ArtForm: A Tool for Exploring the Codebase of Form-Based Websites

Ben Spencer
University of Oxford, UK

Michael Benedikt
University of Oxford, UK

Anders Møller
Aarhus University, Denmark

Franck van Breugel
York University, Canada

ABSTRACT

We describe ArtForm, a tool for exploring the codebase of dynamic data-driven websites where users enter data via forms. ArtForm extends an instrumented browser, so it can directly implement user interactions, adding in symbolic and concolic execution of JavaScript. The tool supports a range of exploration modes with varying degrees of user intervention. It includes a number of adaptations of concolic execution to the setting of form-based web programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Dynamic analysis*; • **Information systems → Web crawling**; *Site wrapping*; *Deep web*; *Search interfaces*;

KEYWORDS

JavaScript, concolic testing, web forms, symbolic execution

ACM Reference format:
Ben Spencer, Michael Benedikt, Anders Møller, and Franck van Breugel. 2017. ArtForm: A Tool for Exploring the Codebase of Form-Based Websites. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA'17-DEMOS),* 4 pages.
https://doi.org/10.1145/3092703.3098226

1 INTRODUCTION

A key part of modern e-commerce and information enquiry software systems consist of web-based services in which a user enters and retrieves data via web forms. Understanding, analyzing, and testing the software behind these sites is challenging. A website may consist of a number of web forms and related widgets, including both standard and custom-developed interactive elements. Browser-based JavaScript uses an event-driven execution model, where user actions such as filling form fields or button clicks will trigger code to run which may in turn enable new events or download new code. The functionality is generally distributed over a large number of event handlers and libraries, and often uses third-party code which may be difficult to understand or even unavailable in source format. The relationship of user actions to code actions is obfuscated by a complex system of event handlers and function calls.

In this demonstration paper, we introduce ArtForm, a tool for understanding and analyzing the codebase of modern form-based websites. ArtForm allows a developer to explore the website via interacting with forms, linking these interactions with both the concrete and symbolic behavior of the underlying code. Since client-side scripting with JavaScript is used to add much of the interactive functionality to modern websites, ArtForm focuses on linking user activity and execution of JavaScript.

ArtForm uses an instrumented browser, based on the Artemis framework [2]. The browser tracks the low-level JavaScript instructions executing in response to user interface actions, producing execution traces. These traces include not only the concrete execution, but also symbolic information – tracking the relationship between values used in the code and the original input values. The execution can be driven by inputs that are manually-provided, or suggested automatically by ArtForm. The suggestions are either provided to the user for a semi-automatic exploration or used directly by the tool to generate further runs in a fully-automatic analysis. In the fully- and semi-automatic modes, the generation of input recommendations is done via *concolic analysis* [5, 16], which generates inputs that drive the execution to an as-yet unexplored branch of the code by tracking how those inputs can affect the control flow of the JavaScript code.

Organization. In the remainder of the document we first explain how ArtForm can help a developer to understand, analyze, and detect bugs in form-based websites. We then discuss the infrastructure behind the system. We close with a brief overview of the demonstration plan. *The demo can be seen in the screencast available at www.cs.ox.ac.uk/projects/ArtForm/demo/.*

2 USING THE EXPLORATION TOOL

We now present exploration via ArtForm from a user perspective. ArtForm has three modes: manual, concolic, and advice mode.

The most basic mode is the *manual mode*, where inputs are entered by the user. Manual mode displays a GUI view from ArtForm's instrumented WebKit browser. A 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 thus 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

ISSTA'17-DEMOS, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098226

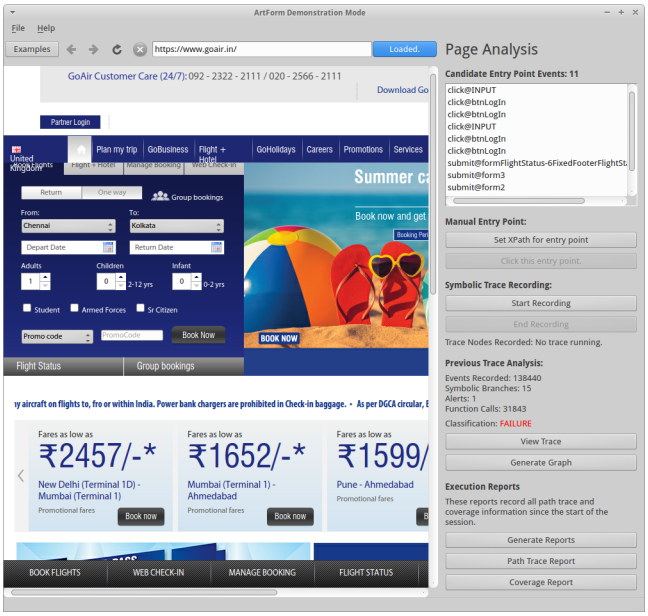

Figure 1: Manual mode

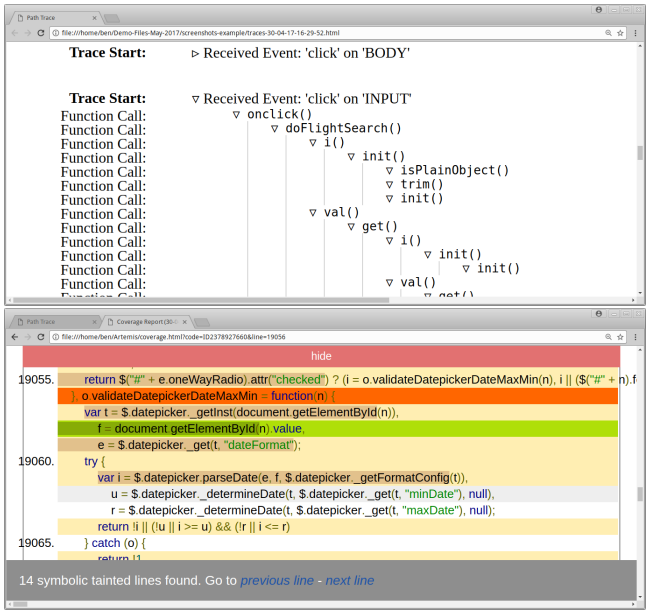

Figure 2: Reports

JavaScript code. These 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 hints are used to determine whether the interaction included a successful form submission. They are also used to detect JavaScript bugs, by checking for calls to `console.error`, or failed assertions.

Manual mode is useful for understanding which JavaScript code corresponds to each user action, and how that code depends on user inputs. Figure 1 shows manual mode in action. 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.

Figure 2 shows a path trace report and a coverage report for the form validation code of the airline flight search form shown in Figure 1. 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), which is fetching the `value` property of an input field.

ArtForm includes a proxy which reformats minimised and obfuscated JavaScript code on-the-fly. This makes the code easier to read, and also makes line-level statistics, such as the line coverage shown in the coverage report, more useful. Because JavaScript libraries such as jQuery are implemented in JavaScript, they are included in the reports and the analysis like any other JavaScript code.

Combining information from multiple runs in order to find bugs or perform other analysis is possible via *concolic mode*, which automatically generates inputs, aiming to explore new code. Initially, the page's default inputs are selected. After each run, the recorded symbolic trace consists of a sequence of *branch conditions* – the tests on input values, and the result of these tests (whether the concrete execution took the *if* or the *else* branch), that were performed during the run. Thus the traces from multiple runs form a tree of execution paths. The system chooses a branch condition from the tree where one of the possible outcomes has not yet been realized in any prior execution. It then uses a constraint solver to derive input values which lead the execution to this unexplored branch. The trace is then re-run with these new values. Concolic execution coupled with customized classification (e.g. of exceptions, as described earlier) can enable quicker discovery of bugs.

Finally, *advice mode* allows a user to mix suggestions from the solver with manual inputs and other heuristics. Inputs are chosen by the user, and actions are recorded symbolically, adding traces to the symbolic execution tree as in the fully automatic concolic mode. The user can ask for *advice* at any point about which inputs to try next. At this point a solver generates a set of values leading to a new execution path. The user may use this advice or choose their own values. By default, advice mode uses a fixed action sequence (e.g. filling the form in a certain order); and the advice is only about the values to enter. But ArtForm can also advise on the "natural order" on which to fill in form fields; using a dependency analysis of the code from previous traces, and looking for an ordering in which the code attached to each input field does not depend on values from fields that have yet to be filled in. The advice is available via an API for scripting or third-party tools.

Figure 3 shows a partially explored concolic tree from the advice mode exploring a simple web form. Nodes are highlighted based on what kind of code event they represent (e.g. a click event or a branch). The target of a user event such as a click is specified by XPath; for example the shaded boxes near the top of Figure 3 represent filling in a form field and clicking a submit button. The very first trace set the `myinput` field to the empty string and took the

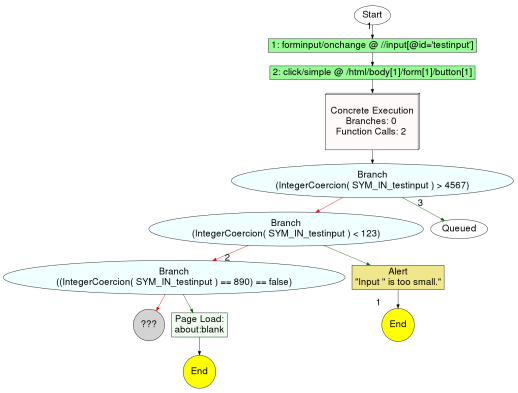

Figure 3: Partial exploration

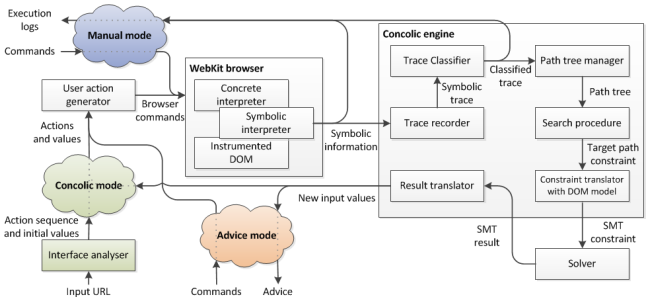

Figure 4: ArtForm architecture

path labelled *1*, leading to the alert found near the bottom right of the tree. The user requested advice from ArtForm, which suggested two options: repeating the trace with `myinput` set to either 123 or 4568. This marked the paths labelled *2* and *3* as *Queued* (i.e. suggested), but not yet explored. For the second iteration, the user chose 123 as the input value, and took the path labelled *2* (ending in a page load). This uncovered an extra execution path which had not been seen before (the leftmost leaf in the example), which is thus neither explored nor queued. The figure shows the state of the search after this second iteration. Note that at this point, only the path labelled *3* is still marked as *Queued*.

Tests on real-world websites show that the execution time of our analysis is dominated by the page loading and interaction; the solver's cost is negligible by comparison. For example, running the concolic mode on the airline example from Figure 1 for 50 iterations took 290.0s. Of this, only 1.7s is spent invoking the solver (50 times, giving 20 SAT and 30 UNSAT results), and the remainder is spent executing and recording 21 traces, with an average execution time per trace of 13.7s. The analysis time is very dependent on the complexity of a page and the amount of JavaScript used.

3 ARCHITECTURE

We now briefly describe how ArtForm performs symbolic execution. Figure 4 shows the components of the analysis platform.

A key component of all modes is the *instrumented browser*. The analysis platform needs both to know what is happening in the browser, and to control certain aspects of the browser. We build on top of Artemis [2], an existing web application testing framework, which itself is built on top of the WebKit browser engine. The browser engine includes the core functionality of a normal web browser, including page fetching, HTML and CSS rendering, and a JavaScript interpreter (called JSC or JavaScriptCore); but excluding the user-interface. Artemis adds instrumentation and hooks to WebKit which are useful for our analysis, providing low-level information about the page (such as the registered event handlers). Using a production web browser provides several benefits. The browser already provides infrastructure for downloading and interpreting web pages and their associated content (JavaScript, CSS, images, and so on). Many of the dynamic features of JavaScript, which pose the most difficult problems for static analysis, can be handled directly by the browser and do not need to be modelled in our analysis. Features such as dynamic code loading, calls to `eval`, and runtime modification of object signatures are implemented by the browser; our analysis simply records which JavaScript code is executed, and avoids modelling these dynamic features directly.

Our symbolic interpreter is an extension of the existing concrete interpreter JSC. Not every concrete operation must be modelled symbolically; some can pass along the existing symbolic information unmodified, which simplifies the interpreter implementation, and more importantly also simplifies the generated constraints.

Values in the interpreter initially have no symbolic value. As user input values are read from the DOM by JavaScript, they are tagged with a symbolic variable name. ArtForm's goal is to track how form inputs are used, so the `value` property of form fields is instrumented, as well as the `checked` property of checkboxes and radio buttons, the `selectedIndex` property of drop-down lists, and so on. This "symbolic tagging" is implemented by instrumenting the internal getter methods which implement DOM property lookups in WebKit. These getters are modified to return values with a symbolic tag showing from which input field that value originated. Each time a branch instruction (for example an `if` statement or a loop condition) is executed we check if it is a *symbolic branch*, i.e. whether the branch condition uses any symbolic value.

Built-in methods in JavaScript must also be instrumented. WebKit implements JavaScript's built-in functions with C++ methods internally: When the WebKit interpreter reaches a call to a built-in, it calls the corresponding C++ code in WebKit (external to the main interpreter) and passes the returned value back to the calling JavaScript code. In ArtForm the WebKit-internal methods are instrumented so they propagate symbolic values correctly.

Concolic engine. In concolic or advice mode, one needs to track multiple symbolic traces, as well as get new suggestions for values that will move towards unexplored code. This is the job of the concolic engine. It includes a search procedure, which is responsible for choosing an as-yet unexplored branch in the partial exploration tree as the next exploration target and generating a *path constraint*, a logical formula over the input values which, if satisfied, implies that re-running the same program (or action sequence, in our case) with those values will lead to exploring the specified execution path. We currently support both depth-first search and a search that prioritizes nodes which are most likely to lead to new code.

Once a path constraint is generated, it is translated into the constraint solver's input language. We also add *realizability constraints*

to model restrictions on the set of input values which are realizable by a user at the interface. For example, an input to a drop-down list is restricted to one of the values available from the drop-down by an extra constraint. We make use of the CVC4 constraint solver [3], which is well-suited to the types of constraints generated by ArtForm. It supports solving a rich variety of string constraints with useful built-in string functions, including substring extraction, find-and-replace, indexing, and regular expression tests. Critically, CVC4 also supports coercions between different types.

Advice mode. Advice mode tracks "queued" suggestions which have already been made but have not yet been tested. When a target branch 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. Now the concolic execution engine is free to operate as normal, recording new traces and making new suggestions, but without repeatedly suggesting previously suggested execution paths. When the main analysis decides to test a suggestion is was given, then the queued branch will become explored in the concolic tree.

4 DEMONSTRATION DETAILS

The demonstration will walk the user through the use of ArtForm's 3 modes, focusing on how the tool supports exploration of the code in an event-driven manner, automated analysis and testing of code. We will see both trace reports and coverage reports. In concolic and advice mode users will see not only the suggested values, but also some of the internals, including the automatically-generated browser events that are needed to simulate user actions, and the generated constraints whose solution corresponds to each suggestion. We will also show some of our preliminary experimental results, including (1) testing ArtForm on real websites; (2) benchmarking the concolic execution using both synthetic and real-world programs; (3) comparing ArtForm with Jalangi, another concolic execution tool for JavaScript; and (4) testing the form and DOM support using synthetically generated example forms.

5 RELATED WORK

Testing of form-based websites can be done using randomized and feedback-directed testing, importing methods used for other event-driven systems [1, 7, 8]. While these techniques can provide coverage of the user action space, concolic testing can give more coverage at the level of code. Concolic testing was first introduced for C with the DART [5] and CUTE [16] tools; later tools include SAGE [6] and KLEE [4], which provide many features for increasing the accuracy of concolic analysis. There are tools for static analysis of JavaScript [9, 11, 17], however, the dynamic nature of JavaScript makes static analysis problematic [12, 13].

SymJS [10] also attempts concolic execution on web JavaScript, based on an instrumented browser. The concolic execution includes many sophisticated features to reduce the search space. SymJS uses the open source Rhino JavaScript engine, which can only parse and interpret the JavaScript code from a limited number of real-world websites. In addition, there is no modelling of form restrictions (corresponding to the realizability constraints of ArtForm).

Jalangi [15] is a framework allowing instrumentation and runtime monitoring of JavaScript code. Jalangi could be seen as a base for testing applications such as ArtForm, working at the JavaScript source level rather than via an instrumented browser. An implementation of concolic execution for stand-alone JavaScript functions is included with Jalangi. One key limitation in the web setting is that Jalangi requires pre-processing of the JavaScript source, which can be time-consuming even when all source is available to the tester.

Kudzu is an automated test-generation tool for JavaScript-based web applications, based on concolic execution [14]. Although designed to generate tests for web applications, Kudzu does not appear to include modelling of the DOM, the browser APIs, or user inputs.

ArtForm builds on Artemis, a web application testing framework [2]. Artemis explores form-related code with random inputs or strings from the page's JavaScript, not via symbolic execution.

6 CONCLUSION

ArtForm provides a means for a developer or tester to explore, understand, and debug the event-driven code of a form-based website. Being based on an instrumented production browser, it faithfully models the actions of a live user. As ArtForm's instrumentation works at the level of bytecode, it does not require pre-processing of source, and can even work with third-party code. It allows a variety of interaction modes giving flexibility about the trade-off between user-guided exploration and fully automated testing. Its automation support is based on concolic analysis that includes modelling specific to form-based websites, limiting the automated exploration to actions that can be realized by a real user filling a form.

ArtForm is available to download at https://github.com/cs-au-dk/Artemis/blob/master/ArtForm.md.

REFERENCES

[1] S. Anand, M. Naik, M. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, 2012.
[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, 2011.
[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.
[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[5] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.
[6] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
[7] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *EuroSys*, 2014.
[8] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ISSTA*, 2013.
[9] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, 2009.
[10] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *FSE*, 2014.
[11] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *ECOOP*, 2015.
[12] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *ECOOP*, 2011.
[13] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010.
[14] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript, 2010.
[15] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *ESEC/FSE*, 2013.
[16] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[17] T.J. Watson Libraries for Analysis. WALA. http://wala.sf.net.