Automated Unit Testing of JavaScript Code through Symbolic Executor SymJS

Hideo Tanida, Tadahiro Uehara

Information System Technologies Laboratory
Fujitsu Laboratories Ltd.
Kawasaki, Japan
Email: {tanida.hideo,uehara.tadahiro}
@jp.fujitsu.com

Guodong Li, Indradeep Ghosh

Software Systems Innovation Group
Fujitsu Laboratories of America, Inc.
Sunnyvale, CA, USA
Email: {gli,indradeep.ghosh}
@us.fujitsu.com

***Abstract*—JavaScript is expected to be a programming language of even wider use, considering demands for more interactive web/mobile applications and deployment in server-side software. While reliability of JavaScript code will be of more importance, testing techniques for the language remain insufficient, compared to other languages. We propose a technique to automatically generate high-coverage unit tests for JavaScript code. The technique makes use of symbolic execution engine for JavaScript, and symbolic stub/driver generation engine, which injects symbolic variables to system under test. Our methodology allows for automatic generation of input data for unit testing of JavaScript code with high coverage, which ensures quality of target code with reduced effort.**

Keywords–JavaScript; test generation; symbolic execution; symbolic stub and driver generation.

I. INTRODUCTION

Extensive testing is required to implement reliable software. However, current industrial practice rely on manually-written tests, which result in large amount of effort required to ensure quality of final products or defects from inadequate testing.

Verification and test generation techniques based on formal methods are considered to be solutions to overcome the problem. One such technique is test generation through symbolic execution, which achieves higher code coverage compared to random testing [1]–[7].

In order to symbolically execute a program, input variables to the program are handled as symbolic variables with their concrete values unknown. During execution of the program, constraints to be met by values of variables in each execution path are obtained. After obtaining constraints for all the paths within the program, concrete values of input variables to execute every paths can be obtained, by feeding a solver such as Satisfiability Modulo Theory (SMT) [8] solver with the constraints. Normal concrete execution of the program using all the obtained data, results in all the path within the program went through.

Manually-crafted test inputs require effort for creation, while they do not guarantee exercising all the execution path in the target program. In contrast, test generation based on symbolic execution automatically obtains inputs to execute all the path within the program. As the consequence, it may find corner-case bugs missed with insufficient testing.

There are tools for symbolic execution of program code, including those targeting code in C/C++ [2], [3], [5], Java [4], and binary code [6], [7]. It is reported that the tools can automatically detect corner-case bugs, or generate test inputs that achieve high code coverage.

JavaScript was historically introduced as an in-browser scripting language of light weight use. However, it is now heavily used for implementation of feature rich client-tier within interactive web applications. The language is also used to implement software product of other kind, including application servers based on Node.js [9] and standalone mobile applications implemented with PhoneGap [10].

The wider adoption of the language has brought efficient testing technique for JavaScript code into focus. In order to exercise tests in an efficient manner, unit testing frameworks such as Jasmine [11], QUnit [12] and Mocha [13] are developed and used in the field. However, only execution of once developed tests can be supported through the tools, and large amount of effort is still required to prepare test cases that ensure quality of target code. Therefore, automatic test generation techniques for the language are becoming of more importance, and symbolic execution is again considered one key technology to play a role.

Existing symbolic execution tools for JavaScript code include Kudzu [14] and Jalangi [15]. Kudzu automatically generates input data for program functions, with the aim of automatically discovering security problems in the target. Jalangi allows for modification of path constraints under normal concrete executions, in order to obtain results different from previous runs. However, the tools could not be applied to unit testing of JavaScript code in field, due to limitations in string constraint handling and need for manual coding of driver and stub used in testing.

We propose a technique to generate test inputs for JavaScript code through symbolic execution on a tool SymJS [1], [16]. Test inputs generated by the tool exercise target code with high coverage. After augmenting generated test inputs with user-supplied invariants, application behavior conformance under diverse context can be checked in a fully automatic fashion. Our proposal includes automatic generation of symbolic stubs and drivers, which reduces need for manual coding. Therefore, our technique allows for fully automatic generation of input data used in unit testing of JavaScript code. Test inputs generated by our technique exercise every feasible

execution paths in the target to achieve high coverage.

Our methodology has the following advantages to existing works. Our JavaScript symbolic execution engine SymJS is applicable to JavaScript development in practice for the following reasons.

- Our constraint solver PASS [17] allows test generation for programs with various complex string manipulations
- SymJS does not require any modification to the target code, while the existing symbolic executors for JavaScript [14], [15] needed modifications and multiple runs
- Our automatic stub/driver code generation allows for fully automatic test data generation

An existing work [15] could be employed for generation of unit tests. However, it required manual coding of stub/driver, which requires knowledge on symbolic execution and error-prone. The engine also suffered from limitations in string constraint handling. Our fully automatic technique can be applied to development in practice.

The rest of this paper is organized as follows. Section II explains the need for automatic test generation/execution with an example, and introduces our test input generation technique through symbolic execution. Section III describes our method to automatically generate stub/driver code used in test generation/execution. Evaluation in Section IV shows applicability and effectiveness of our technique on multiple benchmarks. We discuss the lessons learnt in Section V. Finally, we come to the conclusion in Section VI and show possible future research directions.

II. Background and Proposed Test Generation Technique

A. Demands for Automatic Test Generation

Generally, if a certain execution path in a program is exercised or not, depends on input fed to the program. Therefore, we need to carefully provide sufficient number of appropriate test input data, in order to achieve high code coverage during testing.

For example, function `func0()` shown in Figure 1 contains multiple execution path. Further, whether each path is exercised or not depends on input fed to the program, which are value of arguments `s,a` and return value of function `Lib.m2()`. Current industrial testing practice depends on human labor to provide the inputs. However, preparing test inputs to cover every path within software under test requires large amount of efforts. Further, manually-created test inputs might not be sufficient to exercise every path within the target program.

Figure 2 shows possible execution path within the example in Figure 1. In the example, there are two sets of code blocks and whether blocks are executed or not depend on branch decisions. The first set of the blocks contains blocks 0-3, and the second set contains blocks A-B. Conditions for the blocks to be executed are shown at the top of each block in Figure 2. Block 2 has a contradiction between conditions for

```
function func0(s,a) {
  if("".equals(s)) { // block 0
    s = null;
  } else {
    if(s.length <= 5) { // block 1
      a = a + status;
    } else {
      if("".equals(s)) { // block 2
        Lib.m0(); // Unreachable
      } else { // block 3
        Lib.m1();
      }
    }
  }
  if(a <= Lib.m2()) { // block A
    a = 0;
  } else { // block B
    a = a + s.length; // Error with null s
  }
}
```

Figure 1. Code Fragment Used to Explain our Methodology: s, a, Lib.m2() may Take Any Value

Table I. Constraints to Execute Paths in Figure 2 and Satisfying Test Inputs (under assumption status=-1)

Test No.	Blocks Executed	Path Conditions	Test Data
1	0,A	`"".equals(s)` ∧ `a<=Lib.m2()`	`s="", a=0` `Lib.m2()=0`
2	0,B	`"".equals(s)` ∧ `a>Lib.m2()`	`s="", a=0` `Lib.m2()=-1`
3	1,A	`!"".equals(s)` ∧ `s.length <= 5` ∧ `a-1<=Lib.m2()`	`s="a", a=0` `Lib.m2()=0`
4	1,B	`!"".equals(s)` ∧ `s.length <= 5` ∧ `a-1>Lib.m2()`	`s="a", a=1` `Lib.m2()=0`
5	3,A	`!"".equals(s)` ∧ `s.length > 5` ∧ `a<=Lib.m2()`	`s="aaaaaa", a=0` `Lib.m2()=0`
6	3,B	`!"".equals(s)` ∧ `s.length > 5` ∧ `a>Lib.m2()`	`s="aaaaaa", a=0` `Lib.m2()=-1`

execution, which are `s.length()>5` and `"".equals(s)`, and will never be executed. However, the other blocks have no such contradiction and are executable. Tests to execute every possible combination of blocks 0,1,3 and blocks A,B correspond to $3 \times 2 = 6$ set of values for the inputs.

Table I shows combinations of blocks to execute, path condition to be met by arguments `s,a` and return value of `Lib.m2()`. In the example, it is possible to obtain concrete values meeting the conditions for the inputs, and the values can be used as test inputs. We will discuss how to automatically obtain such test inputs in the sequel.

B. Test Input Generation through Symbolic Execution

We propose a methodology to automatically generate test inputs with SymJS, a symbolic execution engine for JavaScript. During symbolic execution of a program, constraints to be met in order to execute each path (shown as "Path Conditions" in

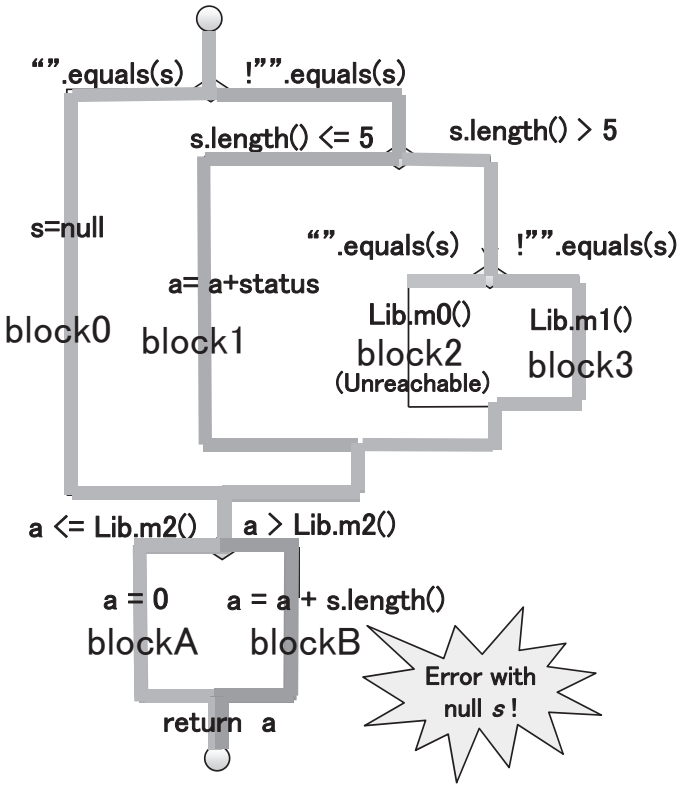

Figure 2. Execution Paths within Code Shown in Figure 1

Table I) are calculated iteratively. After visiting every possible path within the program, constraints corresponding to all the path are obtained. Concrete values of variables meeting the constraints can be obtained with solvers such as SMT solver. Obtained values are input data to exercise paths corresponding to the constraints, which we can use for testing.

While JavaScript functions are often executed in a event-driven and asynchronous fashion, our technique focuses on generation of tests that invoke functions in deterministic and synchronous orders. We assume the behavior of generated tests are reasonable, considering what is inspected in current JavaScript unit tests in field, as opposed to integration/system testing. Each generated test data exercise single path within the target, and only single data set is generated for each path.

SymJS allows for symbolic execution of JavaScript code. SymJS interprets bytecode for the target program, and symbolically executes it in a way KLEE [3] and Symbolic JPF [4] do. SymJS handles program code meeting the language standard defined in ECMAScript [18].

SymJS is an extended version of Rhino [19], an open-source implementation of JavaScript. Our extensions include symbolic execution of target code, constraint solving to obtain concrete test input data, and state management. While there are existing symbolic executors for JavaScript, SymJS does not reuse any of their code base. Table III shows comparison between SymJS and existing symbolic executors.

SymJS interprets bytecode compiled from source code of target program. This approach is taken by existing symbolic executors such as KLEE [3] and Symbolic PathFinder [4].

Table II. Instructions with their Interpretations Modified from Original Rhino

Arithmetic/Logical Operations	ADD, SUB, MUL, DIV, MOD, NEG, POS, BITNOT, BITAND, BITOR, BITXOR, LSH, RSH, URSH etc.
Comparisons	EQ, NE, GE, GT, LE, LT, NOT, SHEQ, SHNE etc.
Branches	IFEQ, IFNE, IFEQ_POP etc.
Function Calls	RETURN, CALL, TAIL_CALL etc.
Object Manipulations	NEW, REF, IN, INSTANCEOF, TYPEOF, GETNAME, SETNAME, NAME etc.
Object Accesses	GETPROP, SETPROP, DELPROP, GETELEM, SETELEM, GETREF, SETREF etc.

Handling bytecode instead of source code allows for implementation of symbolic executors without dealing with complex syntax of the target language. SymJS is implemented as an interpreter of Rhino bytecode, which updates the program state (content of heap/stack and path condition) on execution of every bytecode instruction. Upon hitting branch instruction, it duplicates the program state and continues with the execution of both the branches.

In order to implement symbolic execution of target programs, we have modified interpretation of the instructions shown in Table II from the original Rhino. Handling of instructions for stack manipulation, exception handling, and variable scope management remain intact.

For example, an instruction ADD op1 op2 is interpreted as follows.

1) Operands op1 and op2 are popped from stack. The operands may take either symbolic or concrete value.
2) Types of the operands are checked. If both the operands are String, the result of computation is the concatenation of the operands. If they are Numeric, the result is the sum of the operands. Otherwise, values are converted according to ECMAScript language standard, and the result is either concatenation or addition of the obtained values.

As JavaScript is a dynamically typing language, types of operands for Rhino instructions are not known prior to execution. Therefore, types of results also need to be determined at run time. For example, evaluation of instructions v1 = ADD e1:number v:untyped; v2 = ADD v1 "abc" yields, v1 = e1 + v:number; v2 = toString(e1 + v:number) concat "abc". Types of variables v1,v2 are fixed just at run time in a dynamic fashion.

Comparison instructions are followed by branch instructions in Rhino bytecode. SymJS handles compare and branch instruction pairs as in the following. First, it creates Boolean formula corresponding to result of comparison after necessary type conversions. Assuming the created formula is denoted by symbol c, we check if c and its negation $\neg c$ are satisfiable together with current path condition pc. In other words, we check for satisfiablity of $pc \wedge c$ and $pc \wedge \neg c$. If both are satisfiable, we build states s_1, s_2 corresponding to $pc \wedge c$ and $pc \wedge \neg c$ and continue with execution from states s_1 and s_2. If one of them is satisfiable, the state corresponding to the satisfiable one is chosen and execution resumes from that point.

SymJS supports two ways to manage states, which are forked on hitting branches etc. The first method is to store all program state variables including content of heap/stack, as is

Table III. Comparison of Symbolic Executors

Tool	Target Lang.	Sym. VM	Dep./Cache Solving	String Solving
SymJS	JavaScript	Yes	Yes	Yes
KLEE [3]	C	Yes	Yes	No
SAGE [7]	x86 binary	No	Yes	No
Sym JPF [4]	Java	Yes	No	No
Kudzu [14]	JavaScript	No	No	Yes
Jalangi [15]	JavaScript	No	No	Limited

Table IV. Representation of States in Fuzzing after Executing Code on Figure 1 under Path Conditions in Table I

Test No.	Blocks Executed	State Representation
1	0,A	L;L
2	0,B	L;R
3	1,A	R;L;L
4	1,B	R;L;R
5	3,A	R;R;R;L
6	3,B	R;R;R;R

done in [3], [4]. The second method is to remember only which side is taken on branches. This method needs to re-execute the target program from its initial state on backtracking. However, it benefits from its simple implementation and smaller memory footprint. The method is called "Fuzzing" and similar to the technique introduced in [5], [7]. However, our technique is implemented upon our symbolic executor and does not need modification of target code required in the existing tools [14], [15] for JavaScript.

During symbolic execution of programs through fuzzing, states are represented and stored only by which side is taken on branches. The information can be used to re-execute the program from its initial state and explore the state space target may take. States after symbolically executing the target program in Figure 1 with path conditions corresponding to tests 1-6 in Table I, are represented as shown in Table IV during fuzzing. Symbols *L,R* denote left/right branch is taken on a branching instruction.

For each of state representations shown in Table IV, corresponding path condition can be obtained. Table I includes path conditions for the states in Table IV. If it is possible to obtain solutions satisfying the constraints, they can be used as inputs used during testing. Constraints on numbers can be solved by feeding them into SMT solvers. However, SMT solvers cannot handle constraints of strings, which is heavily used in most of JavaScript code. Therefore, we employ constraint solver PASS [17] during test input generation.

PASS can handle constraints over integers, bit-vectors, floating-point numbers, and strings. While previous constraint solvers with support for string constraints used bit-vectors or automata, PASS introduced modeling through parameterized-arrays, which allows for more efficient solving. PASS converts string constraints into quantified expressions. The expressions are solved through an efficient and sound quantifier elimination algorithm. The algorithm speeds up identification of unsatisfiable cases. As the consequence, it can solve complex constraints corresponding to string manipulations within ECMAScript standard. Multiple optimizations are also introduced on incorporating PASS into SymJS. Such optimizations include dependency solving, cache solving and expression simplification to reduce computation within the solver.

```
symjs_assume(arg0.length==16);
```

Figure 3. Use of symjs_assume() to Constrain Length of arg0 to be 16

As the nature of symbolic execution, SymJS may suffer from path explosion on targeting programs with large state space. In order to eliminate program behavior of uninterest, SymJS can make use of `symjs_assume(assumption)` function call, which prunes state space violating the assumption. The code snippet shown in Figure 3 shows an example of constraining length of string `arg0` to be *16*.

C. Symbolic Stubs and Drivers

Symbolic variables are targets of test input generation through symbolic execution. SymJS allows definition of symbolic variables through function calls. The code snippet below shows an example of defining symbolic string variable. `var s = symjs_mk_symbolic_string();` While the example defines a symbolic variable of string type, functions `symjs_mk_symbolic_int()`, `symjs_mk_symbolic_bool()` and `symjs_mk_symbolic_real()` allow definition of symbolic variables with their type being integer, Boolean, and floating-point, respectively. While SymJS allow only string, integer, Boolean, and floating-point numbers to be symbolic, their constraints are retained on assignments/references as fields of more complex objects, allowing generation of tests with values of object fields varied.

In order to determine test inputs for the function `func0()` in Figure 1, additional code fragments are required. First, a symbolic driver shown in Figure 6 is required. The driver declares symbolic variables and passes them to the function as arguments. Stubs to inject dependencies are also required. A symbolic stub in Figure 7 includes a symbolic variable declaration. With the stub, return values of function `Lib.m2()` are included to test inputs obtained by SymJS.

D. Test Execution within Web Browsers

Functions `symjs_mk_symbolic_*()` used to define symbolic variables are interpreted as expressions to define new symbolic variables during test generation. SymJS itself allows for normal concrete test execution with the generated test inputs. During concrete execution, the functions return concrete values contained in test inputs.

SymJS can export test inputs into external test runners based on a test framework Jasmine [11]. The runners contain test playback library, which returns corresponding test input data on `symjs_mk_symbolic_*()` function calls. Figure 4 shows an example of test runner generated. Each of tests contains automatically generated test data in an array structure, and users can easily create new tests through duplication and modification of existing tests.

The runners can be loaded into typical web browser and allow for execution of generated tests with no custom JavaScript interpreter. The runner has an extension to Jasmine, which

```
describe("Test with underscore.string.symjs.js-camelize.js", function() {
    it("should run test 1", function() {
        replayList.init(
[ [ "arg0#0", null ], [ "arg1#1", false ] ]
        );
        var arg0 = symjs_mk_symbolic_string("arg0");
        var arg1 = symjs_mk_symbolic_bool("arg1");
        var retval = camelize(arg0, arg1);
        expect(true).toBe(true);     // default assertion
    });
});
```

Figure 4. Test Runner Code with Test Data

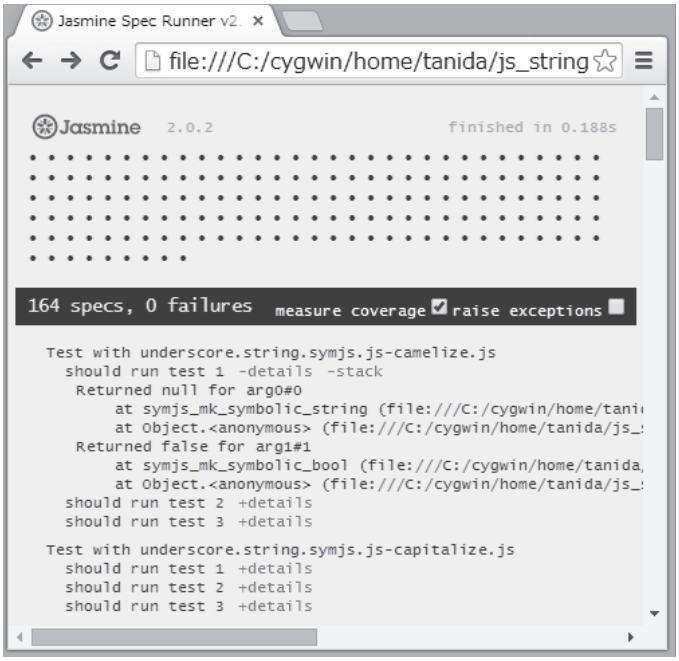

Figure 5. Test Runner View with Test Data and Stacktrace

prints test data and stacktrace on use of test input as shown in Figure 5.

III. Automatic Generation of Symbolic Stubs and Drivers

As explained in Section II-C, symbolic stubs and drivers are required to symbolically execute target functions and obtain test inputs. Symbolic stubs that return symbolic variables are used to generate return values of functions, which are called from functions under test. Symbolic drivers are needed to vary arguments passed to functions tested.

While it is possible to employ manually implemented symbolic stubs and drivers, additional cost is required for implementation. Therefore, it is desirable to have symbolic stubs and drivers automatically generated. Hence, we have decided to generate symbolic stubs and drivers in an automatic manner, and use them for test generation and execution.

A. Strategy for Generating Symbolic Stubs and Drivers

Our symbolic stub generation technique produces stub for functions and classes specified. Our driver generation technique emits code that invokes program under test.

As for stub generation, we have decided to generate functions, which just create and return objects according to type of return value expected by caller. The following is the mapping between expected type and returned object:

- String, integer, Boolean and floating-point numbers which SymJS can handle as symbolic
  (Hereafter referred to as SymJS primitives):
  Newly defined symbolic variable of the corresponding type.
- Other classes:
  Newly instantiated object of the expected type. If the class is targeted for stub generation, newly instantiated stub object is returned.
- Void: Nothing is returned.

In order to create stubs for classes, stubs for constructors also need to be generated. Here, we generate empty constructors, which result in all stateless objects. Our approach assumes there is no direct access to fields of stub classes, and does not generate stubs for fields.

We have to note even in case type of return value from a stub is a non-SymJS primitive, we may get multiple test inputs on invocation on the stub. That is the case if returned objects contain functions that return symbolic variables. The situation happens if the non-SymJS primitive class contain functions that return objects of SymJS primitive class, and the non-SymJS primitive class is stubbed. Therefore, it is possible to obtain more than one set of test inputs by calling functions returning non-SymJS primitive.

Symbolic drivers generated with our technique have the following functionality:

- If the function to be tested is not static and needs an object instance to be executed, instantiate an object of the corresponding class and call the function
- If the function is a static one, just call the function

As arguments passed to the function, drivers give the following objects according to the expected types:

```
var s = symjs_mk_symbolic_string("arg0");
var a = symjs_mk_symbolic_float("arg1");
func0(s,a);
```

Figure 6. Symbolic Driver to Execute Code in Figure 1

```
Lib.m2 = function() {
  return symjs_mk_symbolic_float();
};
```

Figure 7. Symbolic Stub Providing Lib.m2() Used in Figure 1

```
/** @return {Number} m2 value */
Lib.m2 = function() { ... };
```

Figure 8. Function Definition with an Annotation to Automatically Generate Symbolic Stub in Figure 7

```
/** @return {tx.Data} data */
tx.Ui.getValue = function(){ ... };
```

⇓

```
tx.Ui.getValue = function(){
    return new tx.Data();
};
```

Figure 9. Function with an Annotation Returning non-SymJS Primitive and Generated Symbolic Stub

```
/** @param {String} s
  * @param {Number} a */
function func0(s,a) { ... }
```

Figure 10. Annotations for Function under Test to Automatically Generate Symbolic Driver in Figure 6

- SymJS primitives:
  Newly defined symbolic variable of corresponding type.
- Other classes:
  Newly instantiated object of the expected type. If the class is targeted for stub generation, newly instantiated stub object is passed.

The manner to choose arguments is similar to the one resolves what to return in symbolic stubs.

B. Generating Symbolic Stubs and Drivers from Annotations

Symbolic stub/driver generation strategy proposed in Section III-A requires type information from target code. Types of return values expected by caller are required for stub generation. Types of arguments passed to functions under test are required to generate drivers.

However, JavaScript is a dynamically typing language, which makes it difficult to determine type of return values and arguments prior to run time. On the contrary, many JavaScript programs have some expectations in types of return values and arguments, which are often defined in Application Programming Interface (API) etc. Further, there is a way to express type information for JavaScript code in a machine readable manner, which is JSDoc-style annotation. Therefore, we have decided to obtain type information from annotations in JSDoc3 [20] convention, and generate symbolic drivers and stubs.

Symbolic stubs are generated from original source code of functions to generate stubs for. Functions need to contain annotations, which provide type information on return values of functions. Symbolic stub for a function can be generated if the type of its return values is obtained from annotations.

JSDoc3 allows for declaration of return value type, mainly through `@return` annotations. In order to generate symbolic stub for function `Lib.m2()` used in code snippet on Figure 1, an annotation like the one shown in Figure 8 is required. If such annotation is attached to original source code of the function, it is possible to figure out type of return values. From the obtained type of return values, the symbolic stub in Figure 7 can be generated in a fully automatic manner. The example demonstrates generation of symbolic stub for a function returning a SymJS primitive. An example of generating symbolic stub for a function that returns a non-SymJS primitive is shown in Figure 9.

Symbolic drivers are generated from source code of functions to be tested. Source code need to contain annotations expressing type of arguments passed to the function, in order to automatically generate symbolic driver to invoke the function.

Types of parameters passed to functions are often given through `@param` annotation for JSDoc3. Symbolic driver for the function `func0()` can be generated from the annotations in Figure 10, attached to the function. The annotations give types of parameters for the function, allowing generation of the symbolic driver in Figure 6.

The proposed technique for automatic generation of symbolic stub and drivers is implemented as plugins for JSDoc3. JSDoc3 allows implementation of custom plugins, and they may contain hooks to be invoked on finding classes or functions. Within the hooks, it is possible to obtain types for return values and parameters. The developed plugins automatically generate symbolic stubs and drivers for classes and functions found in input source code.

While we have proposed a technique to automatically generate symbolic stubs and drivers based on type information obtained from annotations, it is also possible to use type information from other sources. Such sources of type information include API specification documents.

IV. Evaluation

A. Experimental Setup

In order to confirm that our proposed technique can automatically generate and execute unit tests achieving high code coverage, we have performed experiments using two JavaScript programs with their statistics shown in Table V.

The first subject (INDUSTRIAL) corresponds to the client part of web application implemented upon our in-house framework for web application implementation. Within the target

Table V. Statistics on the Target Program

Name	INDUSTRIAL	UNDERSCORE.STRING
#Statement	123	427
#Public Function	22	57

program, calls to API not defined in ECMAScript standard are wrapped in our framework. As the consequence, it contains only calls to standard API or our framework. We have to note common API to manipulate HTML Document Object Model (DOM) or to communicate with servers are not part of ECMAScript standard and not used directly in the program.

The second subject (UNDERSCORE.STRING) is a free and open-source string manipulation library Underscore.string [21]. It provides many useful string operations, which are not standardized in JavaScript programming language. The library has no functionality involving HTML DOM manipulation or communication, and implemented using only functionality defined in ECMAScript standard. It can be employed on server-side Node.js platform as well as web browsers on clients. We have manually annotated source code of the target to provide argument type information required for automatic driver generation.

All experiments are performed on a workstation with Intel Xeon CPU E3-1245 V2@3.40GHz and 16GB RAM.

B. Generation of Symbolic Stubs and Drivers

In order to perform automatic generation of test input proposed in Section II, we have generated symbolic stubs and drivers with the technique explained in Section III.

Symbolic stubs to target INDUSTRIAL are generated from source code of the framework used to implement the application. Source code has annotations meeting JSDoc3 standard, which allow for retrieval of types for return values of functions. Stubs are successfully generated for all the classes and functions defined in the framework. UNDERSCORE.STRING that depends only on functionality provided by ECMAScript standard required no stub to be generated. As the program is implemented only upon API defined in ECMAScript language standard and the framework, all the stubs required for symbolic execution of the program are ready at this stage.

Symbolic drivers are generated from source code of the program under test. INDUSTRIAL had JSDoc3-style annotations as is. Manually annotated source code of UNDERSCORE.STRING is used to extract type information for function arguments within the subject. Types of arguments in UNDERSCORE.STRING could be found in its document, and the annotation process was straightforward. Drivers for all functions within the two targets are generated successfully.

C. Test Input Generation

All functions within the target programs are symbolically executed using the automatically generated drivers and stubs. Table VI contains statistics on the generated tests.

In the first trial, the subject programs are symbolically executed with no special configuration. While all functions in INDUSTRIAL are processed within *1* second, analysis of *2* functions in UNDERSCORE.STRING did not finish within timeout of *1* minute. In order to process the functions within reasonable time, we have introduced the following constraints during symbolic execution of UNDERSCORE.STRING.

- `count(string,substring)`, which counts number of `substring` occurrences in `string`, resulted in timeout assuming arbitrary string as `string` and `substring`. This is due to non-terminating loop and resulting large number of forked states during symbolic execution. we have limited maximum number of branches a state may go through to `7` from its default configuration of `20`.
- `words(string,delimiter)`, which counts number of words within `string` separated by `delimiter` also resulted in timeout, assuming arbitrary string as the two parameters. We have constrained length of two arguments to equal to *16* and *1*, respectively. Constraints can be introduced through insertion of `symjs_assume()` calls to the driver.

After introducing the constraints, all functions within UNDERSCORE.STRING are processed within *2* seconds.

D. Test Execution

Test inputs for all functions in INDUSTRIAL and UNDERSCORE.STRING are automatically exported to test runners based on Jasmine test framework. Code coverage during testing is measured with Blanket.js [22], and line coverage of *92.7%* and *76.0%* on average is obtained for INDUSTRIAL and UNDERSCORE.STRING, respectively.

Figure 11 shows distribution of statement coverage for functions in the benchmarks. The result shows our technique can generate unit test input with high coverage. For instance, 100% statement coverage is achieved with more than 60% of the functions. However, some of the benchmarks are not fully covered due to limitations in symbolic execution or stub/driver generation. In the sequel, we discuss the source of failure to cover some statements.

E. Code Not Covered in the Experiments

While the experimental results show that the proposed method can generate test input achieving high code coverage, 100% coverage is not reached, implying some portion of the target program is not exercised. Automatically generated test runner code shown in Figure 4 allows for manual modification and insertion of cases in order to test such code. However, additional labor is required and it is desirable to have such code automatically covered. The followings are the classes of code not executed, and possible enhancements to our methodology, which allows for coverage of missed code.

1) Code Exercised on Matches to Regular Expressions

JavaScript features regular expression library within its core ECMAScript language standard. The functionality is very useful to perform string manipulations and heavily used in UNDERSCORE.STRING. However, SymJS cannot obtain and manage path conditions corresponding to matches and unmatches on some regular expressions. The limitation results

Table VI. Statistics on the Tests Generated

Target	INDUSTRIAL	UNDERSCORE.STRING (w/o constraints)	UNDERSCORE.STRING (w/ constraints)
Functions with generated tests	22	55	57
#Test per function(max./avg./min.)	27/2.6/1	201/24.4/1	48/6.4/1
Max. test generation time per function (sec.)	< 1	> 60 (timed out)	< 2
Statement coverage(%)	92.7	73.5	76.0

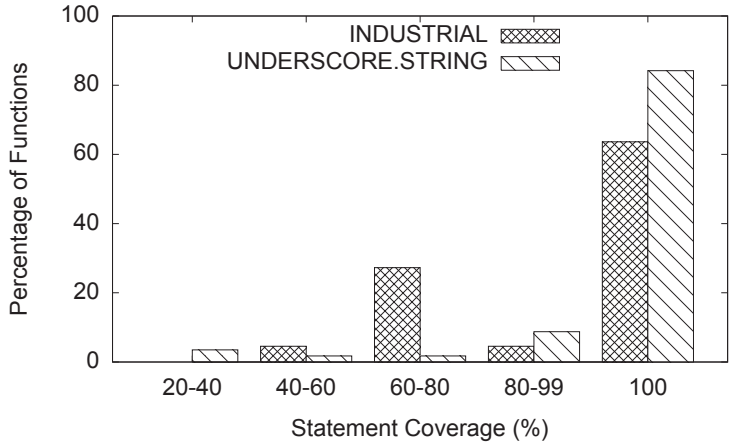

Figure 11. Distribution of Statement Coverage for Functions in INDUSTRIAL and UNDERSCORE.STRING

```
// _fmt is a symbolic String variable
// created from arguments to the function
if ((match = /^[^\x25]+/.exec(_fmt))
    !== null) {
    parse_tree.push(match[0]);
}
```

Figure 12. Code Exercised on Matches to Regular Expressions from sprintf() Function in UNDERSCORE.STRING

```
/**
 ...
 * @param {Number} position
 */
function endsWith(str, ends, position) {
  ...
  if (typeof position == 'undefined') {
    position = str.length - ends.length;
  }...
}
```

Figure 13. Code Handling Objects of Unexpected Type from endsWith() Function in UNDERSCORE.STRING

in failure to cover code to be exercised on matches of input string to such regular expressions.

11 functions in UNDERSCORE.STRING are not fully covered due to this limitation in handling regular expressions. Figure 12 shows such code fragment found in `sprintf()` function from UNDERSCORE.STRING. The restriction results in 57/75 statements within the large function missed, downpressing total coverage achieved with UNDERSCORE.STRING.

We are planning to enhance string constraint solver PASS, in order to model and handle larger class of regular expressions.

2) Code Handling Objects of Unexpected Type

As JavaScript is a dynamically typing language, objects of unexpected type might be returned from functions. In order to handle such scenario, the target programs contained type checking and subsequent branching code. However, symbolic stubs generated through our technique, always return an object of type specified in source code annotation. Such stubs fail to cover code portions handling objects of type different from annotations.

1 function in INDUSTRIAL and *2* functions in UNDERSCORE.STRING contain such code, and full coverage is not achieved. Figure 13 shows the corresponding code fragment from `endsWith()` function in UNDERSCORE.STRING.

Code handling objects of unexpected type can be exercised by making use of multiple symbolic stubs/drivers, which return/pass objects of different type. Currently, we support `@return` and `@param` annotations each specifying single type. However, JSDoc3 includes support for annotations with multiple possible types of arguments and return values. Our symbolic driver and stub generator can be extended easily to support such annotations.

3) Code with No Premise on Object Type

INDUSTRIAL also contained *1* function, which determines type of objects at run time and process them accordingly.

However, our technique cannot cover such procedures. From functions with types of their return values unknown, we generate stubs that return the default JavaScript "Object". As the consequence, code interacting with objects of custom class is uncovered.

Object types a variable may take can be extracted by means of static analysis or random tests. Our symbolic execution technique can be employed to create test input variation within the obtained type.

4) Code Iterating through Entries in Hash or Array

1 function within INDUSTRIAL had loop iterating through members in objects returned from a symbolic stub. Such control structure is often observed in JavaScript code, in order to make use of a plain "Object" as a hash table. However, automatically generated symbolic stub returns newly created "Object" with no members, and loop body is missed in the experiment. Code that inspects content of arrays from symbolic stubs is also expected to missed, as symbolic stubs generated by our tool return newly created empty arrays.

Loop bodies in such functions can be exercised by stubs that returns "Object" or array with one or more members contained.

5) Catch Blocks Handling Exceptions

1 function from INDUSTRIAL contained catch blocks for exceptions thrown from the framework used in the program. However, after replacing the framework with the automatically generated symbolic stubs, exceptions are never thrown and catch blocks are not exercised.

In order to cover catch blocks in target, we have to generate symbolic stubs that throws exception during execution, in addition to those do not throw exceptions.

V. DISCUSSION

A. Completeness

Our symbolic execution technique depends on modeling of computation performed, including complex one such as string manipulations. While we have employed constraint solver PASS, which is capable of handling complex string constraints, some constraints such as the one corresponding to code with regular expressions could not be handled.

Another limitation comes from automatically generated symbolic stubs/drivers from our technique. Type annotations used as input of our stub/driver generation technique, do not provide enough information to model environments where target software is executed. For instance, target software might be fed objects of unexpected type, or thrown exceptions. Our symbolic stub/driver generation technique does not take such situations into account.

However, compared to existing industrial practice of testing based on manually written tests, our technique can test behavior of wider scope in an automatic fashion.

B. Scope

Like other automated test input generation techniques based on symbolic execution, we do not automatically generate assertions to check target application behavior conformance with the obtained test inputs. In other words, users need to provide assertions/invariants to ensure target code is functioning as expected.

However, invariants and many assertions can be shared between multiple test cases, and costs for writing them is much smaller compared to those required to write tests from scratch. Further, our Jasmine-based test runner allows for insertion of global invariants, as well as to assertions specific to test cases.

C. Automation Level

Our test generation technique may require some user inputs on targeting complex applications such as UNDERSCORE.STRING. In the experiment, maximum number of branches and constraint to string length are required to end test generation for some functions. However, the number of parameters that needed to be adjusted are quite small. In addition, more than 95% of functions in the subject could be handled with the default configuration of the symbolic executor.

D. Correctness

Symbolic stubs generated with our technique always return newly defined symbolic variable. Such behavior may result in over-approximation of real system behavior before introducing stubs. For example, an expression containing multiple calls to a single function `getValue()!=getValue()` is unlikely to be satisfied, assuming target variable of the getter functions is not accessed from other execution contexts between two function calls. However, as the stubbed `getValue()` function returns newly defined symbolic variable on every call, it is possible to generate tests that make the expression true.

Combination of the generated test input and stub allows for reproduction of behavior that makes the expression true during test execution. However, the behavior might be quite different and hard to reproduce in real system under development. In that case, stubs need to be manually modified or developed, in order to mock behavior of environment in which software under test is executed.

E. Scalability

Our case studies confirm that our technique can be applied to real-world JavaScript code used in field. While we need to adjust some parameters used in test generation, we were able to target all functions in the benchmarks within *2* seconds.

However, we need to perform experiments on applications with their size varied. In order to target software of larger scale, test generation techniques such as DART [23] would be required, in addition to pure symbolic execution used in current SymJS.

F. Threats to Validity

Issues related to the external validity of our evaluation are handled in the discussions above. The internal validity of our experiments may depend on software tools used. We have minimized the chance by writing tests for the toolchain developed, which are completely different from the benchmarks used in the evaluation.

VI. Conclusions and Future Work

A. Conclusions

We proposed a technique to automatically generate unit test input data for JavaScript code. The technique makes use of a symbolic executor SymJS, in order to achieve high code coverage during testing. The technique is a two-phase approach, consisting of the following fully-automatic steps:

1) Symbolic stub/driver generation based on type information obtained from annotations
2) Test input generation through symbolic execution of target code

The experiments were conducted targeting client part of proprietary web application and open-source string manipulation library. Our technique generated tests that achieve line coverage higher than 75%, and more than 60% of functions in the subject are fully covered with the generated tests. The results show the technique can automate generation and execution of high-coverage unit tests for large portion of JavaScript code in the field.

B. Future Work

Future work includes more verification trials with variety of target programs. While we have performed experiments with programs of relatively small size, experiments on larger targets are also required.

In order to exercise target code missed in the experiment automatically, constraint solver PASS and symbolic stub/driver generator need to be improved. However, our methodology allows for manual modification of generated tests to cover such code, which is found in less 40% of the subject in the evaluation.

In the experiment, we have targeted JavaScript code with HTML DOM API encapsulated in our custom framework (INDUSTRIAL) and code that involves API defined in ECMAScript standard associated only (UNDERSCORE.STRING). As the consequence, symbolic stubs required for test generation and execution in the experiment were only those corresponding to our custom framework. However, in order to target JavaScript code, which has interactions with server-side API such the one in Node.js or client-side API for HTML DOM manipulations, symbolic stubs for corresponding APIs need to be developed. To target mobile applications, it is required to prepare symbolic stubs for frameworks used in their implementation. Development support techniques for new symbolic stubs are necessary, in order to support larger set of platforms with JavaScript code deployed with reduced effort.

References

[1] H. Tanida, G. Li, M. Prasad, and T. Uehara, "Automatic Unit Test Generation and Execution for JavaScript Program through Symbolic Execution," in Proceedings of the Ninth International Conference on Software Engineering Advances, 2014, pp. 259–265.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in Proceedings of the 13th ACM Conference on Computer and Communications Security, 2006, pp. 322–335.

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 209–224.

[4] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2010, pp. 179–180.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in Proceedings of the 10th European Software Engineering Conference, 2005, pp. 263–272.

[6] N. Tillmann and J. De Halleux, "Pex: White Box Test Generation for .NET," in Proceedings of the 2nd International Conference on Tests and Proofs, ser. TAP'08, 2008, pp. 134–153.

[7] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," Queue, 2012, pp. 20:20–20:27.

[8] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," Commun. ACM, vol. 54, no. 9, 2011, pp. 69–77.

[9] "Node.js," https://nodejs.org/, [Online; accessed 2015.05.28].

[10] "PhoneGap — Home," http://phonegap.com/, [Online; accessed 2015.05.28].

[11] "Jasmine: Behavior-Driven JavaScript," http://jasmine.github.io/, [Online; accessed 2015.05.28].

[12] "QUnit: A JavaScript Unit Testing framework," http://qunitjs.com/, [Online; accessed 2015.05.28].

[13] "Mocha - the fun, simple, flexible JavaScript test framework," http://mochajs.org/, [Online; accessed 2015.05.28].

[14] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in Proceedings of the 2010 IEEE Symposium on Security and Privacy, 2010, pp. 513–528.

[15] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 488–498.

[16] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 449–459.

[17] G. Li and I. Ghosh, "PASS: String Solving with Parameterized Array and Interval Automaton," in Proceedings of Haifa Verification Conference, 2013, pp. 15–31.

[18] ECMA International, Standard ECMA-262 - ECMAScript Language Specification, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[19] "Rhino," https://developer.mozilla.org/en-US/docs/Rhino, [Online; accessed 2015.05.28].

[20] "Use JSDoc," http://usejsdoc.org/index.html, [Online; accessed 2015.05.28].

[21] "underscore.string," https://epeli.github.io/underscore.string/, [Online; accessed 2015.05.28].

[22] "Blanket.js —Seamless javascript code coverage," http://blanketjs.org/, [Online; accessed 2015.05.28].

[23] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 213–223.