $= 0$. These constraints are added to the corresponding path trees for each action, and are also marked as known-unextendible due to the aborts. The command on line 12 will then choose a fresh path to target from one of the $\mathsf{Unresolved}_{\mathsf{Act}_i}$ sets. Suppose the algorithm selects $\mathsf{Act}_{\mathsf{Date}}$ to explore next and then chooses the path $\neg$(to.length $= 0$) from $\mathsf{Unresolved}_{\mathsf{Act}_{\mathsf{Date}}}$. Assuming the default value D_{To} is the empty string, the corresponding ordered constraint $\mathsf{Ord}_{\mathsf{Date}}$ simplifies to

$$(\mathsf{Act}_{\mathsf{To}} \prec \mathsf{Act}_{\mathsf{Date}} \wedge \neg(\mathsf{to.length} = 0))$$

To form the full over-approximation constraint, we also include the linear order axioms, and $\mathsf{MayTerminate}_{\mathsf{Date}}$, which in this case (when simplified) is simply the negation of the single aborting trace in $\mathsf{Act}_{\mathsf{To}}$:

$$(\mathsf{Act}_{\mathsf{From}} \prec \mathsf{Act}_{\mathsf{To}} \wedge \neg(\mathsf{from.length} = 0))$$

The call on line 13 will solve this combined constraint for both an order and values. The only valid order is From $\prec$ To $\prec$ Date. Suppose the returned values are To_1 = "A", From_1 = "A", and Date_1 = "01/01". The code is re-tested with these values, and symbolically traced, leading to the second alerts in both validate_aux and validate_date. This second trace is associated with local symbolic paths for each action, for example $\neg$(from.length $= 0$) $\wedge$ from = to for $\mathsf{Act}_{\mathsf{To}}$, which are added to the corresponding trees on line 16, and again marked as known-unextendible. In the second iteration of the **while** loop we would choose another unresolved path at line 12, and this would return, for example, the path $\neg$(from.length $= 0$) $\wedge$ $\neg$(from = to) from $\mathsf{Act}_{\mathsf{To}}$. The ordered version of this path simplifies to

$$(\mathsf{Act}_{\mathsf{From}} \prec \mathsf{Act}_{\mathsf{To}} \wedge \neg(\mathsf{from.length} = 0) \wedge \neg(\mathsf{from} = \mathsf{to}))$$

This time, $\mathsf{MayTerminate}_{\mathsf{To}}$ is required to avoid both of the known-aborting paths in $\mathsf{Act}_{\mathsf{Date}}$. Thus, the full over-approximation constraint, omitting the linear order constraints, simplifies to

$$\begin{aligned} & [\mathsf{Act}_{\mathsf{From}} \prec \mathsf{Act}_{\mathsf{To}} \wedge \neg(\mathsf{from.length} = 0) \wedge \neg(\mathsf{from} = \mathsf{to})] \\ \wedge\ & [\mathsf{Act}_{\mathsf{To}} \prec \mathsf{Act}_{\mathsf{Date}} \wedge \neg(\mathsf{to.length} = 0)] \\ \wedge\ & \left[\mathsf{int}(\mathsf{substr}(\mathsf{date}, 3, 5)) \geqslant m_1 \wedge \begin{pmatrix} \vee & \mathsf{int}(\mathsf{substr}(\mathsf{date}, 3, 5)) \neq m_1 \\ & \mathsf{int}(\mathsf{substr}(\mathsf{date}, 0, 2)) \geqslant d_1 \end{pmatrix}\right] \end{aligned}$$

where m_1 and d_1 are the concrete numbers representing the month and day observed during the previous execution.

A second call to the solver will return the ordering $\mathsf{From} \prec \mathsf{To} \prec \mathsf{Date}$ and values $\mathsf{From}_2, \mathsf{To}_2, \mathsf{Date}_2$, where From_2 and To_2 are distinct and non-empty, and Date_2 is later than the current date. These new values are again tested, this time giving rise to a trace which terminates in every action and successfully submits the form. □

4.3 Modelling form fields

As well as modelling the user-level action sequences, the application of concolic testing to real web forms also requires modelling of form input fields. This modelling includes (a) the JavaScript events and event handlers which relate to filling a field, (b) the types of the fields and their properties, and (c) the possible values which can be entered into each field by a user.

It is important to consider which JavaScript events and input values are realisable by a user at the interface—as opposed to the values it is possible for a specific tool to inject—because our goal is to understand the user-facing interface. As such, the analysis should account for user-level restrictions. An example of a field with such a restriction is a drop-down select box. The select box has a list of possible values which the user can select. An automated analysis tool, or a user with access to the browser's web development tools, can easily inject unexpected or invalid values into this field, but this is not useful for understanding the expected user behaviour. Even when the goal is finding bugs, it is useful to have a model of which execution paths are reachable by normal users.

Simulating form-filling actions

In order to get a realistic picture of what happens when a user fills the form (and thus to expose the real validation rules to our analysis), we must execute each action in the sequence in a realistic way. Otherwise, we will miss constraints which are applied during form filling. This simulation must be done at the same time as the new input values are injected, so that the code always sees a consistent application state.

ArtForm supports two methods of simulating a user filling form fields. In the first, the new input value is injected into the field (typically into the value property), and then a change event is triggered on the field. The causes any associated change handlers to run, allowing us to record the field-related code. This simple scheme works for the vast majority of sites, but sometimes it is not precise enough. In some cases, sites listen for other events than change, and in these cases this method does not trigger the correct code. For example, a site using an autocomplete might listen for keyup events so it can react to partial input. ArtForm's second form-filling simulation method uses a set

of JavaScript events triggered on the field in sequence, with the actual value injection performed at the appropriate point. The events triggered are the same ones a real user would trigger by filling the field manually in a browser. For example, the following sequence is used to fill a text field:

- The field's value property is set to the empty string
- focus
- For each character in the input string:
  - keydown (for the shift key, if this is a capital letter)
  - keydown
  - keypress
  - The field's value property is updated to include the new character
  - input
  - keyup
  - keyup (for the shift key, if this is a capital letter)
- change
- blur

In either mode, the form-filling is simulated even if the value is not changed from its default, a departure from the general goal to simulate a user filling the form as accurately as possible. This is necessary so that the field's associated code is recorded consistently in each iteration. If the executed code relating to a field is different in each iteration, then the concolic engine will not be able to merge the different traces and will not know about branches which are seen in the default traces. In particular, it would be difficult for the analysis to get started without knowing ahead of time which fields use interesting validation rules. Although not entirely realistic, this behaviour is a practical trade-off to get the most information from each trace while remaining broadly faithful to actions a real user would perform.

As well as filling form fields, ArtForm also includes simulations for clicking on elements (typically submit buttons). As before, we support different levels of granularity: (a) triggering the click event on the button, (b) triggering a sequence of events corresponding to those triggered in a normal browser, and (c) performing a GUI-level click action. Although as GUI click is the most faithful—it comes from outside the browser infrastructure, so WebKit sees it exactly as if from a real user—it can be tricky to implement correctly. For example, the element must be visible on screen, which may require scrolling to bring the element into view before a GUI click can be issued. If the element is partially obscured, it is possible that the click will hit the obscuring element instead, and therefore not trigger the expected behaviour. As such, some sites work better with JavaScript-level event simulation and some with GUI-level clicks.

Creating symbolic values from form fields

The general concolic testing platform discussed in the previous chapter uses special-purpose input functions to receive symbolic inputs. This is a common pattern in traditional concolic testing; for example Jalangi does the same. However, when applying concolic testing to real web forms we cannot control how the code being tested reads its inputs, so the inputs must be made symbolic at the DOM level. ArtForm instruments the value property (as well as a few others, discussed below) of input fields. When JavaScript code accesses these properties, the value returned is tagged as a symbolic input and can be traced by the symbolic interpreter as normal.

WebKit implements element properties via internal C++ getter methods. For example, to retrieve the value property of an input element, the interpreter calls the WebKit-internal method JSHTMLInputElement::jsHTMLInputElementValue() which returns an interpreter JSValue object representing the value of this field. There are similar methods for other properties. In ArtForm, these methods are instrumented so that the values they return are tagged as symbolic inputs. That is, whenever the JavaScript code reads the value from a form field, a new symbolic value is created and returned. The symbolic source of the value contains the identifier of the original field, and a variable name derived from it. This provides the "origin" of all symbolic information in the system.

The WebKit implementation of the DOM API which we instrument is auto-generated from a high-level description of the DOM objects and their properties. Thus we have modified both these high-level descriptions and the auto-generation code to generate a symbolically-instrumented DOM API when WebKit is compiled. The DOM interface is described as a series of IDL (Interface Description Language [111]) files, describing the different types of DOM element and their properties. We have added a new Symbolic annotation to object properties. When the code generator parses the IDL files and generates the C++ code of the DOM implementation for WebKit, our modifications check for the Symbolic annotation and generate the code required to create symbolic values in the DOM object property getter functions.

ArtForm supports different symbolic instrumentation policies for the different input field types as follows:

- *Text inputs.* For standard text inputs, the value property returns a symbolic value, as described above.
- *Checkboxes and radio buttons.* Here, the value property is not instrumented, and the checked property (which is a Boolean) is instrumented instead.
- *Select boxes.* As with normal text inputs, the value property gives the selected value, and is instrumented. We also instrument the selectedIndex property, which gives the index of the selected item (an integer). The value property of option elements is also instrumented and returns a symbolic value in certain situations.

This is to handle the pattern where the value of a select box is accessed via the option element:

```
mySelect.options[mySelect.selectedIndex].value
```

- *Hidden and non-interactive inputs.* Hidden inputs never return a symbolic value, as they are not editable by a user. Note that this refers to input elements with the attribute type="hidden", rather than to normal text inputs which are simply not visible, for example by using CSS display:none. Various other types of input field are also not editable by a user, and as such are also not instrumented: button, reset, image, and submit.

- *Other input types.* All other input types are treated as normal text fields (in the context of generating symbolic values) and have their value properties instrumented to return a symbolic value. This includes all the HTML5 input types, such as email. The textarea element is ignored, simply because we have not seen it in our test sites. In principle it would be handled like any other text field, but allowing line breaks in the input.

By creating this symbolic information at the DOM property level we avoid propagating these values through the DOM API—only the JavaScript interpreter needs to be instrumented to deal with them. This means that the symbolic values are created as close to the symbolic interpreter as possible (as that is where they are used). The alternative—to create a fresh symbolic value and use that during the form filling part of the action sequence, which would then implicitly leave the DOM property holding a symbolic value—would require much more extensive instrumentation of the browser and DOM API, and would make form-filling simulation more difficult.

Restricting the possible values for each field

In ArtForm, different field types return different types of symbolic value—the value for a text box is a string, for a checkbox a Boolean, and so on. This provides a basic level of modelling the possible values each field can take. HTML permits other field types which impose extra restrictions on which values can be input. For example a select element produces a drop-down list with fixed options for the user to choose from. These constraints are enforced directly by the browser, and do not require any client-side JavaScript code, meaning they are never explicitly stated. To produce user-realisable input values, the analysis must model these input fields and only generate values which a user would be able to provide using a normal web browser. This is done by encoding facts about the specific fields in the DOM as extra constraints to be included with each path constraint, restricting the possible values the solver may select.

Example. In the running example, the To field would be restricted to a set of airport codes “ORD”, “JFK”, etc. In any constraint involving the corresponding variable to, we would add a constraint saying to = “ORD” ∨ to = “JFK” ∨ . . ., where the list of codes is populated from the values observed in the DOM.

The client-side code may also check the index of the selected item, rather than its value. When we encounter such code, we will also add a variable, to_index, standing for the index. The new variable is restricted by including it in the disjunction of possible airport values:

$$(\text{to} = \text{“ORD”} \wedge \text{to\_index} = 1) \vee (\text{to} = \text{“JFK”} \wedge \text{to\_index} = 2) \vee \ldots$$

This links the value and index so that only valid combinations of to and to_index may be selected by the solver. □

The full list of field types and how their values are restricted in the generated constraints is as follows:

- *Text fields.* Text fields are modelled by an unrestricted string variable, and no extra validation rules are required. Some extra attributes, such as maxlength can be used in HTML to restrict the possible values, but we have not encountered these commonly on real sites, so they are not modelled.
- *Checkboxes.* Similarly, checkboxes can be correctly modelled by a single Boolean variable, with no extra restrictions required.
- *Radio buttons.* Radio buttons represent a choice of a single element from a fixed list of options. A group of individual button elements are linked by a common name attribute, and no two radio buttons with the same name may be selected at once. Each radio button is represented by a Boolean variable, and each group is restricted by an extra exclusive-or constraint, meaning that exactly one must be selected. There is one exception: if the page has no button selected by default, then it is also allowed for all the buttons to be deselected in the solver’s result, representing no action being taken by the user.
- *Select boxes (by value).* A select input provides a drop-down list of possible values to choose between.[23] The selected value is accessed in JavaScript via the value property. As shown in the example above, this value is restricted by including a disjunction in the constraints where it must take one of the possible values from the concrete DOM. Note that the available values are those of the value attributes of the option elements inside the select; these are the values the code sees, such as “ORD”, as opposed to the values the user sees, such as “Chicago

[23]HTML also allows for multi-select boxes, but these are extremely rare in practice and are not modelled in ArtForm.

O'Hare". Figure 2.1 (on page 18) shows this distinction between the user-visible value and the internal value property.

- *Select boxes (by the selected index).* The setting of a select box can also be accessed via the selectedIndex property, which gives the index of the chosen item. Some sites use this index instead of, or as well as, the selected value to check what the user has chosen. In ArtForm, a selected index is modelled by an integer variable. This variable is constrained to be within the range of valid indices given the concrete options available in the DOM. If both the index and value of the same field are used in a path constraint, then the constraint sent to the solver includes an extra restriction to link them, as shown in the example above.

- *HTML5 input fields.* HTML5 introduces a new set of form input types on top of those described earlier. These new fields include implicit validation enforced by the browser. They are listed in Table 4.1. HTML5 also includes new attributes for standard fields to restrict the possible values, such as pattern, which provides a regular expression used to validate the field. None of these new field types are currently modelled in ArtForm.

The extra constraints restricting the possible values of these fields are only written when they are needed. If the variable corresponding to a field is used in a given path constraint, then the value restriction constraints for that field will also be included; if a variable is not used, its corresponding restriction constraints are not included. This simplifies the path constraints, and prevents the solver from returning values for variables which do not appear in the path constraint.

The new HTML5 input types are uncommon in practice, and we have not seen them in use in the sites we have investigated. As such, ArtForm does not include support for them—they are treated as standard text boxes with no special validation handling. Because browser support for the new fields is not yet universal, the validation rules are often implemented in JavaScript as well on sites where the new fields are used, meaning we can still analyse this JavaScript to explore the site's behaviour. As browser support improves, the use of the new field types and attributes is bound to increase.

If we were to model the new HTML5 input types, most would not pose a big problem. All the new input types can be thought of as a text field with an extra regular expression specifying the allowable values. For example, the color field could be modelled with the following regular expression:

```
#[0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f]
```

These expressions would be added as new form restrictions whenever a field of that type was used in a path constraint, and would limit the solver to suggesting values which are valid inputs for that field type.

Table 4.1: The new HTML5 input types and their implicit validation rules. The browser provides standard widgets to aid filling these fields, which is what allows the formats to be so specific.

Input type	Validation	Example
color	Allows input of colours in hex format.	#002147
date	Allows dates in RFC 3339 *full-date* format [78, §5.1].	1989-09-17
datetime	Allows date and time (in RFC 3339 *date-time* format [78, §5.1]), specifying a time zone.	1989-09-17T00:30:00+01:00
datetime-local	Allows date and time with no time zone.	1989-09-17T01:30:00
email	Uses a standard regular expression to validate email addresses.	ben@example.org
month	Allows dates consisting of only a year and month.	1989-09
number	Allows floating point numbers.	26.72
range	Allows numbers, typically integers but floating point numbers are also possible.	26.72
search	No extra validation; same as text by default.	
tel	No extra validation; same as text by default	
time	Allows a time with no time zone.	12:30:00
url	Allows the empty string or a valid absolute URL, according to a standard regular expression.	http://www.example.org
week	Allows dates consisting of a year and week.	1989-W37

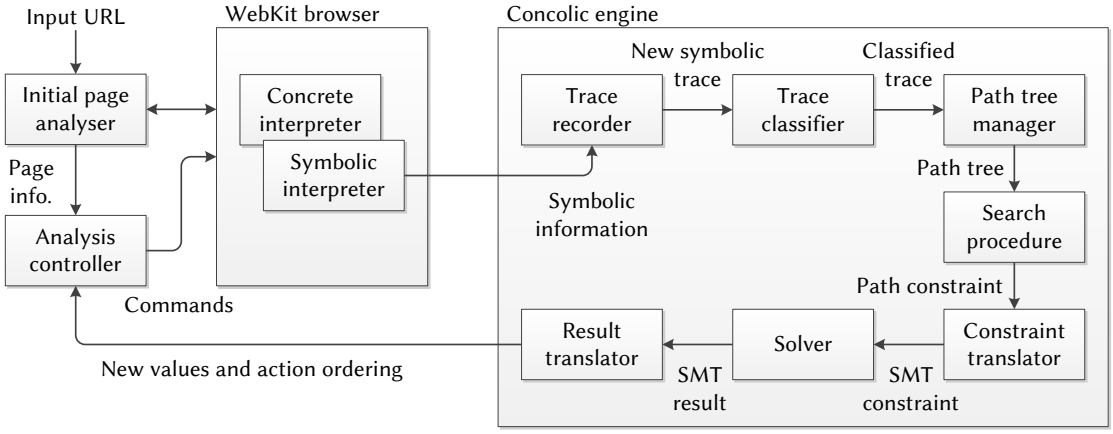


Figure 4.1: ArtForm's high-level system architecture.

Some attributes can cause problems however. For example, although the valid datetime inputs can be matched with a regular expression, enforcing the min and max attributes on these fields is more difficult. This would require special handling for dates and times, or solver support for a date-time type.

4.4 System architecture

Figure 4.1 shows the overall components of ArtForm. The concolic engine stores the state of the exploration, selects new unexplored paths to target and interacts with the constraint solver. Execution and symbolic tracing is done by the instrumented browser. After the initial page analyser obtains a list of fields to fill and a submit button, the main analysis controller can begin recording the initial execution trace. It then uses concolic testing, as in Algorithm 2, to find new sequences of actions and input values to test on subsequent iterations.

Most of the components of the concolic engine were already described in the previous chapter, and function the same way here. The generation of new symbolic values in the interpreter and the form-specific modelling were described in the previous section. This section describes the final outstanding components which are specific to web-form analysis: the page analyser and trace classifier.

Initial page analysis

Analysing a web page to detect the interface components requires two steps in ArtForm: identifying the fields to be filled, and identifying the form's submit button. This can be difficult because web pages may contain multiple forms, and often do not use standard markup for forms or buttons, so it is not enough to look at the HTML form elements to

reliably detect forms. In general, web form analysis also includes form labelling and other tasks [43], which can be avoided in our context.

ArtForm includes some simple heuristics for detecting submit buttons by considering the element type, whether the element is the descendant of a form element, and whether it has a JavaScript event handler registered. These heuristics work for very simple pages, but normally return many elements for any complex page, including many false positives, and ArtForm has no way to choose between these candidates.

For analysing real sites, we have added some simple integration with DIADEM. DIADEM is a fully automated data extraction wrapper generation framework, which is described in more detail in Section 4.11. DIADEM bases its analysis on domain knowledge of how web forms are constructed, both at the level of HTML and at the level of their visual structure, so it has much more robust detection and labelling of buttons (and other form components) than ArtForm can achieve. We developed a new "form observer" mode for DIADEM, which extracts and returns the forms and buttons identified by their system. The submit buttons identified by DIADEM are passed to ArtForm (as XPath expressions), which uses them as the entry-point for its analysis. Because the form observer produces many fewer false positives than the internal entry-point finding heuristics, it is feasible to run ArtForm on each entry-point for a site, to explore all the submit buttons identified.

As well as submit buttons, the form input fields must also be determined. This is simpler, as almost all input widgets must be backed by one of the standard HTML input fields (listed on page 20), which can easily be detected in the DOM. Even complex input widgets such as date pickers which are implemented in HTML and JavaScript typically save their values to standard input elements. In these cases, even though ArtForm cannot interact directly with the date-picker widget, it can inject values into the corresponding input field which is sometimes enough to successfully simulate a user input. Some widgets use hidden inputs, which are not directly editable by a user, so ArtForm does not modify them and cannot analyse these widgets.

ArtForm takes the list of form input elements on the page as the list of actions to execute. In most cases, this achieves useful results. It is also possible to manually customise the list of input fields to be tested for a certain site.

Trace classification for form submission

After each trace is recorded, it is classified before being merged into the path tree. The trace classifier analyses the annotations on each trace (the markers for page loads, alerts, and so on) to determine whether this trace represents a successful form submission or not. At the point in the trace where this is decided, the trace classifier inserts either an "End (Success)" or "End (Failure)" marker, which the search procedure will consider the end of the trace—there is no need to explore any further branches if we already know the result. Adding these markers as soon as the result can be determined is also

useful for the dynamic action reordering algorithm, which needs to know which actions caused a given abort.

Currently, our trace classification is very simple; alert boxes are considered a sign of failure, and new page loads are considered a sign of success. ArtForm has several other metrics for classification, such as whether error text such as "Warning", "Error", and so on was added to the DOM during the trace. However, this check is expensive and therefore is not enabled by default.

The core platform also includes other markers, such as detecting JavaScript exceptions, which can be used for other types of trace classification. These are not used in the main ArtForm tool, as they are not relevant to detecting successful form submissions. However, they can be useful for other applications, such as bug-finding.

4.5 Alternative approach: A fixed action ordering

An alternative to the dynamic-reordering algorithm presented in Section 4.2 is to use a single, fixed ordering for the actions. This is the most straightforward adaptation of concolic testing for the web analysis setting. Under this static-ordering approach, a single action sequence is chosen at the start of the analysis, and each trace recorded consists of the actions being executed (with their corresponding inputs) in that fixed sequence. It uses a variant of Algorithm 1 (page 41), adapted to trigger a sequence of user-level actions with inputs rather than executing a single function. The algorithm can be thought of as running on a single "top-level" function $F(v_1, \ldots, v_n)$, which is simply the concatenation of the individual actions $\text{Act}_1, \ldots, \text{Act}_n$ with each action Act_i being triggered with its corresponding input value v_i in turn. The definition of this top-level function is shown in Figure 3.3 (on page 49), which also shows how the action sequence is decomposed into the individual JavaScript functions executed by the browser. Keeping the same action sequence for each test is important so that the recorded traces are consistent with each other and can be merged into the same symbolic path tree.

The fixed-ordering approach can clearly lose some coverage, since testing multiple orderings is sometimes essential to uncovering certain behaviours. If the form satisfies the strong assumptions from our formal model given earlier (the input values of actions are fixed, and actions cannot interact except by reading each other's input values), then a fixed ordering can never achieve better coverage than Algorithm 2. However, when these assumptions are not satisfied—which is common in reality—a fixed-order approach is sometimes able to explore paths which the dynamic-reordering algorithm cannot reach. This is possible because interactions between actions which are not tracked by the analysis are still executed consistently, and are therefore predictable, when a fixed action sequence is used on every iteration.

For example, suppose a field's action code checks the field's validity, and sets a

global flag to show whether it is valid or not. This global flag will not typically have a symbolic value. Then the submit button handler reads the flag to decide whether to allow submission. When analysed separately, the two actions do not appear related, and a submission cannot be found. When analysed together, the paths in the field event handler which set the flag will go on to succeed in the button handler, whereas those which do not set the flag will go on to be rejected. This allows the submission to be discovered by the concolic testing.

Using a fixed action sequence also permits some extra features to be added to the analysis. We will show an example of this, where ArtForm is able to model the possible values for form fields in the fixed-order mode even if they are dynamically updated, because the full system state is known at each branch.

Dynamic changes to the fields' possible values

Recall from Section 4.3 that ArtForm records the possible values for form fields and uses them in the constraints to ensure the solver only provides suitable new input values to test. A relatively common pattern is to update the possible values for a field based on previous actions in the form. The most common examples are fields to choose a category and a sub-category (such as choosing a country and a city within that country, or the make and model of a car), and "matching pairs" constraints (such as the allowed pairs of origin and destination airports for an airline). In these cases, it is important to model how the possible values can change as the action sequence progresses.

Because the possible values of a field can change dynamically, they must be recorded at the point in the action sequence where the field is filled. Using this method, the recorded possible values used in the constraints correctly reflect the possible values which could have been chosen at the point in the trace where the choice had to be made. Constraining the possible values to those which were available when the page was loaded, for example, would not give an accurate representation of the possible values which are really available.

Under this model, different paths in the tree are associated with different sets of possible values, depending on updates made by previous actions. To implement this, ArtForm stores the possible values seen at injection-time in markers within the trace which denote the start of each form-filling action. These annotations link the action and its possible values, and also allow the possible values to be different in different parts of the tree. The search procedure can generate a path constraint where the input restrictions are tailored to the particular branch being explored.

This is only possible in the fixed-order mode because when the search procedure selects a certain branch to explore, that choice includes the paths through every previous action in the sequence. In fixed-order mode each trace covers the whole action sequence, so the paths through the execution tree encode the whole behaviour of the page up to that point. In the dynamic-reordering mode, we cannot guarantee that the possible-

values at a certain action will be the same on different iterations (not knowing how the concrete state will have been updated by earlier actions), so this feature cannot easily be supported.

Example. Consider the From and To fields in the example form from Chapter 3 (introduced on page 39). They are drop-down select boxes, and a common pattern on real airline sites is to update the possible values of To after From is updated. This is a useful filter in case the airline does not have routes between every pair of airports they serve. If the event ordering is From $\prec$ To (which is in fact required by the validation code in our example), then depending on the setting of From, there will be different possible values available when the action sequence reaches To. The analysis must be able to model the dynamic updates to this field in order to fully explore this form.

Suppose the airline has routes between London and New York, and New York and Chicago, but not directly between London and Chicago. In a path where from = "JFK" is selected earlier in the trace, the appropriate extra constraint for the possible values of To is to = "ORD" $\vee$ to = "LHR". When from = "LHR" is selected, the corresponding restriction constraint for To will be just to = "JFK". This changes which constraints might be satisfiable in later actions (representing which paths will be reachable in the application), depending on the paths taken and the possible values observed during previous actions.

Figure 4.2 shows how this information is stored in the tree so that it can be used to generate path constraints with different sets of possible values on each path. For example, any path explored in the right-hand subtree will necessarily use values from = "LHR", because of the first branch condition, and to = "JFK", because that is the only valid value for variable to along that path according to the annotations in the tree. □

ArtForm implements support for dynamically updated select boxes, which are common on real sites. We do not support dynamically-updated radio button groups, which are much rarer, although it would be straightforward to extend the same approach to radio buttons as well. Other, more general types of updates or modifications to the form, such as changing the code attached to fields, or dynamically adding and removing fields, are much harder to support.

Implementing the fixed-order mode

Recall that in generating our constraints, we need to take into account the ordering, since it will determine whether a reference to a form field value corresponds to the default value for that field or the corresponding input value. In the dynamic-reordering algorithm from Section 4.2 this is modelled by order-modified assignments σ^i, and corresponding ordered constraints Ord_i.

When a fixed-order action sequence is used, it is possible to avoid these extra constraints by using concrete information from the execution. Each input field is only

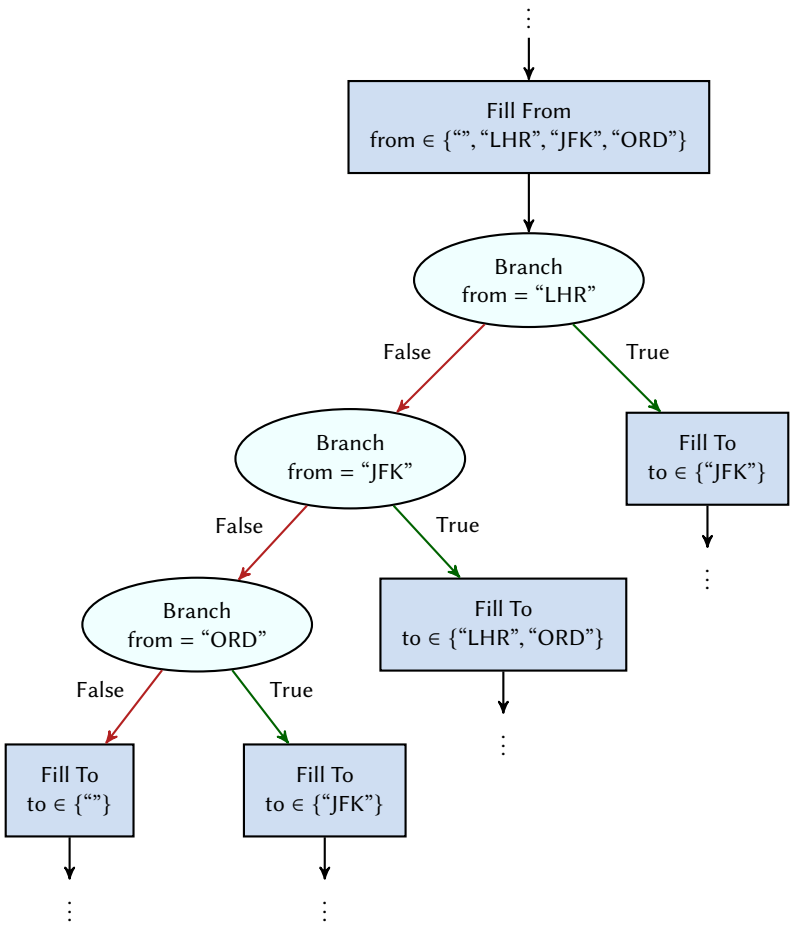


Figure 4.2: A section of the symbolic path tree for our airline example with dynamically updated possible values for the airports. New markers are added to show where each action is triggered, and the corresponding possible input values.

made symbolic at the point in the action sequence where it is filled, instead of having all fields being symbolic from the start. This means that when an action Act_i reads a variable v_j associated with Act_j which comes later in the action sequence ($i \prec j$), then v_j has no symbolic value and just appears as a constant in the program. This is appropriate because until the field is filled, its value *is* a constant; the default value D_j. By this method, each constraint in Act_i only includes variables corresponding to actions Act_k with $k \preceq i$. This effectively substitutes concrete values into the constraints in appropriate places, simplifying them.

ArtForm implements this solution in its static-ordering mode with a process known as *symbolic triggering*. As the action sequence is executed, a notification, or "trigger", is sent to each input field as its value is filled but before the input filling events are

simulated. This trigger switches the field from concrete to symbolic. As described earlier in Section 4.3, WebKit's DOM API is instrumented so that symbolic values are returned from form inputs. In the static-ordering mode this instrumentation waits for a symbolic trigger to be received for before the field begins returning symbolic values.

The technique of keeping form field values concrete until they are filled hides some symbolic information from the analysis. This is intentional and beneficial. The symbolic values which are hidden are from fields which have not been filled, so they *must* have their default value, and this value cannot be changed. Without the ordering constraints Ord_i (which are not useful when we use a fixed ordering) constraints on fields which have not yet been injected are useless, and can only lead to explorations which miss their intended target.

Choosing an action ordering

When using the static-ordering mode, it is not necessarily obvious how a good ordering should be chosen. A sensible default for web form analysis is to fill the fields in the order they appear in the DOM. This typically corresponds to the visual order on the page, and therefore the order a user would expect to fill the fields. Intuitively, one would expect it to be possible to submit the form using this ordering. ArtForm also includes the ability to manually choose any custom ordering of the fields.

To choose a fixed ordering more intelligently than this DOM-ordering default, we would prefer an ordering $\preceq_0$ which increases the amount of symbolic information visible to the analysis. The symbolic triggering system described above means that a variable v_j is only seen as symbolic in action Act_i if Act_j precedes Act_i in the action sequence (i.e. if $j \preceq_0 i$). An ordering $\preceq_0$ which maximises the available symbolic information has the property that every condition encountered in action Act_i mentions only input variables v_j associated with actions Act_j, where $j \preceq_0 i$. An ordering with this property is called *backward-looking* because each branch condition only uses variables which have already been filled in earlier in the sequence. A backward-looking ordering is desirable because it allows the concolic testing to observe the most branch conditions, and therefore explore the most paths during the analysis of a single ordering.

Since we do not have access to the program code for actions in advance, ArtForm executes the code using an arbitrary default order, and records an *action dependency graph*. The action dependency graph has an edge from index i to index j if the code for Act_j reads the input variable v_i associated with Act_i. Using this graph, we can search for a suitable backward-looking ordering and re-run the analysis under that ordering.

It is possible, and reasonably common in practice, for the action dependency graph to contain cycles, and this precludes any backward-looking ordering. A common example would be two date fields where the first date must be earlier than the second. When either field is filled in, it checks the other to make sure this rule has not been violated, causing a circular dependency in the graph.

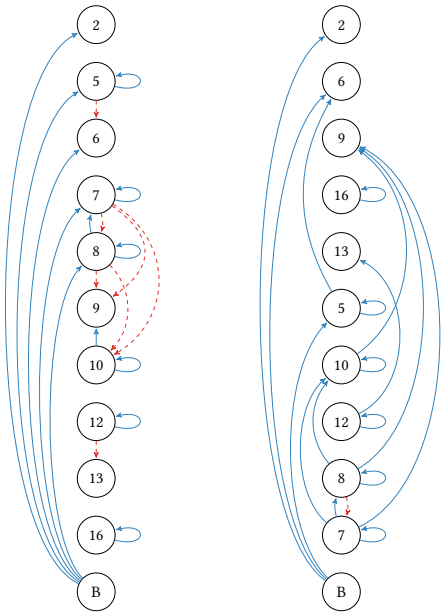


Figure 4.3: The action dependency graph after running ArtForm on airtran.com with the default document-ordered (left) and reordered (right) form filling actions.

When cycles are present, it is still desirable to find the "most backward-looking" ordering possible, by minimising the number of forward-looking edges. This gives an ordering allowing as many actions as possible to be fully exercised. We also calculate the smallest set of orderings which together cover every edge in the backward-looking direction. By rerunning ArtForm and testing each of the orderings in such a set, we can exercise each symbolic branch at least once.

Figure 4.3 shows the action dependency graph generated while testing an airline search form on airtran.com. The graph contains 20 fields, plus the submit button action, but those with no dependency edges are not shown. The blue edges pointing upwards are backward-looking edges which represent accessing the values of fields which have already been filled. The red dashed edges represent forward-looking edges, where a value is accessed before it is filled. There are several forward-looking edges in this graph, meaning the concolic testing will not be able to exercise any branches which depend on those reads. Note the cycle between fields 7 and 8 means that it is impossible to choose a fully backward-looking action ordering. The same site was re-tested with a new action ordering minimising the forward-looking edges; its dependency graph is shown on the right. The only remaining red edge $7 \longrightarrow 8$ is part of the cycle and therefore can't be explored without running tests on two different action orderings. This edge is forward-looking on the left and backward-looking on the right, so these two orderings together are enough to test each symbolic branch in the code.

4.6 ArtForm's manual analysis tools

As well as the automated concolic testing of web forms described in the previous sections, ArtForm also includes a manual demonstration mode. This mode allows a developer to explore the website's code via interacting with forms, and linking these interactions with both the concrete and symbolic behaviour of the underlying code. Showing this low-level JavaScript and browser information is useful for investigating a website's JavaScript code and seeing how the automated analysis will work.

In manual mode, inputs are entered by the user via a GUI view from ArtForm's instrumented WebKit browser. The developer can interact with a web page as an end-user would, and can understand the codebase by looking at different reports produced by recording this interaction. There is a *trace report* which shows the tree of function calls made, and a linked *coverage report* which shows the JavaScript source code that has been explored so far. In addition, symbolic execution traces can be recorded, which show how symbolic values (that is, those which depend on user inputs) were used during the interaction, and in particular how they affected the control-flow of the JavaScript code. As in the automated modes, the traces include events from our instrumented browser which connect code execution with user interaction, such as when a new page was loaded, or an alert box was shown. These markers can also be used to detect JavaScript bugs, by checking for calls to console.error, failed assertions, and so on.

Manual mode is useful for understanding which JavaScript code corresponds to each user action, and how that code depends on user inputs. Figure 4.4 shows the manual mode's browser view. The user sees a web page in the main window, and can record their interactions with the page. While recording, the JavaScript events corresponding to each user action are recorded in a symbolic form as a *symbolic trace*. The user can inspect an individual trace or view a summary of the whole browser session using the buttons on the right. Being able to see what JavaScript code executed and where data flowed for a particular action in the context of the global function call tree is very useful for investigating third-party sites.

Figure 4.5 shows a path trace report and a coverage report for the form validation code of the airline flight search form shown in Figure 4.4. They are linked so that the implementation of functions seen in the trace report can be looked up in the coverage report. The highlighting in the coverage report shows which lines were covered during the run; in this example, it is most of the displayed functions. It also shows which lines make use of symbolic information. In the example this is only one line (the fourth one shown; highlighted in green), which is fetching the value property of an input field. Because JavaScript libraries such as jQuery are implemented in JavaScript, they are included in the reports like any other JavaScript code.

ArtForm includes a simple proxy which can intercept the pages downloaded by its browser and reformat minimised or obfuscated JavaScript code on-the-fly. This makes

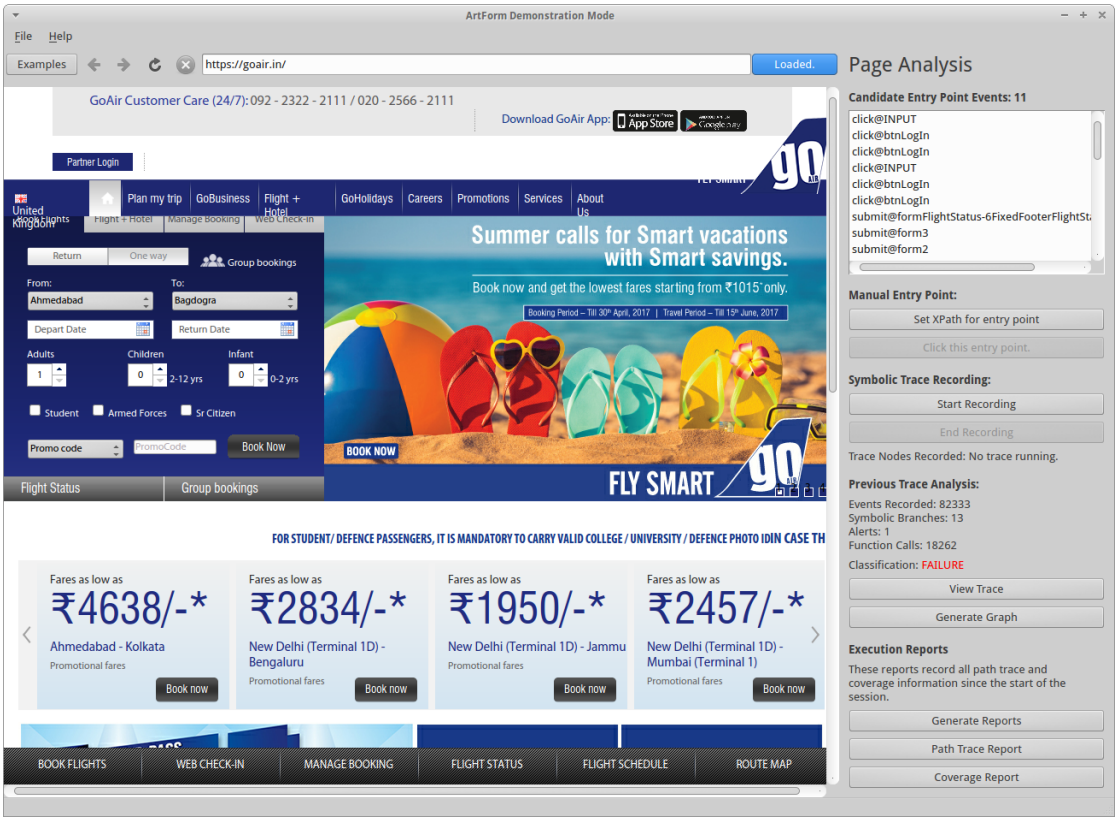


Figure 4.4: ArtForm's manual mode. The left hand panel is an interactive browser window. The controls on the right can be used to view information about the page, and record and inspect symbolic traces of actions performed.

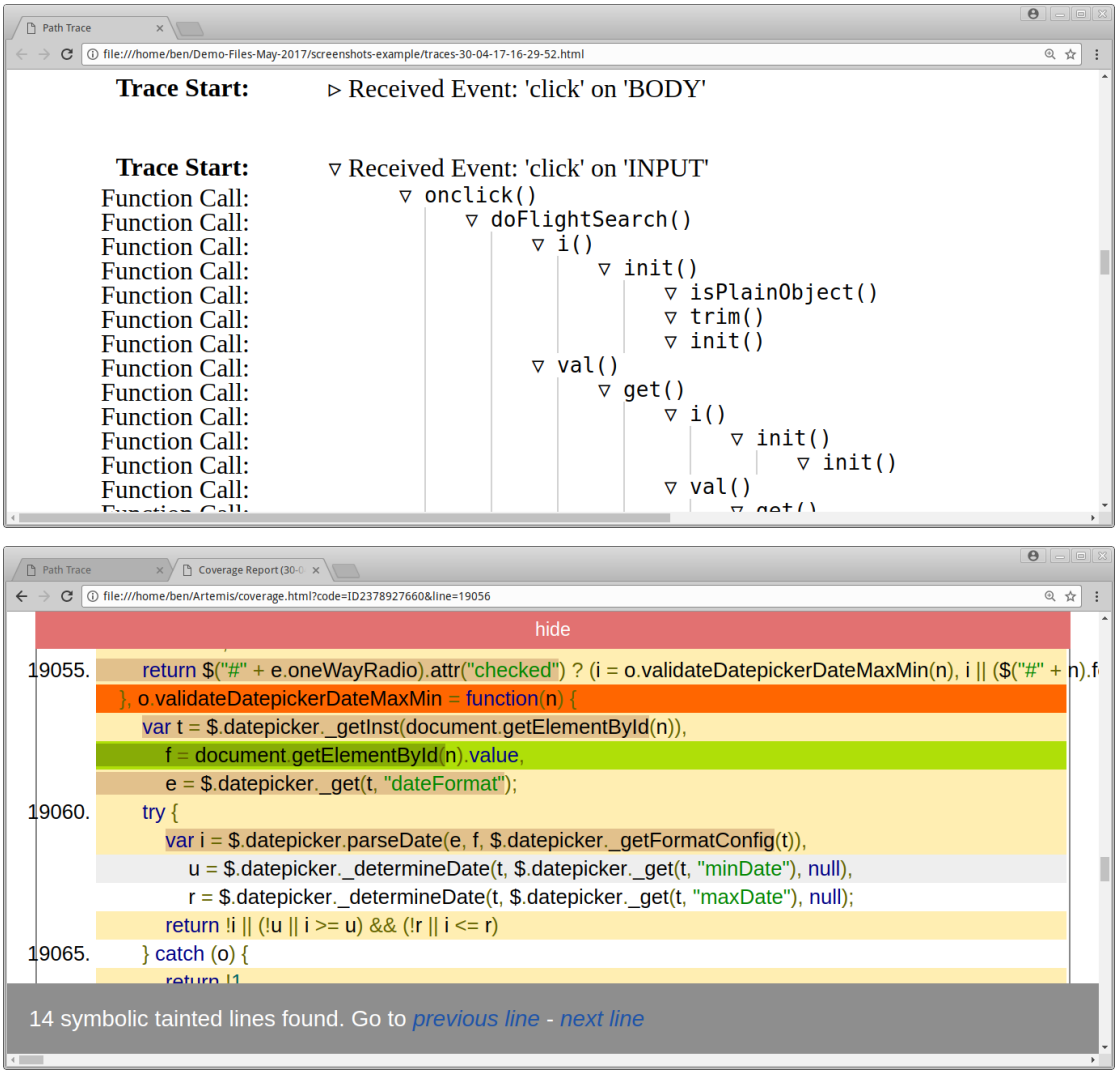


Figure 4.5: The path trace report (above) and code coverage report (below).

the code easier to read manually, and also makes line-level statistics, such as the line coverage shown in the coverage report, more useful.

4.7 Providing concolic advice to other tools

As well as its standalone analysis modes, ArtForm can also be used as a source of concolic testing advice for third-party tools. In this mode ArtForm acts as an advice server which is controlled by the client tool—or directly by a user—to execute actions on a web page and return various analysis results. This allows the client tool or user to mix suggestions from the solver with manual inputs and other heuristics.

For example, a tool like DIADEM considers the user-level view of a web page in order to generate data extraction wrappers. By connecting it to ArtForm's advice mode, it would be possible to also include more implementation-level information about the page in DIADEM's analysis. This can allow traditional testing or data extraction tools to be augmented with a more powerful understanding of the page's low-level operation.

In the advice mode, the client takes control over ArtForm's browser by issuing commands, such as clicking buttons and filling form fields. Inputs are chosen by the client, and the actions are recorded symbolically, adding traces to a symbolic execution tree just as in the fully automatic concolic mode. The user can ask for advice at any point about which inputs to try next, as well as and other low-level information about the page. At this point ArtForm's concolic testing procedure runs on the symbolic execution tree and generates a set of values leading to a new execution path. The client may use this advice immediately or choose their own values to test. The advice mode does not impose any special requirements on the behaviour of the client tool; the provided advice is optional and ArtForm can continue to work and provide results even if some of its results are never used.

ArtForm's advice mode supports running multiple concolic testing instances in parallel. This allows the client tool to interleave testing of different parts of a website in a single session. If there are multiple forms on the page, or multiple pages being tested, ArtForm does not require that one must be completely tested before recording traces and providing advice to explore another. Each separate sequence of actions to be tested is assigned an identifier, which is used to match up traces from matching action sequences and provide advice on a per-sequence basis.

The extra restrictions on values discussed in Section 4.3 to reflect the possible inputs seen in the DOM are enabled in advice mode. Advice mode also supports restricting the possible values of dynamically updated fields, as described in Section 4.5. This is possible because the advice mode uses a fixed action sequence for each testing instance, filling a certain set of fields in a particular order. Some of the flexibility of the dynamic-reordering mode is still retained. The client tool is free to run several short tests on different fields in sequence, and ArtForm can report on the dependencies between

them, using the same action dependency analysis described in Section 4.5, and these dependencies will suggest new orderings which may be useful to test. The individual analysis instances are robust enough that testing the same trace starting from different application states (for example after filling different sets of fields in different ways) can still provide useful results. How strictly the client tool follows ArtForm's model for concolic testing (by resetting the browser state, following the same sequence of actions, and so on) is a trade-off; the best results are achieved by following our model, but ArtForm will still provide useful "best-effort" advice for whatever actions the client tool chooses to make.

The advice is available via a JSON-based API for third-party tools or scripting. The API includes commands to control the browser and manipulate the web pages, including loading pages; clicking on elements, via GUI-level or simulated clicks; filling form fields, with various levels of form-filling simulation; using the browser's back button, and executing arbitrary JavaScript snippets. There are also a set of commands for retrieving information about the current page: returning the URL, page title, DOM statistics, and optionally the serialised DOM itself; listing the event handlers registered on the page; showing information about a particular element; and evaluating arbitrary XPath expressions. Finally there are commands which expose the analysis results from ArtForm: the coverage report, showing the line-coverage of all the JavaScript included by the page; the action dependency graph, showing which JavaScript event handlers read from which form fields; and new input value suggestions from the concolic testing.

Example. Figure 4.6 shows an example session using the API commands to record a symbolic trace and get back new value suggestions from ArtForm's concolic testing. This example analyses the simple airline form from Chapter 3 (page 39). In this case the commands were sent manually using Postman,[24] but they could have been scripted via curl[25] or sent by any other tool which can make HTTP requests.

First, the page is loaded in ArtForm's browser. The event handlers registered in JavaScript are inspected, to see which interactions with the page will cause code to be executed. Next, we begin recording a symbolic trace and fill the form by sending "forminput" commands for each of the three form fields and finally issuing a command to click the form's submit button. Having recorded the trace, we end the recording and request advice from ArtForm's concolic testing. Setting the amount to 0 means ArtForm will return as many different value suggestions as it can find in the current concolic tree. In this case there are three suggestions:

- {From = ""}
- {From = "BHX", To = "BHX"}
- {From = "BHX", To = "", Date = "01/02"}

[24] https://www.getpostman.com/
[25] https://curl.haxx.se/

Request	Response
{ *"command"*: *"pageload"*, *"url"*: *"http://localhost:8088/airline.html"* }	{ *"pageload"*: *"done"*, *"url"*: *"http://localhost:8088/airline.html"* }
{ *"command"*: *"handlers"* }	{ *"handlers"*: [{ *"element"*: *"//input[@id='button']"*, *"events"*: [*"click"*] }, { *"element"*: *"//select[@id='date']"*, *"events"*: [*"change"*] }, { *"element"*: *"//select[@id='from']"*, *"events"*: [*"change"*] }, { *"element"*: *"//select[@id='to']"*, *"events"*: [*"change"*] }] }
{ *"command"*: *"concolicadvice"*, *"action"*: *"begintrace"*, *"sequence"*: *"MySequence"* }	{ *"concolicadvice"*: *"done"* }
{ *"command"*: *"forminput"*, *"field"*: *"id('from')"*, *"value"*: *"LHR"* }	{ *"forminput"*: *"done"* }
{ *"command"*: *"forminput"*, *"field"*: *"id('to')"*, *"value"*: *"EDI"* }	{ *"forminput"*: *"done"* }
{ *"command"*: *"forminput"*, *"field"*: *"id('date')"*, *"value"*: *"01/01"* }	{ *"forminput"*: *"done"* }
{ *"command"*: *"click"*, *"element"*: *"//input[@type='submit']"* }	{ *"click"*: *"done"* }
{ *"command"*: *"concolicadvice"*, *"action"*: *"endtrace"*, *"sequence"*: *"MySequence"* }	{ *"concolicadvice"*: *"done"* }
{ *"command"*: *"concolicadvice"*, *"action"*: *"advice"*, *"sequence"*: *"MySequence"*, *"amount"*: 0 }	{ *"concolicadvice"*: *"done"*, *"sequence"*: *"MySequence"*, *"values"*: [[{ *"field"*: *"//select[@id='from']"*, *"value"*: *""* }], [{ *"field"*: *"//select[@id='to']"*, *"value"*: *"BHX"* }, { *"field"*: *"//select[@id='from']"*, *"value"*: *"BHX"* }], [{ *"field"*: *"//select[@id='to']"*, *"value"*: *""* }, { *"field"*: *"//select[@id='date']"*, *"value"*: *"01/02"* }, { *"field"*: *"//select[@id='from']"*, *"value"*: *"BHX"* }]] }
{ *"command"*: *"fieldsread"* }	{ *"fieldsread"*: [{ *"element"*: *"//select[@id='date']"*, *"event"*: *"forminput/simulate-js"*, *"reads"*: [{ *"count"*: 1, *"field"*: *"//select[@id='date']"* }, { *"count"*: 1, *"field"* : *"//select[@id='from']"* }] }, { *"element"*: *"//select[@id='to']"*, *"event"*: *"forminput/simulate-js"*, *"reads"*: [{ *"count"*: 1, *"field"*: *"//select[@id='from']"* }, { *"count"*: 1, *"field"*: *"//select[@id='to']"* }] }] }

Figure 4.6: An example use of the advice-mode API to analyse a simple web form.

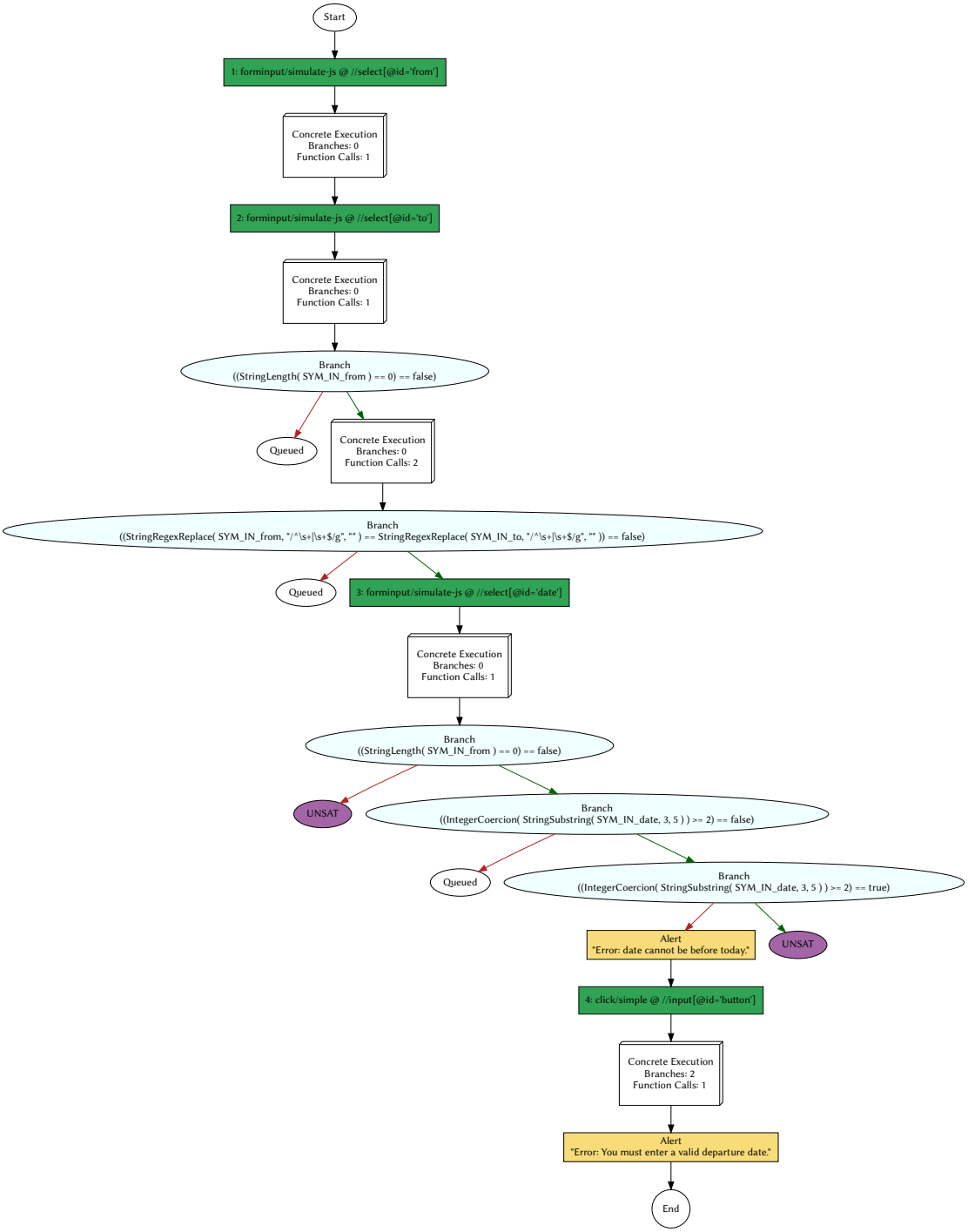


Figure 4.7: The symbolic path tree after recording a trace and generating three suggested new inputs.

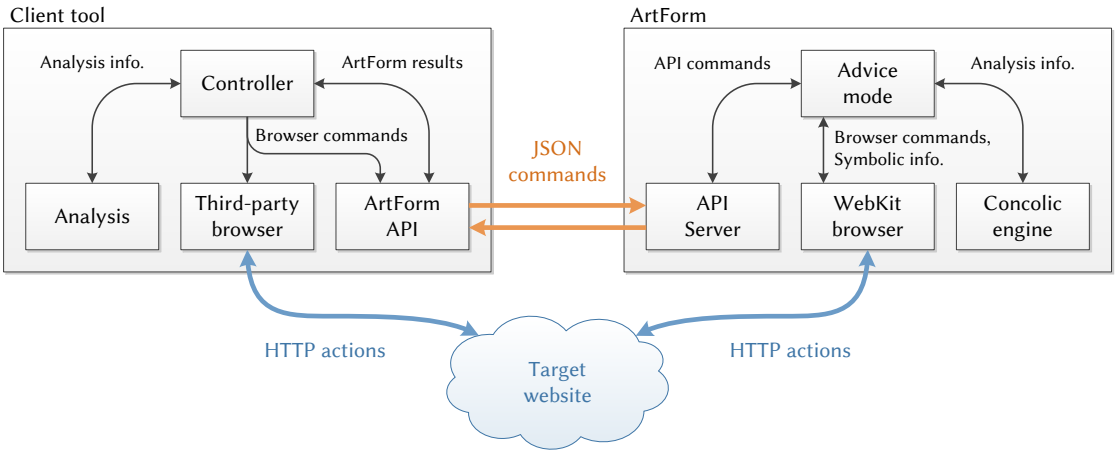


Figure 4.8: The proposed architecture of a combined tool where the concolic advice server works in parallel with an existing web analysis tool.

The symbolic path tree is shown in Figure 4.7, and contains a single trace. The advice mode operates by marking suggested paths as "queued" so that they are not suggested again in later requests for advice. In this case the three suggestions correspond to the three queued paths in the tree. These value suggestions all assume that the same action sequence will be used (in this case From; To; Date; Click). The final "fieldsread" command shows which form fields were read during which actions, which may suggest further orderings to test. □

Figure 4.8 shows the architecture for a proposed system combining ArtForm's advice mode with a third-party analysis system. Under this model, ArtForm would mimic the browser actions of the third-party tool so each would maintain a consistent state. ArtForm would then be able to return extra analysis information which the other tool could not determine itself.

4.8 Determining new user-realisable actions

So far, we have assumed that the actions relevant to filling and submitting a form are known in advance. However, this is not necessarily the case when analysing real sites.

JavaScript events are typically registered on the DOM element they affect, using the onclick attribute or the addEventListener method. These events can be identified by watching the DOM and tracking when event listeners are added or removed; producing a list of DOM elements which have handlers for certain events. However, there are cases when detecting actions statically is not possible. Many JavaScript libraries, including jQuery, provide an alternative event registration mechanism known as *event delegation*, which decouples the event handling code from the element it corresponds to. Under this

scheme only a single JavaScript event handler is registered and known to the browser. If an analysis tool tries to trigger this event directly it will have no effect, unless an appropriate *target* element for the event is specified.

Concolic testing can be applied to observe the top-level event handler checking the target element for certain properties, and thus dynamically determine which target elements are relevant. This is another instance where low-level JavaScript analysis information can be applied to learn about user-level actions and events.

Another consideration which is ignored when simply checking for JavaScript event handlers is element visibility. By checking which elements in the DOM are really visible on the page we can limit the analysis to only consider *user-realisable* events.

By detecting delegated events and determining which event targets are visible, we can provide a much more accurate simulation of which events can be triggered by a real user. Identifying user-realisable actions is important to understanding how a web interface is *expected* to be used, and therefore which actions can lead to valid new application states.

Figure 4.9 shows which types of events are triggered when using different detection methods for interactive elements. DOM-based tools use heuristics to determine which elements "look like" buttons, represented by the red area in the middle of the figure. Artemis and other JavaScript-focused tools can detect JavaScript event handlers, and therefore only explore actions which may have some effect on the page state (represented by the blue area to the left). Ideally, a tool should be able to detect delegated events (shown in the green area on the right) as well as native JavaScript events, while also rejecting events attached to invisible elements (the bottom half of the figure). This desired behaviour is represented by the union of the dark blue and dark green shaded areas. Note that a delegation or visibility analysis alone would detect respectively too many and too few elements for a true simulation of a real user.

Event delegation

Event delegation is an alternative to JavaScript's built-in event registration process which is provided by many JavaScript libraries, most notably jQuery. Events can be registered on all elements matching a certain specification (typically a CSS selector), even if new matching elements are added to the DOM dynamically. This is a powerful, and flexible, system for developers. Event delegation is implemented with a single top-level event handler registered at the root of the document which contains a lookup table of CSS selectors and corresponding developer-provided callback functions. When any element on the page is clicked, a process called *event bubbling* (a standard part of the JavaScript event model) causes the event to be fired on the target element which was actually clicked, and then subsequently on each parent element in turn, until the event is eventually handled and the bubbling is cancelled. Under event delegation, the event bubbles up to the top-level handler, which checks the target property of the event

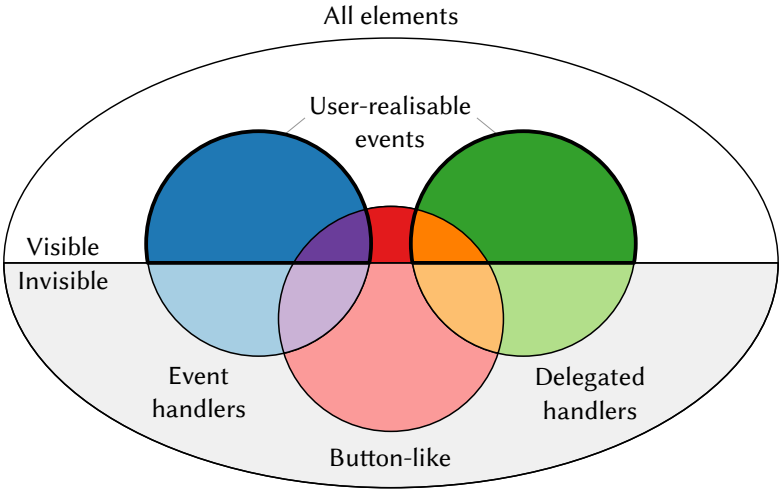


Figure 4.9: The different kinds of elements which may be considered interactive by different tools.

object (which is a reference to the element which was originally clicked) against its list of CSS selectors. If any selector matches, the corresponding callback is executed, which runs the "application-level" event handler for the clicked element.

Listing 5 shows a simplified example of event delegation, developed from, and using similar underlying code to, a real example from the jQuery documentation.[26] The on function is used to register a new event handler for clicks on paragraph elements using event delegation. The top-level event handler is added to the document's body element. Figure 4.10 shows how clicks on the paragraph elements bubble up to the top-level event handler. When this top-level handler receives a bubbling click event, it builds a list of all p elements on the page. Each event has an associated *event object*, whose target property references the DOM node where the event originated; in this case, the element which was originally clicked. If the event's target element is in the set of p elements detected, then the developer-provided event handling function is executed. Because the CSS selector matching is done dynamically at each click event, it can match newly-added paragraphs without having to track changes to the DOM, but this also makes it impossible to statically determine the list of elements which will respond to a given event. In the normal event-registration scheme, an event handler would have to be registered on each p element individually, and when a new one was added to the DOM, a corresponding event would need to be attached.

Most interface-analysis or web testing tools are not based on detecting JavaScript event handlers in the first place. They have rules or heuristics for which events should be triggered on which elements without checking beforehand whether any code is attached.

[26] http://api.jquery.com/on/#example-9

Listing 5 An example of event delegation. This code is a simplified version of the delegation code found in jQuery, showing how a single top-level event handler is used to handle click events for all child elements matching a certain selector.

```
<!doctype html>
<html>
<head><title>Delegation example</title></head>
<body>
  <p>Click me!</p>
  <span></span>
  <script>
  // A simple re-implementation of the jQuery code to show roughly how it works internally.
  function on(container, type, selector, handler) {
    container.addEventListener(type, function (event) {
      // Fetch an array containing all descendants of container which match the selector.
      var matches = Array.prototype.slice.call(container.querySelectorAll(selector));
      // If the target matches the selector then fire the event.
      if (matches.indexOf(event.target) >= 0) {
        handler(event);
      }
    });
  }

  var count = 0;
  on(document.body, "click", "p", function (e) {
    var newPara = document.createElement("p");
    newPara.appendChild(document.createTextNode("Another paragraph! " + (++count)));
    // Add the new paragraph after the clicked target.
    e.target.parentNode.insertBefore(newPara, e.target.nextSibling);
  });
  </script>
</body>
</html>
```

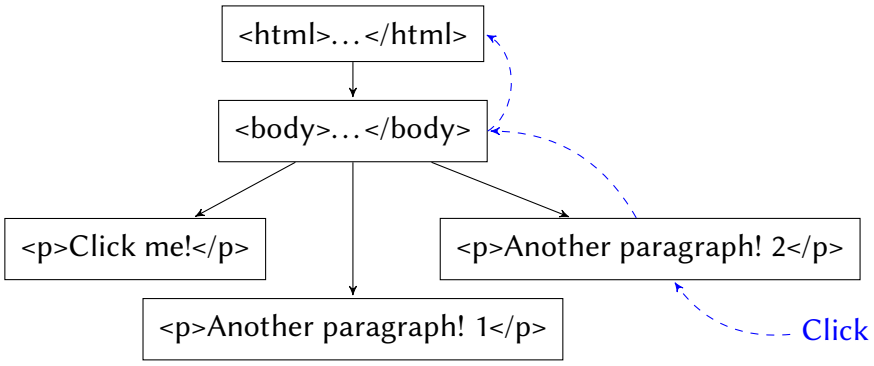


Figure 4.10: The DOM tree for the delegation example (after a couple of clicks have been made). A click event on the third p element bubbles up the tree as shown by the dashed arrows. In our example the event is handled at the body element.

Crawljax, an AJAX-aware web crawler, is an example taking this approach [100]. This means that delegated events will be triggered and tested correctly, but only if the tool happens to trigger an appropriate event on an appropriate element; the search space for finding interesting code is enormous. Other tools *do* track the JavaScript event handlers, and therefore have a more direct problem with delegation (which prevents them from triggering the interesting code at all). Artemis, for example, is a web application test generation tool [8] (discussed in more detail in Section 4.11) which has been extended to include special-purpose detection of events registered using jQuery, so they can be added to the list of event handlers alongside the natively-registered events. Artemis is then able to trigger these events correctly and test the corresponding code. The downside to this approach is that it is library-specific. I am not aware of any tool which detects delegated events in a general-purpose way.

Detecting delegated events using concolic testing

By considering the target property of an event object as an input parameter to the event, we can apply concolic testing to discover new target elements which exercise new code paths in the top-level event handler, and thus eventually reach the "application-level" event handler.

We instrument the event target objects to make their properties symbolic. When the event handler checks whether the target is relevant (i.e., whether it matches the desired CSS selector), this will give rise to symbolic constraints over the properties of the target object.

As with the select box and radio button constraints discussed earlier, we will include extra constraints to restrict the possible values. In this case, the possible values for the target object are those DOM elements which are descendants of the element where the top-level event handler is registered. Those are the elements whose events could bubble up to the handler being analysed. Therefore, this set of DOM nodes are encoded into the constraints to provide the possible values for the new symbolic variables.

Example. Suppose that the running airline example uses jQuery to register its final button click handler. The following line of JavaScript sets up the delegated event handler to call the validate_button function when the submit button (identified by CSS selector #button) is clicked.

```
$("#form").on("click", "#button", validate_button);
```

The browser sees a single click handler registered on the form element, whose event handling function contains the delegation code. This code is similar to that shown in Listing 5, but is obviously much more complex in jQuery's real implementation, in order to handle extra features and corner cases.

To analyse this example, our delegation analysis first triggers the event with the default target being the same form element where the event is registered (in this case

the body element), and records a symbolic trace. The concrete jQuery event-handling code responds to this event by checking whether the event's target node matches the given CSS selector. In the version of jQuery tested, this particular invocation[27] uses the Sizzle selector engine,[28] so the CSS selector is evaluated directly in JavaScript, instead of being passed to the browser. This means we see individual property checks against the target node in the symbolic trace. The constraints at these symbolic branches are solved, as in any concolic testing, generating a new choice for the target element to be tested at the next iteration.

Figure 4.11 shows the completed tree after using concolic testing to explore the jQuery delegation code in this example. There are two paths explored, the initial path which does not lead to any code execution, and the right-hand path, which leads to the application-level event handler. Several interesting constraints are visible. For example, the final constraint on the right-hand path is

$$\textbf{StrOp}(\textbf{SymProp}(\textbf{SymObj}(\textit{"target"}),\ \textit{"id"}),\ \textbf{STR\_SEQ},\ \textbf{ConstStr}(\textit{"button"}))$$

which checks that the target element has id attribute "button".

The possible values for the target element must be constrained to those which are really realisable in the current DOM. These are the descendants of the form element where the top-level delegation handler is registered. Listing 6 shows a simplified version of this part of the DOM, and Listing 7 shows how those DOM nodes are encoded into SMT constraints. The SMT constraints model the symbolic DOM node as a collection of properties, each represented by a variable. These variables are used in the main path constraints whenever a property of the symbolic object is accessed. The possible DOM nodes are represented as a disjunction of possible nodes, where each clause specifies the concrete property values for that node, as seen in the current snapshot of the real DOM. Thus, the target DOM node selected by the solver is restricted to one of these valid possibilities.

Property variables are included in the DOM constraint whenever they are observed in the main path constraint. This example shows the id and nodeType properties. The extra pseudo-property SOLUTIONXPATH is never used in the constraints. It contains an XPath expression uniquely identifying the corresponding DOM element. This is included so that the solver's solution references the target element in a stable way across iterations; the XPath can be used to look up the same element again during a later test. In our example, the solver finds a solution using the form submit button (identified in the last few lines of Listing 7) as the new target element, and was therefore able to execute the application-level event handler. □

[27] jQuery determines whether to use the browser's querySelectorAll or its own JavaScript-based Sizzle CSS engine depending on the selector being checked.

[28] https://sizzlejs.com/

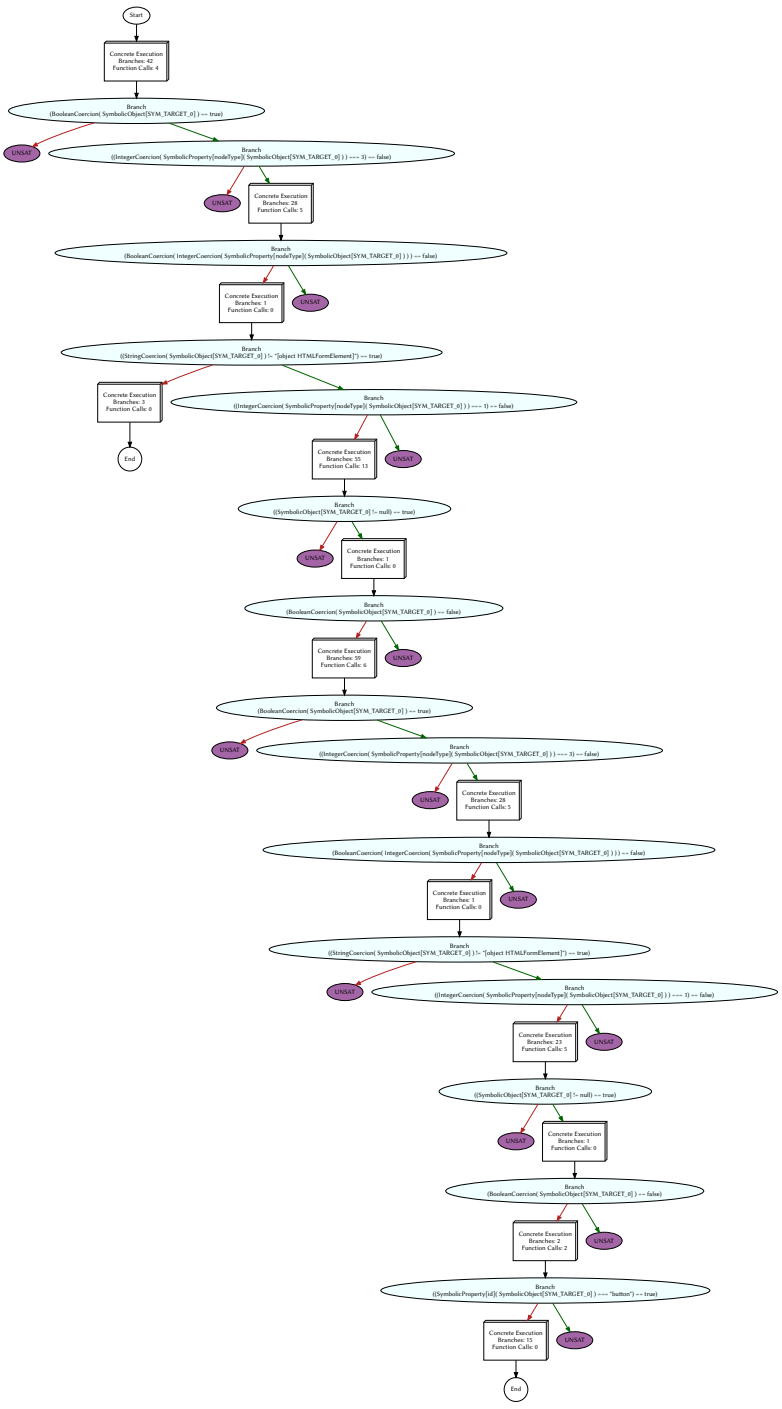


Figure 4.11: The concolic testing tree for the jQuery delegation code in the running example.

Listing 6 The DOM subtree rooted at the form element in our example.

```
<form action="search.php" method="POST" id="form" >
    <label for="from" >Departure Airport:</label>
    <input type="text" id="from" name="from" ><br>
    <label for="to" >Destination Airport:</label>
    <input type="text" id="to" name="to" ><br>
    <label for="date" >Departure Date:</label>
    <input type="text" id="date" name="date" >
    <span class="instructions" >(Format: dd/mm)</span><br>
    <input type="submit" value="Search" id="button" >
</form>
```

Listing 7 The SMT constraints encoding the possible target elements.

```
(assert (or
    (and (= SYM_TARGET_0 702)
        (= SYM_TARGET_0_SOLUTIONXPATH "//form[@id='form']")
        (= SYM_TARGET_0__TOSTRING "[object HTMLFormElement]")
        (= SYM_TARGET_0__id "form")
        (= SYM_TARGET_0__nodeType "1"))
    (and (= SYM_TARGET_0 704)
        (= SYM_TARGET_0_SOLUTIONXPATH "//form[@id='form']/label[1]")
        (= SYM_TARGET_0__TOSTRING "[object HTMLLabelElement]")
        (= SYM_TARGET_0__id "")
        (= SYM_TARGET_0__nodeType "1"))
    (and (= SYM_TARGET_0 707)
        (= SYM_TARGET_0_SOLUTIONXPATH "//input[@id='from']")
        (= SYM_TARGET_0__TOSTRING "[object HTMLInputElement]")
        (= SYM_TARGET_0__id "")
        (= SYM_TARGET_0__nodeType "1"))
  ⋮   ⋮
    (and (= SYM_TARGET_0 731)
        (= SYM_TARGET_0_SOLUTIONXPATH "//input[@id='button']")
        (= SYM_TARGET_0__TOSTRING "[object HTMLInputElement]")
        (= SYM_TARGET_0__id "button")
        (= SYM_TARGET_0__nodeType "1"))
))
```

Element visibility

Another important part of determining which events are user-realisable is detecting which elements are visible on the page. For example, an element may be placed off-screen, hidden behind another element, or removed from the page flow by CSS rules. Even though such elements appear in the DOM and are accessible to automated tools, it is useful to detect them and exclude them from user-simulated events, seeing as a real user would not be able to see them or trigger events on them.

It is not always straightforward to define which elements should be considered visible on a page. In fact, the rules depend on the event type being considered. For example, some form fields can be focused and filled using keyboard controls, even if they are not visible on the screen, or a form can be submitted by pressing enter even if the submit button is not visible. An element is considered visible to a certain event type when a normal user can trigger that event on that element.

Click events have the most natural notion of visibility, and are most relevant to the delegation analysis. Table 4.2 shows the different ways that elements might be considered visible or invisible to click events. A click event is considered invisible when: (a) the element has been removed from the DOM completely, or from the page layout by a CSS rule, (b) the element is placed outside the page's visible area, and it cannot be revealed by scrolling, (c) the element is totally obscured by another element, or (d) the element has no size (i.e., its width or height is 0). Note that being transparent does not necessarily make an element invisible if it can still be clicked. Because of event bubbling, child elements must also be considered; an invisible may still have an event triggered if it bubbles up from a visible descendant element.

There is no direct check available for our notion of visibility in the existing DOM or browser APIs. For example, it is not enough to check for an element's presence in the DOM, or for certain CSS properties. Because ArtForm includes an instrumented web browser engine, we are able to create customised visibility checks using the internal layout and rendering information of the browser.

Each element has a bounding box calculated by the browser within which it can receive click events. An element is considered visible to click events if the union of its bounding box with those of all its descendant nodes (including both normal elements and text nodes) intersects with the page's visible area. The descendants must be included because events triggered on them are able to bubble up to the event being considered and be fired there as well. This information must be recalculated dynamically in case the element is covered or uncovered as a result of a previous action. This is still not a perfect test, but it gives a reasonably close over-approximation of the visible elements, so it provides an accurate and safe way to reject known-invisible elements.

In a survey of 888 random URLs sampled from the Common Crawl search index [30], we found that on average, 61% of the button elements[29] and 64% of the form input

[29] That is, button elements and input elements of types button and submit.

elements[30] on the pages were user-visible according to the above method. This shows that there can be significant reductions in the number of elements or actions to be explored if element visibility is taken into account.

In general, more complex pages (which are more expensive to test) have more invisible elements, and therefore benefit the most from checking for user-visibility. Of the 888 test URLs described above, 144 (16%) had 10 or more form input elements, and on those pages only 40% of the form input elements were detected as user-visible on average. Similarly, for the 198 URLs (22%) with at least 5 button elements, 47% of the buttons were user-visible on average.

4.9 Experimental evaluation

Our evaluation of ArtForm consists of three parts. First, we extend the synthetic JavaScript generator described in the previous chapter to generate web forms with random validation rules to explore. These examples are tested in ArtForm and two alternative tools. Second, we test a suite of small, self-contained example forms taken from the code-sharing website JSFiddle [74]. These provide realistic code samples, without the added complexity of being embedded in a full web page. Finally, we test ArtForm on some real-world websites with complex query interfaces. Although the complexity of the sites (such as their use of custom widgets which cannot be analysed) and the limitations of our prototype tool make it difficult to run a comprehensive evaluation, these tests show that concolic testing is a powerful and promising approach to web interface understanding.

Synthetic web forms

We first evaluate ArtForm on a suite of 1000 synthetically generated web forms with JavaScript validation. The generator first generates a random form skeleton—consisting of the number of fields, their types, and so on—and then supplements it with validation rules using the JavaScript code generator described in Section 3.6. The forms use an average of 8.57 form fields, and 47.16 lines of JavaScript validation code.

We compared ArtForm with two alternative approaches, Crawljax and Artemis. Crawljax is an automatic crawler for dynamic websites which is based on a dynamic analysis of AJAX-powered web applications [100]. It builds a state graph describing the actions available on a page and how they modify the application state. It explores different sequences of actions until no more states can be discovered. In the absence of any domain knowledge, Crawljax can be seen as a state-of-the art approach to crawling complex web applications, including web forms, although it does not directly perform

[30] That is, input elements of types text, radio or checkbox; select and textarea elements; and the HTML5 input types listed on page 104.

Table 4.2: The different ways an element might be considered invisible.

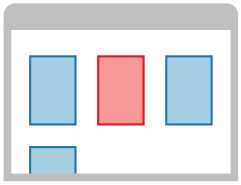
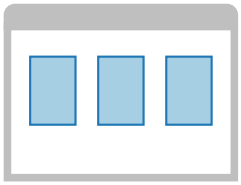
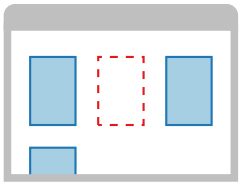
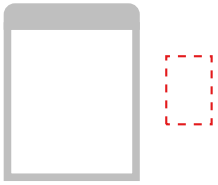
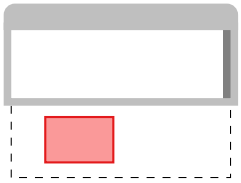
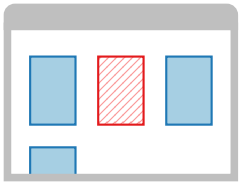
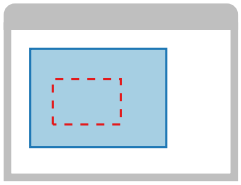
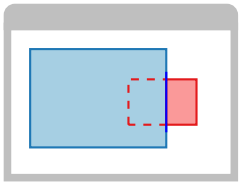
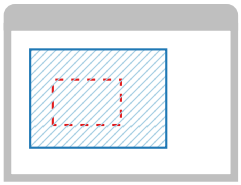
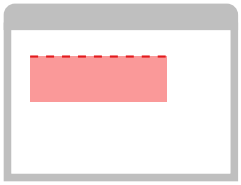
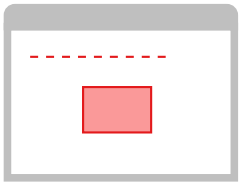
Element type		Visibility to click events
Normal		**Visible** In normal layout, elements are visible and clickable.
CSS display:none		**Invisible** The element is not included in the page layout at all.
CSS visibility:hidden		**Invisible** Space for the element is included in the layout but the element itself is not rendered.
Outside page (e.g. using absolute positioning)		**Invisible** Although the element is in the DOM a user cannot reach it.
Outside viewport		**Visible** Although the element is not shown on screen it is still part of the page and can be clicked after scrolling.
Transparent		**Visible** The element is still on the page and can be clicked even if it cannot be seen.

Table 4.2: (continued)

Element type		Visibility to click events
Obscured		**Invisible** The element is covered so it can't be clicked.
Partially obscured		**Visible** Some area of the element is visible so it can be clicked.
Obscured by transparent element		**Invisible** Although the element is visible on the screen, it is covered by a transparent element and can't be clicked.
No size		**Invisible** If the element has no size there is nothing to click.
No size, with overflow		**Visible** Although the element has zero height the contents is visible and therefore clickable.[31]
No size, with child outside		**Visible** A click on the visible child can reach the parent via bubbling.

[31] The overflowing content could be child elements or even just a text node.

Table 4.3: Comparison of ArtForm, Crawljax and Artemis on 1000 synthetic form validation examples.

	ArtForm	Crawljax	Artemis
Analysis time for 1000 tests (s)	969	272 153	2 206
Average line coverage	88.2%	—	87.1%
Examples with 100% line coverage	10.4%	—	0.7%
Avg. no. iterations to first submit	1.86	40.95	43.29
Successful form submissions	31.4%	19.8%	17.7%

any JavaScript analysis. Artemis is a web application testing framework which uses feedback-directing testing [8], and is discussed in more detail in Section 4.11. In each iteration, Artemis generates a sequence of actions to test, as well as corresponding values to enter into form fields. Although it has no direct JavaScript analysis, it uses metrics like JavaScript line coverage to choose actions which appear most promising for new exploration in the next iteration.

Each tool (ArtForm, Crawljax and Artemis) is run for a maximum of 50 iterations. For ArtForm, an iteration is defined as in Algorithm 2, that is, each test with a new set of values is one iteration; for Crawljax, it is a full run of the tool, until no new state is discovered, and different iterations use different random seeds; for Artemis, an iteration is a pre-selected sequence of actions that is run in the browser. Note that iterations in Crawljax can take a large amount of time and involve numerous interactions with the browser, while they are comparatively short in ArtForm and Artemis.

ArtForm and Artemis each have a notion of resetting to a clean state and running a new test. Crawljax, by contrast, executes its actions in one continuous browser session, so there is no obvious breaking-point where individual interactions or tests can be separated.[32] This is why our notion of iterations allows Crawljax more power than the other tools. On the other hand, this notion gives Crawljax less marginal benefit when running more iterations; ArtForm and Artemis require many iterations to analyse a page, whereas Crawljax runs its whole analysis within each "iteration", so suffers more from diminishing returns. In any case, the limit of 50 is high enough that none of the tools had problems caused by the iteration limit.

The results are shown in Table 4.3. The table includes line coverage of the event-handling code, measured at the JavaScript source level, for all tools except Crawljax, which does not record this. The average line coverage when performing a single form filling with the default values is 82.0%, so this should be considered the baseline for line

[32]Crawljax does not *require* a continuous session; it can even split its crawling across multiple browser instances, for example; but it has no clearly defined "reset step" where the algorithm returns to a clean state and begins the exploration again with new inputs or actions.

Table 4.4: Comparison of Algorithm 2's dynamic reordering with statically-ordered concolic testing on 1000 synthetic form validation examples.

	Single-order	Alg. 2
Analysis time (s)	585	969
Average iterations to submit	3.04	1.86
Average line coverage	88.0%	88.2%
Examples with full line coverage	4.7%	10.4%
Successful form submissions	24.1%	31.4%
Cases with multiple traces explored	69.8%	46.8%
… their average line coverage	90.6%	95.5%
… of which full line coverage	6.7%	22.2%
… of which successful submissions	34.0%	66.2%

coverage (most of the lines are in the scaffolding for setting up the event handlers and calling the validation functions). The average number of iterations required to reach a page submission is taken as 50 if the tools reach the iteration limit before finding a submission. ArtForm scores well on this metric because the concolic testing stops once there are no more satisfiable branches, and does not run to the full 50 iterations, even when it cannot find a successful submission. Similarly, the running time for Crawljax and Artemis includes 50 iterations for every example, whereas for ArtForm, it only includes as many iterations as the concolic testing required. Note that as the validation rules are randomly generated, it is not guaranteed that the examples are really satisfiable; the proportion of successful form submissions can be compared between tools, but it is not possible to score 100%.

The results of course do not represent the full complexity of analysing real websites; these experiments focus only on the problem of analysing complex constraints. They indicate that ArtForm can deal with complex constraints that do *not* obey the restricted global state restrictions that our completeness result relies on, while also indicating that existing methods do not suffice for complex constraints.

Impact of dynamic reordering

We ran both the dynamic reordering algorithm from Section 4.2 and the single-order algorithm from Section 4.5 on the same collection of 1000 synthetic forms. For each configuration, we measured the line coverage as before, and whether we were able to generate a successful form submission. The results are shown in Table 4.4.

The big advantage of Algorithm 2 over a static ordering is pruning out explorations in one event handler which are known to lead to an abort action in a later handler. The algorithm is able to prove that certain unexplored paths must later lead to abort actions,

and therefore avoid wasted iterations. Trace execution is very expensive compared with the rest of the analysis, so this is a significant advantage.

An example of this can be seen in the second part of the table. Algorithm 2 only attempted to explore 46.9% of the examples, compared to 69.6% for the variant with a static ordering. The difference represents cases rejected by Algorithm 2 but where the static ordering algorithm spent time exploring branches in event handlers which would all eventually lead to an abort action in a later handler. In contrast, calls of Algorithm 2 to a constraint solver came back unsatisfiable, allowing the algorithm to ignore them. This explains the dramatic difference in the number of successful form submissions found in those examples where more than one trace was explored (34.0% for the static order and 75.3% for Algorithm 2). Intuitively, when Algorithm 2 *did* run a test, it was much more likely to find a terminating trace, having already pruned out many more of the aborting traces than the single-order algorithm.

Form validation demo examples

A more realistic benchmark is given by a set of demonstration-style examples of web form validation. We used JSFiddle [74], a code-sharing website for web-based code. JSFiddle is used by developers to easily test new techniques, to share solutions, and to host tutorial code. Since form-related examples are not easily isolated in JSFiddle itself, we performed a Google search for "JSFiddle form validation". We extracted 18 examples, after removing those which did not perform JavaScript form validation. The examples were modified where necessary to allow them to be run in the tools with common submission and error behaviour, and we manually confirmed that it is possible to submit each form. These test cases represent real-world forms—they include common JavaScript libraries and use complex constraints—but do not include the full complexity of real sites. There is no other content to contend with apart from the forms themselves. When selecting examples to test, we also excluded those using the Angular framework for form validation. ArtForm's symbolic tracing cannot currently track form inputs being read by Angular, so none of these examples could produce any useful results.

We again compared ArtForm with Crawljax and Artemis, and the results are shown in Table 4.5. Some examples required only non-empty results; Artemis still has difficulty handling some of these, since it does not fully simulate the user-level actions required for form filling. ArtForm has a modest gain over Crawljax, managing to solve several of the examples which require string comparisons. All three tools seemed to be limited by custom-built interface widgets on several of the examples.

The running time for ArtForm is dominated by a single example which uses string functions to parse an ID number, including a checksum. This example alone took took 30 minutes 45 seconds out of the total analysis time of 37 minutes 25 seconds for all 18 examples. This example requires solving many difficult constraints over both strings and integers. At one point characters from the ID are converted to integers, multiplied

Table 4.5: Comparison of ArtForm, Crawljax and Artemis on the 18 JSFiddle examples.

	ArtForm	Crawljax	Artemis
Analysis time for 18 forms (s)	2 245	6 509	319
Avg. no. iterations to first submit	26.17	33.67	45.94
Successful form submissions	61.1%	33.3%	11.1%

by a constant, converted back to strings to extract the digits, which are then converted to integers and summed; and the analysis must solve a final integer constraint on the cumulative total of these sums. As such, ArtForm does not find a submission within 50 iterations, and spends 99.6% of its running time making 181 calls to the solver. However, if left to run with no iteration limit, it is eventually able to find a valid ID number and successfully submit the form. In total, there were three examples which ran to the iteration limit, all of which can be solved with enough time. Including those examples would bring ArtForm's successful submissions to 77.8%.

Real-world form validation

We also tested ArtForm on a collection of real-world airline websites. Airline search forms were chosen because they include complex validation rules spread over several different fields. The eight examples presented here were selected as having a search form visible on the front page, client-side validation which prevents form submission, using relatively simple form widgets, and being relatively easy to localise for more consistent testing. For these experiments, ArtForm was restricted to only consider input fields within a manually-specified area covering the main form on each page.

The results of running ArtForm in the single-order mode are shown in Table 4.6. The corresponding results for the reordering mode are shown in Table 4.7. Finally, the different analysis features which were used on each site are shown in Table 4.8.

In this test run ArtForm only managed a successful form submission on one site, the Australian airline Rex, where it submitted the form 242 times in single-order mode and 83 times in the reordering mode. It is worth noting that the results are not totally stable run-to-run, and although we have attempted to localise the sites, some of them still depend on remote AJAX calls, and therefore their behaviour can still change as the original sites are updated. Of these eight localised sites, ArtForm has found success on four of them in prior runs. Even though a successful submission is difficult to discover, ArtForm is still able to achieve some significant exploration of the sites, covering many branches in the form-related code.

An interesting example is the archived website of AirTran Airways. The validation rules include that origin and destination must be entered, that there must be valid combinations of the number of infants, children and adults (e.g. not more infants than

Table 4.6: Single-order mode results on the airline examples.

Site	Running time (s)	Iterations	Distinct traces	Form submit	Constraints solved
airtran	2 138	316	229	N	1 634
cathaypacific	38	8	4	N	28
cheapflights	22	3	2	N	2
csa	44	10	10	N	25
easyjet	1 239	98	97	N	2 620
flightnetwork	6	1	1	N	0
goair	1 341	768	706	N	769
rex	9 349	2 081	1 501	Y	5 494

Table 4.7: Reordering mode results on the airline examples.

Site	Running time (s)	Iterations	Form submit	Actions explored	Unique orderings	Constraints solved
airtran	438	15	N	13	12	677
cathaypacific	961	162	N	12	2	470
cheapflights	17	3	N	12	1	2
csa	20	5	N	16	1	4
easyjet	750	81	N	12	4	1 419
flightnetwork	4 317	200	N	34	20	358
goair	3	1	N	15	1	7
rex	1 271	329	Y	15	18	695

Table 4.8: Analysis features used in single-order mode on the airline examples.

Site	Radio constraints	Select constraints	Dynamic select	Regex translated	Coercion optimisation
airtran	1 470	8 680	385	0	75
cathaypacific	0	28	0	115	0
cheapflights	0	0	0	3	0
csa	0	0	0	89	0
easyjet	0	7 122	196	2 460	7 096
flightnetwork	0	0	0	0	0
goair	0	2 260	1 445	0	0
rex	3 096	39 125	2 009	0	7 936

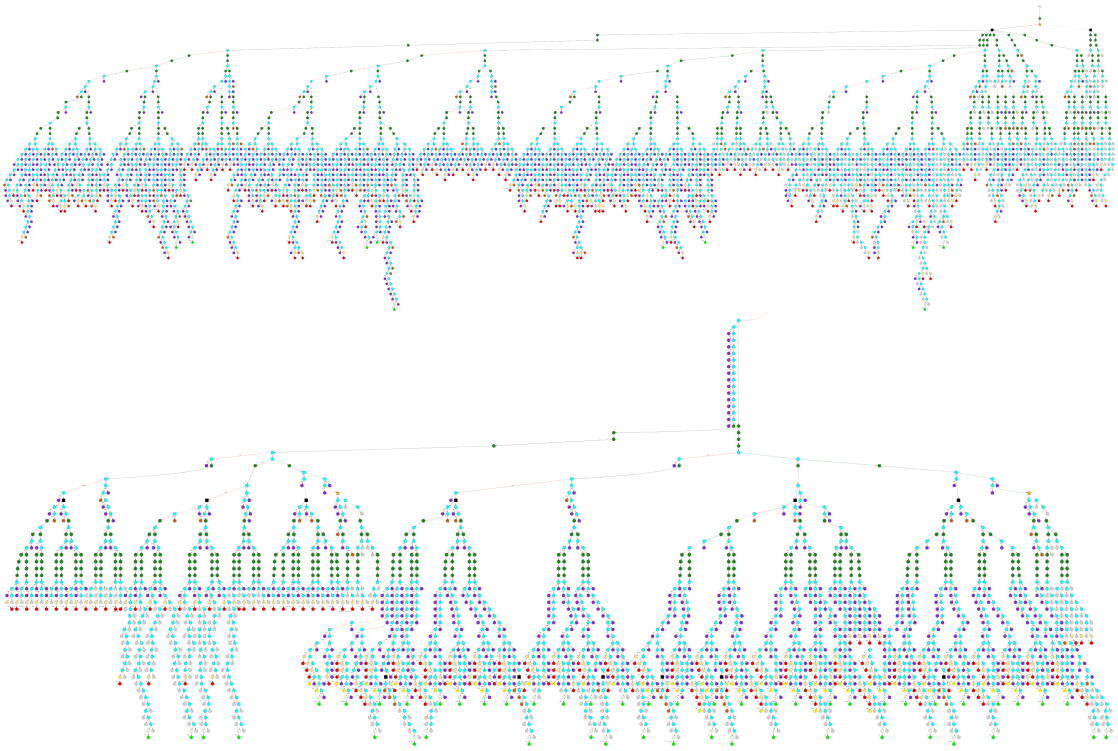


Figure 4.12: The concolic execution trees for local mirrors of the AirTran (top) and Rex (bottom) airline search forms, as tested by ArtForm in single-order mode. Only an excerpt of the Rex tree is shown, as it is much too wide to include in its entirety.

adults), and that the departure and return dates must be ordered correctly and be no earlier than today's date. Handling of the date and passenger constraints exercises the support for integer arithmetic in the solver. The AirTran site also makes use of dynamically-updated select boxes. When a departure airport is selected from the drop-down, the list of destinations is filtered to only show the permitted destinations. ArtForm (in single-order mode) is able to successfully track these dynamically-changing possible values, and select appropriate pairs of airports.

The Rex site has similar passenger restrictions, and also includes dynamically-updated possible values for the destination airports. ArtForm finds 242 distinct successful paths to submission, out of a total of 1501 total paths explored.

The concolic execution trees generated by ArtForm (in single-order mode) for these two examples are shown in Figure 4.12.

As expected, the reordering mode provided a great improvement over the static-ordering mode on some sites, but was limited on others. On the FlightNetwork site, for example, ArtForm did not find any symbolic branch which was suitable for exploration in single order mode, and therefore did not even make any attempts. In the action

dependency graph, all edges were pointing to actions which had already been executed. This made it a prime target for reordering, and the reordering mode was able to run for 200 iterations, using 20 different action orderings during its testing. ArtForm was able to find a fully-terminating trace (a trace which terminated in every action) for six of the eight examples in the reordering mode. However, these do not reflect form submissions; many of the sites do not use alert boxes when showing errors, so they are not marked as aborts in our traces. More sophisticated detection of rejected form inputs would be able to mark many of these traces as aborting based on changes to the DOM, for example. These experiments only considered alerts as abort actions, so as not to accidentally mis-classify any correct traces as aborting.

The amount of time spent solving constraints was comparable between the two modes, but in general the reordering mode uses fewer, more complex constraints. On average, the reordering mode's solver calls took nearly twice as long as those made by the single-order mode. Because loading the pages and executing JavaScript is so expensive, the time spent solving the constraints is only a small part of the overall analysis time, around 4.4% in single-order mode.

The experiments on these real-world airline sites are only anecdotal, partly due to the development version of WebKit the ArtForm prototype is based on, and partly due to the prevalence of custom-built widgets which ArtForm cannot currently analyse. Further experiments with a more modern browser engine and improved handling of custom form input fields will be needed to draw any firm quantitative conclusions.

4.10 Discussion

ArtForm makes significant progress toward the problem of exploring web forms. It can explore much of the form-related code on real-world sites, even if finding a successful form submission is still relatively rare.

ArtForm's approach has the advantage that it can generate not just single submissions but constraints which characterise all the valid submissions. This is particularly relevant for wrapper-generation. Clearly concolic testing alone cannot replace existing techniques for form filling and wrapper generation; in particular, techniques which apply domain knowledge will always be able to tackle different aspects of the problem. However, ArtForm's concolic testing does have the advantage of being domain-independent and not requiring any prior knowledge of the site being analysed. This makes ArtForm's approach more generally applicable than more specialised tools which require per-application customisation. ArtForm's advice mode is a practical way to combine concolic testing with more domain-focused analysis tools and techniques in cases where this is useful.

The current version of ArtForm does not explicitly generate a logical representation of a form's integrity constraints, which was one of our initial goals. Currently they are

only represented implicitly in the symbolic path tree. The overall constraint for a valid form submission can be easily "read off" from the tree, by taking the union of those symbolic paths which are classified as successful. It would be possible to use BDDs (binary decision diagrams) to automatically generate a simplified form of the overall constraints [5], but this has not yet been implemented.

In general, our approach has been to add features to ArtForm in response to issues discovered while testing on real websites. It is not possible to fully analyse the whole of JavaScript and the DOM APIs, especially given how quickly some web technologies can change. We instead chose the most important and most common patterns from our example sites and extended the analysis to support them. Our evaluation shows that the features we implement are already enough for a useful analysis of many forms. Obviously in future it will be beneficial to continue extending this set of supported features and patterns to support more and more real sites.

One prominent limitation is custom form controls (date pickers, auto-completed fields, and so on), which limit ArtForm's ability to discover a form submission on many sites. ArtForm has no way to identify and interact with these fields—which might be solved with some form-specific domain knowledge—but even if it could, it has no way to model the inputs symbolically. Currently there is no way, for example, that a button click in a date picker can be linked with the corresponding interpreter value representing the selected day of the month. The interpreter value has to be recognised as being an input value, and any new values generated by the solver need to be linked to clicking a *different* button in the date picker on a subsequent iteration in order to input the new date. It is not clear how this can be achieved in general.

ArtForm is built on top of the WebKit browser engine. As described in the previous chapter, this is critical to our approach, allowing us to run on real sites and take advantage of WebKit's JavaScript-to-bytecode compilation to simplify our interpreter and the generated constraints. However, it also limits how easily ArtForm can keep up with new developments in web technologies. Because ArtForm heavily instruments many parts of WebKit's JSC interpreter (as well as other parts of WebKit to a lesser extent) it is tied to a fixed WebKit version. Updating the version of WebKit is a significant undertaking, as any changes to its internal operation may require reworking much of our own instrumentation. For the moment this is not a big limitation, but as web technology develops, the browser engine will become outdated. For example, our version of WebKit does not support the WebSocket or web worker APIs, which provide persistent client-server communications and parallel execution, respectively. Although they are not yet widely used on general sites, lack of support for these features means ArtForm is already unable to analyse some JavaScript-based games or real-time messaging applications, which simply do not run in our browser. Another issue is stability; ArtForm is not as reliable as we would hope, especially when analysing especially JavaScript-heavy pages. The crashes come from WebKit, partly because we use an older, development version of WebKit, and seemingly partly because of our own instrumentation. One of Jalangi's

biggest benefits is that its analysis runs in JavaScript itself, so the same analysis can easily be re-run in different browsers and can easily keep up to date with new web technologies as they are developed.

A final implementation-level limitation, which could certainly be improved with further work, is ArtForm's input simulation. Currently, ArtForm supports simulating form-filling either by injecting the new value and calling the field's change handler, or by creating and triggering a sequence of JavaScript-level events to simulate a real form filling (as described in Section 4.3). Neither of these methods provides a perfect simulation of a real user with a real web browser, and a few sites' validation code is not triggered by our simulations. The next step would be to issue GUI-level mouse and keyboard events to WebKit itself, so the browser sees no difference between simulated and real form filling. As a trial, we have implemented GUI-level mouse click events, but even this has limitations: the target element needs to be scrolled into view, the clickable area must be determined, and so on.

Some further complexities to fully simulating form filling cannot be resolved by improving the simulation. Different sites expect different user behaviour, and if the form filling is simulated perfectly, then this must be accounted for. For example, many auto-complete search fields pop up a list of suggestions while typing, but if the user moves away from the field, triggering the blur event, the suggestions are hidden again, preventing the analysis from clicking on them. However, many other sites *only* apply their validation rules once the user has finished filling each field and the blur event is received. Whether or not the blur event should be included in a given form-filling action depends on the type of field being filled, and what the subsequent actions will be. Another example is key-by-key input simulation. This is often useful for sites which auto-complete or validate the input as the user types, but it creates a problem for concolic testing where the length of an input string affects the code being executed and the trace recorded. This problem could be resolved by introducing pseudo-branches checking the length of the input before each key event, at the cost of artificially inflating the search space for the concolic testing.

ArtForm's different analysis modes are based on the same core functionality, and share much of their underlying implementation, but are separate modes which are invoked individually and cannot easily share information. Being able to better integrate the different modes would be a big benefit. The dynamic-reordering algorithm described in Section 4.2 and the fixed-order analysis described in Section 4.5 are run as separate modes, for example. There are trade-offs to each mode, with some sites being analysed more effectively by each. Currently there is no way for the modes to be combined or merged to get results representing the best of both; the analysis must simply be run twice to compare the two modes.

Another part of the analysis which would benefit from closer integration is the event delegation analysis described in Section 4.8. Currently, this analysis is invoked separately from the main concolic testing mode for analysing form validation. Ideally,

they could be combined so that a single invocation could determine the interesting form fields and buttons on the page using the delegation and visibility analysis, interleaved with determining interesting values and orderings to fill the fields using the main form-filling analysis. There is no technical reason the modes could not be merged and work together; it is simply a matter of designing exactly how the interaction would work and implementation effort.

The delegation analysis itself is still an ongoing work-in-progress. Although it can support many real-world constraints, and reach interesting code which an event-handler based tool could never access, it is still limited. For example, different versions of jQuery implement their delegation support in slightly differing ways, leading to slightly different results in our analysis. The delegation analysis is sensitive to some of these changes and therefore is not able to detect delegated events in all versions of jQuery. Further work, helped by the manual tracing and debugging tools described in Section 4.6, will be able to improve the robustness of this analysis.

Our work on ArtForm has applications in other areas of web application testing. The extension of concolic testing to cover web-based JavaScript alone could be directly useful for web application developers. Many of the new contributions in ArtForm relate to modelling which actions or values a real user could provide. This analysis can be applied in other contexts to check, for example, which error states, bugs or warnings are reachable *via normal user actions*, as opposed to which are reachable in theory via some code path which the user may or may not ever be able to exercise.

4.11 Related work

This section presents some related work on traditional web analysis, focusing on web crawlers, form understanding and information extraction. We will then look at two specific tools, Artemis and DIADEM, which my own work builds on. Finally, we then discuss some existing approaches to analysing other event-driven systems.

Web analysis and information extraction

Automated analysis of interactive data-driven websites has been an active research topic since the advent of the web. Web information extraction tools aim to capture information about a website's interface, for example separating out distinct forms, identifying the input fields and their labels, and transforming the results of a query into structured records. Often the output of these tools is a description of the website which shows how data should be found and extracted—a data extraction wrapper.

Most of these tools rely on the regularities in the web page structure to drive the analysis. For example, to transform the results of a query into structured records, a tool would look for repeated patterns within the output page representing individual output records. In particular, these tools are typically unaware of the client-side JavaScript

code which forms part of the website's implementation, although browser-based tools will still execute it.

Traditional web crawlers cannot search through web forms; therefore the data behind them is known as the deep web—the part of the web hidden underneath access restrictions (search forms) which the automated tools cannot pass. Several crawlers have been developed which specifically target deep-web data, including Google's deep web crawler [97], Metaquerier [140], and DeepPeep [14].

Form understanding is a critical component of deep-web data extraction, and typically involves form identification and labelling, form filling, and result-page analysis. Given the flexibility of HTML, it is sometimes difficult even to determine which parts of a page constitute a search form. This problem is known as entry-point identification [12]. Once a form is identified, along with its fields and buttons, the fields are labelled—or *annotated*—with their names and types [34, 43, 61, 107, 139]. Next, suitable values are chosen to fill the form [13, 73, 76, 89, 95, 131, 137]. Once the form is submitted, the result page is analysed and annotated so the target data can be extracted [31, 96, 143]. Techniques from each of these areas can then be bought together into full-featured wrapper generation and data extraction tools, such as DIADEM (discussed below).

Dynamically updated and AJAX-powered websites make web crawling significantly more challenging. Instead of several pages which are statically linked together, each of which can be fetched and examined individually, many modern web pages now load their content dynamically. An AJAX-aware crawler needs to be able to click buttons, fill fields, and extract data from a working version of the page—that is, in a real web browser. There is often no way to skip straight to the desired data without interacting with the web application as a normal user would.

AJAX-aware crawlers, such as Crawljax [100], AJAX Crawl [35], and others [33, 99, 112], typically build a state machine to represent the state of the page being searched. Each transition represents an action taken by the crawler (such as clicking a button) and each state represents a different page state. The tools must include techniques for merging states which are considered similar to each other, to avoid trivial change to the page state causing a huge blow-up in the state graph size.

Because of the sheer number of actions and states to be modelled, this type of exploration is very expensive. The state space is made even larger when forms are considered, and choosing appropriate values to fill is difficult in itself, so most existing AJAX crawlers do not even attempt to search through web forms, other than filling random values when performing other actions.

Another application of AJAX-based interface analysis is for web accessibility in the area of human-computer interaction. Dynamic pages and web applications using custom interface widgets can be difficult for accessibility tools (such as screen readers) to understand [60]. They face the similar problems to the automated crawling data-

extraction tools discussed above. DynaRIA aims to aid understanding of the structure and behaviour of dynamic web applications, with a view to improving accessibility and understandability [2]. Other work focuses on the problem of identifying and classifying custom widgets which are made up from many HTML elements and JavaScript code so they can be better understood by assistive technologies [28]. In general, solving these problems requires connecting the high-level user-facing interactions with the low-level JavaScript events and HTML elements which implement them [1]. The low-level information about a web page can further be applied to optimise page loading times for users, as in Polaris [106]. Polaris analyses the loading behaviour of different assets on a web page to work out which can be loaded in parallel, and which must be blocked until other assets are loaded.

Different analysis tools use different methods for interfacing with a web browser. Many web-related tools, including Crawljax, DIADEM and ProFoUnd use Selenium WebDriver [120]. WebDriver gives control of a web browser, allowing the client tool to load pages, inspect them, and perform actions. The actions simulate real user interactions, such as filling form fields, or hovering over and clicking on elements. WebDriver uses a standard third-party browser, complete with JavaScript execution and plugins, which means it accurately reflects how a real user sees the page. The trade-off for this simplicity and power is that the API cannot give full low-level control of the browser's operation. This makes some types of lower-level information about the page or its JavaScript more difficult to obtain than with tools which bundle their own instrumented or custom browsers.

DIADEM partially gets around this problem by installing a custom Firefox extension in its WebDriver-controlled browser, which allows extra control and reporting which WebDriver would not normally provide. This permits the extra features required for DIADEM, but in general still doesn't provide *full* control (such as over JavaScript execution, timers, or AJAX requests, for example). OpenWPM is a web privacy analytics framework, which collects data about how real-world websites track users and handle user data [38]. It combines WebDriver with a proxy, mitmproxy [101], which also allows it to modify and instrument the pages before they arrive at the browser.

While WebDriver is very common, many tools take other approaches. SymJS uses HtmlUnit and Rhino, Java-based implementations of a browser engine and JavaScript interpreter, respectively, rather than a traditional browser [88]. Jalangi relies on JavaScript source code instrumentation (which can be done automatically via mitmproxy in the latest version [66]), and does not include any browser or page rendering infrastructure itself. Artemis takes yet another approach, and is built on a custom-instrumented version of the WebKit browser engine, so information from the browser can be accessed and manipulated directly [8]. Each of these approaches provides a different trade-off between the features supported, simplicity, and flexibility [38].

Artemis

Artemis is an automated testing tool for JavaScript web applications [8]. It includes an instrumented version of the WebKit browser to inspect the JavaScript event handlers registered on a page (such as clicks, timers, form inputs and AJAX events). Sequences of these events are executed to explore new parts of the application and find any errors.

It uses a feedback-directed testing algorithm, where new event sequences are selected for testing based on the results of previous iterations. Four strategies can be used to prioritise the search:

- Testing each sequence as it is discovered,
- Choosing sequences at random,
- Choosing sequences which include new events and increase coverage, and
- Choosing events with high dependence on previous events in the sequence.

Dependence is measured by an analysis of the read/write sets for JavaScript object properties. It is designed to predict which event handlers have a high likelihood of exploring new execution paths—and therefore covering new code—when executed after certain other handlers. Artemis explores form-related code by generating random input values or taking static strings from the page's JavaScript code.

Our concolic testing platform reuses and extends the instrumented browser and base functionality from Artemis, without its feedback-directed testing infrastructure.

DIADEM

DIADEM is a web information extraction framework which automatically generates data-extraction wrappers given a URL as input [44]. Domain knowledge, patterns in the DOM and the visual structure of a page are exploited to identify forms, fill them in sensibly, identify the result pages, and finally generate a wrapper for that particular site, allowing automatic extraction of the data into a structured database. DIADEM has a mechanism for encoding domain knowledge in Datalog rules, and thus can be customised for different domains. This domain knowledge is very powerful, making it simple to apply useful heuristics and "intelligent" reasoning to classify different parts of the page and determine how to interact with them or extract data.

We have been able to use some of DIADEM's infrastructure to identify form submission buttons as a starting point for ArtForm's initial page analysis.

DIADEM's form analysis module is called OPAL [43]. It matches form fields with their labels, and uses these to infer how they should be filled. For example minimum and maximum price fields are grouped and their relationship is labelled. Example strings from the page or the DOM can be used as inputs, and known field types such as cities or post codes can be chosen from a database of domain knowledge.

To maximise the number of results returned from each submission OPAL understands how to make the most general queries, for example by setting the minimum and

maximum price to opposite extremes. More restricted queries can be used when the number of returned results is limited, so the result pages can be systematically crawled.

OXPath is a language for specifying data extraction wrappers, and a tool which executes these wrappers to extract the data [45]. OXPath is the language used for the wrappers generated by DIADEM. It extends the XPath DOM query language which adds support for interacting with a page. For example, extracting certain data might involve loading a page, clicking a button (specified by standard XPath) to display a form, filling certain values and then (using another XPath query) finding the target elements in the result page. Because OXPath is based on XPath, and is executed in a running web browser, the selectors describing which elements to interact with or extract are applicable even on dynamic and interactive pages. The wrappers generated by DIADEM are robust in the sense that they are not sensitive to adverts or other unimportant changes to the page between different executions.

Analysing event-driven systems

Browser-based JavaScript uses an event-driven model of execution—different event handlers are called in response to user or system events. There are many tools which analyse other types of event-driven systems. These tools face many of the same problems to those in web interface analysis, and often at much larger scales. The techniques they employ are therefore relevant to the analysis of web-based JavaScript.

A common example, which shares many similarities with web-based JavaScript analysis, is the analysis of Android applications. In Android apps, both system- and user-level events (taps, swipes, button clicks, and so on) trigger application code which responds to them. As the user interacts with the app, they change the state of the application, which affects the subsequent events which are now possible. Searching for sequences of events which can be used to reach or explore certain functionality, or to uncover certain classes of bugs has been a common and important area of investigation [4, 65, 68]. My work differs from these approaches by being able to apply known or expected properties of web pages and the possible interactions a user can make to reduce the amount of inter-event dependence which needs to be analysed.

My work on ArtForm, including the new dynamic event reordering algorithm, was presented at ICWE 2018 [125]. A screencast from our demonstration paper [124], showing the manual testing tools, the fully automatic concolic testing, and the advice server mode, is available at:

http://www.cs.ox.ac.uk/projects/ArtForm/demo/

The evaluation data for ArtForm, including the synthetic form generator, the generated forms, the JSFiddle examples, and localised versions of four of the tested airline websites, is available on GitHub [42].

5 FastWrap: Browserless data extraction

This chapter departs from web form analysis, and tackles a different problem in web information extraction, that of optimising data extraction wrappers once they have been generated. As before, we are able to exploit low-level details of a page's implementation to understand the user-level actions taken by a wrapper.

Most modern web data extraction wrappers use an embedded browser to render web pages and to simulate user actions. The wrappers are therefore relatively expensive to execute, in terms of time and network traffic. In contrast, it is much more efficient to use a "browserless" wrapper which makes direct HTTP requests to a web server, and takes the desired data directly from the raw responses. However, creating and maintaining browserless wrappers is difficult, requiring a careful analysis of the target website. This process is too slow and labour intensive to be feasible at a large scale.

We have developed an approach, called FastWrap, to automatically translate browser-based wrappers into equivalent browserless ones. This chapter presents our approach, along with an evaluation of our prototype tool on a suite of real-world and realistic data extraction wrappers.

5.1 Motivation and goals

Most modern data extraction tools use visual clues and user-level interactions in the wrapper generation process and relying on graphical information such as the distance between two elements on the page. For example, there is a rule in the DIADEM knowledge base which says, in simplified form, "the closest text chunk below or above an input field on a web page is (with high probability and in absence of better information) the label for this field". These types of "visual" rules make up for the fact that the page's HTML is only semi-structured, and there is no requirement for the DOM to encode the user-level page structure in any machine-accessible way. Visual clues and "page geography" information drastically simplify wrapper generation. Defining, modifying, and testing wrappers in a GUI tool means that wrapper designers can avoid deciphering the HTML code of each target page and writing scripts to decode it. This way, generating and testing a new wrapper for a moderately complex site typically takes only a few hours, instead of days using the traditional method. Moreover, the use of visual clues can lead to more precise and more robust wrappers.

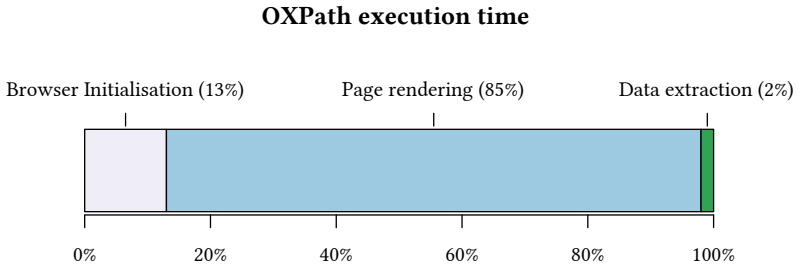


Figure 5.1: The time spent on each part of OXPath execution [45].

The costs of visual wrappers

While these visual-clue based wrappers, referred to as *visual wrappers*, are a great benefit for *wrapper design*, they come with a significant penalty at execution time: they are executed by loading and rendering the page in an embedded web browser, which is very expensive in terms of both time and resources used. This rendering consists of (a) parsing the web page and building a DOM tree, (b) applying the CSS rules, and (c) executing JavaScript (typically the most computationally expensive part, often having wide-ranging effects on the page's DOM). Note that in this context, rendering does not necessarily include displaying the resulting page on a screen—in fact many tools run in a "headless mode" with no GUI. Even with a headless browser, all the significant costs are still required, as the page has to be prepared *as if* it were to be displayed on screen, so as to provide an accurate simulation of a normal browser.

Based on data from OXPath, Figure 5.1 shows a coarse analysis of the relative costs of each phase of executing a visual wrapper: the browser initialisation, the page loading and rendering, and the OXPath execution and data extraction itself. Initialisation and rendering jointly require around 50 times more time than the actual data extraction. For visual-clue based wrappers in other languages and tools than OXPath, the proportions will be similar. Note that in this data the time required to request and download the data is included in the page rendering segment. A browserless wrapper still needs to perform some of these downloads, and will still have some initialisation costs, so it is not realistic to expect the full 50 times speedup. This is merely the optimistic goal; or an informal upper-bound.

When data extraction is done at scale, the overhead of executing wrappers in a web browser becomes very expensive. Many companies run hundreds of wrappers continually, so the computational costs of wrapper execution have a significant real-world impact. For some applications, the wrappers also need to be run in real-time (such as to check flight prices after a user's search, or to gather data for real-time options trading). If these wrappers are too slow, it can directly affect the end users, sometimes to the point of making it infeasible to use the wrappers on-the-fly at all.

HTTP wrappers

To avoid the high runtime-cost of browser-based visual wrappers, many organisations replace the most heavily-used wrappers with hand-programmed alternatives which interact directly with the remote web server, without a browser. We refer to these wrappers as *HTTP wrappers*, as they work at the level of HTTP requests and responses. HTTP wrappers send similar requests to the remote server to those a browser would use, but analyse the responses without rendering them in a browser. The desired data can often be extracted directly from the raw content (for example, using an XPath expression if the content is HTML-formatted). As well as avoiding rendering costs, HTTP wrappers also usually make many fewer HTTP requests. Only those requests which are really necessary to obtain the desired data are sent, and the majority of requests that a browser would normally make to load irrelevant images, fonts, CSS style sheets, JavaScript code, text, and adverts can be ignored.

In our experiments, we found that HTTP wrappers are on average 23.8 times faster than the original visual wrappers they were derived from, while extracting the same data. In terms of network traffic, 96.8% of the data transferred by visual wrappers (which already ignore images) was found to be unrelated to the desired data and could be elided from an HTTP wrapper.

To build HTTP wrappers, a programmer must initially perform the corresponding user interactions through a browser and analyse the HTTP traffic between the browser and server. This is typically a manual, labour-intensive process, using the browser's integrated web developer tools, and requires some specialist knowledge of web development and web page analysis. Once they have understood the browser-server interaction, and after they have identified how the relevant data flows through the application, they write a parametrised HTTP wrapper that simulates the browser-server interaction. For each set of input parameters, the wrapper obtains and returns the desired output data from the remote server without any rendering. After this work, HTTP wrappers are often less robust than similar visual-clue based wrappers; they are typically more sensitive to changes in the structure of the target site, requiring more frequent and more complex maintenance to keep them running.

Generating HTTP wrappers from existing visual wrappers

Both visual and HTTP wrappers have their advantages and drawbacks. A tool which could combine the ease-of-creation of visual wrappers with the efficient execution of HTTP wrappers would be very valuable. Such a tool would gain the advantages of each approach, while avoiding the main drawbacks of both.

Our approach to this problem is to automatically transform browser-based visual wrappers into browserless HTTP wrappers which extract the same data. The visual wrappers can still be generated easily with existing tools—either automatically, as in DIADEM, or via user-assisted GUI tools such as Visual OXPath [81]. Then the

automatically transformed HTTP wrapper can be used to avoid the run-time costs of executing the visual wrapper directly. Our implementation and experiments have demonstrated that this transformation is feasible and effective.

The transformation must of course be "intelligent" and produce HTTP wrappers which only perform the necessary steps and suppress useless requests which load pictures, adverts, and other irrelevant data. Note that the robustness of the generated HTTP wrappers is no worse than that of the original visual wrapper: if there is a change to the page which invalidates the HTTP wrapper but not the visual wrapper, then the HTTP wrapper can simply be re-generated, assuming of course that the changes to the site are not so far-reaching that the transformation now fails. Making the transformation itself robust enough to handle a wide variety of possible sites and site changes is a key research challenge.

User-level actions and their HTTP-level implementation

As with the form analysis discussed in previous chapters, visual wrappers targeting complex web applications must simulate specific sequences of user interactions to reach certain target web pages or states in a web application. For example, the wrapper may need to fill and submit a web form to reach a result page, or click certain buttons to load detailed product information via an AJAX request. The wrappers also include input and output instructions, for example that a certain form field takes a location as input and that a particular part of the search result page is a phone number to be extracted.

Figure 5.2 shows an example of the actions a visual wrapper might execute to extract data from the "Find a Retailer" search interface at vauxhall.co.uk. The actions executed are as follows: (1) entering the search string using the input data "Preston", (2) selecting a suggestion from the autocomplete, (3) clicking the search button, which displays a list of corresponding dealerships, (4) clicking a link to a more detailed view which shows the target output data, and finally (5) extracting the target data, which is then returned by the wrapper.

There are two levels of interaction to consider: the user-level interaction with the browser, and the HTTP-based communication between the browser and the server. The simulated user actions listed above trigger JavaScript code, which itself makes HTTP request-response exchanges (*HTTP interactions*) with the server and parses and displays the results. In our example, when an input is filled into the search form, the field's change handler makes an AJAX request to the server with the search string. The server returns a JSON object which is converted by the client-side JavaScript into an HTML snippet to populate the autocomplete list. These interactions are recorded as *HTTP traces*; the logs of HTTP interactions performed during each test, along with the corresponding input and output data.

Our goal is to eliminate the first level of interaction, along with the browser itself, and conduct web data extraction exclusively using HTTP interactions. A secondary goal

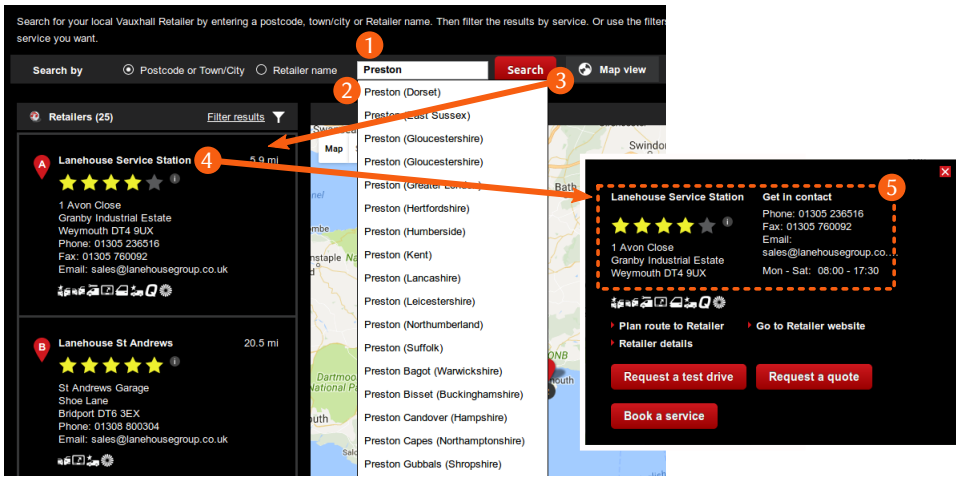


Figure 5.2: The five-steps a visual wrapper takes to extract dealership data from the Vauxhall site, in this case using the search input “Preston”.

is to minimise execution cost of these HTTP-based wrappers, measured by the number of HTTP interactions, time, and network usage required to extract the target data. A tool which could simply replay the same HTTP interactions that a browser-based wrapper would have made would fulfil the first goal—already a significant gain—but would still be very wasteful.

Project contributions

To our knowledge, this is the first attempt to automatically transform visual wrappers into HTTP wrappers. This is not surprising, given that it is a difficult problem, both theoretically and technically. The main contributions of this work are as follows:

- *The presentation of our FastWrap algorithm, and its implementation.* We have developed an algorithm to analyse a given visual wrapper and produce, where possible, an HTTP wrapper which extracts the same target data. The algorithm is based on the concept of a *dependency graph*, showing how different data values flow through the website’s multiple HTTP interactions. We present both the high-level approach, as well as a concrete algorithm.

- *A description of the challenges and limitations of the approach.* To make the transformation feasible, while still being useful and effective, we must carefully set the scope of the wrappers we aim to handle. In particular, there are certain classes of sites which our current prototype cannot support, and we will discuss how and why those limitations were chosen.

- *The evaluation of FastWrap on real websites.* Our implementation of the new algorithm is tested on real sites, using both real-world and hand-made OXPath

wrappers as input. This proves the feasibility of our approach for large-scale, general-purpose wrapper transformation.

- *An analysis of problematic cases and suggested solutions.* By analysing the tool's performance on real sites, we are able to make some observations about where the current prototype is limited, and suggestions on how they could be addressed. Some of the problems are purely technical, requiring more implementation work, while others are significant research challenges in their own right.

5.2 The FastWrap approach

This section introduces FastWrap, our approach to automatically generating HTTP wrappers. The algorithm takes as input a visual wrapper, along with examples of its input parameters, and produces as output a corresponding browserless HTTP wrapper. The output wrapper extracts the same data as the input wrapper by directly interacting with the website's servers and does not require loading and rendering the web page.

FastWrap works by executing the input wrapper with the given input parameters to collect the sequence of request-response exchanges between the input wrapper and the web server. The individual HTTP interactions from these recorded traces can be thought of as forming a dependency graph: each interaction requires certain input parameters, and once executed with those inputs, provides certain corresponding output parameters. Once we have identified the target data which we need to extract, the algorithm explores this dependency graph in order to satisfy all the input parameter requirements of the interaction which returns the target data.

The output wrappers are generated solely based on analysis of the HTTP traces and some on-the-fly tests against the remote server. They do not require any live interaction with the visual wrapper tool recording the traces. This means our approach is applicable to a wide range of visual wrapper formats and tools, and is not tied to any particular system.

Algorithm description

We will now give a more thorough description of the algorithm, including formal descriptions of the main objects it uses: concrete and abstract HTTP interactions, the dependency graph, and HTTP wrappers. The full pseudo-code for FastWrap's algorithm is given at the end, in Algorithm 3.

Step 1. *Execute the input wrapper on the given sample input parameters and record the corresponding HTTP trace for each run, along with the input parameters and output data. In the special case where input wrapper does not take input, simply execute the input wrapper a single time.*

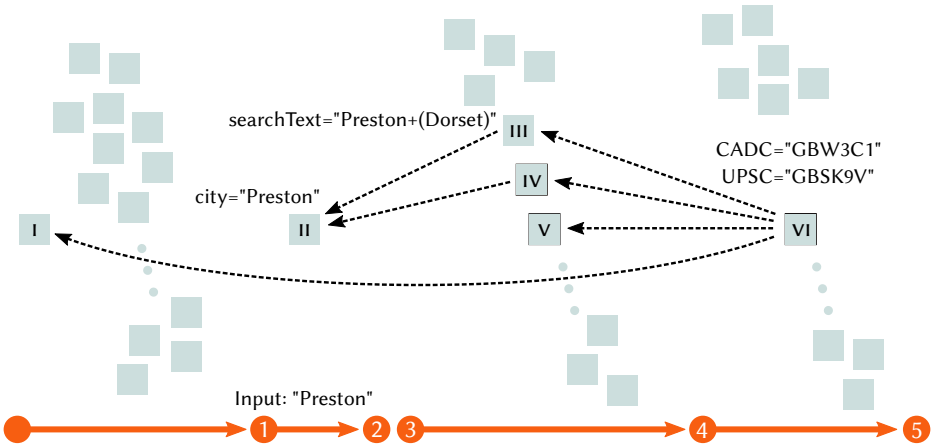


Figure 5.3: An example of a single HTTP trace for the Vauxhall wrapper, shown alongside the corresponding wrapper actions from Figure 5.2. Each block represents an HTTP interaction.

An HTTP trace for the Vauxhall site with input "Preston" is illustrated in Figure 5.3. In the diagram, the HTTP interactions are grouped according to which step of the input wrapper triggered them. For example, interaction I is the initial page load, and the group of interactions immediately following it (before the first user-level action) are the assets downloaded while rendering the page. Interaction II is the AJAX call to populate the autocomplete drop-down. The boxed interactions IV, V and VI contain the target output data in their responses. However, as we shall see later, interaction IV is not reliably repeatable, and FastWrap is unable to successfully extract the data from interaction V, leaving interaction VI as the only viable source of the target data.

The dashed lines show matching parameter values between different interactions. For example, the string "GBW31C" appears in the request parameters for interaction VI, and in the response bodies of interactions I, III, IV and V. These matches are not part of the HTTP trace itself, but are computed and used later in the algorithm, and are included here as an illustration of the relevant parts of the data-flow.

Definition. An HTTP interaction is a tuple (U, V, P, B), representing an HTTP request-response exchange, where:

- U is the request URL template: the URL without its query string and with values of segments split by "/", removed);
- V is the HTTP verb, "GET" or "POST";
- P is a set of HTTP request parameters, each of which are pairs $p_i = (n_i, v_i)$ specifying parameter names and values. A *parameter name* n_i is a pair (o_i, s_i) of an *origin* and an *origin-specific name*. The origin describes where in the request the parameter comes from, and is one of "header", "path", "query", or "post". This disambiguates parameters with the same name in different parts of the request.

- *B* is the body of the response, and is labelled with one of the types "html", "json", "jsonp", or "xml".

An HTTP trace is a sequence of HTTP interactions, along with their associated input and output data from the visual wrapper. □

Step 2. *Analyse the traces in order to identify those HTTP interactions that contain the target records. These are called the target interactions. Initially, the target records are the output records discovered in Step 1. In subsequent iterations, they are provided by the recursive call in Step 4.7.*

HTTP interactions which contain the target data are shown boxed in Figure 5.3. In this case there are three: interactions IV, V, and VI.

The traces are analysed "backwards", starting from the target data and working back towards the input parameters. In subsequent iterations, Step 2 will analyse interactions I, II, III, IV and V. Using this method, only interactions which actually satisfy some dependency which contributes to the target interactions are considered, and the vast majority of totally irrelevant interactions (such as adverts, CSS files, and so on; shown without labels in the figure) can be ignored. In Figure 5.3 the HTTP interactions without labels are irrelevant in this sense; they do not contribute any data which is used (even indirectly) by any of the target interactions. Therefore, the algorithm will not consider them further.

Step 3. *Group the target interactions from different traces into equivalence classes.*

Definition. Two interactions are *equivalent* if they have the same URL template, the same HTTP verb, the same set of parameter names, and the same response content type. The parameter values and response content itself may vary. □

For example, Figure 5.3 shows a single trace for input "Preston"; the interaction marked VI takes input parameters "CADC" and "UPSC", and provides the parameters "name" and "opening hours" as output. In the other test executions with other input data, there will be corresponding interactions VI′, VI″, and so on, which together form the equivalence class [VI].

The grouping appears simple in our example, but it can sometimes be complex in reality. The traces for different inputs may not record exactly the same interactions, for example, so the traces may not correspond exactly.

Step 4. *For each group of of target interactions from Step 3, perform Steps 4.1–4.8.*

Note that this iteration will run until we find a suitable wrapper which extracts the target data (i.e., a wrapper which can successfully execute any one of the target interactions), at which point the loop will be broken and the wrapper returned. If the loop terminates without returning a wrapper, then return "none".

Step 4.1. *Generate a* data selector *to extract the required data (which we know is present from Step 2) from the HTTP response of each interaction in the group. If a data selector cannot be generated, then skip this group and return to Step 4.*

A data selector is an expression (for example an XPath expression) which describes exactly how the target data is to be extracted from the HTTP response body. For tree-structured data, like HTML, XML or JSON, FastWrap searches through the structure to find the path which leads to a given target data item. The paths for each data item in a trace, and across the different traces in a group, are then combined to generate a single selector capable of extracting all the target data. For example, if the three data items for a parameter are found at the XPaths /ul[1]/li[1]/a, /ul[1]/li[2]/a and /ul[1]/li[3]/a, then these three expressions can be abstracted to the final selector /ul[1]/li/a.

It is often the case that any single parameter can be found in multiple groups. For example, interaction VI requires the parameters "CADC" and "UPSC", which can be obtained from any of the interactions I, III, IV or V. However, interactions I and V both contain a list of all possible values for "CADC", and it is impossible to generate a selector which can choose the correct value across different traces. Thus, no working data selector can be generated for [I] or [V]. A selector *can* be successfully generated for both [III] and [IV]. The response type for interactions in [VI] is "html", so an XPath-based data selector is generated. In other cases, other selector types would be used. Sometimes simple transformations of the extracted data are required, in which case they are recognised by our algorithm and applied.

Step 4.2. *Ensure that all HTTP interactions from the equivalence class are repeatable. If not, skip this group and return to Step 4.*

Each interaction is re-executed outside the context of the interactive browser session to make sure that it can be replayed and the selector can extract the target data. This shows that FastWrap has captured all the information necessary to successfully replay each HTTP interaction. FastWrap may fail to replay an interaction successfully if it includes some session information which is no longer valid, or requires some parameters which were not identified. For example, the current prototype does not support parameters in cookies.

Step 4.3. *Classify each parameter of the interactions in the group as either constant or variable. Constant parameters have the same value across all interactions in the group, while variable parameters can change from trace to trace.*

In our example, both "CADC" and "UPSC" are variables, as they occur with different values in different traces. There are many constant parameters in each request, such as "x-brand", which is always "vauxhall", or "x-language", which is always "en". As these parameters never vary, they do not need to be included in the algorithm's search of the dependency graph.

Step 4.4. *Use query probing to remove variable parameters from the interactions if they are not necessary to retrieve the target data.*

Not all parameters are directly related to the data-flow required for the wrapper being analysed. Some simply contain tracking or session data, for example, and do not

affect the results. By removing these parameters we reduce the number of dependencies required to invoke each HTTP interaction, and therefore reduce the search space for the algorithm.

Step 4.5. *Create an abstract HTTP interaction from the equivalence class. An abstract interaction is an HTTP request with placeholders replacing each of the variable parameters. These abstract HTTP interactions will be the nodes in our dependency graph.*

Definition. An *abstract HTTP interaction* is a template for a new HTTP interaction, represented as a tuple (U, V, N, C, r), where:

- U and V are the URL template and HTTP verb, respectively, as for concrete HTTP interactions;
- N is the set of names of variable parameters which were found to be required in Step 4.4;
- C is the set of constant parameter names and their values; and
- r is the HTTP response type. □

Figure 5.4 shows the abstract HTTP interactions for our Vauxhall example, generated from the concrete HTTP interactions in Figure 5.3. For instance, the equivalence class [VI] corresponds to abstract interaction **VI** in the dependency graph.

Definition. A dependency graph is a directed acyclic graph whose nodes are abstract HTTP interactions. Each edge is annotated by a tuple (n, s, t), where n is a parameter name, s is a data selector and t is a *value transformer*. A *value transformer* is a function which converts the data extracted from the response of one interaction into the format required by the following interaction. In addition, the graph contains special nodes *input* and *output*, each annotated with a set of parameter names. □

Note that the algorithm does not need to explore the whole dependency graph. It is explored lazily until a suitable wrapper is found. In particular, the dependency graph never needs to be materialised; the algorithm searches it implicitly to find a suitable wrapper.

Definition. An HTTP wrapper is a dependency graph with the following properties:

- It contains the special *output* node;
- All parameter dependencies are resolved. That is, for each abstract interaction in the graph, if it contains an input parameter named n, then there is an outgoing edge labelled with n.

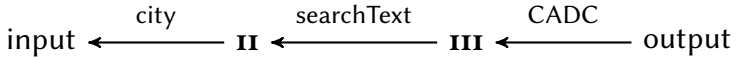
The generated HTTP wrappers are sub-graphs of the full dependency graph. □

Step 4.6. *Identify any variable parameters for the group whose values come directly from the input parameters.*

For example, the parameter "city" in interaction II is provided directly in the input data. In this case the parameter occurs exactly in the input, but in other cases transformations may be required, as in Step 4.1.

Step 4.7. *For each variable parameter p that cannot be obtained from the input, run the algorithm recursively from Step 2. The target data during the recursion is the set of parameter values of p. To avoid loops, remove all interactions from the current equivalence class from consideration. The recursion produces a wrapper W_p for each such variable parameter p, or returns that no wrapper can be found. If the recursive call for any variable fails to find a wrapper, then skip this group and return to Step 4.*

For example, when considering abstract interaction **VI**, the algorithm runs recursively to find suitable values for the variable parameters "CADC" and "UPSC". These recursive calls return "local" wrappers W_{CADC} and W_{UPSC}, which will ultimately become subgraphs of the final wrapper constructed by the algorithm. Both of these local wrappers have the form:



A useful effect of this recursion is that the final wrapper doesn't necessarily follow the chronological order of HTTP requests as they were recorded in the trace; they can be rearranged as needed to obtain a more efficient wrapper.

Step 4.8. *Merge all the local wrappers W_p into a new graph W. Also add the current abstract interaction, along with edges from it to each W_p, labelled by their parameter name p. Exit the loop and return W.*

For example when processing abstract interaction **VI**, we have two subgraphs, each with the form shown above, where **III** provides the target data. As such we add two edges from **VI** to **III** labelled with "CADC" and "UPSC".

The full dependency graph for our example is shown in Figure 5.4, including all dependency edges which *could* provide the required data at each step. The graph contains one wrapper, which is returned by the algorithm (comprising nodes **II**, **III** and **VI**). In our experiments, the second candidate wrapper (comprising nodes **II**, **IV** and **VI**) is rejected at Step 4.2 because abstract interaction **IV** could not be repeated.

Some parameter matches from the HTTP traces do not give rise to corresponding abstract dependencies. In our example, Figure 5.4 has no edges between abstract interactions **VI** and **V**, or **VI** and **I**, even though those edges *do* appear in Figure 5.3. This is because the result content of interactions I and V includes all the possible values for certain parameters, and it is therefore impossible to generate a data selector which can precisely extract only the relevant data corresponding to the given user input. Thus, these nodes are rejected at Step 4.1 as sources of those parameters. Note that the arrows in the figure show dependencies, so the data flow in the resulting wrapper is in fact from left to right.

The HTTP wrapper graph returned by this algorithm is simple to execute concretely as a usable wrapper. The definition of an HTTP wrapper means that it either requires no input, or only requires parameters from the special *input* node, which are those

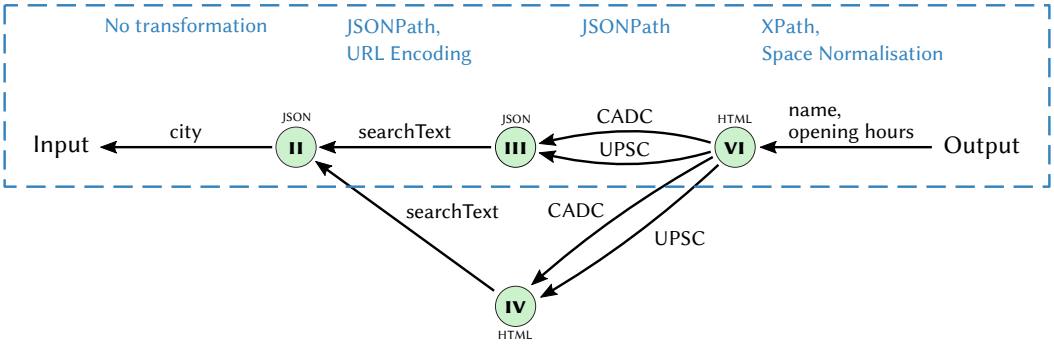


Figure 5.4: The full dependency graph for the Vauxhall example. Abstract interaction **IV** is included for illustration, although in reality it could not be reliably repeated, and so would have been removed at Step 4.2. The boxed area shows the resulting HTTP wrapper, and the value transformations applied at each step.

we would have used for the original visual wrapper. The execution is by depth-first traversal of the wrapper graph, beginning at the output node. Each interaction is executed—that is, the request is sent to the server and the response received—once all the values for its input parameters are obtained, whether they come directly from the input or from prior HTTP interactions.

As presented, the FastWrap algorithm returns the first wrapper it is able to discover. It is relatively simple to extend the algorithm to return all wrappers from the full dependency graph. There are two changes required to Step 4.8: First, it would not break the loop once one wrapper has been found, but would continue the search. Second, it would need to build output wrappers for each combination of the different sub-graphs available for each parameter returned from Step 4.7. Some heuristics would likely be required to determine which of these candidate wrappers were worth keeping, and which were redundant and could be discarded.

In our FastWrap implementation, we have extended the algorithm to return a wrapper for every abstract interaction containing the *final* output data (as returned by the visual wrapper), but it does not attempt to return every possible wrapper. In our experiments, very few examples gave rise to more than one HTTP wrapper.

Algorithm 3 shows the pseudo-code implementing the FastWrap process described above. The algorithm takes as input a visual wrapper V, and a set of input values I for this wrapper. It returns an HTTP wrapper which extracts the same data as V, or $\perp$ if none could be found. The main work of the algorithm is performed by the recursive procedure Resolve-Dependencies, which takes as input a set of HTTP traces $\mathbb{T}$, a set of input values I, and a set of target records $\mathbb{D}$, and returns an HTTP wrapper or $\perp$.

Algorithm 3 The FastWrap algorithm.

```
1:  procedure FastWrap(V, I)                                                    Step
2:      𝕋 ← ∅; D ← ∅                                                            1
3:      for input ∈ I do
4:          trace, extracted-data ← Execute-Visual-Wrapper(V, input)
5:          𝕋 ← 𝕋 ∪ {trace}
6:          D ← D ∪ {extracted-data}
7:      end for
8:      return Resolve-Dependencies(𝕋, I, D)
9:  end procedure
10: procedure Resolve-Dependencies(𝕋, I, D)
11:     Find the HTTP interactions H from traces in 𝕋 which contain the target data D.   2
12:     Group H into a set of equivalence classes C.                                     3
13:     for c ∈ C do                                                                     4
14:         S, F ← Generate-Selector(c, D)                                               4.1
15:         if selector S and value transformation function F are successfully generated then
16:             ▷ Confirm the interactions can extract the target data with S and F.
17:             D′ ← {Execute-HTTP-Wrapper(i, S, F) | i ∈ c}                             4.2
18:             if D′ = D then
19:                 P_var, P_con ← Identify-Variable-Parameters(c)                       4.3
20:                 P′_var ← {p ∈ P_var | Check-Parameter-Required(c, p)}                4.4
21:                 Create an abstract interaction A from C.                             4.5
22:                 ▷ Check for parameters available directly from the input
23:                 input-params ← ∅
24:                 for p ∈ P′_var do
25:                     if values(p) are available from I via value transformer F_in then
26:                         Mark p as coming from the input via F_in.                    4.6
27:                         input-params ← input-params ∪ {p}
28:                     end if
29:                 end for
30:                 P″_var ← P′_var \ input-params
31:                 ▷ Resolve dependencies for the remaining parameters
32:                 𝕎 ← ∅; 𝕋′ ← {t \ c | t ∈ 𝕋}
33:                 for p ∈ P″_var do                                                    4.7
34:                     𝕎 ← 𝕎 ∪ {Resolve-Dependencies(𝕋′, I, values(p))}
35:                 end for
36:                 if ⊥ ∉ 𝕎 then
37:                     Initialise a wrapper W.
38:                     Add node A, connected to the output with annotation S, F.
39:                     Add each sub-wrapper W_p ∈ 𝕎, linked to A with annotation name(p).   4.8
40:                     return W
41:                 end if
42:             end if
43:         end if
44:     end for
45:     return ⊥
46: end procedure
```

Implementation details

In our implementation, the HTTP traces are recorded by an instrumented version of OXPath [45]. The input wrapper is provided as an OXPath expression, and whenever the OXPath browser makes an HTTP call to a remote server, it is recorded in the HTTP trace. These HTTP traces are saved as log files by OXPath, and are given as input to FastWrap. FastWrap itself is implemented in Python.

To extract the target data from HTTP responses (whether for output or as input to a subsequent interaction), FastWrap supports either XPath or JSONPath expressions as data selectors. XPath expressions extract data from HTML or XML content, and JSONPath is an analogous query language for JSON-formatted data. Additionally, once the XPath or JSONPath expression is applied, the data selectors support serialising the resulting object as a string and either splitting it on certain characters or extracting certain substrings. This makes the data selector system very flexible, and means that data can be extracted even if it is not available in a perfectly "clean" form in the response.

Steps 4.1 and 4.6 in the algorithm can apply transformations to extracted data so that it matches the format required—either for output or as input to a subsequent interaction. FastWrap supports a library of simple transformation functions which can be identified and applied automatically. They are as follows:

- Trimming leading and trailing spaces, and removing double spaces;
- Extracting substrings and concatenation with constant strings;
- Various methods for URL encoding and decoding;
- Replacing HTML escape sequences;
- Truncation of numeric strings to a certain precision (which is common in APIs which deal with geographic coordinates).

The functions we have chosen to implement so far are based on our observations of common patterns in real-world sites. We also support some simple combinations of the above transformations, where two are applied in sequence. It is easy to extend the system with new value transformers or combinations as they are required.

Setting the scope of supported wrappers

In order to develop a first prototype, we have made certain assumptions about the wrappers and websites being analysed. These set the scope of applicability of the current algorithm. With further work, we hope that many of these assumptions can be relaxed or removed, making the system more widely applicable.

Extraction from a single source. The input wrapper may make any number of user-level actions, and visit multiple pages during its execution, but the data extracted must all come from a single page. Further, our algorithm assumes that the target data is available from a single HTTP interaction, whether it be a web page or AJAX request. For example, if a site combines restaurant names obtained from one AJAX call with

postcodes from another to display them together, the current version of FastWrap would not support extracting the combined records including both fields. The vast majority of real-world wrappers only extract data from a single target page, and it is relatively rare for sites to assemble their output data from multiple sources on the client-side. Most wrappers which extract from multiple pages do so by following "next page" links, which could be handled separately.

We also assume that the input wrappers are "linear" in the sense that they follow a single path to provide any required input data and extract their target output records. We do not support iteration to apply the same actions over multiple items (for example, wrappers such as "for each product on the product listing page, navigate to its product detail page and extract the price"), which produce "tree-shaped" traces.

Extracting flat records. Our implementation assumes that extracted records (whether part of the output or for intermediate parameters) have a fixed, flat structure. Our records have no optional parameters and no hierarchical structure. OXPath wrappers commonly extract structured entities, for example a wrapper might extract product listings including attributes of the products (some of which are not present for all products), and the list of user reviews for each product. FastWrap does not yet support such wrappers; for the above example, supported wrappers would extract fixed information about each product, or extract a list of reviews for a specific product.

Reproducible HTTP interactions. FastWrap assumes that HTTP interactions can be replayed to attain the same response content, and also that the abstract HTTP interaction templates can be used to generate valid new interactions which return useful data. Some sites use dynamically-generated session IDs, or short-term session cookies which violate this assumption. In these cases, the server can recognise that a replayed request is no longer valid and return an error message instead of the previous content. As such FastWrap is unable to generate HTTP wrappers for these sites.

Fixed parameter names. Our approach assumes that parameter names are fixed and can be used to identify the same parameter between traces. Some sites, however, use variable parameter names, and in particular encode the parameter *values* into the names of the parameters in their requests. For example, many sites use the Google Maps geocoding API to convert location searches to geographic coordinates which are then used to look up branch locations near those coordinates. When searching for "oxford", the request to Google's geocoding API (namely a GET request to GeocodeService.Search) includes the parameters "4soxford", "7sgb" and "9sen-US", each with no value, as well as some API key and callback-related parameters. Here the parameter names are used to transfer the query data to the server. As FastWrap depends on the parameter names for matching equivalent interactions across traces, sites which use this pattern as part of the critical-path for looking up the target data cannot be supported. Even if a general-purpose solution is difficult, Google Maps and its Geocoding API are so prevalent that it would be worthwhile to add specific special-purpose handling for this particular API.

5.3 Experimental evaluation

This section presents our evaluation of FastWrap on a variety of real-world sites and realistic wrappers. The input wrappers were provided as OXPath expressions and the HTTP traces were recorded by an instrumented version of OXPath, as described above. These traces are then passed to our Python implementation of FastWrap which generates a corresponding HTTP wrapper in a custom JSON-based format.

Choice of test cases
Our evaluation uses wrappers for real-world websites, following realistic data-extraction tasks. Where possible, we used a suite of existing wrappers generated by DIADEM, and in other cases created new OXPath wrappers manually.

Our different classes of wrappers are as follows:

- *No-input wrappers.* A set of 41 wrappers for US restaurant chains which do not require input. They typically extract information about the restaurant branches which are all visible on a single page, so the wrapper simply navigates to that page and extracts the data. Note that this does not mean there is nothing for FastWrap to optimise; the data may still be loaded via AJAX, and cutting out the browser still provides a benefit.

- *Restaurants.* A set of 12 wrappers for US restaurant chains which do require input to look up the desired output.

- *Cars.* A set of 11 wrappers for popular car manufacturers, which extract information about their dealerships.

- *Random URLs.* A set of 25 wrappers for a set of random URLS. The URLs were sampled from the Common Crawl search index [30], which contains over 3 billion pages. We chose to include random URLs to test how FastWrap performs on "long tail" sites. Most of the sites were much simpler than those from the other domains. Therefore most of these wrappers simply use the site's search box to look up certain keywords and return the corresponding results.

- *Multi-step wrappers.* A set of 12 wrappers which require multiple interactions to make a query and extract data, both at the user-action level and the HTTP level. For example, a typical wrapper from this group would enter a search string into a search box, choose the first item from the auto-complete list, and finally extract information from the detail page of the suggested product. A corresponding HTTP wrapper would need to be able to link the AJAX request for the auto-complete field with the corresponding product page being loaded. These examples are mostly retail sites.

In total, we evaluate 101 wrappers across these five verticals. The 56 wrappers for the combined restaurants domains were generated by DIADEM's fully-automated wrapper induction [44]. They represent real-world data extraction wrappers, and were only minimally updated to reflect changes in the target sites. The 45 wrappers for the three remaining verticals were created manually. These wrappers extract similar types of data (car dealership information, article information from news sites, and so on) using similar methods to the DIADEM wrappers. The wrappers for the multi-step set were carefully constructed to ensure that a corresponding HTTP wrapper is required to chain multiple requests in order to access the target data.

The wrappers listed above all match the scope restrictions described in the previous section. Wrappers which we could easily see were out of scope were excluded directly, and others were removed from the evaluation once their out-of-scope features were discovered. In all, we tested 130 wrappers and excluded 16 for using Google's geocoding API as a critical part of the query process, and 13 for having necessary HTTP interactions which FastWrap was not able to successfully repeat. This left the 101 valid wrappers for the evaluation.

Experimental setup

The experiments are run in two phases. First, the input wrappers were executed in OXPath to record the HTTP traces. Our version of OXPath is extended with a proxy server which watches all the HTTP request-response exchanges made by the OXPath browser and records them into the trace. It also records the input and output data. Each wrapper was executed 8 times with different input parameters (chosen manually beforehand), except those which require no input, which only need to be executed once.

Second, the saved HTTP traces were passed to our FastWrap system, which attempted to generate a corresponding HTTP wrapper. For wrappers which take input, five of the recorded traces were used for wrapper induction, and three were reserved for the evaluation. These remaining three sets of input parameters and output data are used as a test set to check the new wrapper on inputs which were never seen during the induction phase and confirm it still extracts the expected data. The wrappers which do not take input were transformed and evaluated on a single trace; for these, we only need to confirm that they can reliably return the expected data.

If the HTTP wrapper returns identical data to the original OXPath wrapper for a given test input, it is marked as a *successful evaluation*. If all three of the test inputs for a given wrapper are successful, then the wrapper is marked as *fully correct*.

Results

Overall, FastWrap was able to successfully generate HTTP wrappers for 76 (75.2%) of our examples, with 72 of those (94.7%) being fully correct. However, the results varied significantly between the different classes of test wrappers.

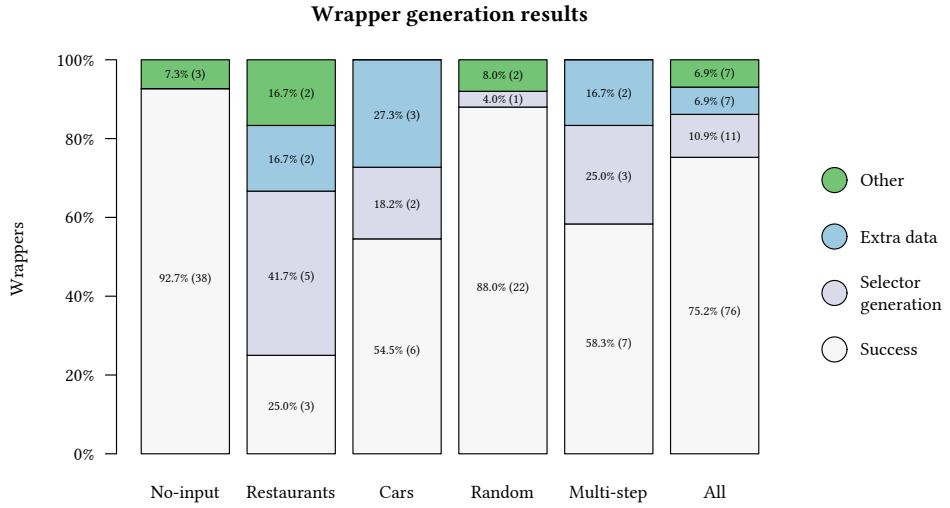


Figure 5.5: The different outcomes of the wrapper induction process for wrappers in each of the five groups.

Figure 5.5 shows the different results of the wrapper induction over the five groups. *Selector generation* refers to cases where FastWrap could not generate a data selector to extract the target data from an HTTP response. Composite parameters were a common problem here, where a single parameter would encode multiple data values. For example, a list of geographic coordinates are sometimes encoded in a single parameter value such as "lat|lon;lat|lon;lat|lon;..." and then split apart by the receiving code. A few simple cases can be handled by our string-splitting value transformations, but in general FastWrap has no way to decode such values into the structures they represent and extract the data. *Extra data* refers to wrappers which extract a superset of the expected data. This is often because a full data set, such as a list of search results, is retrieved via AJAX, and then only a filtered subset is shown on the page (which is what the visual wrapper then extracts). As such, when FastWrap extracts the data directly from the AJAX response it finds more records than its goal, with no way to know which should be returned. In many applications this is a benefit, because the goal of the wrapper is to eventually extract all the data—finding a short-cut to getting more data than is available to the visual wrapper represents an additional speed-up. However, for our evaluation we use the strict goal of extracting *identical* data to the visual wrapper, so these cases are not counted as successes. *Other* encompasses other issues which prevented a wrapper from being generated. This was commonly due to "heterogeneous content", where data is nested within multiple "layers" of different types of content. For example, the data might be encoded as an HTML snippet, which is serialised into a string in a JSON object. Our implementation of FastWrap does not support data selectors which span multiple formats like this.

Table 5.1: The results of FastWrap's wrapper induction process and the evaluation of the generated wrappers.

Wrappers	Avg. gen. time (s)	Ignored params (%)	Successfully generated (%)	Successful eval. (%)	Fully correct (%)
No-input	2.14	0.0	92.7	100.0	100.0
Restaurants	35.40	0.0	25.0	100.0	100.0
Cars	241.48	25.0	54.5	100.0	100.0
Random	33.07	8.7	88.0	97.0	90.9
Multi-step	99.89	14.8	58.3	91.7	77.8
All	44.91	13.8	75.2	98.2	94.9

On average, it took 44.9 seconds to generate an HTTP wrapper, excluding the time taken to record the original HTTP traces (which was 33.3 seconds per trace on average, and therefore 166.7 seconds for the five traces used for wrapper induction). For the simple wrappers in the no-input class, the average wrapper generation time was only 2.1 seconds, while the more complex multi-step and cars classes took 99.9 and 241.5 seconds respectively per wrapper on average. This is because the more complex wrappers recorded many more HTTP interactions, slowing the algorithm's interaction grouping and parameter-matching stages significantly.

Table 5.1 shows the results of the wrapper induction and evaluation process. Across all wrappers, FastWrap found that 13.8% of variable parameters checked could be removed without affecting the results. This indicates the importance of query probing for identifying unnecessary parameters to reduce the size of the dependency graph and increase the chance of resolving all dependencies. In total, 98.2% of the final wrapper evaluations were successful (the HTTP wrapper extracted the same data as the OXPath wrapper on previously unseen inputs), with 94.9% of the wrappers being fully correct (extracting the correct results on all three test inputs). Note that these numbers include three cases where two HTTP wrappers were generated for the same input wrapper.

Table 5.2 compares the performance of the generated HTTP wrappers against the original OXPath wrappers. In all cases, the new HTTP wrappers significantly outperform the corresponding OXPath wrappers. On average, they took 4.2% of the time (a 23.8-times improvement), used 3.2% of the network resources, and made 1.4% as many HTTP interactions. This confirms that removing the browser leads to a significant performance benefit for wrapper execution.

The majority of the HTTP wrappers required only a single HTTP interaction, plus any value transformations applied to its input and output. In the no-input, restaurants and random-URLs classes, for example, *all* the generated wrappers only require a single interaction. In the multi-step class, seven OXPath wrappers were successfully converted,

Table 5.2: A comparison of the average time taken (excluding browser initialisation), data transferred (with images disabled), and number of HTTP interactions made when executing the OXPath wrappers and their HTTP wrapper equivalents.

	OXPath wrappers			HTTP wrappers			Comparison		
Wrappers	Time (s)	Data (KB)	Int.	Time (s)	Data (KB)	Int.	Time (%)	Data (%)	Int. (%)
No-input	23.88	1 487	38.6	0.84	68	1.0	3.5	4.6	2.6
Restaurants	15.38	1 990	73.6	1.03	16	1.0	6.7	0.8	1.4
Cars	22.31	5 272	117.5	0.87	46	1.2	3.9	0.9	1.0
Random	22.30	2 773	79.7	1.31	75	1.0	5.9	2.7	1.3
Multi-step	42.53	3 884	76.6	1.29	191	1.9	3.0	4.9	2.5
All	26.92	2 942	76.9	1.14	94	1.1	4.2	3.2	1.4

with six of the new wrappers using two interactions, and one using three. There is also a single example from the cars group which required two interactions. These examples represent successes for the algorithm's dependency tracking, which was able to track the parameters' data-flow through the traces and "chain together" multiple interactions in sequence to reach the target data. Finally, for two of the multi-step examples, FastWrap was able to find a second wrapper using only a single interaction, which is why the average number of interactions shown in Table 5.2 is 1.9. This highlights one of the great advantages of our approach; even with examples specifically chosen to require multiple interactions, FastWrap was sometimes able to find a short-cut to a more efficient wrapper.

Once generated, the vast majority of HTTP wrappers were evaluated successfully and found to be fully correct. Only two wrappers each from the random-URLs and multi-step groups failed any evaluations at all.

FastWrap's HTTP wrappers perform interactions fetching HTML data 76% of the time, JSON data 22% of the time, and XML only 2% of the time. Unsurprisingly, the more complex sites made heavier use of JSON, as the target data was more often accessed on-the-fly using AJAX. For example, the simpler sites from the no-input and random-URLs groups used HTML interactions 92% and 91% of the time, respectively. In contrast, the more complex cars and multi-step groups used HTML only 13% and 55% of the time, and JSON 75% and 45% of the time, respectively.

The wrappers make use of the following value transformations (excluding transformations of the output data, all of which simply trim extra white-space): no transformation (68% of the time); URL encoding or decoding (21%); string operations (5%); truncating numeric data to a certain precision (4%); and composite transformations involving URL encoding before or after other transformations (2%).

5.4 Discussion

Our experiments confirm that it is feasible to automatically transform traditional browser-based data extraction wrappers into browserless HTTP wrappers. The FastWrap approach is effective, and the generated wrappers are dramatically more efficient than the original browser-based ones. Our techniques could already provide huge benefit to many real-world data extraction problems.

However, there are obviously still many improvements which can be made. The main goal of our future work will be to increase the number of sites and input wrappers which FastWrap can operate on. While the scope restrictions discussed at the end of Section 5.2 permit many useful and powerful wrappers to be tackled, they still exclude many which we would also wish to handle. Some ideas for loosening these restrictions, and addressing the failures seen in our evaluation, are discussed below. In particular, it seems important to address the issues observed in the more complex wrappers. As illustrated in Figure 5.5, even among wrappers which fell within our algorithm's current scope, 25% of wrappers could not be translated.

Heterogeneous content. During our evaluation, we discovered several instances of "nested" content, where a structure of one content type is serialised within another structure, often of a different content type. The most common example is snippets of HTML being serialised as strings in a JSON object returned from an AJAX call. These snippets are then added directly to the page, displaying the data. Either the HTML or JSON content alone can be parsed, but currently there is no way to extract data from these nested structures. It should be possible to create *hybrid data selectors* which are able to select data through multiple "levels" of serialisation. This would solve many of the common cases. It may also be useful to apply some existing techniques for selecting known data from unknown or variable structures [26, 63, 85].

Composite parameters. Several of our evaluation sites encode multiple pieces of data into a single parameter. This can vary from simple pairs of values which can sometimes be handled by our string-splitting value transformers up to much more complex structures which cannot be parsed. There is currently no way for FastWrap to parse these parameters to extract data or, if they are input parameters for an interaction, reconstruct them to make new HTTP requests. Domain knowledge could help in many cases, for example addresses might be identified from the combination of the various separate parameters for the house number, street name, and so on. Techniques from data matching may also be applicable [21, 80], once the variable parts have been identified by comparing interactions from the same equivalence class.

Extracting too much data. It was relatively common (6.9% of the wrappers tested) to find interactions which could provide more data than the original OXPath wrapper returned. For example, a car dealership search interface may return all the models from a certain manufacturer, while a client-side JavaScript filter then restricts the models

shown to the user to only those which match their search criteria. As FastWrap has no way to filter out the "expected part" of the data, and our evaluation aimed at replicating the input wrappers exactly, these cases were marked as failures. As noted earlier, in many real applications, this would not be a problem and may in fact be an improvement over the original wrapper. In other cases, it may be possible to apply JavaScript analysis to track how the results are filtered on the client-side, and re-apply the same filter in the browserless wrapper. Alternatively, with enough training examples it may be possible to use machine learning techniques to identify which results should be filtered out.

Tracking complex transformations. As well as filtering, JavaScript analysis may be useful to track other kinds of complex value transformations. FastWrap currently identifies value transformations by comparing the output data of one interaction with the input data required for another and checking if any of its library of known transformations can be used to make the necessary conversion. More types of transformation can be supported by extending this library. An alternative would be to use symbolic tracing, as described in Chapter 3, to track the real data-flow in the client-side JavaScript code to see exactly what transformations are applied. This would allow much more complex and site-specific value transformations to be tracked. It may also be possible to automatically extract snippets of JavaScript which perform the required transformation from the site, by combining program slicing [138, 142] with the symbolic tracing.

Supporting new content formats. FastWrap can extract data from HTML, XML, JSON, and JSONP documents. However, we have also found other formats in use, and supporting them would increase the number of sites FastWrap can successfully analyse. A surprising example we found on several sites is data embedded into JavaScript source code. The sites will load a JavaScript source file from the server, which contains assignments of literal values into variables, and these variables are built up into complex objects. Once the final result object is built, it is passed to a callback function, defined in the sites main code, which handles the downloaded data. This pattern is used in a similar way to JSONP (JSON data "padded" with a callback function) to access data from other domains without violating the browser's same-origin policy, but the returned structure is instead built up piece-by-piece on the fly. In these examples, it may be possible to extract flat records by text-mining the downloaded JavaScript source. A more interesting approach may be to execute these JavaScript snippets in isolation (that is, without a full browser), and extract the final result object at the end.

Non-reproducible HTTP interactions. FastWrap was not able to successfully reproduce all the recorded HTTP interactions. This was often related to session state being shared with the server in ways which our analysis did not pick up. For example, an initial interaction may contact the server to obtain a session ID, or generate one on the client-side. If these IDs are not obtained or generated correctly, new requests to the same endpoint will fail. Research in the area of web automation may be helpful towards automatically obtaining these session IDs as if a real browser were being used [15, 18].

Interactions with varying parameters. Our process for matching interactions across traces relies on corresponding interactions having matching sets of parameters. If parameters have different names in different traces, or a certain parameter is used in some traces and not in others for the same HTTP request, then FastWrap will not be able to recognise that these interactions should be grouped. Handling these cases would require more advanced approaches to identifying similar HTTP requests that are able to deal with such a high degree of variability.

Complex wrapper navigation. To simplify our first prototype, we required the input wrappers to follow a single linear path and extract data from a single source. Although these restrictions still permit many useful real-world wrappers, it would be better to support a wider variety of wrappers as inputs. Some of the restrictions on the input wrappers are simple to remove, but some will require further research. For example, the algorithm can easily be adapted to extract the final output data from multiple sources. A new node would be introduced to the dependency graph representing the final combination of the different output fields required. This node would provide the final output, and its input parameters would be the individual parameters of the output data, which the existing algorithm could then link to the various different abstract interactions which can provide them. Wrappers which include certain well-defined looping patterns, such as following "next page" links to extract results from multiple pages, have been studied (for example in DIADEM), but supporting more general looping and branching in wrappers requires more work.

Potential extensions

As well as addressing the limitations described above, we have identified several interesting extensions to the approach which would be interesting to investigate.

Wrapper optimisation. The current approach aims to generate a single HTTP wrapper, and stops as soon as a suitable one is discovered. From our tests it seems that many sites do not permit multiple different paths to the same data, so this is enough. However, in some cases, and in particular if more complex sites and APIs are targeted, it may be interesting to generate a suite of equivalent wrappers and choose the best among them. Returning all possible HTTP wrappers which can be found from the given traces only requires a relatively small changes to the algorithm, as described in Section 5.2. Then they could be compared based on the number of HTTP interactions required, or benchmarked to determine the time taken or data transferred while executing them. It would also be possible to include this optimisation problem in the wrapper generation process itself, and avoid exploring new ways to reach the same data if it appears they will be more costly.

Short-cuts to certain values. In our experiments FastWrap sometimes encountered interactions which could provide a list of the possible values for a certain parameter, while only one value at a time is necessary for each query. This was seen in the Vauxhall

example earlier, where interaction **I** provided all the possible values for the parameters "CADC" and "UPSC". If the goal of the wrapper is to make multiple queries to extract all the data available from a site, then these interactions could provide a short-cut. Instead of running the whole wrapper from the beginning for each query, it should be possible to extract the set of possible values for some intermediate parameter and use those directly as inputs to the following interactions. If the values for "CADC" and "UPSC" were known ahead of time, then the wrapper could be reduced to calling interaction **VI** and extracting the data, rather than calling interactions **II**, **III** and **VI** in sequence.

Partially-browserless wrappers. Finally, it would be interesting to investigate ways to generate "partially browserless" wrappers. It is not always possible to generate a fully browserless wrapper for every site. The APIs being used might require some client-side processing which FastWrap cannot model, or the site may use session IDs which FastWrap cannot track. In these cases, we would like to generate a wrapper which runs without a browser where possible, but invokes the browser for any specific steps where it is required. If FastWrap can identify steps in the wrapper which can be reproduced successfully in a browser but not directly by its own HTTP interactions, then the result may be a much more flexible system which can improve wrappers for a much wider variety of sites.

5.5 Related work

To our knowledge, this is the first work to tackle the problem of automatically transforming visual wrappers into HTTP wrappers. This work is an extension of an earlier master's project [113].

There are many sophisticated semi-automated tools which allow simple, GUI-based wrapper generation. These include STALKER [105], Lixto [19, 56], import.io,[33] Mozenda,[34] FMiner,[35] iMacros,[36] Visual Web Ripper,[37] BODE [126], and Visual OXPath [81]. The wrappers produced by these tools are executed using traditional browser engines. DIADEM can generate data-extraction wrappers fully automatically, given only a list of target sites' URLs [44]. This is done using a set of Datalog rules which encode prior knowledge about the particular application domain being targeted (e.g. property or used car sites).

Web automation tools and AJAX-aware crawlers share some similarities with visual wrappers, in that they use a web browser to simulate user actions on a web page. Web automation tools generate "test scripts" (essentially wrappers with little or no data extraction) to check the correct behaviour of a web application [15, 18, 22, 84, 86]. Our approach could be used on these test scripts and could be used to verify that the

[33] https://www.import.io/
[34] http://www.mozenda.com
[35] http://www.fminer.com
[36] http://imacros.net
[37] http://visualwebripper.com

application's back-end produces the expected results. However, as our goal is to avoid interaction with the application's front-end as much as possible, it is obviously not well-suited to testing that part of an application.

Modern web crawlers account for AJAX-based sites by building a state-graph of the different application states which can be reached by performing different sequences of actions on the page [33, 35, 100]. Some examples of these tools are described in Section 4.11. Our approach could be adapted to convert these user-action–level state graphs into new graphs at the HTTP-interaction level. If the purpose of crawling is to extract data (including, e.g., for site indexing), then the back-end HTTP-level state graph would provide a significantly faster way to retrieve that data than the original front-end state graph.

There has been some prior work on analysing logs of HTTP interactions, to infer something about the behaviour of the user generating those logs [91, 141]. At a high-level, this is similar to our work (both analyse traces of HTTP interactions to reconstruct some higher-level information about the system or interaction) but the similarity is mainly superficial. We focus on observing interactions with a single website, without any control over that site. In contrast, HTTP log analysis usually assumes many users accessing a single instrumented back-end. As far as we are aware, none of the existing work on HTTP log analysis considers the problem of discovering the underlying back-end API (which is usually assumed to be known). Polaris is another tool which uses dependency tracing to optimise web pages [106], but the focus is quite different: Polaris tracks which page objects (HTML files, JavaScript code, etc.) cause other objects to be loaded, with the goal of optimising page load times.

Our work on FastWrap was presented at WWW 2018 [41].

6 Conclusion

This thesis presents several techniques for improving the state of the art in web interface analysis by combining traditional web analysis techniques with JavaScript analysis. This combined approach has not been extensively explored by prior work in the context of interface analysis or web information extraction. Existing approaches typically focus either on web page and DOM analysis, using heuristics and domain knowledge to reason about a page, or on JavaScript analysis alone, largely ignoring its interaction with the parent web page.

This work develops certain key components necessary for a full-fledged web page analyser, information extraction system, or deep web crawler. ArtForm is able to explore and analyse a page's user-level actions and use the information gathered to generate machine-accessible descriptions which could be applied to wrapper induction. These user actions must be executed in certain sequences to achieve particular effects on a web page, and ArtForm is able to model and explore these sequences of events.

By using a production web browser as its base, our analysis platform avoids or simplifies many of the most difficult aspects of traditional JavaScript analysis, such as JavaScript's highly dynamic features, the object system, and so on. It also simplifies many of the web-specific issues with traditional analysis approaches, such as code obfuscation and the use of libraries. JavaScript analysis alone is not enough, a useful interface analysis tool must also account for how the JavaScript code interacts with the web page via the DOM API. Again, instrumenting a real web browser simplifies much of this modelling, allowing our platform to pull concrete facts from the DOM during the symbolic trace recording.

As well as interface analysis, which is focused on generating data extraction wrappers, similar approaches can be used to optimise existing wrappers. The work on FastWrap shows that associating user-level browser actions with their JavaScript- and HTTP-level implementations is an effective way to model the behaviour of a visual wrapper and thus to generate more efficient equivalents which do not require a full browser environment.

As discussed in the introduction, this thesis presents the following key contributions:

- Development of a concolic testing platform for web JavaScript, which is based on a production web browser engine and which is effective in practice;
- Application of this analysis platform to the problem of exploring and understanding web forms on real-world websites without prior domain knowledge, including:

  - Integration of specialised modelling of the form input restrictions from the DOM into the concolic testing,
  - Development of the new dynamic action reordering algorithm for concolic testing of event-driven form validation code,
  - Development of a set of manual website investigation and analysis tools using the same analysis infrastructure,
  - Development of a concolic testing advice server which can provide low-level concolic testing information to third-party tools in a simple way, and
  - Application of the concolic testing platform to the discovery and analysis of delegated event handlers;
- Automatic optimisation of existing data extraction wrappers by connecting the high- and low-level actions performed by these wrappers, resulting in more than an order of magnitude performance improvement on many sites.

While they do not constitute a complete information extraction or crawling system in themselves, each of these contributions improves on the corresponding components used in existing systems, and would make valuable additions to such systems.

A final contribution is an overview of the limitations of our approach and our implementation, as well as those of various alternative approaches. We have also outlined some seemingly promising directions for future work, both as extensions of our system and as integrations with other tools.

References

[1] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 367–377. ACM, 2014.

[2] Domenico Amalfitano, Anna Rita Fasolino, Armando Polcaro, and Porfirio Tramontana. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering*, 10(1):41–57, March 2014.

[3] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08, pages 367–381. Springer, 2008.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12. ACM, 2012.

[5] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture notes, Technical University of Denmark, April 1998.

[6] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP '05, pages 428–452. Springer, 2005.

[7] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '14, pages 17–31. ACM, October 2014.

[8] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580. ACM, May 2011.

[9] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, July 2010.

[10] Tab Atkins, Jr., Elika J. Etemad, and Florian Rivoal. CSS Snapshot 2015. W3C Working Group Note, W3C, 2015. http://www.w3.org/TR/2015/NOTE-css-2015-20151013/.

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *CoRR*, arXiv:abs/1610.00502, 2016.

[12] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 441–450. ACM, 2007.

[13] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. *JIDM*, 1(1):133–144, 2010.

[14] Luciano Barbosa, Hoa Nguyen, Thanh Nguyen, Ramesh Pinnamaneni, and Juliana Freire. Creating and exploring web form repositories. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1175–1178. ACM, 2010.

[15] Shaon Barman, Sarah Chasins, Rastislav Bodík, and Sumit Gulwani. Ringer: Web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 748–764. ACM, 2016.

[16] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, CAV '11, pages 171–177. Springer, 2011.

[17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. http://www.smt-lib.org.

[18] Alberto Bartoli, Eric Medvet, and Marco Mauri. Recording and replaying navigations on AJAX web sites. In *International Conference on Web Engineering*, ICWE '12, pages 370–377. Springer, 2012.

[19] Robert Baumgartner, Oliver Frölich, and Georg Gottlob. The Lixto systems applications in business intelligence and semantic web. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference*, ESWC '07, pages 16–26. Springer, June 2007.

[20] Michael Benedikt, Tim Furche, Andreas Savvides, and Pierre Senellart. ProFoUnd: Program-analysis–based form understanding. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 313–316. ACM, April 2012.

[21] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *Proceedings of the VLDB Endowment*, 4(11):695–701, 2011.

[22] Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: Enabling blind users to blaze trails through the web. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, IUI '09, pages 177–186. ACM, 2009.

[23] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[24] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[25] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams. Selectors Level 3. W3C Recommendation, W3C, 2011. https://www.w3.org/TR/2011/REC-css3-selectors-20110929/.

[26] Michal Ceresna. *Supervised Learning of Wrappers from Structured Data Sources*. PhD thesis, Vienna University of Technology, 2005.

[27] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages*, OOPSLA, October 2017.

[28] Alex Q. Chen, Simon Harper, Darren Lunn, and Andrew Brown. Widget identification: A high-level approach to accessibility. *World Wide Web*, 16(1):73–89, January 2013.

[29] David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):207–242, 2016.

[30] Common Crawl. http://commoncrawl.org/. Crawl index from CC-MAIN-2017-04.

[31] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 109–118. Morgan Kaufmann, 2001.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS '08, pages 337–340. Springer, 2008.

[33] Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. A statistical approach for efficient crawling of rich internet applications. In *Web Engineering*, pages 362–369. Springer, 2012.

[34] Eduard C. Dragut, Thomas Kabisch, Clement Yu, and Ulf Leser. A hierarchical approach to model web query interfaces for web source integration. *Proceedings of the VLDB Endowment*, 2(1):325–336, August 2009.

[35] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. AJAX Crawl: Making AJAX applications searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89, March 2009.

[36] Bruno Dutertre. Yices 2.2. In *Computer-Aided Verification*, CAV '14, pages 737–744. Springer, July 2014.

[37] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS '10. Springer, 2010. https://panopticlick.eff.org/about.

[38] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1388–1401. ACM, 2016.

[39] Esprima. http://esprima.org/.

[40] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2Colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, pages 37–47. ACM, 2013.

[41] Ruslan Fayzrakhmanov, Emanuel Sallinger, Ben Spencer, Tim Furche, and Georg Gottlob. Browserless web data extraction: Challenges and opportunities. In *Proceedings of The Web Conference 2018*, WWW '18. ACM, 2018.

[42] FormSolve test data. https://github.com/BenSpencer/FormSolve-Test-Data. *(FormSolve is the name used for ArtForm's dynamic-reordering algorithm in our ICWE '18 paper [125]).*

[43] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, and Christian Schallhart. The ontological key: Automatically understanding and integrating forms to access the deep web. *The VLDB Journal*, 22(5):615–640, 2013.

[44] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, Christian Schallhart, and Cheng Wang. DIADEM: Thousands of websites to a single database. *Proceedings of the VLDB Endowment*, 7(14):1845–1856, October 2014.

[45] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Jon Sellers. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, 2013.

[46] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, CAV '07, pages 519–531. Springer, July 2007.

[47] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44. ACM, 2012.

[48] Patrice Godefroid. "SAGE in one slide", SAGE: Whitebox fuzzing for security testing. Presented at PLDI '13, June 2013.

[49] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215. ACM, 2008.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[51] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '08. The Internet Society, 2008.

[52] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33. ACM, 2011.

[53] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 357–368. ACM, 2015.

[54] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA '15, pages 94–105. ACM, 2015.

[55] Google. Closure Compiler. https://developers.google.com/closure/compiler/.

[56] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The Lixto data extraction project: Back and forth between theory and practice. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 1–12. ACM, 2004.

[57] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187. ACM, 2011.

[58] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250. ACM, 2012.

[59] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperor's new APIs: On the (in)secure usage of new client side primitives. In *Proceedings of the 4th Web 2.0 Security and Privacy Workshop*, w2SP '10, 2010.

[60] Juliet L. Hardesty. Bells, whistles, and alarms: HCI lessons using AJAX for a page-turning web application. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 827–840. ACM, 2011.

[61] Hai He, Weiyi Meng, Yiyao Lu, Clement Yu, and Zonghuan Wu. Towards deeper understanding of the search interfaces of the deep web. *World Wide Web*, 10(2):133–155, June 2007.

[62] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5. W3C Recommendation, W3C, 2014. https://www.w3.org/TR/2014/REC-html5-20141028/.

[63] Andrew W. Hogue and David R. Karger. Thresher: automating the unwrapping of semantic content from the world wide web. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 86–95, May 2005.

[64] David Van Horn and Matthew Might. Pushdown abstractions of JavaScript. *CoRR*, arXiv:abs/1109.4467, 2011.

[65] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15. ACM, 2014.

[66] Jalangi2. https://github.com/Samsung/jalangi2.

[67] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1930–1937. ACM, 2009.

[68] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proc. 22nd International Symposium on Software Testing and Analysis*, ISSTA '13, July 2013.

[69] Simon Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. MemInsight: Platform-independent memory profiling for JavaScript. Technical report, Samsung Research America, 2015.

[70] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis*, ISSTA '12, July 2012.

[71] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '11, September 2011.

[72] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer, August 2009.

[73] Lu Jiang, Zhaohui Wu, Qian Feng, Jun Liu, and Qinghua Zheng. Efficient deep web crawling using reinforcement learning. In *Advances in Knowledge Discovery and Data Mining*, PAKDD '10, pages 428–439. Springer, 2010.

[74] JSFiddle. https://jsfiddle.net/.

[75] Samy Kamkar. evercookie. http://samy.pl/evercookie/, September 2010. Accessed 2016-08-30.

[76] Gustavo Zanini Kantorski, Tiago Guimaraes Moraes, Viviane Pereira Moreira, and Carlos Alberto Heuser. Choosing values for text fields in web forms. In *Advances in Databases and Information Systems*, 2013.

[77] Anne van Kesteren, Aryeh Gregor, Ms2ger, Alex Russell, and Robin Berjon. W3C DOM4. W3C Recommendation, W3C, 2015. https://www.w3.org/TR/2015/REC-dom-20151119/.

[78] Graham Klyne and Chris Newman. Date and Time on the Internet: Timestamps. Request for Comments 3339, IETF Network Working Group, June 2002. http://tools.ietf.org/html/rfc3339.

[79] Clemens Kolbitsch, Benjamin Livshits, Ben Zorn, and Christian Seifert. Rozzle: Decloaking internet malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 443–457. IEEE Computer Society, May 2012.

[80] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data and Knowledge Engineering*, 69(2):197–210, 2010.

[81] Jochen Kranzdorf, Andrew Sellers, Giovanni Grasso, Christian Schallhart, and Tim Furche. Visual OXPath: Robust wrapping by example. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 369–372. ACM, 2012.

[82] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1st edition, 2008.

[83] Pratap Lakshman and Allen Wirfs-Brock. Standard ECMA-262 5.1 Edition. ECMAScript Language Specification, Ecma International, 2011. http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf.

[84] Tessa Lau, Julián Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey Bigham, and Jeffrey Nichols. A conversational interface to web automation. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 229–238. ACM, 2010.

[85] A. Lemay, J. Niehren, and R. Gilleron. Learning n-ary node selecting tree transducers from completely annotated examples. In *Proceedings of the 8th International Conference on Grammatical Inference: Algorithms and Applications*, ICGI '06, pages 253–267. Springer, 2006.

[86] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa A. Lau. CoScripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, April 2008.

[87] Martin Lester, Luke Ong, and Max Schaefer. Information flow analysis for a dynamically typed language with staged metaprogramming. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 209–223, June 2013.

[88] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459, 2014.

[89] Panagiotis Liakos, Alexandros Ntoulas, Alexandros Labrinidis, and Alex Delis. Focused crawling for the hidden web. *World Wide Web*, 19(4), July 2016.

[90] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification*, CAV '14, 2014.

[91] Jun Liu, Cheng Fang, and Nirwan Ansari. Identifying user clicks based on dependency graph. In *2014 23rd Wireless and Optical Communication Conference*, WOCC '14, 2014.

[92] Ben Livshits and Salvatore Guarnieri. Gulfstream: Incremental static analysis for streaming JavaScript applications. Technical report, Microsoft Research, January 2010.

[93] Francesco Logozzo and Herman Venter. RATA: Rapid atomic type analysis by abstract interpretation. Application to JavaScript optimization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC/ETAPS '10, pages 66–83. Springer, 2010.

[94] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN '17, pages 196–199. ACM, 2017.

[95] Jianguo Lu, Yan Wang, Jie Liang, Jessica Chen, and Jiming Liu. An approach to deep web crawling by sampling. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, WI-IAT '08, pages 718–724. IEEE Computer Society, 2008.

[96] Yiyao Lu, Hai He, Hongkun Zhao, Weiyi Meng, and Clement Yu. Annotating structured data of the deep web. In *2007 IEEE 23rd International Conference on Data Engineering*, ICDE '07, pages 376–385, April 2007.

[97] Jayant Madhavan, David Ko, Łucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proceedings of the VLDB Endowment*, 1(2):1241–1252, August 2008.

[98] Bruce McKenzie. Generating strings at random from a context free grammar. Technical report, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 1997.

[99] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, 2009.

[100] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, March 2012.

[101] mitmproxy. https://mitmproxy.org/.

[102] Mozilla Developer Network. HTML Forms Guide. https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms. Accessed 2016-08-18.

[103] Mozilla Developer Network. <input>. https://developer.mozilla.org/en/docs/Web/HTML/Element/Input. Accessed 2016-08-18.

[104] Mozilla Foundation. Rhino. http://www.mozilla.org/rhino.

[105] Ion Muslea, Steven Minton, and Craig A. Knoblock. A hierarchical approach to wrapper induction. In *Agents*, pages 190–197, 1999.

[106] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI '16, 2016.

[107] Hoa Nguyen, Thanh Nguyen, and Juliana Freire. Learning to extract form labels. *Proceedings of the VLDB Endowment*, 1(1):684–694, August 2008.

[108] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[109] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2015.

[110] Nu Html Checker. https://validator.w3.org/nu/about.html.

[111] Object Management Group. Interface Definition Language. Specification, Object Management Group, April 2016. http://www.omg.org/spec/IDL/4.0/.

[112] Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong Ren. Graph-based AJAX crawl: Mining data from rich internet applications. In *Proceedings of the 2012 International Conference on Computer Science and Electronics Engineering*, volume 3, pages 590–594, 2012.

[113] Richard Penman. *Web Data Extraction Optimization: From User Interaction To Web Server Communication.* MSc Thesis, University of Oxford, 2016.

[114] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP '11, pages 52–78. Springer, 2011.

[115] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12. ACM, 2010.

[116] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528. IEEE Computer Society, 2010.

[117] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proc. of the 17th Annual Network and Distributed System Security Symposium*, NDSS '10, 2010.

[118] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 225–236. ACM, 2009.

[119] Michael I. Schwartzbach. Lecture notes on static analysis. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2008.

[120] Selenium WebDriver. http://www.seleniumhq.org/projects/webdriver/.

[121] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, pages 615–618. ACM, 2013.

[122] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, pages 263–272. ACM, 2005.

[123] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853. ACM, 2015.

[124] Ben Spencer, Michael Benedikt, Anders Møller, and Franck van Breugel. ArtForm: A tool for exploring the codebase of form-based websites. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '17. ACM, 2017.

[125] Ben Spencer, Michael Benedikt, and Pierre Senellart. Form filling based on constraint solving. In *Web Engineering*, ICWE '18. Springer, 2018.

[126] Jui Yuan Su, Der Johng Sun, I. Chen Wu, and Lung Pin Chen. On design of browser-oriented data extraction system and the plug-ins. *Journal of Marine Science and Technology*, 18(2):189–200, 2010.

[127] Hideo Tanida, Tadahiro Uehara, Guodong Li, and Indradeep Ghosh. Automated unit testing of JavaScript code through symbolic executor SymJS. *International Journal on Advances in Software*, 8(1&2), 2015.

[128] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP '05, pages 408–422. Springer, 2005.

[129] Peter Thiemann. A type safe DOM API. In *Database Programming Languages*, pages 169–183. Springer, 2005.

[130] T.J. Watson Libraries for Analysis. WALA. http://wala.sf.net.

[131] Guilherme A. Toda, Eli Cortez, Altigran S. da Silva, and Edleno de Moura. A probabilistic approach for automatically filling form-based web interfaces. *Proceedings of the VLDB Endowment*, 4(3):151–160, December 2010.

[132] TypeScript. http://www.typescriptlang.org.

[133] W3C Markup Validation Service. Why Validate? https://validator.w3.org/docs/why.html#debug. Accessed 2016-08-25.

[134] W3Techs Web Technology Surveys. Historical yearly trends in the usage of JavaScript libraries for websites. https://w3techs.com/technologies/history_overview/javascript_library/all/y, accessed 2018-03-12.

[135] W3Techs Web Technology Surveys. Usage of JavaScript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all, accessed 2018-03-12.

[136] W3Techs Web Technology Surveys. Usage of JavaScript libraries for websites. https://w3techs.com/technologies/overview/javascript_library/all, accessed 2018-03-12.

[137] Yan Wang, Jie Liang, and Jianguo Lu. Discover hidden web properties by random walk on bipartite graph. *Information Retrieval*, 17(3), June 2014.

[138] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981.

[139] Wensheng Wu, AnHai Doan, Clement Yu, and Weiyi Meng. Modeling and extracting deep-web query interfaces. In *Advances in Information and Intelligent Systems*, volume 251 of *Studies in Computational Intelligence*, pages 65–90. Springer, 2009.

[140] Wensheng Wu, Clement Yu, AnHai Doan, and Weiyi Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 95–106. ACM, 2004.

[141] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. ReSurf: Reconstructing Web-Surfing Activity From Network Traffic. *Proc. IFIP Networking Conference*, pages 1–9, 2013.

[142] J. Ye, C. Zhang, L. Ma, H. Yu, and J. Zhao. Efficient and precise dynamic slicing for client-side JavaScript programs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1 of *SANER '16*, pages 449–459, March 2016.

[143] Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement Yu. Fully automatic wrapper generation for search engines. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 66–75. ACM, 2005.

[144] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, pages 114–124. ACM, 2013.