Form Filling based on Constraint Solving

Ben Spencer[1], Michael Benedikt[1], and Pierre Senellart[2]

[1] Department of Computer Science, University of Oxford
[2] DI ENS, ENS, CNRS, PSL Research University & Inria Paris

Abstract. We describe a system for analyzing form-based websites to discover sequences of actions and values that result in a valid form submission. Rather than looking at the text or DOM structure of the form, our method is driven by solving constraints involving the underlying client-side JavaScript code. In order to deal with the complexity of client-side code, we adapt a method from program analysis and testing, concolic testing, which mixes concrete code execution, symbolic code tracing, and *constraint solving* to find values that lead to new code paths. While concolic testing is commonly used for detecting bugs in stand-alone code with developer support, we show how browser instrumentation can allow it to be applied to the very different problem of filling Web forms. We evaluate our system on a benchmark of real and synthetic forms.

1 Introduction

Finding data on the Web is useful for search, information extraction, and aggregation. The massive scale of the Web and the data itself means these tasks must necessarily be automated. Interesting data is often hidden behind user interfaces, and in particular *Web forms*. For example, airline or real-estate websites provide free access to their data via search forms, but there is no public API or standard way to access this information. This data, being "hidden" from Web search engines and other automated tools, makes up the *deep* or *hidden Web* [8, 21].

An automated tool accessing Web forms must find some sequence of user actions (for example clicking buttons or filling input fields) which leads to a successful form submission: one that leads to the target data. These actions might involve complex interactive user-interface elements such as drop-down lists, date pickers, and tabs. In addition, they must satisfy certain restrictions on the actions and inputs given such as mandatory and optional fields, and validation rules for input values. This combination of actions and input values creates a huge search space for form filling tools to consider. The restrictions on input actions and values are normally 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.

Much research attention has been devoted to effective, automatic form-filling [5, 25, 27, 31, 33, 47], or even further, to generate *wrappers* which use the form to look up and extract data [18, 41, 45]. A common target is search fields, which generally have no validation constraints [5, 27, 40]. Other work assumes domain knowledge, encoded in heuristics or rules [18]. For complex fields in the absence of domain knowledge, approaches to finding field values include sampling text on the page [29], and the application of machine learning techniques [25, 35]. In the absence of both domain knowledge and a corpus of examples, these approaches cannot infer constraints enforced by client side code, and thus cannot find satisfying values. We thus look to supplement these approaches with *constraint-driven form-filling*, which analyzes client-side code to determine constraints being enforced on form actions and values, and then solves these constraints to yield a successful submission. Constraint-driven form filling would allow form exploration tools to take advantage of the rapid maturation of constraint-solving technology [6, 13, 28].

Example 1. Consider an airport form, which includes fields From and To (with identifiers "from" and "to", respectively) with values populated from a drop-down list, along with field Date (with identifier "date") populated by a date picker. A snippet of the form's validation code is shown in Listing 1, where validate_to is attached to the To field, and validate_date is attached to the Date field. For To, the code checks that the From field has already been filled and that it is not equal to To. For Date it checks that To is already filled and that the date entered is later than today's date (for simplicity, all dates are assumed to be in the current year).

Listing 1 Validation code for the example airline form.

```
function validate_to() {
    var from = document.getElementById("from").value;
    var to = document.getElementById("to").value;
    return validate_aux(from, to);
}
function validate_aux(from, to) {
    if (from.length == 0) { alert("Error: Departure airport must be set"); return false; }
    if (from == to) { alert("Error: Departure must differ from Destination"); return false; }
    ... return true;
}
function validate_date() {
    var to = document.getElementById("from").value;
    var date = document.getElementById("date").value;
    if (to.length == 0) { alert("Error: Destination Airport must be set"); return false; }
    var day = parseInt(date.substr(0, 2), 10), month = parseInt(date.substr(3, 5), 10);
    var today = new Date();
    var valid = (month>=today.getMonth()+1 && (month!=today.getMonth()+1 || day>=today.getDate()));
    if (!valid) { alert("Error: date cannot be before today"); return false; }
    ... return true;
}
```

Note that the validation code involves restrictions on both the values of the fields, and on the order in which they are filled. In addition to the constraints explicitly enforced by the event handlers, there are a number of implicit constraints on the field: for example, From and To are implemented by drop-down lists, so the values can only be chosen for a certain pre-determined set. A form-filling or wrapper-generation tool would need to find values that satisfy all of these constraints.

Static analysis of the underlying code can in principle determine the set of restrictions enforced by the code (and thus values leading to form submission). Unfortunately, analyzing JavaScript code used on the Web is notoriously difficult [7]. Thus a popular intermediate position in testing JavaScript code is *dynamic analysis*, with a well-known approach being *concolic testing* [20, 44]. Concolic testing applies to a stand-alone function, with the goal of generating test inputs that explore all paths within the code. First, concrete starting values (e.g., randomly selected) are chosen, and the function is executed with these values as its input. The execution is *symbolically traced*, recording all *symbolic branches*: branching statements where the branch condition involves a comparison with a value depending on one or more of the inputs. The system then looks for a symbolic path that has not yet been exercised, and uses a constraint solver to generate values that will reach that path. This process is repeated to cover progressively more code, and hence exercise a wider set of behaviors.

We introduce FormSolve, which applies the idea of mixing execution, symbolic tracing, and constraint solving to the exploration of Web forms. FormSolve will generate input actions and values for Web forms, symbolically trace the code executed while processing these inputs, and then solve constraints that will lead to new inputs which reach new code paths.

Applying constraint-solving and symbolic tracing to form-filling involves many challenges that are not encountered in concolic testing of standalone code. In order to inject values and symbolically trace the resulting code, we require new infrastructure for controlling and monitoring browser behavior. When generating constraints for the constraint solver, we are not merely describing input values for a single function, but sequences of user events that may trigger a set of event-handling functions. In order to get interesting output, our goal is not to generate arbitrary input values (as in exhaustive testing), or interesting corner cases (as in functional testing), but rather values that a typical user might provide via the interface.

FormSolve deals with each of these challenges. We propose a refinement of concolic testing tuned for Web forms. Instead of using a constraint solver to find input values for a single function, our concolic testing algorithm discovers sequences of user-actions and their corresponding input values together. We introduce browser

Algorithm 1 The high-level algorithm for concolic testing.

```
1: procedure CONCOLIC-ANALYSIS(program)
2:     values ← CHOOSE-INITIAL-VALUES( )
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-UNREACHABLE(path-tree, target-path)
13:        end if
14:    end while
15: end procedure
```

infrastructure that allows the symbolic tracing of event-handling code, which can be written either directly in pure JavaScript or using popular libraries such as jQuery. It also provides fine control over the browser, which is necessary for faithfully—and deterministically—emulating user form-filling actions. We provide a constraint-generation technique that allows us to focus on *user-realizable values*, thus accelerating the discovery of interesting form submissions. We evaluate our technique on both synthetic and real-world forms, comparing it to alternative means of form-filling.

Related work. Indexing and extraction from deep Web sites involves a number of tasks, including entry point finding [4, 34, 35], form label identification [17, 38, 48], form-filling [5, 25, 27, 31, 33, 47], and result page analysis [11, 49]. Work on form-filling is focused on finding values that extract a good set of results. For example, some work attempts to choose keywords for a text field which return relevant results [5]. Tools such as Crawljax [37] and AJAX Crawl [14] take into account the state of the user interface while crawling. The emphasis there is in identifying changes in the DOM (*Document Object Model*, the tree structure representing the content, structure, and styling properties of a Web page) caused by user actions and events, not on getting through forms. Our work is on form-filling, but the focus is only on getting results that satisfy validation rules. Additional desired properties can be overlaid on top of this. One of our challenges is choosing an interesting ordering for the form filling events. This challenge also arises in the analysis of other event-driven systems, such as detecting bugs in Android apps [1, 22, 24].

Concolic testing is a well-established topic, applied to many programming languages, including JavaScript. Still, there are few attempts to apply it to Web JavaScript. SymJS [30] uses concolic analysis, but is based on the Rhino JavaScript engine, which can parse only a small fraction of real-world Web JavaScript. Jalangi [43] is a framework allowing instrumentation and runtime monitoring of JavaScript. Neither SymJS nor Jalangi support Web forms. The only two systems we are aware of for exploring forms via some form of JavaScript analysis are the demonstration systems ProFoUnd [7] and ArtForm [46] (the precursor to this work).

2 Concolic testing and adaptation to Web form exploration

Concolic testing, or directed automated random testing [20, 44], is a testing technique which uses concrete executions of a program to drive a symbolic analysis. 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 not understood, or only partially understood, by the symbolic analysis alone.

A generic concolic testing algorithm is given in Algorithm 1. Concolic testing begins by choosing some default starting values for the variables. The function is then executed concretely (that is, using a real interpreter) and 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 condition*: the set of individual branch conditions (expressed

in terms of the input values) which must be satisfied or unsatisfied in order for the program execution to follow that particular path. Each path condition 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 characterized by the set of path conditions of explored traces, which form a tree. The testing algorithm proceeds in an execute-and-analyze loop. After execution of a trace, the new information gathered during that trace is added to the tree. 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 been explored, generates the corresponding path condition, and sends it to a *constraint solver*. 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.

Example 2. Consider the function validate_aux(from,to) used in event handler validate_to() of the running example. Concolic testing would first execute the program on random values for the arguments from and to, say empty strings for both. This would bring the program to a trace that terminates after the first alert. Tracing this path symbolically, the algorithm identifies it with the constraint $\mathsf{from.length} = 0$, which is the first path added to the tree (line 4). The search command on line 6 will isolate the path consisting of the single constraint $\neg(\mathsf{from.length} = 0)$ as an unexplored path of the code, and this will be sent to a constraint solver (line 7), which will return values for from and to that it. For example, the solver might return the values $\mathsf{from} = \text{'a'}$ and $\mathsf{to} = \text{'a'}$. In the next iteration, these values are used to execute the function, which drives the code to the second alert. This second execution is symbolically traced, and associated with the path $\neg(\mathsf{from.length} = 0) \wedge \mathsf{from} = \mathsf{to}$. The search procedure on line 6 will now select $\neg(\mathsf{from.length} = 0) \wedge \neg(\mathsf{from} = \mathsf{to})$ as an unexplored path to target. Solving this constraint will give values that drive the execution to avoid each alert.

Note that concolic testing normally includes *classification* of a trace—in the context of testing, this would be determining whether an error occurs in it. We have omitted this in Algorithm 1, since the classification can be done offline. In our application to form crawling, 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 watching for certain commands such as alert, or checking for certain text in the DOM.

Adaptation to form exploration. In adapting concolic testing to exploration of Web forms, a tempting analogy is that the form represents a "virtual function", whose arguments are the form fields. However, a function takes its arguments all at once, whereas a Web form takes inputs one at a time interactively. The code which responds to a form filling can be triggered by a number of events, such as filling fields or hovering over them. These event handlers may interact, and the order in which they are fired can affect the output. Thus a more realistic model of a form is as an association of each field f with one or more actions Act_f, where an action writes the value v_f into the field and executes some program code. In using concolic testing to explore a form, we must discover both the values to put in form fields and *also* the ordering of these actions.

Our algorithm is shown in Algorithm 2. We explain all the details, and in particular the constraint solving call on line 12, further below. For now we note that it is similar in structure to a classic concolic algorithm. The high-level distinction is that we explore a tree of branches for each action. At each iteration we choose an action with some unexplored code, and then generate both a value for each form action and an action ordering.

We now formalize the problem of simultaneous detection of input values and the ordering of actions. We use an extremely idealized model to explain the details of Algorithm 2. Although the assumptions of this model do not hold on real Web forms, the approach is still effective.

In the model we have a set of *form actions*, each with a corresponding program which manipulates a set of variables $\vec{v}$, including one distinguished variable v, the *input variable* of the form action. Informally this is the value supplied by the user when invoking this action. Each variable v has a *default value* D (provided by the Web page's HTML) which it holds until the action is run. In our basic model, we assume program code in a simple procedural language built up from the following grammar:

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

Above τ ranges over terms built up from set of atomic functions (e.g., $+, \times$) from variables and constants, while φ ranges over some set of conditions (e.g., Boolean combinations of atomic conditions $\tau_1 \{\leqslant, \neq, =\} \tau_2$,

Algorithm 2 The high-level algorithm for form exploration.

procedure FORM-CONCOLIC-TESTING(*form-actions*)
  initial-order, *initial-values* ← CHOOSE-INITIAL-ORDER(*form-actions*)
  trace ← EXECUTE-AND-RECORD(*initial-order*, *initial-values*)
  path-tree ← INIT-TREE(*trace*)
  if *trace* is terminating **then**
    Mark all local symbolic paths within *trace* as Known-Extendible
  else
    Mark those local symbolic paths which aborted as Known-Unextendible
  end if
  while $\exists \mathsf{Act} \cdot \mathsf{Unresolved}_{\mathsf{Act}}(\textit{path-tree}) \neq \emptyset$ **do**
    target ← CHOOSE(sympth, Act) with sympth $\in \mathsf{Unresolved}_{\mathsf{Act}}(\textit{path-tree})$
    order, values ← SOLVE-PATH-CONSTRAINT($\mathsf{OverApprox}_{\mathsf{Act}}(\textit{target})$)
    if *target* was successfully solved with *order*, *values* **then**
      trace ← EXECUTE-AND-RECORD(*order*, *values*)
      EXTEND-TREE(*path-tree*, *trace*)
      if *trace* is terminating **then**
        Mark all local symbolic paths within *trace* as Known-Extendible
      else
        Mark those local symbolic paths which aborted as Known-Unextendible
      end if
    else
      Mark *target* as Known-Unextendible
    end if
  end while
end procedure

where τ_i are terms). C in do_C ranges over some set of commands that do not impact submission or control flow. 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*. The semantics of the language are standard. Given an expression E and a binding σ for the free variables of E, the semantic function returns the sequence of ground conditions and atomic actions Abort and do_C that are generated during an execution. We will be particularly interested in the truth values of conditions 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 by σ. The *trace* of E on an assignment σ to the free variables of E is the sequence of conditions encountered and their values. If E executes Abort at any point when running on σ it is said to *abort on* σ, while otherwise we say it *terminates* on σ.

We name our actions by numbers, with Act_i denoting the program associated with action i, while v_i and D_i denote the distinguished input variable and default value of Act_i, respectively. A set of such indexed actions $\mathsf{Act}_1 \ldots \mathsf{Act}_n$ has *restricted global state* if for every Act_i, each free variable v occurring in it's program expression is one of the input variables v_j and further no input variable is ever assigned. That is, the actions do not have any shared global state except the input variables, which are only set as each action is executed.

The trace of an action on a binding σ is simply the trace of its program code expression. The behavior of a single expression is well-defined given a binding for all variables. We now need to explain the outcome of a sequence of form actions, which involves binding each free variable v encountered in conditions to either the user-specified input value of v or to its default value.

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 $(\mathsf{Act}_1, c_1) \ldots (\mathsf{Act}_n, c_n)$ as the sequence $(E_1, \sigma^1), \ldots, (E_n, \sigma^n)$, where σ^i is the *order-modified assignment* mapping v_j to c_j if $j \leqslant i$ and to D_j otherwise. 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. A form input is said to abort if some E_i aborts on σ_i; otherwise we say it terminates.

We now extend our discussion from concrete traces to symbolic descriptions of these traces. We start with a discussion of symbolic descriptions of "local behavior": a form action, given a binding for all variables.

For binding σ to the free variables of expression E, the *local symbolic path* of $E(\sigma)$ is a formula describing the values of conditions in the trace of $E(\sigma)$. That 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 in executing E on σ.

We have just discussed the symbolic paths that emerge from concrete traces. We now describe formulas that represent possible concrete traces that we would like to discover. Given a trace t for an action, consisting of conditions $(\varphi_1, \mathsf{TVal}_1) \ldots (\varphi_k, \mathsf{TVal}_k)$, where TVal_i is the truth value for condition φ_i, a *symbolic modification* is a local symbolic path of the form

$$(\varphi_1, \mathsf{TVal}_1) \ldots (\varphi_p, \mathsf{TVal}_p), (\varphi_{p+1}, \neg\mathsf{TVal}_{p+1})$$

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 in T that satisfies this condition. Given set of traces T and action A, we let $\mathsf{Unexplored}_A(T)$ be the set of symbolic modifications of traces in A that are unexplored.

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 trace to A. Let $\mathsf{Abort}_A(T)$ be the traces in $T(A)$ that are aborting, and $\mathsf{Terminate}_A(T)$ the set of traces that are terminating. By $\mathsf{Symbolic}(\mathsf{Abort}_A(T))$ we denote the set of local symbolic paths of traces in T that are aborting in A, and similarly for $\mathsf{Symbolic}(\mathsf{Terminate}_A(T))$.

Above we have a symbolic version of the behavior of a single form action A, describing it via conditions. We now need to lift this to a sequence of form actions. In doing this we have to take into account the role of the ordering in determining whether we use the default value or the user-supplied value. Further, since we are interested in determining whether a path is explored in a terminating trace, we have to track symbolically whether other actions abort.

Given a local symbolic path $\psi(\vec{v})$ of a trace for Act_i, the *ordered version of* ψ, denoted $\mathsf{Ord}_i(\psi)$, is the formula ψ with each variable v_j is replaced by v'_j and conjoined with the constraint

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

using an additional relation $\preceq$. That is, $\mathsf{Ord}_i(\psi)$ is a symbolic representation of an ordering and a binding σ such that ψ holds on the order-modified assignment σ^i defined above.

Given a path sympth for action Act_i, and a set of concrete traces T, we let $\mathsf{OverApprox}^i(\mathsf{sympth})$ be the formula $\mathsf{Ord}_i(\mathsf{sympth}) \wedge \mathsf{OrderAx} \wedge \bigwedge_{j \neq i} \mathsf{MayTerminate}_j$, where $\mathsf{MayTerminate}_j$ is

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

and $\mathsf{OrderAx}$ is a formula asserting that $\preceq$ is a linear order on the indices of actions $1 \ldots n$. That is, the formula $\mathsf{OverApprox}^i(\mathsf{sympth})$ symbolically represents the orderings and values that will achieve the behavior sympth in action Act_i and will not drive any other action to a known-aborting trace of T. Informally, it describes form inputs that may explore sympth without aborting in any action, based on the current knowledge of aborts in T.

We similarly let $\mathsf{UnderApprox}^i(\mathsf{sympth})$ be the formula $\mathsf{Ord}_i(\mathsf{sympth}) \wedge \mathsf{OrderAx} \wedge \bigwedge_{j \neq i} \mathsf{MustTerminate}_j$, where $\mathsf{MustTerminate}_j$ is:

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

This is a formula representing orderings and values that will achieve the behavior sympth in action Act_i and will drive every other action to known-terminating trace of T. Informally, this describes form inputs and orderings that we are sure will explore sympth based on what we know about aborts in T. The following lemmas give the critical properties of these symbolic descriptions.

Lemma 1. *If* $\mathsf{UnderApprox}^i(\mathsf{sympth})$ *is satisfiable by values* $\vec{c}$ *and ordering* $j_1 \dots j_n$*, then the form input* $(\mathsf{Act}_{j_1}, c_{j_1}) \dots (\mathsf{Act}_{j_n}, c_{j_n})$ *gives a trace that does not abort outside of action i and extends* sympth*.*

Lemma 2. *If* sympth *is a symbolic modification for* Act_i*, and form input* $(\mathsf{Act}_{j_1}, c_{j_1}) \dots (\mathsf{Act}_{j_n}, c_{j_n})$ *generates a trace that does not trigger an abort outside of* Act_i*, and which extends* sympth*, then* $\mathsf{OverApprox}^i(\mathsf{sympth})$ *is satisfied by values* $\vec{c}$ *and ordering* $j_1 \dots j_n$*.*

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 traces and their symbolic modifications, which form a tree. We also classify the symbolic paths and their modifications. We distinguish the *known-extendible* paths, those which 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-unextendable are said to be *unresolved*. The set $\mathsf{Unresolved}_{\mathsf{Act}_i}(T)$ contains the local symbolic paths from T which have not been marked as known-extendible or known-unextendible, as well as the symbolic modifications of local symbolic paths in T 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 whether $\mathsf{OverApprox}^i(\mathsf{sympth})$ is satisfiable. 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 $\vec{c}$ and ordering $j_1 \dots j_n$, 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 by Lemma 1 we know that $\vec{c}$ and $j_1 \dots j_n$ must not have satisfied $\mathsf{UnderApprox}^i(\mathsf{sympth})$. Therefore, there must be some other action Act_j for which t does not extend an explored branch, and where t aborts. That is, in Act_j, t follows a previously unexplored path and discovers an abort. Thus one local symbolic modification is resolved in Act_j (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 symbolic 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 1. *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.*

Example 3. Let us illustrate Algorithm 2 on Example 1. 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 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 $\mathsf{from.length} = 0$, and for $\mathsf{Act}_{\mathsf{Date}}$ it is $\mathsf{to.length} = 0$. These constraints are added to the corresponding path trees for each action. The command on line 11 will then choose an unexplored path to target, suppose it is $\neg(\mathsf{to.length} = 0)$ in $\mathsf{Act}_{\mathsf{Date}}$. 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 MayTerminate, which in this case (when simplified) is simply the negation of the single abort 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 12 will solve this combined constraint for both an order and values. The only valid order is $\mathsf{From} \prec \mathsf{To} \prec \mathsf{Date}$. Suppose the returned values are $\mathsf{To}_1 =$ "A", $\mathsf{From}_1 =$ "A", and $\mathsf{Date}_1 =$ "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 $\neg(\mathsf{from.length} = 0) \wedge \mathsf{from} = \mathsf{to}$ for $\mathsf{Act}_{\mathsf{To}}$, which are added to the corresponding trees on line 15. In the second iteration of the **while** loop we would choose another unexplored path (line 11), and this would return, for example, the path $\neg(\mathsf{from.length} = 0) \wedge \neg(\mathsf{from} = \mathsf{to})$ 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, the MayTerminate constraint is required to avoid both aborts in $\mathsf{Act}_{\mathsf{Date}}$. Thus, the full over-approximation constraint, omitting the linear order constraints, simplifies to:

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

where m_1 and d_1 are the concrete date and month numbers observed during the 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.

The completeness result is based on strong assumptions. It requires completeness of the solver, and requires the analysis to track all conditions symbolically. The code must also conform to the simple structure in which form actions set their input variables but otherwise do not update global state shared by other actions. Real-world code does not obey these assumptions, but we can still apply Algorithm 2 on arbitrary code, tracking only the input fields symbolically across actions, while dropping the completeness guarantees.

3 Implementing concolic testing for form exploration

Implementing concolic testing in the Web context is challenging. First, one needs to control the browser, both to simulate user actions faithfully and ensure that the browser behaves deterministically from one iteration to the next. Frameworks such as Selenium WebDriver [42] support this, but give limited control over certain low-level events such as timers and AJAX events. Second, we need to get information from the browser, and in particular need to analyze the executed code to record the symbolic paths taken. There are frameworks which can be used for symbolic tracing of stand-alone JavaScript, with the goal to help testing and debugging [43]. However, applying these to third-party JavaScript on the Web can be problematic, since they require instrumentation of the JavaScript source, which is not available when crawling of third-party sites.

FormSolve's approach works directly with a browser engine. We build on top of WebKit, an open-source production Web browser engine, used by Apple's Safari browser. The browser engine includes page fetching, HTML and CSS rendering, and a JavaScript interpreter, but excludes the browser's user-interface. We instrument the source code to add hooks to control the browser, and we instrument the interpreter to perform symbolic-tracing, extending the WebKit interpreter to track a symbolic value with every concrete value. WebKit's JavaScript interpreter uses an internal JavaScript bytecode language, so our symbolic tracing works at the *byte-code level*, rather than the JavaScript source level. This simplifies the generated constraints.

The symbolic interpreter runs alongside WebKit's existing concrete JavaScript interpreter and computes the symbolic values used in the analysis. When values are read from form fields, they are tagged as symbolic, and these symbolic values are propagated by the symbolic interpreter as the input values are processed. A branch instruction is called symbolic if its branch condition uses any symbolic value. That is, if it includes any value derived from a symbolic input field. When a symbolic branch is encountered, the symbolic condition is extracted and passed to the analysis.

Symbolic values are created when certain properties of DOM objects are accessed by the site's JavaScript code. For example the value property of text fields, the checked property of checkboxes and radio buttons, or the value or selectedIndex properties of drop-down boxes. To make certain DOM properties symbolic, the WebKit

Table 1. JavaScript level, bytecode-level, and symbolic execution.

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

implementation of the DOM API is instrumented. In WebKit, a DOM property access in JavaScript is implemented by a corresponding C++ function in the WebKit DOM implementation which returns the appropriate value. These getter functions are instrumented to return symbolically tagged values.

The concrete and symbolic interpreters run in lock-step to propagate the symbolic values during execution. When the concrete interpreter executes an instruction, it also calls the corresponding symbolic instruction in the symbolic interpreter. JavaScript's built-in methods must also be instrumented, separately from the interpreter. These are implemented by C++ methods in WebKit, which are modified in FormSolve to add a symbolic tag to the return value. Not all built-in functions are instrumented. We have only instrumented 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, and thus allow analysis to continue past certain difficult functions or patterns which cannot easily be encoded as SMT constraints.

Note that JavaScript library functions and any other functions which are not JavaScript built-ins are themselves 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.

Example 4. We explain our symbolic tracing on the following JavaScript snippet adapted from Example 1:

var from = document.getElementById(*"From"*).value;
if (from.length == 0) { alert(*"Error: Departure Airport must be set"*); }

Table 1 shows the corresponding JavaScript bytecode, the resulting register-level state change, and the changes in the symbolic state generated by the symbolic interpreter. For example, the first line of the table shows that the initial JavaScript command generates two bytecode instructions: **op_call** r1 getElementById *"from"*, which calls the document.getElementById function to look up the DOM node with identifier "from" and store it in the register r1, followed by **op_get_by_id** r2 r1 *"value"*, which fetches the "value" property of the DOM object in r1 and stores 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 corresponding symbolic instrumentation will set r2 to the symbolic value of the property lookup, which will be **SymStr**(*"from"*), representing a symbolic input originating from a field with identifier "from".

Finding user-realizable values. Line 12 of Algorithm 2 uses the constraint solver to generate a new set of input values. In the context of form filling, we want to avoid inputs that could not be performed by a user at the interface, since these are unlikely to produce a useful output. We thus need to add additional constraints on the value space that enforce *user realizability*.

HTML permits various types of input fields, each with their own restrictions on which values can be input. For example a *select* element produces a drop-down list with fixed options for the user to select from. To produce user-realizable input values, the analysis models these input fields to only generate values a user would be able to provide using a normal Web browser. This is done by encoding DOM facts as extra constraints which are included with each path condition.

Example 5. In the running example, the To field is implemented by a drop-down to choose between a set of airport codes: JFK, ORD, etc. In any constraint involving the corresponding variable to, we add a constraint saying $\mathsf{to} = \mathrm{JFK} \vee \mathsf{to} = \mathrm{ORD} \vee \ldots$, where the list of codes is populated from the DOM. The client-side code may also check the index of the selected item, rather than its value. In this case, we also add a variable to_index to represent this index, which must correspond to the main to variable. In our example, the extra constraints

would become $(\mathsf{to} = \mathrm{JFK} \wedge \mathsf{to\_index} = 15) \vee (\mathsf{to} = \mathrm{ORD} \wedge \mathsf{to\_index} = 23) \vee \ldots$, forcing the solver to choose a matching index and value.

Constraint solving. A critical component in the architecture is the constraint solver. In general, concolic testing requires a high-performance solver which supports standard program variable types: integers, Booleans, reals, bit-vectors, and arrays. Some examples of solvers used for concolic analysis or symbolic execution include Boolector [39], STP [19], Yices [15], and Z3 [12]. JavaScript requires very strong support for reasoning about strings, so this is the most important feature for our application. In particular transformations between strings and other data types are common, so it is very useful if the solver supports these datatype coercions. We made use of *CVC4*, which supports a wide variety of theories compared with other solvers [9], strong string support [32], as well as support for constraints which involve coercions between multiple theories.

Using constraint solving to detect new relevant actions. Algorithm 2 assumes that the set of actions relevant to submitting the form is known in advance. JavaScript events are typically registered on the DOM element they affect, either using HTML attributes such as onclick or the addEventListener function. These events can be identified by inspecting 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 static action detection is not possible. Many JavaScript libraries, including jQuery, provide an alternative event registration mechanism known as *event delegation*. Using these libraries, events can be registered on all elements matching a certain CSS selector, even if new matching elements are added to the DOM dynamically. This is implemented with a single top-level event handler registered at the root of the document whose handler contains a lookup table of CSS selectors and corresponding callback functions. When any element on the page is clicked, a process called *event bubbling* (a standard part of the DOM 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 object (which is a reference to the element which was actually clicked) against its list of CSS selectors. For any matching selector the corresponding callback is executed, which runs the developer-provided event handler for the clicked element. Under this scheme only one 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, because the event's target does not match any of the CSS selectors.

It is possible to observe the top-level event handler checking the target element for certain properties. Thus, concolic testing can be applied to dynamically determine which DOM elements would trigger application-level event handlers and are thus relevant to analyze. We consider the event handler associated with the top-level element to have an additional argument target, representing the DOM element that triggered the event. We instrument the browser to make the target property of the event object in the top-level event handler 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 a symbolic branch involving the variable target. The concolic testing procedure can then explore the event handling code, selecting different target elements as the input, until one is discovered which satisfies the constraints and triggers the application-level event handler. As with drop-down lists (see Example 5) we generate extra constraints to restrict the new input element to real elements from the page's DOM.

Example 6. In the running example, assume that the validation code for the Date field is registered using jQuery. The jQuery events are registered on the document element, at the root of the DOM. The event-handling code at the root delegates processing to validate_date if the user has changed the field Date by including the line:

$(document).on("change", "#date", validate_date);

where #date is the CSS selector of the Date field. Initially FormSolve sees only the event handler on the root, and thus triggers a change event on the root. The concrete jQuery event-handling JavaScript code responds to this event by checking whether the event's target node (that is, the node where the event originated) matches the CSS selector #date. Our instrumented browser makes all properties of the target node symbolic, so the symbolic interpreter generates the symbolic branch:

StrBinOp(**SymProp**(**SymObj**("target"), "id"), **STR_EQ**, **ConstStr**("date")).

To satisfy this condition, FormSolve will choose the Date field as the event target, and thus discover—and later explore—the application-level handler validate_date.

Table 2. Comparison of FormSolve with Jalangi on 1000 synthetic JavaScript programs.

	FormSolve	Jalangi
Avg. no. distinct paths explored	5.75	2.54
Avg. line coverage	91.49%	87.69%
Examples with full line coverage	46.44%	37.05%

Table 3. Comparison of FormSolve with Crawljax and Artemis on 1000 synthetic forms.

	FormSolve	Crawljax	Artemis
Analysis time (s)	969	272 153	2 206
Avg. iterations to submit	1.86	40.95	43.29
Ex. with full line coverage	10.4%	—	0.7%
Successful submissions	31.4%	19.8%	17.7%

Table 4. Comparison of FormSolve with Crawljax and Artemis on 18 examples from JSFiddle.

	FormSolve	Crawljax	Artemis
Analysis time (s)	2 245	6 509	319
Avg. iterations to submit	26.17	33.67	45.94
Successful submissions	61.1%	33.3%	11.1%

Table 5. Comparison of Algorithm 2 with a single static order 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%

4 Experimental Evaluation

Standalone JavaScript code. The first goal of our evaluation was to compare our bytecode-based code exploration with alternatives based on JavaScript source. We compared against Jalangi [23, 43], a framework for JavaScript testing that includes concolic testing for standalone (that is, not Web-based) JavaScript. We compared the tools on 1000 synthetic, randomly-generated standalone JavaScript programs. To generate the examples, we began with a context-free grammar for the program expressions, and then used a standard CFG-sampling algorithm [36] to generate random expressions from it. We then choose a random implementation for each operation in the generated program skeleton, choosing from a fixed set of simple JavaScript implementations. For example, for a term $a + b$, we choose between a single-expression implementation adding the two child expressions and one which makes use of intermediate variables; for a Boolean operation such as conjunction we choose between a direct implementation using a primitive and an implementation using conditionals. The aggregate results are shown in Table 2.

We see that FormSolve explores many more paths than Jalangi. FormSolve benefits from symbolic instrumentation of a number of built-ins, native string operations, and datatype coercions which are not supported (and may be difficult to support) at the source level in Jalangi. There is also some benefit from FormSolve using CVC4, which has more advanced solving capabilities than the earlier version of CVC used by Jalangi.

Synthetic Web forms. The second goal was to test FormSolve's ability to deal with complex Web forms. We ran FormSolve on a suite of 1000 randomly generated test forms. The generator first produces a random form, and then supplements it with validation code using the JavaScript generator described above. Each field's validation code uses the value of its own field, and optionally those of other fields. The generated forms use an average of 8.57 form fields and 47.16 lines of JavaScript validation code. We compared FormSolve with two alternative approaches. Crawljax [10, 37] is an automatic crawler for dynamic websites which is based on a dynamic analysis of the Web application. In particular, Crawljax tracks changes to the state of the DOM, and explores user events 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 websites, including Web forms. Artemis [2, 3] is a tool that does feedback-directing testing of websites. In each iteration, Artemis generates DOM and JavaScript events as well as values to enter in form fields. It uses metrics like line coverage to rank actions that are most promising for use in the next iteration. Each tool (FormSolve, Crawljax, Artemis) is run for a maximum of 50 *iterations*: for FormSolve, an iteration is defined as in Alg. 2; for Crawljax, it is a full run of the tool, until no new state is

discovered—different iterations use different random seeds; for Artemis, an iteration is a predefined 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 FormSolve and Artemis.

The results are shown in Table 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 number of iterations for each example is taken to be that of the first submission, or the total if no submission was found. Of course the results are not truly indicative of the kinds of constraints found on real websites. They indicate that FormSolve can deal with complex constraints that do *not* obey the restricted global state restrictions that our completeness result relies on, while also indicating that the prior methods do not suffice for complex constraints.

Real-world form validation. A more interesting benchmark is given by a set of simple real-world examples of Web forms. We used JSFiddle [26], a general code-sharing website for Web-based code. Since form-related examples are not easily isolated in JSFiddle, 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 behaviors. 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.

We again compared FormSolve with Crawljax and Artemis, with results shown in Table 4. Some examples required only non-empty results; Artemis still has difficulty handling these, since it does not faithfully simulate the form-filling events triggered by a real user. FormSolve has a modest gain over Crawljax, handling several of the examples that require string comparisons.

The running time for FormSolve is dominated by a single example which uses string functions to parse an ID number, including a checksum; which took 30 minutes 45 seconds out of the total analysis time for all 18 examples of 37 minutes 25 seconds. This example (named *VSKNx* in our test data [16]) requires solving many difficult constraints over both strings and integers. At one point characters from the ID are converted to integers, multiplied 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, FormSolve 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 FormSolve's successful submissions to 77.8%.

Impact of dynamic re-ordering. We compared Algorithm 2 to a variant of the algorithm where a fixed field order (taken from the field ordering in the DOM) is used. The experiments were conducted on the same set of 1000 synthetic forms described above. The JSFiddle examples are ignored, because they do not include interesting interactions between different fields' event handlers, so the results are the same for both modes. For each configuration, we measured the line coverage as before, and also tracked whether we were able to generate a successful form submission. Note that as the examples are randomly generated, some of them have no valid combination of inputs, and submission is impossible. The results are shown in Table 5.

We observe that 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 when testing real sites, 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 46.8% of the examples, compared to 69.8% for the static-ordering variant. In the cases explored by the static ordering algorithm but rejected by Algorithm 2 (23.0% of the total), the static ordering algorithm spent time exploring branches in certain event handlers which would all eventually lead to an abort action in a later handler. In contrast, Algorithm 2's calls to the constraint solver showed these branches to be futile, allowing the algorithm to ignore them. This explains the dramatic difference in the number of successful form submissions found in examples where multiple traces were explored (34.0% for the static order and 66.2% for Algorithm 2).

Real websites. Finally we experimented with 4 real airline websites that included complex validation rules. We report the numbers for the single-order mode discussed above. In that mode, FormSolve was able to find correct submissions while Crawljax could not (despite taking more than 16 hours in total) and Artemis could find only 2. An illuminating 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 children and adults (e.g., not more infants than adults), and that the departure and return dates must be ordered correctly and be no earlier than today's date. FormSolve finds 10 successes while exploring 207 code paths. Handling of the date and passenger constraints exercises the support for integer arithmetic in the solver. The Australian airline Rex has similar passenger restrictions, but in addition each departure airport allows a different set of arrival airports. FormSolve can find 458 distinct successful paths to submission, out of a total of 1539 total paths explored. The real-world set is only anecdotal, due to our prototype's basis in the development version of WebKit, which limits its ability to run on recently-updated websites. Further, the dynamic reordering-mode has not yet been made sufficiently robust—for example, it has weaker support for restricting to user-realisable values. This causes it to miss half of the submissions found by the single-order mode on our small set of real-world sites. More thorough of the approach with a more modern browser codebase will be needed to draw firm quantitative conclusions on its scope of applicability.

5 Conclusion

In this paper we make the first step at applying constraint-solving technology to finding valid submissions for Web forms. The approach has the advantage that it can be used not just to find single submissions but a representation of all valid submissions, in terms of a constraint; this is particularly relevant for wrapper-generation. Clearly this approach does not replace prior techniques of form-filling and wrapper generation, particularly in the presence of domain knowledge or a corpus of examples. Preliminary results with the FormSolve prototype show that the high-level approach, along with our bytecode-based implementation, has promise in practice. A full-featured implementation of bytecode-based symbolic tracing in a state-of-the-art browser remains a major engineering challenge. In the future we will look at variants of our approach with symbolic tracing at the JavaScript level, working with the code on-the-fly.

All our evaluation data for FormSolve, as well as the benchmark generators, are available on GitHub [16].

References

1. Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *FSE*, 2012.
2. Artemis. https://github.com/cs-au-dk/Artemis/tree/ff6ee7ee, 2017.
3. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript Web applications. In *ICSE*, 2011.
4. Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-Web entry points. In *WWW*, 2007.
5. Luciano Barbosa and Juliana Freire. Siphoning hidden-Web data through keyword-based interfaces. *JIDM*, 2010.
6. Clark Barrett, Leonardo de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT. In *HVC*, 2010.
7. Michael Benedikt, Tim Furche, Andreas Savvides, and Pierre Senellart. ProFoUnd: Program-analysis–based form understanding. In *WWW*, 2012. Demonstration.
8. Michael K. Bergman. The deep Web: Surfacing hidden value. *J. Electronic Publishing*, 7, 2001.
9. David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2014.
10. Crawljax, version 3.5. http://crawljax.com/, 2014.
11. Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large Web sites. In *VLDB*, 2001.
12. Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
13. Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, September 2011.
14. Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. AJAX crawl: Making AJAX applications searchable. In *ICDE*, 2009.
15. Bruno Dutertre. Yices 2.2. In *CAV*, 2014.

16. FormSolve test data. https://github.com/BenSpencer/FormSolve-Test-Data.
17. Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, and Christian Schallhart. OPAL: Automated form understanding for the deep Web. In *WWW*, 2012.
18. Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, Christian Schallhart, and Cheng Wang. DIADEM: thousands of websites to a single database. *PVLDB*, 7(14):1845–1856, 2014.
19. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.
20. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, 2005.
21. Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep Web. *Communications of the ACM*, 50(5):94–101, May 2007.
22. Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *EuroSys*, 2014.
23. Jalangi. https://github.com/SRA-SiliconValley/jalangi/tree/22f130ec, 2015.
24. Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *ISSTA*, July 2013.
25. Lu Jiang, Zhaohui Wu, Qian Feng, Jun Liu, and Qinghua Zheng. Efficient deep Web crawling using reinforcement learning. In *PAKDD*, 2010.
26. JSFiddle. https://jsfiddle.net/.
27. 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.
28. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2nd edition, 2016.
29. Juliano Palmieri Lage, Altigran S. da Silva, Paulo B. Golgher, and Alberto H. F. Laender. Automatic generation of agents for collecting hidden Web pages for data extraction. *Data Knowl. Eng.*, 49(2):177–196, May 2004.
30. Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript Web applications. In *FSE*, 2014.
31. Panagiotis Liakos, Alexandros Ntoulas, Alexandros Labrinidis, and Alex Delis. Focused crawling for the hidden Web. In *WWW*, 2016.
32. 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 *CAV*, 2014.
33. Jianguo Lu, Yan Wang, Jie Liang, Jessica Chen, and Jiming Liu. An approach to deep Web crawling by sampling. In *WI-IAT*, 2008.
34. Y. Lu, H. He, H. Zhao, W. Meng, and C. Yu. Annotating structured data of the deep Web. In *ICDE*, 2007.
35. Jayant Madhavan, David Ko, Łucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep-Web crawl. In *VLDB*, 2008.
36. Bruce Mckenzie. The generation of strings from a CFG using a functional language, 1997. https://ir.canterbury.ac.nz/handle/10092/11231.
37. Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based Web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.
38. Hoa Nguyen, Thanh Nguyen, and Juliana Freire. Learning to extract form labels. *PVLDB*, 1(1):684–694, August 2008.
39. Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2015.
40. Alexandros Ntoulas, Petros Zerfos, and Junghoo Cho. Downloading textual hidden Web content through keyword queries. In *JCDL*, 2005.
41. Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden Web. In *VLDB*, 2001.
42. Selenium WebDriver, 2017. www.seleniumhq.org/projects/webdriver/.
43. Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *FSE*, 2013.
44. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.
45. Pierre Senellart, Avin Mittal, Daniel Muschick, Rémi Gilleron, and Marc Tommasi. Automatic wrapper induction from hidden-Web sources with domain knowledge. In *WIDM*, 2008.
46. Ben Spencer, Michael Benedikt, Anders Møller, and Franck van Breugel. ArtForm: A tool for exploring the codebase of form-based websites. In *ISSTA*, 2017.
47. Yan Wang, Jie Liang, and Jianguo Lu. Discover hidden Web properties by random walk on bipartite graph. *Information Retrieval*, 17(3), Jun 2014.
48. Wensheng Wu, AnHai Doan, Clement Yu, and Weiyi Meng. Modeling and extracting deep-Web query interfaces. In *Advances in Information and Intelligent Systems*. 2009.
49. Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement Yu. Fully automatic wrapper generation for search engines. In *WWW*, 2005.