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

Ben Spencer  
University of Oxford, UK

Anders Møller  
Aarhus University, Denmark

Michael Benedikt  
University of Oxford, UK

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:

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 a-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
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 reformat 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
We now briefly describe how ArtForm performs symbolic execution. The browser, which itself is built on top of the WebKit browser engine, is an existing web application testing framework, which itself is built on top of the WebKit browser engine. 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. Most of the time 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 selected 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.
We will also show some of our preliminary experimental results, in-
cluding (1) testing ArtForm on real websites; (2) benchmarking the
accuracy of concolic analysis. There are tools for static analysis,
SAGE [13], and KLEE [14], 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 run-
time 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 implemen-
tation 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 de-
signed 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 frame-
work [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, un-
derstand, and debug the event-driven code of a form-based website.
Being based on an instrumented production browser, it faithfully
models 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 automa-
tion 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

testing of JavaScript web applications. In ICSE, 2011.
automatic generation of high-coverage tests for complex systems programs. In
OSDI, 2008.
In PLDI, 2005.
In NDSS, 2008.
study of the use of eval in JavaScript applications. In ECOOP, 2011.
study of the use of eval in JavaScript applications. In ECOOP, 2011.
concolic testing, selective record-replay, and dynamic analysis of JavaScript. In
ESEC/FSE, 2013.