# Automatic Unit Test Generation and Execution for JavaScript Program through Symbolic Execution

Hideo Tanida, Tadahiro Uehara

Software Engineering 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. 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 code, and stub/driver generation engine which automatically generate stub for code of uninterest. Our methodology allows for fully 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, stub 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 approaches are considered to be solutions for the problem. One such technique is test generation through symbolic execution, which achieves higher code coverage compared to random testing [1]–[6].

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) [7] 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 running 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++ [1][2][4], Java [3], and binary code [5][6]. It is reported that the tools can automatically detect corner-case bugs, or generate test inputs to achieve high code coverage.

Existing tools for JavaScript code include Kudzu [8] and Jalangi [9]. Kudzu automatically generates input data for program functions, with the aim of automatically discovering security problems in the target. Jalangi allows 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 creation of driver/stub used for testing.

We propose a technique to generate test inputs for JavaScript code through symbolic execution on a tool SymJS. Test inputs generated by the tool allows for automatic unit test execution. 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 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 field for the following reasons. First, our constraint solver PASS [10] allows test generation for programs with various complex string manipulations. Secondly, SymJS does not require any modification to the target code, while the existing symbolic executors for JavaScript [8][9] needed modifications and multiple runs.

Further, our automatic stub/driver code generation allows for fully automatic test data generation. An existing work [9] could be employed for generation of unit tests. However, it required manual coding of stub/driver, which requires knowledge

```
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

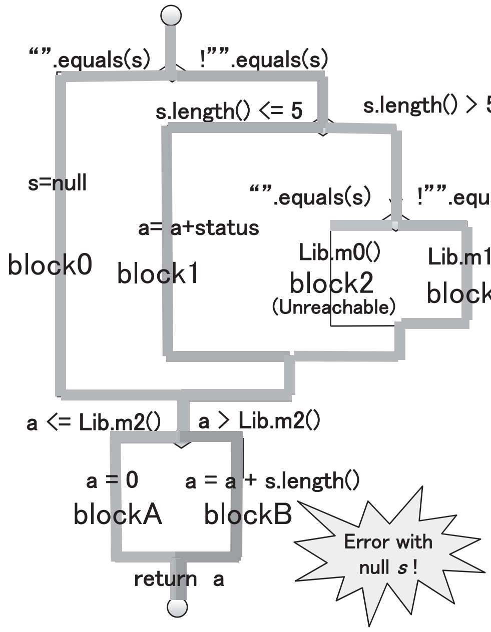

Figure 2. Execution Paths within Code Shown in Figure 1

on symbolic execution and error-prone. 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. Finally, we come to the conclusion in Section V and discuss 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 set 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

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` |

in Figure 2. Block 2 has a contradiction between conditions for execution and will never be executed. However, the other blocks have no such contradiction and 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 and 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

TABLE II. INSTRUCTIONS WITH THEIR INTERPRETATIONS MODIFIED FROM ORIGINAL RHINO

| | |
|---|---|
| Arithemetic/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. |

TABLE I) are calculated iteratively. After visiting every possible path within the program, constraints for 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 which 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 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 [2] and Symbolic JPF [3] do. SymJS handles program code meeting the language standard defined in ECMAScript [11].

SymJS is an extended version of Rhino [12], 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 target program source code. This approach is taken by existing symbolic executors such as KLEE [2] and Symbolic PathFinder [3]. Handling bytecode instead of source allows for implementation of symbolic executors without dealing with complex syntax of the 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.

TABLE III. COMPARISON OF SYMBOLIC EXECUTORS

| **Tool** | Target Lang. | Sym. VM | Dep./Cache Solving | String Solving |
|---|---|---|---|---|
| SymJS | JavaScript | Yes | Yes | Yes |
| KLEE [2] | C | Yes | Yes | No |
| SAGE [6] | x86 binary | No | Yes | No |
| Sym JPF [3] | Java | Yes | No | No |
| Kudzu [8] | JavaScript | No | No | Yes |
| Jalangi [9] | 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 |

Comparison instructions are followed by branch instructions in Rhino bytecode. SymJS handles comparison 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 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 created on hitting branches etc. The first method is to store program state variables including content of heap/stack, as is done in [2][3]. 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 [4][6]. However, our technique is implemented upon our symbolic executor and does not need modification of target code required with the existing tools [8][9] 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 [10] during test input generation.

PASS can handle constraints over integers, bit-vectors, floating-point numbers, and strings. While previous constraint solvers supporting string constraints used bit-vectors or automata, PASS introduced modeling through parameterized-arrays which allows for more efficient solving. As the consequence, it can solve most of constraints corresponding to string manipulations within ECMAScript standard.

### 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 members of more complex objects, allowing generation of tests with value of object members changing.

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 3 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 4 includes a symbolic variable declaration. With the stub, the return value of the function call to `Lib.m2()` is included to test inputs obtained by SymJS.

Functions `symjs_mk_symbolic_*()` used to define symbolic variables is interpreted as expressions to define new symbolic variables during test generation. SymJS allows for normal concrete 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 files in JavaScript Object Notation (JSON) format. The files can be read by test playback library which returns corresponding test input data on `symjs_mk_symbolic_*()` function calls. The library loaded into typical web browser enables execution of generated tests with no custom JavaScript interpreter.

## 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 which 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 be 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 which calls functions specified.

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 is a non-SymJS primitive, we may get multiple test inputs. That is the case if functions defined in the returned object return symbolic variables. The situation happens if the non-SymJS primitive class contain functions which 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:

- 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.

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

Figure 3. Symbolic Driver to Execute Code in Figure 1

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

Figure 4. Symbolic Stub Providing Lib.m2() Used in Figure 1

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

Figure 5. Function Definition with an Annotation to Automatically Generate Symbolic Stub in Figure 4

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

$\Downarrow$

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

Figure 6. Function with an Annotation Returning non-SymJS Primitive and Generated Symbolic Stub

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

Figure 7. Annotations for Function under Test to Automatically Generate Symbolic Driver in Figure 3

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 given 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 [13] 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 type of its return values is obtained from annotations.

JSDoc3 allows 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 5 is required. If such annotation is attached to original source code of the function, it is possible to figure out type for return values. From the obtained type for return values, the symbolic stub in Figure 4 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 which returns a non-SymJS primitive is shown in Figure 6.

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.

Type of parameters passed to functions are often given with `@param` annotation for JSDoc3. Symbolic driver for the function `func0()` can be generated from the annotations in Figure 7, attached to the function. The annotations give types of parameters for the function, allowing generation of the symbolic driver in Figure 3.

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 contained in program source code fed to JSDoc3.

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

## IV. Evaluation

In order to confirm that our proposed technique can automatically generate and execute unit tests achieving high code coverage, we have performed experiments using an industrial JavaScript program. The program corresponds to the client part of web application implemented upon our custom framework for web application implementation. The program calls to API not defined in ECMAScript standard wrapped in our framework, and it contains only calls to standard API or our framework. We have to note common API to manipulate HTML Document Object Model (DOM) or communicate with servers are not part of ECMAScript standard and they are not used directly in the program. TABLE V shows statistics on the target program.

### *A. 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 through technique explained in Section III.

Symbolic stubs are generated from source code of the framework used to implement the application. Source code has annotations meeting JSDoc3 standard which allow for retrieval

TABLE V. Statistics on the Target Program

| #Line | #Function | #File |
|---|---|---|
| 431 | 23 | 1 |

TABLE VI. Statistics on the Framework Source Code Used for Stub Generation and Generated Stub

| #Line(Orig.) | #Line(Stub) | #Function | #File |
|---|---|---|---|
| 2843 | 1304 | 154 | 13 |

of types for return values of functions. Stubs are sucessfully generated for all the classes and functions defined in the framework. TABLE VI lists figures on the original framework source code and generated symbolic stubs. 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 for the program under test. JSDoc3-style annotations can be found in the target program as well, and it is possible to determine types of arguments required for generation of symbolic drivers. Drivers for all 23 functions within the program are generated.

## B. Test Input Generation and Test Execution

Each function within the target program is executed in a symbolic manner, using the automatically generated drivers and stubs. Test inputs containing concrete values of symbolic variables are obtained at the end of executions.

Symbolic execution of all functions finished *within 1 second* and test inputs are generated. Number of test inputs, which are assignments of concrete values into symbolic variables, differed between target functions. Only *1* test input is generated for functions with no branch, while number of tests varied to *27* obtained with a more complex function.

Target functions are concretely executed with test inputs obtained. Test playback library running on a web browser is used to replay the tests. Code coverage during testing is measured with JSCover [14], and line coverage of *92%* was obtained. The result shows our technique can generate unit test input achieving high code coverage fully automatically.

## C. 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. The followings are the classes of code not executed with our methodology.

Code handling objects of unexpected type is not covered. As JavaScript is a dynamically typing language, objects of unexpected type might be returned by functions. In order to handle such scenario, the target program contained type checking and subsequent error handling code. However, symbolic stubs generated through our technique, always return a object of type described in source code annotation. Such stubs fail to utilize code portions handling objects of type different from annotations.

Code with no premise on object type is also missed. The target program contained code fragments which determine 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 returning default JavaScript “Object”. Therefore, code interacting with objects of custom class is uncovered.

Catch blocks handling exceptions is left. The target program contained catch blocks for exceptions thrown from the framework used in the program. However, after replacing the framework with the automatically generated symbolic stubs which throw no exception, they are not exercised.

## V. 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 execution engine, 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 experiment with an industrial JavaScript program shows the technique can generate tests achieving line coverage of 92%. The result shows our technique can automate generation and execution of unit tests for JavaScript code.

### 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, symbolic stubs need to be improved. Code handling objects of unexpected/unknown type can be kicked by symbolic stubs which return various types of objects. Code handling exceptions can be triggered with symbolic stubs throwing exceptions. In addition to more complex automatic stub generation strategies, manual modifications to automatically generated stubs are considered effective to increase coverage.

In the experiment, we have targeted JavaScript code with HTML DOM handling encapsulated in our framework, allowing test generation and execution only with creation of symbolic stub for the framework. In order to target JavaScript code containing manipulation on HTML DOM, symbolic stubs for HTML DOM API need to be developed. To target mobile applications, it is required to write symbolic stubs for frameworks used in mobile application implementation.

## References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] "Rhino," https://developer.mozilla.org/en-US/docs/Rhino, [Online; accessed 2014.08.15].

[13] "Use JSDoc," http://usejsdoc.org/index.html, [Online; accessed 2014.08.15].

[14] "JSCover - JavaScript code coverage," http://tntim96.github.io/JSCover/ http://usejsdoc.org/index.html, [Online; accessed 2014.08.15].