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

Ben Spencer
Michael Benedikt
Anders Møller
Franck van Breugel

## ABSTRACT

We describe ArtForm, a tool for exploration of the codebase of dynamic data-driven websites where users enter data via forms. ArtForm extends an instrumented browser, so that it can directly implement user interactions, adding on top of it 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.

**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 ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 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 will generally be 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 to a developer. 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, 17], 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. *A walk-through of the demo can be found 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 of ArtForm is the *manual mode*, in which inputs are entered by the user. Manual mode displays a GUI view via the instrumented WebKit browser on which ArtForm is based. 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 the recording of this interaction. There is a *trace report* and a linked *coverage report*, which shows the tree of function calls made, and the parts of the JavaScript source code that have 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 JavaScript code. These traces can include any events detectable by our instrumented browser to connect code execution with user interaction, such as whether a new page was loaded, 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.


*ISSTA'17, July 2017, Santa Barbara, CA, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

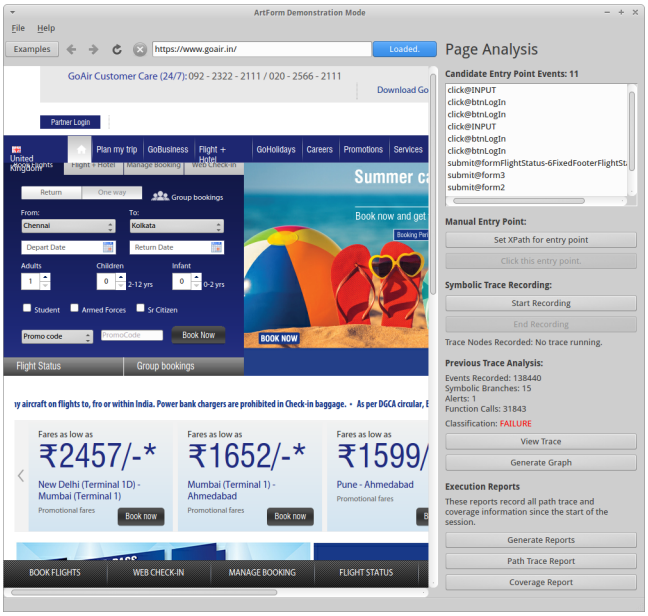


**Figure 1: Manual mode**

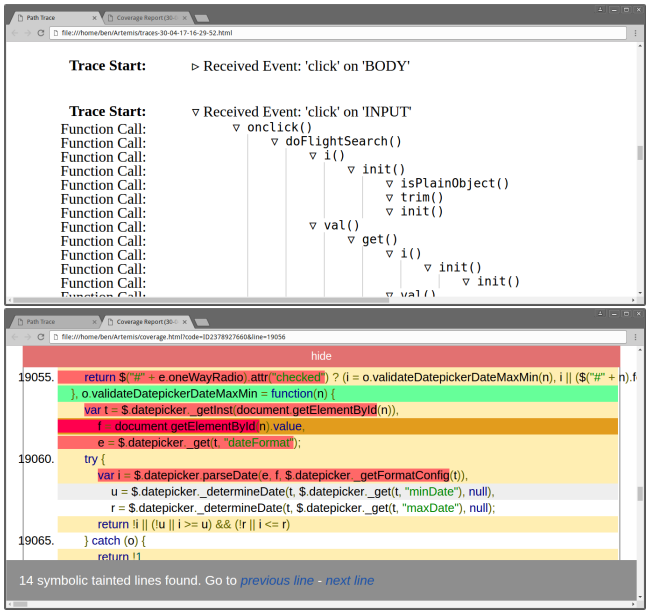


**Figure 2: Reports**

The manual mode is useful for understanding which JavaScript code is executed after certain user actions, and how that code depends on user inputs. Figure 1 shows manual mode in action. The user can see a web page in the main window, and on the right hand side is given the option to record interactions with the page. If this option is selected the user can type in values and the corresponding JavaScript events are recorded in a symbolic form as a *symbolic trace*. The user can either inspect an individual trace or look at a summary of the whole browser session by selecting the appropriate button on the right pane.

Figure 2 shows a path trace report and a coverage report for the form validation code of the same 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.

Combining information from a large number of runs in order to find bugs or perform other code analysis tasks is possible via *concolic mode*, which automatically generates interactions, driving towards exploring new code paths. 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 in one of the executions such that one of the possible outcomes has not yet been realized in any prior execution. It then uses a constraint solver to derive input values that should lead the execution to this unexplored outcome. The system 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. Concolic execution ties to the coverage reports to generate a profile of the code; this can also be customized to show particular code metrics.

Finally, *advice mode* allows a user to mix suggestions from the solver with either manual inputs or other heuristics. Inputs are generated by the user or program, but at any point the input actions can be *symbolically recorded*, causing the resulting trace to be added to the tree of symbolic traces, 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 is called to generate a set of values that will lead to a new execution path. The user is free to use this advice or to generate its own values. The default use of advice mode is when the action sequence (e.g. filling in the form in a certain order and clicking submit) is fixed; the advice is then only about the values to enter. But the mode can also advise on the "natural order" on which to fill in form fields; ArtForm does a dependency analysis of the code firing in previous traces, and looks for an ordering in which the code attached to each field does not depend on the values of fields that have yet to be filled in. The advice is also available as an API for scripting via applications.

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 in XPath – e.g. near the top of the figure are shaded boxes representing filling in a form field and clicking on a submit button. The very first trace set the `myinput` field to the empty string and took the 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

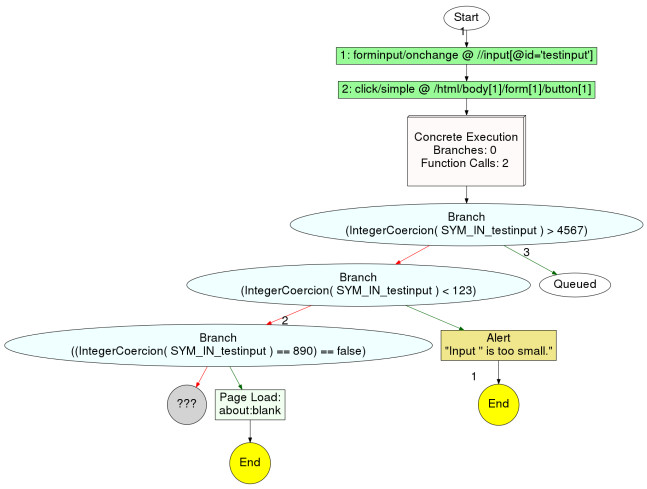


**Figure 3: Partial exploration**

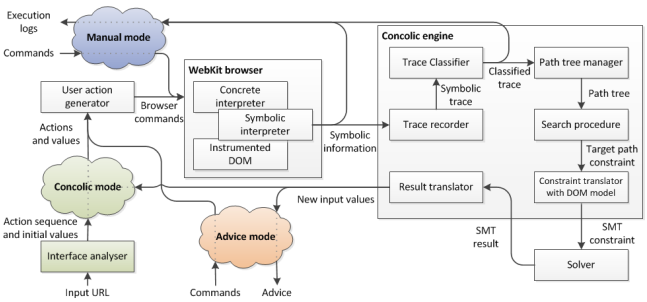


**Figure 4: ArtForm architecture**

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 tree), which is therefore unexplored and not yet marked as queued. The state of the search after this second iteration is what is shown in the figure. Note that at this point, only the path labelled *3* is still marked as *Queued*.

## 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. First, the browser already provides infrastructure for downloading and interpreting web pages and their associated content (JavaScript, CSS, images, and so on). The connections between JavaScript and HTML/CSS are already dealt with by a browser; this does not need to be re-modelled explicitly. Tying the symbolic infrastructure to the existing browser means that it is necessarily accurate in modelling the real system.

The symbolic interpreter is implemented as an extension to the existing concrete interpreter JSC. This means that our symbolic interpreter accurately reflects the concrete traces being executed, and not every concrete operation must be modelled symbolically. Some operations can pass through the existing symbolic information untouched, which simplifies the interpreter implementation, but more importantly also simplifies the generated constraints.

No values in the interpreter have any symbolic value initially. As user inputs are read from the DOM by JavaScript code, those values are tagged with a fresh symbolic variable name. ArtForm's goal is to track how form inputs are used, so the `value` property of form fields are symbolically instrumented, as well as the `checked` property of checkboxes and radio buttons, the `selectedIndex` property of drop-down select boxes, and so on. This "symbolic tagging" is implemented in our browser by instrumenting the WebKit-internal getter methods which are used to implement DOM property look-ups. These getters are modified to return values with a symbolic tag showing from which input field that value originated.

As the concrete value is used by the JavaScript program, the symbolic interpreter tracks its corresponding symbolic value. Each time a branch instruction (for example an `if` statement or a loop condition check) is executed we check if it is a *symbolic branch*, meaning that its 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 standard depth-first search and a search that prioritizes nodes which are most likely to lead to execution of new code.

Once a path constraint is generated, it is translated into the language of a target constraint solver. We also add various constraints that model restrictions on a set of values which are *realizable by a user at the interface*. For example, a constraint captures that the value of a field in a drop-down list must match one of the values. We make use of the CVC4 constraint solver [3] for the actual solving of constraints. CVC4 is well-suited to solving the types of constraint generated in ArtForm. It has strong support for solving

a rich variety of string constraints with useful built-in string functions, including substring extraction, find-and-replace, indexing within strings, and regular expression tests. Critically, CVC4 also supports coercions between different types.

**Supporting advice mode.** Supporting advice mode is achieved by keeping track of "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. Now the concolic execution engine is free to operate as normal, recording new traces and making new suggestions, but will not need to repeatedly suggest previously suggested execution paths. When the calling code decides to test a suggestion generated from the constraint solver, 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 in each of its 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 (1) the automatically-generated browser events that are needed to simulate user actions, and (2) the generated constraints whose solution corresponds to each suggestion.

*ArtForm is available to download from GitHub*[1].

## 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]. CrawlJax [11] tracks JavaScript events to model a website's user-interface states. While these techniques can provide wide coverage of the user action space, symbolic execution and concolic testing give more coverage at the level of code. Concolic testing was first introduced for C with the DART [5] and CUTE [17] tools; later tools include SAGE [6] and KLEE [4]. These tools provide many features for increasing the accuracy of concolic analysis; however they cannot be applied directly to JavaScript or to WebKit bytecode. There are tools for static analysis of JavaScript [9, 12, 18], however, the dynamic nature of JavaScript makes static analysis problematic [13, 14].

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. Due to the unavailability of the tool, we cannot compare the functionality of ArtForm directly with SymJS. However, SymJS is based on the open source Rhino JavaScript engine, and only a small fraction of real-world websites' JavaScript can be correctly parsed and interpreted by Rhino. In addition, there is no modelling of form restrictions (corresponding to the "realizability constraints" of ArtForm).

Jalangi [16] is a framework allowing instrumentation and runtime monitoring of JavaScript code. Jalangi could be seen as another way to implement a testing application 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 in the Jalangi distribution. One main limitation of Jalangi in the web setting is that it 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 [15]. 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 the instrumented browser of Artemis, a web application testing framework [2]. Artemis explores form-related code by generating random input values or taking static strings from the page's JavaScript code, 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.

[1] 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] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1), March 2012.
[12] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *ECOOP*, 2015.
[13] 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.
[14] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010.
[15] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript, 2010.
[16] 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.
[17] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[18] T.J. Watson Libraries for Analysis. WALA. http://wala.sf.net.