# Constraint Programming for Dynamic Symbolic Execution of JavaScript

Roberto Amadini[1], Mak Andrlon[1], Graeme Gange[2], Peter Schachte[1], Harald Søndergaard[1], and Peter J. Stuckey[2]

[1] University of Melbourne, Victoria, Australia
[2] Monash University, Melbourne, Victoria, Australia

**Abstract.** Dynamic Symbolic Execution (DSE) combines concrete and symbolic execution, usually for the purpose of generating good test suites automatically. It relies on constraint solvers to solve path conditions and to generate new inputs to explore. DSE tools usually make use of SMT solvers for constraint solving. In this paper, we show that constraint programming (CP) is a powerful alternative or complementary technique for DSE. Specifically, we apply CP techniques for DSE of JavaScript, the *de facto* standard for web programming. We capture the JavaScript semantics with MiniZinc and integrate this approach into a tool we call Aratha. We use G-Strings, a CP solver equipped with string variables, for solving path conditions, and we compare the performance of this approach against state-of-the-art SMT solvers. Experimental results, in terms of both speed and coverage, show the benefits of our approach, thus opening new research vistas for using CP techniques in the service of program analysis.

## 1 Introduction

*Dynamic symbolic execution* (DSE), also known as concolic execution/testing, or directed automated random testing [21, 35], is a hybrid technique that integrates the *concrete* execution of a program with its *symbolic* execution [28]. The main application is the automated generation of test suites with high coverage relative to their size. In a nutshell, DSE collects the constraints (or *path conditions*) encountered at conditional statements during concrete execution; then, a constraint solver or theorem prover is used to detect alternative execution paths by systematically negating the path conditions. This process is repeated until all the feasible paths are covered or a given threshold (e.g., a timeout) is exceeded.

Key factors for the success of DSE are the efficiency and the expressiveness of the underlying constraint solver. The significant advances made by *satisfiability modulo theories* (SMT) solvers over recent years have stimulated interest in DSE and led to the development of many popular tools [11, 15, 36, 44, 46, 48]. In particular, improvements in expressive power (due the ability to combine different theories) and solver performance have made SMT solvers very attractive for DSE, to the point that they are considered the *de facto* standard for DSE tools. Alternatives such as *constraint programming* (CP) exist, however.

Constraint programming [40] is a declarative paradigm aimed at solving combinatorial problems consisted of variables (typically having finite domains) and constraints over those variables. CP is applied in fields like resource allocation, scheduling, and planning, but apart from some dedicated approaches [14, 16, 23], it has seen limited use in software analysis. Arguably, the main impediment has been lack of support for common data structures such as dynamic arrays, bit vectors, and strings.

In this paper, we show that DSE can benefit from modern CP solving. In particular, we apply CP techniques to solve the path conditions generated by the dynamic symbolic execution of *JavaScript* programs. JavaScript is nowadays the standard programming language of the web, extensively used by developers on both the client and server side, and supported by all common browsers. Its dynamic nature can easily lead to programming errors and security vulnerabilities. This makes the dynamic symbolic execution of JavaScript an important task, but also a highly challenging one. Hence, it is not surprising that only a small number of DSE tools are available for JavaScript.

To capture JavaScript semantics, we first modelled the main language constructs with the CP modelling language *MiniZinc* [38]. It is essential to note that we are using the MiniZinc extension with *string variables* defined by Amadini et al. [3]. Strings play a central role in JavaScript because each JavaScript object is a map from string keys to values, and hence coercions to strings frequently occur in JavaScript programs (notably, arrays are objects and hence array indices are converted to their corresponding string values). Moreover, JavaScript programs often use *regular expressions* to match string patterns [6].

We then developed Aratha, a DSE tool using the Jalangi analysis framework [45]. Aratha can generate path conditions in our MiniZinc encoding, and solve them with G-Strings [7], a recent extension of the CP solver Gecode [20] able to handle string variables. Aratha is also able to generate path conditions in the form of SMT-LIB assertions, allowing us to empirically evaluate our CP approach against the state-of-the-art SMT solvers CVC4 [32] and Z3str3 [13].

Results indicate that a CP approach can easily be competitive with SMT approaches, and in particular the techniques can be used in conjunction. We emphasize that this technique can be replicated and extended to analyze languages other than JavaScript by using different MiniZinc encodings and different solvers (MiniZinc is a solver-independent language). We are not aware of any similar existing approaches for dynamic symbolic execution.

*Paper structure.* Section 2 introduces the basics of CP and DSE. Section 3 explains how we use MiniZinc to model JavaScript semantics. Section 4 describes Aratha. Section 5 presents our experimental evaluation. Section 6 discusses related work. Section 7 concludes by outlining possible future research directions.

## 2 Preliminaries

We begin by summarizing some basic notions related to constraint programming, string solving, DSE, and JavaScript.

For a given finite alphabet $\Sigma$, we denote by $\Sigma^*$ the set of all finite strings over $\Sigma$. The length of a string $x \in \Sigma^*$ is denoted $|x|$.

### 2.1 Constraint Programming and String Constraint Solving

Constraint programming [40] comprises modelling and solving combinatorial problems. This often means to define and solve a *constraint satisfaction problem* (CSP), which is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of *variables*, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of *domains*, where each $D(x_i)$ is the set of the values that $x_i$ can take, and $\mathcal{C}$ is a set of *constraints* over the variables of $\mathcal{X}$ defining the feasible assignments of values to variables. The goal is typically to find an assignment $\xi \in D(x_1) \times \cdots \times D(x_n)$ of domain values to corresponding variables that satisfies all of the constraints of $\mathcal{C}$.

Most CSPs found in the literature are defined over *finite domains*, i.e., $\mathcal{D}$ only contains finite sets. This guarantees the decidability of these problems, that are in general NP-complete. Typically, only integer variables and constraints are considered. However, some variants have been proposed. In this work, we also consider constraints over *bounded-length strings*. Fixing a finite alphabet $\Sigma$ and a maximum string length $\lambda \in \mathbb{N}$, a CSP with bounded-length strings contains a number $k > 0$ of string variables $\{x_1, \ldots, x_k\} \subseteq \mathcal{X}$ such that $D(x_i) \subseteq \Sigma^*$ and $|x_i| \leq \lambda$. The set $\mathcal{C}$ contains a number of well-known string constraints, such as string length, (dis-)equality, membership in a regular language, concatenation, substring selection, and finding/replacing. In the following, we will refer to constraint solving involving string variables as *string (constraint) solving*.

Different approaches to string constraint solving have been proposed, based on: *automata* [25, 31, 47], *word equations* [13, 32], *unfolding* (using either bit-vector solvers [27, 42] or CP [43]), and *dashed strings* [7, 8]. In particular, dashed strings are a recent CP approach that can be seen as "lazy" unfolding. Thanks to dedicated propagation, dashed strings enable efficient "high-level" reasoning on string constraints, by weakening the dependence on $\lambda$ [5, 6].

Several modelling languages have been proposed for encoding CP problems into a format understandable by constraint solvers. One of the most popular nowadays is *MiniZinc* [38], which is solver-independent (the motto is *"model once, solve anywhere"*), enabling the separation of model and data. Each MiniZinc model (together with corresponding data, if any) is translated into *FlatZinc*—the solver-specific target language for MiniZinc—in the form required by a solver. From the same MiniZinc model, different FlatZinc instances can be derived.

MiniZinc was equipped with string variables and constraints by Amadini et al. [3]. A MiniZinc model with strings can be solved "directly" by CP solvers natively supporting string variables (Gecode+$\mathbb{S}$ [43] and G-Strings [7]) or "indirectly" via the static unfolding into integer variables. Clearly, direct resolution is generally more efficient—especially as $\lambda$ grows.

### 2.2 Dynamic Symbolic Execution

*Symbolic execution* is a static analysis technique that has its roots in the 1970s [28].

The idea of symbolic execution is to assume symbolic values for input and to interpret programs correspondingly, i.e., to use a concept of "value" that is in fact an expression over the variables representing possible input values. The symbolic interpreter can then explore the possible program paths by reasoning about the conditions under which execution will branch this way or that. The set of constraints leading to a particular path being taken is a *path condition*, so that a given path is feasible if and only if the corresponding constraint is satisfiable. The test for satisfiability (and the generation of a witness in the affirmative case) is delegated to a *constraint solver*.

Symbolic execution can be useful to automatically prove a given property of interest, provided that: *(i)* the whole program—including libraries—is available to the interpreter, and *(ii)* the underlying constraint solver is expressive and efficient enough to handle the generated path conditions. Unfortunately, these conditions are often not met.

*Dynamic symbolic execution* (DSE) is a software verification approach that performs symbolic execution along with concrete (or dynamic) execution of a given program. Concrete execution is straightforward: concrete input values are generated according to some heuristics and tested by executing the program.

DSE can mitigate the above issues by: *(i)* directly invoking unavailable functions (a complete symbolic interpreter is not required), and *(ii)* ignoring or approximating unsupported constraints. The idea is to use symbolic values alongside concrete values: during a concrete execution of the program on a given input, symbolic expressions are tracked as in the symbolic execution.

However, in general DSE cannot guarantee full coverage (e.g., in presence of loops or recursion), whereas symbolic execution tries to cover all the possible paths (although still executing a path at a time, e.g., using interpolation to collapse identical or subsumed paths).

After each run, the recorded path conditions are used to generate inputs for the next concrete execution. Indeed, negating a constraint of a path condition will generate a new set of constraints that can either be satisfiable (in which case a new input will be generated to explore the new path) or unsatisfiable (we found an unreachable path in the program). By repeating the process, we can ideally reach the maximum code coverage.

Note that the constraints of the path conditions can be unsupported or too hard to solve in a reasonable time. This can result in over-approximated solutions (when a constraint is ignored or relaxed) or timeouts. This does not compromise the soundness of DSE, however, as it only means that in the worst case, fewer paths might be explored.

### 2.3 JavaScript

Dynamic symbolic execution is language-independent, but the definition and the resolution of path conditions clearly depends on the semantics of the target language. Although this work only considers the JavaScript language, our approach is flexible enough to encode the semantics of other well-known languages.

```
1 var x = symVar();
2 var y = "length";
3 if (x[y] >= 2)
4     console.log("PC1") // path condition (1)
5 else
6     if (y[x] === "g")
7         console.log("PC2") // path condition (2)
8     else
9         console.log("PC3") // path condition (3)
```

Fig. 1: Example of JavaScript program annotated with symbolic variables.

Designed in mid 1990's by Brendan Eich in only ten days, JavaScript has now become a *de facto* standard for web applications. This dynamic, weakly typed language has a number of unconventional rules and pitfalls that sometimes make its behaviour surprising. For instance, there is no concept of class: JavaScript uses prototype-based inheritance between objects. Its weak typing implies a lot of—often implicit—coercions. In particular, coercions to *strings* are very common because apart from few primitive types, in JavaScript everything is an *object*, which is a dictionary with string keys. Each key-value pair is called a *property*, and the key is called a property's *name*. The syntax to access property "$x$" of object $O$ is, equivalently, $O$["$x$"] or $O.x$. For example, JavaScript arrays are actually objects where instead of indices $0, 1, 2, \ldots$ we have properties `"0"`,`"1"`,`"2"`, .... The same applies for strings. For instance, the value of property `"1"` of string `"hello"` is its second character (indexing is 0-based), i.e., `"hello"["1"]` is `"e"`.

The weakness of the semantic rules for JavaScript is an obstacle for program analysis. Static analysis tools have been proposed [9,26,29,42], but the dynamism of the language makes static reasoning difficult and often ineffective. Dynamic techniques such as fuzzing seem more suitable for the analysis of this language. The DSE approach aims to combine the best of the static and dynamic worlds, by orchestrating the dynamic execution via symbolic reasoning.

We conclude this section by providing an example of how DSE works on a snippet of JavaScript code. In Figure 1, variable $x$ is symbolic, i.e., it can take any JavaScript value, while `y` is a concrete variable initialised to `"length"`. When a property of a string primitive is accessed, JavaScript automatically creates a temporary `String` *wrapper object* to resolve the property access.[1] This wrapper inherits all the string methods (e.g., `indexOf`, `toUpperCase`, ...) and also has an immutable `length` property containing the length of the string.

Dynamic symbolic execution starts by initialising $x$ to an arbitrary default value. Let us assume for simplicity that we start with the empty string: $\{x \leftarrow$ `""`$\}$. The first concrete iteration is then executed. Line 3 checks if the `"length"` property of (the `String` object wrapping) $x$ is at least 2. Since $|x| = 0$, this condition is false and the constraint $\neg(|x| \geq 2)$ is added to the path condition. Then, in line

[1] Similarly, Booleans and numbers are wrapped into `Boolean` and `Number` objects.

6 we check if property $x$ of `y` is equal to string `"g"`. But string `"length"` has no property named `""`, so this check also fails and constraint $\neg(\texttt{"length"}[x] = \texttt{"g"})$ is added. We thus reach line 9 by finding that path condition (3), reached with $\{x \leftarrow \texttt{""}\}$, is $\{\neg(|x| \geq 2), \neg(\texttt{"length"}[x] = \texttt{"g"})\}$. This path condition characterizes all inputs $x$ that would take us along a path identical to that of `""`. It is now used to generate a new path. By negating its first constraint we get $|x| \geq 2 \wedge \neg(\texttt{"length"}[x] = \texttt{"g"})$. A suitable solver can find a feasible assignment, e.g., $\{x \leftarrow \texttt{"aa"}\}$. This input leads to path condition (1). Similarly, we negate the second constraint of path condition (3) to get $\neg(|x| \geq 2) \wedge \texttt{"length"}[x] = \texttt{"g"}$. The assignment $\{x \leftarrow \texttt{"3"}\}$ satisfies this constraint: $|x| = 1 \ngeq 2$ and the fourth character of `"length"`, i.e., the one with index 3, is the string `"g"` (of length 1). At this stage, there are no new constraints that can be generated: the set of inputs $\{\{x \leftarrow \texttt{""}\}, \{x \leftarrow \texttt{"aa"}\}, \{x \leftarrow \texttt{"3"}\}\}$ covers all the lines of Figure 1.

## 3 Modelling JavaScript Semantics

Understanding, and then modelling, the semantics of JavaScript is not always straightforward. For example, the comparison `[] == []` between empty arrays fails because JavaScript actually compares their memory locations, which are distinct because two different temporary objects are created. Faithfully modelling the JavaScript semantics also requires the full support of data types like strings, arrays and floating-point numbers. The lack of proper support for these types is probably the main reason CP solvers are not widely used in software analysis, where SMT solvers are typically preferred. However, recent progress in CP (in particular clause learning) makes the modelling and solving tasks more feasible.

In this section we explain how we encode the path conditions generated by DSE as CP models, focusing in particular on how we handle JavaScript variables and objects. It is important to note that correctness and completeness are not strict requirements in this particular context. Indeed, the nature of JavaScript requires a compromise between a faithful mapping of the language's semantics and the complexity of the resulting CP model. Fortunately, difficult JavaScript constructs can be ignored or approximated. While this affects the correctness of the resolution, the ramifications are not dramatic for test data generation: the worst case outcome is that we fail to achieve optimal test coverage. This should be acceptable if "good enough" coverage is reached in a relatively short time.

### 3.1 JavaScript Variables

JavaScript is dynamic and weakly typed. Variables in JavaScript do not have statically defined types, but refer to heterogeneous values that may vary during program execution. A variable can have a *primitive* type (null, undefined, Boolean, number or string) or an *object* type. In particular, null and undefined types each have only one possible value (`null` and `undefined` respectively), a Boolean is either `true` or `false`, strings are encoded in UTF-16 format, and numbers are represented as 64-bit IEEE 754 floating-point values. Objects are collections of

$$type(x) = Undef \;\Rightarrow\; (sval(x) = \texttt{"undefined"} \wedge addr(x) = 0) \quad (1)$$
$$type(x) = Null \;\Rightarrow\; (sval(x) = \texttt{"null"} \wedge addr(x) = 0) \quad (2)$$
$$type(x) = Bool \;\Rightarrow\; sval(x) \in \{\texttt{"true"}, \texttt{"false"}\} \quad (3)$$
$$type(x) = Num \;\Rightarrow\; sval(x) \in \mathcal{NS} \quad (4)$$
$$type(x) = Obj \;\Rightarrow\; sval(x) = \texttt{"[object Object]"} \quad (5)$$
$$0 \leq addr(x) \leq N_{addr} \;\wedge\; (addr(x) = 0 \Rightarrow type(x) \neq Obj) \quad (6)$$

Fig. 2: Invariants for $type(x)$, $sval(x)$, $addr(x)$.

properties mapping a name (a string) to an arbitrary JavaScript value. For each symbolic JavaScript variable $x$, we define a triple $\langle type(x), sval(x), addr(x)\rangle$ of CP variables such that: (i) $type(x)$ encodes the *type* of $x$; (ii) $sval(x)$ represents the *string value* of $x$; (iii) $addr(x)$ models the *memory address* of $x$.

The domain of $type(x)$ is $\mathbb{T} = \{Null, Undef, Bool, Num, Str, Obj\}$.[1] Note that $\mathbb{T}$ can be arbitrarily extended; e.g., the current implementation also considers JavaScript global objects like `Array` and `Function` as standalone types. However, for simplicity, in this paper we do not consider extensions of $\mathbb{T}$. The string variable $sval(x)$ defines the string representation of $x$, i.e., the value returned by JavaScript when coercing $x$ to a string. As aforementioned, these coercions frequently occur during JavaScript program executions. For example, we have that $type(x) \in \{Null, Bool\} \Rightarrow sval(x) \in \{\texttt{"null"}, \texttt{"true"}, \texttt{"false"}\}$, while $sval(x) = \texttt{"42"} \Rightarrow type(x) \in \{Num, Str\}$ because $x$ can be either the number 42 or the string `"42"`. The value of $addr(x)$ is instead a natural number that can be seen as a logical address of $x$. If $addr(x) = 0$ then $x$ is a constant, primitive value; otherwise, $addr(x)$ uniquely identifies object $x$ (see Sections 3.2, 3.3).

Figure 2 shows some of the invariants we enforce to keep $type(x)$, $sval(x)$, $addr(x)$ in a consistent state. Implications 1 and 2 handle the cases where $x$ is undefined or null ($addr(x) = 0$ because $x$ is a constant having no properties). If $x$ is a Boolean variable (implication 3) then $sval(x)$ is either `"true"` or `"false"`. Note that we do not impose any condition on $addr(x)$. If $addr(x) = 0$, we refer to a Boolean constant; otherwise, to the corresponding *wrapper object*. For our purposes, wrappers are only necessary when we explicitly access a property of $x$ (e.g., $x$[`"length"`] when $type(x) = Str$). Otherwise, we can safely treat $x$ as a constant value. Invariant 4 says that if $x$ is a number, then its string value must represent a number. In the current implementation, the language $\mathcal{NS}$ of numeric strings is denoted by the following regular expression:

$$(\texttt{NaN} \mid (\varepsilon \mid \texttt{-})\texttt{Infinity} \mid \texttt{0} \mid (\varepsilon \mid \texttt{-})[\texttt{1-9}][\texttt{0-9}]^*).$$

However, $\mathcal{NS}$ can be extended to handle exponentials, hexadecimal, and floats. Implication 5 defines the string representation of objects according to ECMAScript

[1] We treat $\mathbb{T}$ as an enumeration where $Null = 1$, $Undef = 2, \ldots, Obj = 6$.

specifications [17]. Note that if we also consider other global objects, this invariant is no longer true (e.g., the string value of an `Array` object is the comma-separated concatenation of the array elements). Finally, invariant 6 defines the address space. The constant $N_{addr}$ is the upper bound for each address (no greater than the number of symbolic variables involved in the path condition).

### 3.2 JavaScript Objects

A JavaScript object is essentially a dictionary that maps strings to JavaScript values. While SMT has a well-defined theory of arrays, parametric in the types of keys and values, CP offers no native encodings for array variables and constraints. Thus, in order to model the semantics of JavaScript objects, we devised a proper CP encoding of arrays. Assuming for the moment a fixed and finite set of attributes, a simple encoding would be to introduce a variable for the value of each attribute in each object. Inspecting the value of an attribute just returns the corresponding variable; destructive updates involve creating a new object, equal to the original in all attributes except the updated one. Unfortunately, this encoding is rather large, and tends to propagate poorly.

Francis et al. [18] invert this model: instead of storing the state of each object at each time, the model records the *history of evolution* of the attribute of interest. In this representation, encoding an attribute write $write(O, attr, val)$ simply appends a cell to the "history of writes"—basically, an array storing subsequent attribute writes. The encoding of $read(O, attr)$ must then select the most recent (if any) write to $attr$ on object $O$ from the history array. If no matching write occurred, the read falls through to a default value.

JavaScript poses a further difficulty: since attributes are arbitrary strings, the set of indices is unbounded. Further, because of aliasing, we cannot even determine statically *which* objects are being written to. To handle non-fixed indices, Plazar et al. [39] exploit the observation that, because a finite sequence of reads and writes can only affect a bounded number of indices, it is possible to emulate an unbounded mapping with bounded arrays using an indirection.

To encode destructive update of JavaScript objects, we combine these two approaches: we record the evolution of the program as a sequence of $\langle O, attr, val\rangle$ tuples, plus a bounded number of additional entries which are read without being written (and for built-in attributes, discussed later). Then encoding a read amounts to identifying the latest $\langle O, attr, val\rangle$ tuple for a given $O$ and $attr$.

We define five arrays of CP variables $OAddr, PName, PType, PSval$ and $PAddr$ such that: $OAddr$ stores the address of the objects to uniquely identify them, $PName$ stores the property names (i.e., the keys), while $PType, PSval$, and $PAddr$ store the property values. We model each property write $O[x] \leftarrow y$ with a predicate $write(O, x, y, i)$ such that:

$$write(O, x, y, i) \iff OAddr[i] = addr(O) \land PName[i] = sval(x) \land PType[i] = type(y) \land PSval[i] = sval(y) \land PAddr[i] = addr(y)$$

where $i > 0$ is a *property index* necessary to handle property overwriting: if $O[x] \leftarrow y$ happens *before* a write $O[x] \leftarrow y'$ then we have two corresponding

writes $write(O, x, y, i)$ and $write(O, x, y', i')$ such that $i < i'$. Hence, we have to track the *temporal order* of the writes: a property index is nothing but a sequence number identifying the time instant of a given write. Each time we have a new write, this sequence number must be incremented.

A property read $y \leftarrow O[x]$ is modelled by a function $read(O, x, T)$ returning the proper index $0 \leq i \leq T$ for $O[x]$. Formally:

$$\begin{aligned} read(O, x, T) = i \iff & 0 \leq i \leq T \ \wedge \ type(O) \notin \{Null, Undef\} \ \wedge \\ & O = [O, OAddr[1], \ldots, OAddr[T]][i] \ \wedge \\ & x = [x, PName[1], \ldots, PName[T]][i] \ \wedge \\ & \forall_{j=1,\ldots,T} : (O = OAddr[j] \wedge x = PName[j]) \Rightarrow j \leq i \end{aligned}$$

where the upper bound $T$ is needed to exclude property reads that still are to happen. Note that $T$ is an input constant that can be pre-computed *before* the solving with a counter incremented at each property write.

Index 0 is returned if $O[x]$ is not defined: since in this case JavaScript returns `undefined`, we set $OAddr[0] = PAddr[0] = 0, PType[0] = Undef, PSval[0] =$ `"undefined"`. Let us suppose, e.g., that $O$ is a symbolic object and after $i$ property writes the following statements are executed sequentially:

$$y \leftarrow O[x]; \quad O[x] \leftarrow z; \quad y' \leftarrow O[x]$$

This is modelled by:

$$\begin{array}{rcrcrcr} j = read(O, x, i) & \wedge & write(O, x, z, i+1) & \wedge & j' = read(O, x, i+1) & \wedge \\ type(y) = PType[j] & \wedge & sval(y) = PSval[j] & \wedge & addr(y) = PAddr[j] & \wedge \\ type(y') = PType[j'] & \wedge & sval(y') = PSval[j'] & \wedge & addr(y') = PAddr[j'] & \end{array}$$

It is fundamental to set a precise upper bound for reads: e.g., if the first read was $j = read(O, x, i+1)$, the above constraint would hold only if $z = y$.

JavaScript has a number of *builtin* properties (e.g., the `length` property for strings and arrays) that can be read and overwritten. We handle them by simulating their writing *before* the program execution, i.e., the index of a builtin property will always be lower than the index of the first property accessed in the program execution. For example, we treat the indices 0, 1, 2, ... of a symbolic string $\omega$ as builtin properties `"0"`, `"1"`, `"2"`, ... of a `String` object wrapping $\omega$.

We approximate the deletion of $O[x]$ with a write $O[x] \leftarrow$ `undefined`. This does not agree exactly with the JavaScript semantics but, as already mentioned, we need some relaxations to avoid overloading the solver. For example, although prototype chains are allowed, if object $O$ does not have property $x$, the function $read(O, x, T)$ returns 0 without checking if there exists a prototype $O'$ of $O$ having property $x$.

### 3.3 Other JavaScript Constructs

From the encodings described above we can model most of the other JavaScript operations. For example, a common JavaScript operation is the *strict comparison*

$x$ `===` $y$. This relation holds if $x$ and $y$ have the same type, the same value (different from `NaN`) and, if one is a non-wrapper object, $x$ and $y$ must be exactly the same object (see the example in Figure 1). We encode the strict comparison as:

$$\begin{aligned} x \texttt{===} y \iff & \mathit{type}(x) = \mathit{type}(y) \ \wedge \ \mathit{sval}(x) = \mathit{sval}(y) & \wedge \\ & (\mathit{type}(x) = \mathit{Num} \Rightarrow \mathit{sval}(x) \neq \texttt{"NaN"}) & \wedge \\ & (\mathit{type}(y) = \mathit{Num} \Rightarrow \mathit{sval}(y) \neq \texttt{"NaN"}) & \wedge \\ & ((\mathit{type}(x) = \mathit{Obj} \ \vee \ \mathit{type}(y) = \mathit{Obj}) \Rightarrow \mathit{addr}(x) = \mathit{addr}(y)) \end{aligned}$$

We model the semantics of other JavaScript operations such as ==, !==, !=, $<$, $\leq$, $>$, $\geq$, +, −, /, %, `indexOf`, `charAt`, `concat`, `slice`, `substr`, and regular expression testing. Some of them need special attention because the semantics of the operation depends on the type of the operators. For example, $x < y$ refers to lexicographic order if $\mathit{type}(x) = \mathit{type}(y) = \mathit{Str}$, otherwise arithmetic comparison is performed (via coercion to numbers). Analogously, $z = x + y$ can refer to either the string concatenation or arithmetic addition. Note that in the current implementation we use channelling functions to convert strings to integers and vice versa. An alternative solution might be to use, in addition to the string value $\mathit{sval}(x)$, a CP integer variable $\mathit{ival}(x)$ to keep track of the integer value of $x$. For example, if $x =$ `true` then $\mathit{ival}(x) = 1$. The tricky part in this representation is to encode a non-integer value: if we use a special integer $v \in \mathbb{Z}$ to represent a JavaScript value $x$ not convertible to integers, then we have to discriminate whether $\mathit{ival}(x) = v$ means "not an integer" or the actual number $v$.

The CP encoding we propose is implemented in the MiniZinc language.[1] Each solver supporting MiniZinc can therefore solve the resulting model. We remark that, due to the fundamental role played by strings in JavaScript, we are using the MiniZinc extension with string variables [3]. Clearly a dedicated string solver like G-Strings is currently the best candidate to solve these models, but other solvers could be used by essentially converting strings to arrays of integers.

## 4 Implementation: Aratha

We implemented our DSE framework into a tool we call Aratha.[2] We followed the standard implementation strategy of DSE systems. After annotating the program with symbolic inputs, we begin by running the program with a concrete seed input. During this execution, we record which branches were taken and then construct the path condition corresponding to the input. A set of new path conditions is obtained by negating the last element of each non-empty prefix of the current path condition, which are then appended to the exploration queue. We then take the next path condition in the queue, use a constraint solver to obtain a satisfying input, and repeat the process. Figure 3 presents a graphical summary of the system.

[1] Publicly available at https://bitbucket.org/robama/g-strings/src/master/gecode-5.0.0/gecode/flatzinc/javascript

[2] Publicly available at https://github.com/ArathaJS/aratha

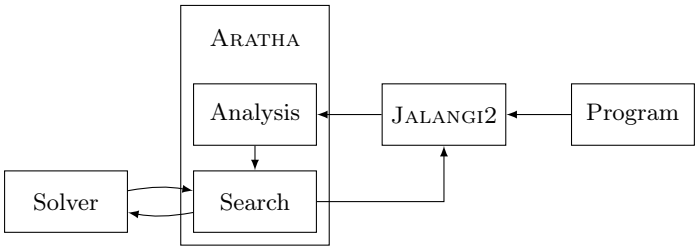


Fig. 3: The architecture of Aratha.

### 4.1 Extracting Path Conditions

Branching information is extracted by running an instrumented program. This is performed via *source-to-source* rewriting using Jalangi2, the successor to Jalangi [45]. That is, instead of writing or modifying a JavaScript interpreter, we insert instrumentation into the source code itself. The instrumenter is invoked whenever new code is accessed, allowing us to analyze code executed using the `eval()` function. We then use the analysis interface of Jalangi2 to intercept, rewrite and record all operations involving symbolic values. All conditional branches depending on symbolic values are added to the current path condition. Apart from if-then-else statements and loops, the logical operators are also treated as conditionals, owing to short-circuit evaluation. This does cause some loss of efficiency, as it pessimistically assumes that any logical operation might involve an expression containing side effects.

In the program, symbolic variables are obtained by calling `J$.readInput()`, returning a pair containing the concrete value to be used in the current run, and a symbolic value representing an input variable. Essentially, any concrete value can have a symbolic expression associated to it. That expression is a record of the operations that produced the associated concrete value. If the program performs an operation we cannot trace symbolically (e.g., a call to a library function), or an operation that we cannot model properly, we return only the concrete result and we throw away the symbolic expression. This is a common approach in DSE.

### 4.2 Source-to-Source Translation

Source-to-source translation brings with it two advantages. Firstly, the analysis is independent of any particular interpreter, and is therefore not sensitive to changes in system architecture. This is especially valuable, as JavaScript interpreters are a moving target, due to the never-ending search for improved performance. Secondly, this allows us to reap the efficiency benefits of those same optimized interpreters. Although instrumentation introduces some overhead, modern interpreters can nearly eliminate its impact. The cost is, however, that our analysis relies on having as much code instrumented as possible.

Though concretization allows DSE to be run on programs containing uninstrumented code, coverage quickly becomes limited as more and more symbolic

expressions become concrete. Notably, though primitive values can be concretized with little impact, any object that is passed to an un-instrumented function must have its entire object graph concretized. This can result in a cascade effect that strips almost all symbolic information from that point.

One thing to note is that, regardless of method, it is difficult to fully mask the presence of instrumentation. As we instrument all JavaScript operations, we can rewrite operations such as introspection functions which could reveal the presence of instrumentation. However, a timing-sensitive program might still be disrupted by the overhead of instrumentation. As most JavaScript programs do not interrogate their execution environment, we have not attempted to handle such things in much depth.

### 4.3 Backend Solving and Optimizations

ARATHA can model path conditions in both the MiniZinc and the SMT-LIB [12] constraint languages. This means it is compatible with any constraint solver supporting either of those languages, as long as it also supports the string extensions. The SMT-LIB output relies on a partial axiomatization of JavaScript's semantics. This axiomatization is itself written in the SMT-LIB 2 language, and is hence independent of any particular solver.

To the best of our knowledge, the only mature SMT solvers that currently support *all* the theories it requires are Z3 and CVC4. Note that previous systems such as Kudzu [41], JALANGI [45], SymJS [30] and ExpoSE [34] were all designed for use with particular constraint solvers. As such, our implementation is the first multi-solver DSE tool. By enabling the use of both CP and SMT solvers, we can potentially benefit from the strengths of both.

Our analysis runs on `Node.js`, which uses the highly efficient V8 JavaScript interpreter. However, it is constraint solving rather than program execution that dominates execution time in DSE. As such, we implemented a number of optimizations in an attempt to make solving more efficient. To reduce the number of solver queries and mitigate the memory impact of storing symbolic expression trees, we perform computations concretely whenever it is possible to do so without losing precision. For instance, the unary `void` operator always returns `undefined`, regardless of its argument. Similarly, we eagerly simplify read operations on properties of symbolic objects. In many such cases, we can determine the property's value unambiguously and hence return just that particular value.

We also attempt to use type information to simplify expressions whenever possible. Though ARATHA fully supports JavaScript's type system, it is beneficial for both performance and understandability to simplify type-dependent operations as early as possible. As many functions only return values of a specific type, we can frequently choose which "overload" of a function to invoke. For instance, if both of the arguments of a + operation are numbers, we can use the more specific numeric addition instead of the general JavaScript addition function.

In terms of back-end optimizations, ARATHA can submit constraint queries incrementally to a supporting solver. Constraint solvers which support incremental solving can reuse previous work when answering a query, potentially yielding

a result much more quickly. However, such functionality is at present generally provided only by SMT solvers because CP does not handle incremental solving.

ARATHA deals with loops and recursions by setting a parameter that limits the maximum number of iterations allowed. This is a common approach in DSE.

## 5 Evaluation

We now use ARATHA to assess the performance of CP and SMT solvers within our tool. We emphasize that it is not our goal to make a comparison of different DSE tools. Such a comparison would be difficult because of the limited availability and development of JavaScript DSE tools so far. Rather, we have wanted to test the hypothesis that CP is a valuable option for software analysis, when used in synergy with (not necessarily in place of) SMT or SAT solving technologies. We also underline that the performance of the individual solvers also depends on the SMT/CP encoding we chose for the JavaScript semantics: different models may lead to a different performance.

We compared four different solvers: the CP-based string solver G-STRINGS and the SMT solvers CVC4 [32], Z3 [37], together with Z3's most recent string solver extension Z3STR3 [13]. For each path condition, we set a small timeout of $T_{pc} = 10$ seconds. This is because DSE implies a high number of path conditions having a limited number of constraints. Moreover, setting a too high value of $T_{pc}$ would be unnecessarily harmful given the heavy-tailed nature of solving: these problems are typically either solved in few seconds, or not solvable at all within hours of computation. We set a maximum number of $N = 1024$ DSE iterations for each problem, and also an overall timeout $T_{tot} = 300$ seconds, because sometimes reaching $N$ iterations can take too long.[1] We ran all the experiments on an Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel® i7 CPU.

Unfortunately, there are no standard benchmarks for JavaScript DSE. Moreover, retrieving large JavaScript benchmarks is tedious because the source-to-source rewriting of ARATHA needs a manual instrumentation for the symbolic input. We therefore tested ARATHA on the test suite of EXPOSE, consisting of 197 already annotated JavaScript sources. This is not a DSE benchmark in the strict sense, but it is however very useful in this context. We did not compare ARATHA against EXPOSE because ARATHA does not yet fully support complex JavaScript operations such as backreferences and greedy matching.

The results are shown in Table 1, where we see the average coverage of statements in the entire test suite, the average time to process each problem in the suite, the average number of unique inputs generated, and the total number of times the overall timeout $T_{tot}$ was reached. Clearly the CP approach implemented by G-STRINGS is competitive with SMT methods: it is fast and provides the best coverage. Note that being fast is not always good in this context, because a solver can terminate its execution in a few seconds without yielding significant inputs. This is the case of Z3 and Z3STR3: they are the fastest solvers, but they have

[1] $T_{tot}$ is also useful because CVC4 may get stuck in presolving regardless of $T_{pc}$ limit. (see http://cvc4.cs.stanford.edu/wiki/User_Manual#Resource_limits).

Table 1: Average results and cross comparisons between solvers

| solver | % statements | time [s] | inputs | timeouts | G-Strings | CVC4 | Z3 | Z3str3 |
|---|---|---|---|---|---|---|---|---|
| G-Strings | **82.85** | 4.74 | **3.54** | 0 | — | **41** | **73** | **112** |
| CVC4 | 77.25 | 33.08 | 2.83 | 21 | 3 | — | 51 | 98 |
| Z3 | 72.81 | 3.06 | 3.01 | 1 | 11 | 19 | — | 66 |
| Z3str3 | 62.69 | **0.46** | 1.80 | 0 | 0 | 2 | 11 | — |

the smallest coverage (in particular Z3str3 appears unstable on these problems). However, we remark that this performance should not be taken as an absolute value because it also depends on the SMT encodings we chose.

The average coverage of CVC4 is closer to that of G-Strings. Its high average time is slightly misleading, as it is mainly due to the high number of timeouts. In fact, for 130 cases CVC4 is faster in reaching (at least) the same coverage as G-Strings. This suggests that CP and SMT solvers should not be seen as mutually exclusive, but possibly cooperating via a *portfolio* approach [4, 10] that aims to select and run the best solver(s)—possibly in parallel and by exchanging information—for a given path condition.

The second part of the table shows the number of times the solver for that row reaches a strictly better coverage than the solver for that column. On this measure, G-Strings has the best performance. However, there are cases where CVC4 and Z3 achieve a better coverage.

## 6 Related Work

The main ideas behind DSE go back to Godefroid, Klarlund and Sen's *DART* project [21]. Since then, advances in solver technology saw DSE tools improve rapidly, in some cases finding large-scale use. For example, Microsoft's *SAGE* [22] DSE tool reportedly detected up to one third of all bugs discovered during the development of Windows 7—bugs that were missed by other testing methods.

DSE was first applied to JavaScript programs in the *Kudzu* project [42]. Existing solvers were found inadequate for the task of reasoning about JavaScript behaviour, for a number of reasons, including JavaScript's orientation towards strings as a default data structure. Hence a major part of the Kudzu project turned out to be the development of a dedicated string + bitvector solver, *Kaluza*.

*SymJS* [30] is a symbolic execution and fuzzing tool for JavaScript. It relies on the *PASS* [31] solver, and it uses a model of the DOM combined with an intelligent, feedback-driven event generator to automatically test web applications.

ExpoSE [34] is the first JavaScript DSE tool able to reason about string matching via (extended) regular expressions, although in a limited fashion. It uses Z3 for constraint solving and it has been applied successfully to several important `Node.js` libraries, though overall coverage is relatively low because of the limited nature of the analysis.

Aratha is the first JavaScript DSE tool capable of reasoning about inputs without resorting to unsound heuristic type assignments or requiring the user to

commit to the type of each input in advance. It allows for easy replacement of constraint solver. It is built using *Jalangi* 2 [45], a framework for implementing dynamic analyses for JavaScript.

Meaningful comparison of the JavaScript DSE tools discussed here is hampered by their limited availability. Comparison of different DSE *backends*, i.e., constraint solvers focused on the types of constraints typically generated in dynamic analysis of JavaScript, is a simpler task, provided we have a DSE tool that allows for easy backend plugging and unplugging. ARATHA does exactly that.

String solvers are still in their infancy and current solvers naturally show varying degrees of robustness. Many are based on the DPLL(T) paradigm [19], including CVC4 [33], Z3str* family [13, 52, 53], S3* family [49–51], and Norn [2]. More recent proposals are Sloth [24] and Trau [1]. These solvers handle constraints over strings of unbounded length; however, they are known to be incomplete. Z3STR in particular claims to be complete for the set of positive formulas in the theory of concatenation and linear integer arithmetic in length, however, its successors are of a different design and have not made such promises.

Some solvers provide finite decision procedures by stipulating an upper bound on the length of strings (e.g., HAMPI [27] and Kaluza [41]). G-STRINGS [7,8] takes a propagation based approach to bounded string solving, where the complexity weakly depends on the length bound.

## 7 Conclusions

In this paper we have described how to build a dynamic symbolic execution tool for JavaScript, utilising an underlying CP solver. Critical to this approach is the ability to translate the complex object behaviour of JavaScript into a set of constraints that are handled by a CP solver. In particular for JavaScript, since strings are essential to almost any operation in the language, our tool makes use of string extensions of the MiniZinc language [3] and the efficient string constraint solving capabilities of the dashed-string solver G-STRINGS [8].

Our experiments suggest that CP solvers can be competitive with state-of-the-art SMT solvers, for the kind of constraints that arise in dynamic symbolic execution of JavaScript. In particular, a *portfolio* consisting of both SMT and CP solvers might turn out to be a good strategy for maximizing code coverage and minimising the DSE time.

Important future work to improve the CP approach is to extend CP constraints to do equality propagation, to propagate more information from object constraints. Extending the string solver to be usable in a nogood solver should also significantly improve CP solving times.

## Acknowledgments

This work is supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

## References


1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 602–617 (2017)
2. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Păsăreanu, C. (eds.) Computer Aided Verification: Proc. 27th Int. Conf. Part I. LNCS, vol. 9206, pp. 462–469. Springer (2015)
3. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with strings. In: Hermenegildo, M., López-García, P. (eds.) LOPSTR 2016: Revised Selected Papers. LNCS, vol. 10184, pp. 59–75. Springer (2017)
4. Amadini, R., Gabbrielli, M., Mauro, J.: A multicore tool for constraint solving. In: Proc. 24th Int. Joint Conf. Artificial Intelligence. pp. 232–238. AAAI Press (2015)
5. Amadini, R., Gange, G., Stuckey, P.J.: Propagating *Lex*, *Find* and *Replace* with dashed strings. In: van Hoeve, W.J. (ed.) Proc. 15th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming. LNCS, vol. 10848. Springer (2018)
6. Amadini, R., Gange, G., Stuckey, P.J.: Propagating regular membership with dashed strings. In: Hooker, J. (ed.) Proc. 24th Conf. Principles and Practice of Constraint Programming. LNCS, vol. 11008, pp. 13–29. Springer (2018)
7. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: Proc. 32nd AAAI Conf. Artificial Intelligence. pp. 6557–6564. AAAI Press (2018)
8. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In: Beck, J.C. (ed.) Proc. 23rd Int. Conf. Principles and Practice of Constraint Programming. LNCS, vol. 10416, pp. 3–20. Springer (2017)
9. Amadini, R., Jordan, A., Gange, G., Gauthier, F., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Combining string abstract domains for JavaScript analysis: An evaluation. In: Legay, A., Margaria, T. (eds.) Proc. 23rd Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, Part I. LNCS, vol. 10205, pp. 41–57. Springer (2017)
10. Amadini, R., Stuckey, P.J.: Sequential time splitting and bounds communication for a portfolio of optimization solvers. In: O'Sullivan, B. (ed.) Proc. 20th Conf. Principles and Practice of Constraint Programming. LNCS, vol. 8656, pp. 108–124. Springer (2014). https://doi.org/10.1007/978-3-319-10428-7_11
11. Artzi, S., Kieżun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A.M., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Software Eng. **36**(4), 474–494 (2010)
12. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.6. Tech. rep., Dept. of Computer Science, University of Iowa (2017), www.SMT-LIB.org
13. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: Stewart, D., Weissenbacher, G. (eds.) Proc. 17th Conf. Formal Methods in Computer-Aided Design. pp. 55–59. FMCAD Inc (2017)
14. Blanc, B., Junke, C., Marre, B., Gall, P.L., Andrieu, O.: Handling state-machines specifications with GATeL. Electr. Notes Theor. Comput. Sci. **264**(3), 3–17 (2010). https://doi.org/10.1016/j.entcs.2010.12.011

15. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. 8th USENIX Conf. Operating Systems Design and Implementation. OSDI, vol. 8, pp. 209–224 (2008)
16. Delahaye, M., Botella, B., Gotlieb, A.: Infeasible path generalization in dynamic symbolic execution. Information & Software Technology **58**, 403–418 (2015)
17. ECMA International: Ecmascript 2018 language specification (2018), available at https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
18. Francis, K., Navas, J.A., Stuckey, P.J.: Modelling destructive assignments. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming: Proc. 19th Int. Conf. LNCS, vol. 8124, pp. 315–330. Springer (2013)
19. Ganzinger, H., Hagen, G., Nieuwenhuis, R., a, A.O., Tinelli, C.: Dpll(t): Fast decision procedures. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification: Proc. 16th Int. Conf. LNCS, vol. 3114, pp. 175–188. Springer (2004)
20. Gecode Team: Gecode: Generic constraint development environment (2016), available at http://www.gecode.org
21. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05). pp. 213–223. ACM (2005)
22. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)
23. Gotlieb, A.: TCAS software verification using constraint programming. Knowledge Eng. Review **27**(3), 343–360 (2012). https://doi.org/10.1017/S0269888912000252
24. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)
25. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Automated Software Engineering **19**(4), 531–559 (2012)
26. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: A static analysis platform for JavaScript. In: Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering. pp. 121–132. ACM (2014)
27. Kieżun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Trans. Software Engineering and Methodology **21**(4), article 25 (2012)
28. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
29. Lee, H., Won, S., Jin, J., Cho, J., Ryu, S.: SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In: Proc. 19th Int. Workshop on Foundations of Object-Oriented Languages (FOOL'12) (2012)
30. Li, G., Andreasen, E., Ghosh, I.: SymJS: Automatic symbolic testing of JavaScript web applications. In: Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering. pp. 449–459. ACM (2014)
31. Li, G., Ghosh, I.: PASS: String solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) Proc. 9th Int. Haifa Verification Conf. LNCS, vol. 8244, pp. 15–31. Springer (2013)
32. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification: Proc. 26th Int. Conf. LNCS, vol. 8559, pp. 646–662. Springer (2014)

33. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**(3), 206–234 (2016)
34. Loring, B., Mitchell, D., Kinder, J.: ExpoSE: Practical symbolic execution of standalone JavaScript. In: Proc. 24th ACM SIGSOFT Int. SPIN Symp. Model Checking of Software. pp. 196–199. ACM (2017)
35. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. 29th Int. Conf. Software Engineering (ICSE 2007). pp. 416–426. IEEE (2007)
36. Majumdar, R., Xu, R.G.: Reducing test inputs using information partitions. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification: Proc. 21st Int. Conf. LNCS, vol. 5643, pp. 555–569. Springer (2009)
37. Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Lecture Notes in Computer Science, Springer (2008)
38. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer (2007)
39. Plazar, Q., Acher, M., Bardin, S., Gotlieb, A.: Efficient and complete FD-solving for extended array constraints. In: Sierra, C. (ed.) Proc. 26th Int. Joint Conf. Artificial Intelligence. pp. 1231–1238. ijcai.org (2017)
40. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier (2006)
41. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proc. 2010 IEEE Symp. Security and Privacy. pp. 513–528. IEEE Comp. Soc. (2010)
42. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proc. 2010 IEEE Symp. Security and Privacy. pp. 513–528. IEEE Comp. Soc. (2010)
43. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: Design and implementation of bounded-length sequence variables. In: Lombardi, M., Salvagnin, D. (eds.) Proc. 14th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming. LNCS, vol. 10335, pp. 51–67. Springer (2017)
44. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification: Proc. 18th Int. Conf. LNCS, vol. 4144, pp. 419–423 (2006)
45. Sen, K., Kalasapur, S., Brutch, T.G., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. Foundations of Software Engineering. pp. 488–498 (2013)
46. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. 10th European Software Engineering Conf. pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
47. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Software Engineering Methodology **22**(4), article 33 (2013)
48. Tillmann, N., De Halleux, J.: Pex—white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs (TAP'08), LNCS, vol. 4966, pp. 134–153. Springer (2008)

49. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proc. 2014 ACM SIGSAC Conf. Computer and Communications Security. pp. 1232–1243. ACM (2014)
50. Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 9779, pp. 218–240. Springer (2016)
51. Trinh, M., Chu, D., Jaffar, J.: Model counting for recursively-defined strings. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. pp. 399–418 (2017)
52. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: CAV. LNCS, vol. 9206, pp. 235–254. Springer (2015)
53. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: Proc. 9th Joint Meeting on Foundations of Software Engineering. pp. 114–124. ACM (2013)