JEST: $N+1$-version Differential Testing of Both JavaScript Engines and Specification

Jihyeok Park
School of Computing
KAIST
Daejeon, South Korea
jhpark0223@kaist.ac.kr

Seungmin An
School of Computing
KAIST
Daejeon, South Korea
h2oche@kaist.ac.kr

Dongjun Youn
School of Computing
KAIST
Daejeon, South Korea
f52985@kaist.ac.kr

Gyeongwon Kim
School of Computing
KAIST
Daejeon, South Korea
gyeongwon.kim@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

***Abstract*—Modern programming follows the continuous integration (CI) and continuous deployment (CD) approach rather than the traditional waterfall model. Even the development of modern programming languages uses the CI/CD approach to swiftly provide new language features and to adapt to new development environments. Unlike in the conventional approach, in the modern CI/CD approach, a language specification is no more the oracle of the language semantics because both the specification and its implementations (interpreters or compilers) can co-evolve. In this setting, both the specification and implementations may have bugs, and guaranteeing their correctness is non-trivial.**

In this paper, we propose a novel *N+1-version differential testing* to resolve the problem. Unlike the traditional differential testing, our approach consists of three steps: 1) to automatically synthesize programs guided by the syntax and semantics from a given language specification, 2) to generate conformance tests by injecting assertions to the synthesized programs to check their final program states, 3) to detect bugs in the specification and implementations via executing the conformance tests on multiple implementations, and 4) to localize bugs on the specification using statistical information. We actualize our approach for the JavaScript programming language via JEST, which performs N+1-version differential testing for modern JavaScript engines and ECMAScript, the language specification describing the syntax and semantics of JavaScript in a natural language. We evaluated JEST with four JavaScript engines that support all modern JavaScript language features and the latest version of ECMAScript (ES11, 2020). JEST automatically synthesized 1,700 programs that covered 97.78% of syntax and 87.70% of semantics from ES11. Using the assertion-injected JavaScript programs, it detected 44 engine bugs in four different engines and 27 specification bugs in ES11.

***Index Terms*—JavaScript, conformance test generation, mechanized specification, differential testing**

I. Introduction

In Peter O'Hearn's keynote speech in ICSE 2020, he quoted the following from Mark Zuckerberg's Letter to Investors [1]:

> The Hacker Way is an approach to building that involves continuous improvement and iteration. Hackers believe that somethings can always be better, and that nothing is ever complete.

Indeed, modern programming follows the continuous integration (CI) and continuous deployment (CD) approach [2] rather than the traditional waterfall model. Instead of a sequential model that divides software development into several phases, each of which takes time, CI/CD amounts to a cycle of quick software development, deployment, and back to development with feedback. Even the development of programming languages uses the CI/CD approach.

Consider JavaScript, one of the most widely used programming languages for client-side and server-side programming [3] and embedded systems [4]–[6]. Various JavaScript engines provide diverse extensions to adapt to fast-changing user demands. At the same time, ECMAScript, the official specification that describes the syntax and semantics of JavaScript, is annually updated since ECMAScript 6 (ES6, 2015) [7] to support new features in response to user demands. Such updates in both the specification and implementations in tandem make it difficult for them to be in sync.

Another example is Solidity [8], the standard smart contract programming language for the Ethereum blockchain. The Solidity language specification is continuously updated, and the Solidity compiler is also frequently released. According to Hwang and Ryu [9], the average number of days between consecutive releases from Solidity 0.1.2 to 0.5.7 is 27. In most cases, the Solidity compiler reflects updates in the specification, but even the specification is revised according to the semantics implemented in the compiler. As in JavaScript, bidirectional effects in the specification and the implementation make it hard to guarantee their correspondence.

In this approach, both the specification and the implementation may contain bugs, and guaranteeing their correctness is a challenging task. The conventional approach to build a programming language is uni-directional from a language specification to its implementation. The specification is believed to be correct and the conformance of an implementation to the specification is checked by dynamic testing. Unlike in the conventional approach, in the modern CI/CD approach, the specification may not be the oracle, because both the specification and the implementation can co-evolve.

In this paper, we propose a novel *N+1-version differential testing*, which enables testing of co-evolving specifications and their implementations. The differential testing [10] is a testing technique, which executes N implementations of a specification concurrently for each input, and detects a problem when the outputs are in disagreement. In addition to N implementations, our approach tests the specification as well using a mechanized specification. Recently, several approaches

1558-1225/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00015

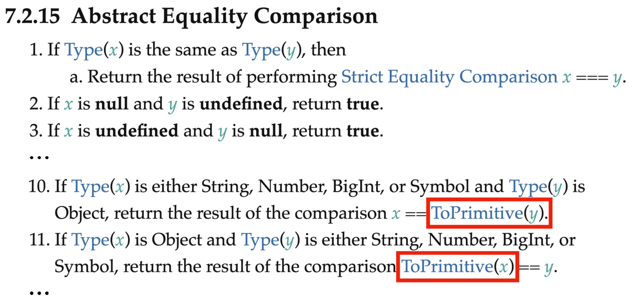
to extract syntax and semantics directly from language specifications are presented [11]–[13]. We utilize them to bridge the gap between specifications and their implementations through conformance tests generated from mechanized specifications. The N+1-version differential testing consists of three steps: 1) to automatically synthesize programs guided by the syntax and semantics from a given language specification, 2) to generate conformance tests by injecting assertions to the synthesized programs to check their final program states, 3) to detect bugs in the specification and implementations via executing the conformance tests on multiple implementations, and 4) to localize bugs on the specification using statistical information.

Given a language specification and N existing real-world implementations of the specification, we automatically generate a conformance test suite from the specification with assertions in each test code to make sure that the result of running the code conforms to the specification semantics. Then, we run the test suite for N implementations of the specification. Because generated tests strictly comply with the specification, they reflect specification errors as well, if any. When one of the implementations fails in running a test, the implementation may have a bug, as in the differential testing. When most of the implementations fail in running a test, it is highly likely that the specification has a bug. By automatically generating a rich set of test code from the specification and running them with implementations of the specification, we can find and localize bugs either in the specification written in a natural language or in its implementations.

To show the practicality of the proposed approach, we present JEST, which is a JavaScript Engines and Specification Tester using N+1-version differential testing. We implement JEST by extending JISET [11], a JavaScript IR-based semantics extraction toolchain, to utilize the syntax and semantics automatically extracted from ECMAScript. Using the extracted syntax, our tool automatically synthesizes initial seed programs and expands the program pool by mutating specific target programs guided by semantics coverage. Then, the tool generates conformance tests by injecting assertions to synthesized programs. Finally, JEST detects and localizes bugs using execution results of the tests on N JavaScript engines. We evaluate our tool with four JavaScript engines (Google V8 [14], GraalJS [15], QuickJS [16], and Moddable XS [17]) that support all modern JavaScript language features and the latest ECMAScript (ES11, 2020).

The main contributions of this paper include the following:

- Present *N+1-version differential testing*, a novel solution to the new problem of co-evolving language specifications and their implementations.
- Implement N+1-version differential testing for JavaScript engines and ECMAScript as a tool called JEST. It is the first tool that automatically generates conformance tests for JavaScript engines from ECMAScript. While the coverage of Test262, the official conformance tests, is 91.61% for statements and 82.91% for branches, the coverage of the conformance tests generated by the tool is 87.70% for statements and 78.30% for branches.



7.2.15 Abstract Equality Comparison

1. If Type(x) is the same as Type(y), then
   a. Return the result of performing Strict Equality Comparison x === y.
2. If x is **null** and y is **undefined**, return **true**.
3. If x is **undefined** and y is **null**, return **true**.

…

10. If Type(x) is either String, Number, BigInt, or Symbol and Type(y) is Object, return the result of the comparison x == ToPrimitive(y).
11. If Type(x) is Object and Type(y) is either String, Number, BigInt, or Symbol, return the result of the comparison ToPrimitive(x) == y.

…

(a) The **Abstract Equality Comparison** abstract algorithm in ES11

```
// JavaScript engines: exception with "err"
// ECMAScript (ES11) : result === false
var obj = { valueOf: () => { throw "err"; } };
var result = 42 == obj;
```

(b) JavaScript code using abstract equality comparison

```
try {
  var obj = { valueOf: () => { throw "err"; } };
  var result = 42 == obj;
  assert(result === false);
} catch (e) {
  assert(false);
}
```

(c) JavaScript code with injected assertions

Fig. 1: Abstract algorithm in ES11 and code example using it

- Evaluate JEST with four modern JavaScript engines and the latest ECMAScript, ES11. Using the generated conformance test suite, the tool found and localized 44 engine bugs in four different engines and 27 specification bugs in ES11.

II. N+1-VERSION DIFFERENTIAL TESTING

This section introduces the core concept of N+1-version differential testing with a simple running example. The overall structure consists of two phases: a conformance test generation phase and a bug detection and localization phase.

A. *Main Idea*

Differential testing utilizes the cross-referencing oracle, which is an assumption that any discrepancies between program behaviors on the same input could be bugs. It compares the execution results of a program with the same input on N different implementations. When an implementation produces a different result from the one by the majority of the implementations, differential testing reports that the implementation may have a bug.

On the contrary, N+1-version differential testing utilizes not only the cross-referencing oracle using multiple implementations but also a mechanized specification. It first generates test code from a mechanized specification, and tests N different implementations of the specification using the generated test code as in differential testing. In addition, it can detect possible bugs in the specification as well when most implementations fail for a test. In such cases, because a bug in the specification could be triggered by the test, it localizes the bug using statistical information as we explain later in this section.

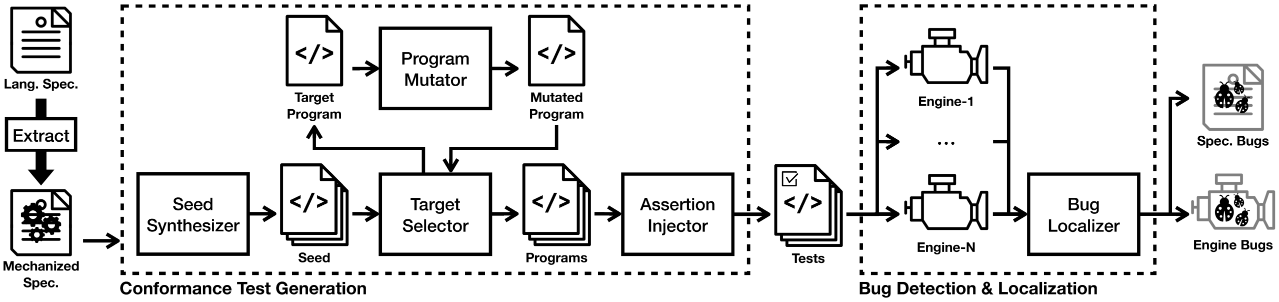


Fig. 2: Overall structure of N+1-version differential testing for N implementations (engines) and one language specification

B. Running Example

We explain how N+1-version differential testing works with a simple JavaScript example shown in Figure 1.

Figure 1(a) is an excerpt from ECMAScript 2020 (ES11), which shows some part of the **Abstract Equality Comparison** abstract algorithm. It describes the semantics of non-strict equality comparison such as `==` and `!=`. For example, `null == undefined` is `true` because of the algorithm step 2. According to the steps 10 and 11, if the type of a value is String, Number, BigInt, or Symbol, and the type of the other value is Object, the algorithm calls **ToPrimitive** to convert the JavaScript object to a primitive value. Note that this is a specification bug caused by unhandled abrupt completions! To express control diverters such as exceptions, `break`, `continue`, `return`, and `throw` statements in addition to normal values, ECMAScript uses "abrupt completions." ECMAScript annotates the question mark prefix (?) to all function calls that may return abrupt completions to denote that they should be checked. However, even though **ToPrimitive** can produce an abrupt completion, the calls of **ToPrimitive** in steps 10 and 11 do not use the question mark, which is a bug.

Now, let's see how N+1-version differential testing can detect the bug in the specification. Consider the example JavaScript code in Figure 1(b), which triggers the above specification bug. In the **Abstract Equality Comparison** algorithm, variables x and y respectively denote `42` and an object with a property named `valueOf` whose value is a function throwing an error. Step 10 calls **ToPrimitive** with the object as its argument, and the call returns an abrupt completion because the call of `valueOf` throws an error. However, because the call of **ToPrimitive** in step 10 does not use the question mark, the specification semantics silently ignores the abrupt completion and returns `false` as the result of comparison. Using the specification semantics, we can inject assertions to check that the code does not throw any errors as shown in Figure 1(c). Then, by running the code with the injected assertions on N JavaScript engines, which throw errors, we can find that the specification may have a bug. Moreover, we can localize the bug using statistical information: because most conformance tests that go through steps 10 and 11 of the algorithm would fail in most of JavaScript engines, we can use the information to localize the bug in the steps 10 and 11 of **Abstract Equality Comparison** with high probability.

C. Overall Structure

Figure 2 depicts the overall structure of N+1-version differential testing for N different implementations (engines) and one language specification. It takes a mechanized specification extracted from a given language specification, it first performs the conformance test generation phase, which automatically generates conformance tests that reflect the language syntax and semantics described in the specification. Then, it performs the bug detection and localization phase, which detects and localizes bugs in the engines or the specification by comparing the results of the generated tests on N engines.

The functionalities of each module in the overall structure are as follows:

1) Seed Synthesizer: The first module of the conformance test generation phase is Seed Synthesizer, which synthesizes an initial seed programs using the language syntax. Its main goal is to synthesize (1) a few number of (2) small-sized programs (3) that cover possible cases in the syntax rules as many as possible.

2) Target Selector: Starting from the seed programs generated by Seed Synthesizer as the initial *program pool*, Target Selector selects a target program in the program pool that potentially increases the coverage of the language semantics by the pool. From the selected target program, Program Mutator constructs a new mutated program and adds it to the program pool. When specific criteria, such as an iteration limit, are satisfied, Target Selector stops selecting target programs and returns the program pool as its result.

3) Program Mutator: The main goal of Program Mutator is to generate a new program by mutating a given target program in order to increase the coverage of the language semantics by the program pool. If it fails to generate a new program to increase the semantics coverage, Target Selector retries to select a new target program and repeats this process less than a pre-defined iteration limit.

4) Assertion Injector: Finally, the conformance test generation phase modifies the programs in the pool to generate conformance tests by injecting appropriate assertions reflecting the semantics described in the specification. More specifically, Assertion Injector executes each program in the pool on the mechanized specification and obtains the final state of its execution. It then automatically injects assertions to the program using the final state.

Algorithm 1: Worklist-based Shortest String

```
Input: $\mathbb{R}$ - syntax reduction rules
Output: $M$ - map from non-terminals to shortest
        strings derivable from them
Function shortestStrings($\mathbb{R}$):
    $M = \varnothing, W =$ a queue that contains $\mathbb{R}$
    while $W \neq \varnothing$ do
        pop $(A, \alpha) \leftarrow W$
        if update$(A, \alpha)$ then propagate$(W, \mathbb{R}, A)$
Function update($A, \alpha$):
    $str =$ an empty string
    forall $s \in \alpha$ do
        if $s$ is a terminal $t$ then $str = str + t$
        else if $s$ is a non-terminal $A' \wedge A' \in M$ then
            $str = str + M[A']$
        else return false
    if $\exists M[A] \wedge ||str|| \geq ||M[A]||$ then return false
    $M[A] = str$
    return true
Function propagate($W, \mathbb{R}, A$):
    forall $(A', \alpha') \in \mathbb{R}$ do
        if $A \in \alpha'$ then push $(A', \alpha') \rightarrow W$
```

Algorithm 2: Non-Recursive Synthesize

```
Input: $\mathbb{R}$ - syntax reduction rules, $S$ - start symbol
Output: $D$ - set of strings derivable from $S$
Function nonRecSynthesize($\mathbb{R}, S$):
    $V = \varnothing, M =$ shortestStrings$(\mathbb{R})$
    return getProd$(M, V, \mathbb{R}, S)$
Function getProd($M, V, \mathbb{R}, A$):
    if $A \in V$ then return $\{M[A]\}$
    $D = \varnothing, V = V \cup \{A\}$
    forall $(A', \alpha) \in \mathbb{R}$ s.t. $A' = A$ do
        $D = D \cup$ getAlt$(M, V, \mathbb{R}, A, \alpha)$
    return $D$
Function getAlt($M, V, \mathbb{R}, A, \alpha$):
    $L =$ an empty list
    forall $s \in \alpha$ do
        if $s$ is a terminal $t$ then
            append $(\{t\}, t)$ to $L$
        else if $s$ is a non-terminal $A'$ then
            append (getProd$(M, V, \mathbb{R}, A')$, $M[A]$) to $L$
    $D =$ point-wise concatenation of first elements of
      pairs in $L$ using second elements as default ones.
    return $D$
```

5) Bug Localizer: Then, the second phase executes the conformance tests on N engines and collects their results. For each test, if a small number of engines fail, it reports potential bugs in the engines that fail the test. Otherwise, it reports potential bugs in the specification. In addition, its Bug Localizer module uses *Spectrum Based Fault Localization* (SBFL) [18], a localization technique utilizing the coverage and pass/fail results of test cases, to localize potential bugs.

III. N+1-VERSION DIFFERENTIAL TESTING FOR JAVASCRIPT

We actualize N+1-version differential testing for the JavaScript programming language as JEST, which uses modern JavaScript engines and ECMAScript.

A. Seed Synthesizer

JEST synthesizes seed programs using two synthesizers.

1) Non-Recursive Synthesizer: The first synthesizer aims to cover as many syntax cases as possible in two steps: 1) to find the shortest string for each non-terminal and 2) to synthesize JavaScript programs using the shortest strings. For presentation brevity, we explain simple cases like terminals and non-terminals, but the implementation supports the extended grammar of ECMAScript such as parametric non-terminals, conditional alternatives, and special terminal symbols.

The shortestStrings function in Algorithm 1 shows the first step. We modified McKenize's algorithm [19] that finds random strings to find the shorted string. It takes syntax reduction rules $\mathbb{R}$, a set of pairs of non-terminals and alternatives, and returns a map M from non-terminals to shortest strings derivable from them. It utilizes a worklist W, a queue structure that includes syntax reduction rules affected by updated non-terminals. The function initializes the worklist W with all the syntax reduction rules $\mathbb{R}$. Then, for a syntax reduction rule (A, α), it updates the map M via the update function, and propagates updated information via the propagate function. The update function checks whether a given alternative α of a non-terminal A can derive a string shorter than the current shortest one using the current map M. If possible, it stores the mapping from the non-terminal A to the newly found shortest string in M and invokes propagate. The propagate function finds all the syntax reduction rules whose alternatives contain the updated non-terminal A and inserts them into W. The shortestStrings function repeats this process until the worklist W becomes empty.

Using shortest strings derivable from non-terminals, the nonRecSynthesize function in Algorithm 2 synthesize programs. It takes syntax reduction rules $\mathbb{R}$ and a start symbol S. For the first visit with a non-terminal A, the getProd function returns strings generated by getAlt with alternatives of the non-terminal A. For an already visited non-terminal A, it returns the single shortest string $M[A]$. The getAlt function takes a non-terminal A with an alternative α and returns a set of strings derivable from α via point-wise concatenation of strings derived by symbols of α. When the numbers of strings derived by symbols are different, it uses the shortest strings derived by symbols as default strings.

For example, Figure 3 shows a simplified *MemberExpres-*

MemberExpression **:**
  PrimaryExpression
  MemberExpression **[** *Expression* **]**
  MemberExpression **.** *IdentifierName*
  new *MemberExpression* *Arguments*

Fig. 3: The *MemberExpression* production in ES11

sion production in ES11. For the first step, we find the shortest string for each non-terminal: `()` for *Arguments* and `x` for the other non-terminals. Note that we use pre-defined shortest strings for identifiers and literals such as `x` for identifiers and `0` for numerical literals. In the next step, we synthesize strings derivable from *MemberExpression*. The first alternative is a single non-terminal *PrimaryExpression*, which is never visited. Thus, it generates all cases of *PrimaryExpression*. The fourth alternative consists of one terminal **new** and two non-terminals *MemberExpression* and *Arguments*. Because *MemberExpression* is already visited, it generates a single shortest string `x`. For the first visit of *Arguments*, it generates all cases: `()`, `(x)`, `(...x)`, and `(x,)`. Note that the numbers of strings generated for symbols are different. In such cases, we use the shortest strings for symbols like `x` for *MemberExpression* as follows:

new	*MemberExpression*	*Arguments*
new	**x**	**()**
new	**x**	**(x)**
	(the shortest string of *MemberExpression*)	**(...x)**
		(x,)

➡

```
new x()
new x(x)
new x(...x)
new x(x,)
```

2) Built-in Function Synthesizer: JavaScript supports diverse built-in functions for primitive values and built-in objects. To synthesize JavaScript programs that invoke built-in functions, we extract the information of each built-in function from the mechanized ECMAScript. We utilize the `Function.prototype.call` function to invoke built-in functions to easily handle the `this` object in Program Mutator; we use a corresponding object or `null` as the `this` object by default. In addition, we synthesize function calls with optional and variable number of arguments and built-in constructor calls with the `new` keyword.

Consider the following `Array.prototype.indexOf` function for JavaScript array objects that have a parameter *searchElement* and an optional parameter *fromIndex*:

Array.prototype.indexOf (*searchElement* **[,** *fromIndex* **])**

the synthesizer generates the following calls with an array object or `null` as the `this` object as follows:

```
Array.prototype.indexOf.call(new Array(), 0);
Array.prototype.indexOf.call(new Array(), 0, 0);
Array.prototype.indexOf.call(null, 0);
Array.prototype.indexOf.call(null, 0, 0);
```

Moreover, `Array` is a built-in function and a built-in constructor with a variable number of arguments. Thus, we synthesize the following six programs for `Array`:

```
Array();        Array(0);       Array(0, 0);
new Array();    new Array(0);   new Array(0, 0);
```

B. Target Selector

From the synthesized programs, Target Selector selects a target program to mutate to increase the semantics coverage of the program pool. Consider the **Abstract Equality Comparison** algorithm in Figure 1(a) again where the first step has the condition “If Type(x) is the same as Type(y).” Assuming that the current pool has the following three programs:

```
1 + 2;   true == false;   0 == 1;
```

because later two programs that perform comparison have values of the same type, the pool covers only the true branch of the condition in the algorithm. To cover its false branch, Target Selector selects any program that covers the true branch like `true == false;` and Program Mutator mutates it to `42 == false;` for example. Then, since the mutated program covers the false branch, the pool is extended as follows:

```
1 + 2;   true == false;   0 == 1;   42 == false;
```

which now covers more steps in the algorithm. This process repeats until the semantics coverage converges.

C. Program Mutator

JEST increases the semantics coverage of the program pool by mutating programs using five mutation methods randomly.

1) Random Mutation: The first naïve method is to randomly select a statement, a declaration, or an expression in a given program and to replace it with a randomly selected one from a set of syntax trees generated by the non-recursive synthesizer. For example, it may mutate a program `var x = 1 + 2;` by replacing its random expression `1` with a random expression `true` producing `var x = true + 2;`.

2) Nearest Syntax Tree Mutation: The second method targets uncovered branches in abstract algorithms. When only one branch is covered by a program, it finds the nearest syntax tree in the program that reaches the branch in the algorithm, and replaces the nearest syntax tree with a random syntax tree derivable from the same syntax production. For example, consider the following JavaScript program:

```
var x = "" + (1 == 2);
```

While it covers the false branch of the first step of **Abstract Equality Comparison** in Figure 1(a), assume that no program in the program pool can cover its true branch. Then, the mutator targets this branch, finds its nearest syntax tree `1 == 2` in the program, and replaces it with a random syntax tree.

3) String Substitutions: We collect all string literals used in conditions of the algorithms in ES11 and use them for random expression substitutions. Because most string literals in the specification represent corner cases such as `-0`, `Infinity`, and `NaN`, they are necessary for mutation to increase the semantics coverage. For example, the semantics of the [[DefineOwnProperty]] internal method of array exotic objects depends on whether the value of its parameter `P` is `"length"` or not.

4) Object Substitutions: We also collect string literals and symbols used as arguments of object property access algorithms in ES11, randomly generate objects using them, and replace random expressions with the generated objects. Because some abstract algorithms in the specification access object properties using **HasProperty**, **GetMethod**, **Get**, and **OrdinaryGetOwnProperty**, objects with such properties are necessary for mutation to achieve high coverage. Thus, the mutator mutates a randomly selected expression in a program with a randomly generated object that has properties whose keys are from collected string literals and symbols.

5) Statement Insertion: To synthesize more complex programs, the mutator inserts random statements at the end of randomly selected blocks like top-level code and function bodies. We generate random statements using the non-recursive synthesizer with pre-defined special statements. The special statements are control diverters, which have high chances of changing execution paths, such as function calls, `return`, `break`, and `throw` statements. The mutator selects special statements with a higher probability than the statements randomly synthesized by the non-recursive synthesizer.

D. Assertion Injector

After generating JavaScript programs, Assertion Injector injects assertions to them using their final states as specified in ECMAScript. It first obtains the final state of a given program from the mechanized specification and injects seven kinds of assertions in the beginning of the program. To check the final state after executing all asynchronous jobs, we enclose assertions with `setTimeout` to wait 100 ms when a program uses asynchronous features such as `Promise` and `async`:

```
... /* a given program */
setTimeout(() => { ... /* assertions */ }, 100)
```

1) Exceptions: JavaScript supports both internal exceptions like `SyntaxError` and `TypeError` and custom exceptions with the keyword `throw`. Note that catching such exceptions using the `try-catch` statement may change the program semantics. For example, the following does not throw any exception:

```
var x; function x() {}
```

but the following:

```
try { var x; function x() {} } catch (e) {}
```

throws `SyntaxError` because declarations of a variable and a function with the same name are not allowed in `try-catch`.

To resolve this problem, we exploit a comment in the first line of a program. If the program throws an internal exception, we tag its name in the comment. Otherwise, we tag `//Throw` for a custom exception and `//Normal` for normal termination. Using the tag in the comment, JEST checks the execution result of a program in each engine.

2) Aborts: The mechanized semantics of ECMAScript can abort due to unspecified cases. For example, consider the following JavaScript program:

```
var x = 42; x++;
```

The postfix increment operator (`++`) increases the number value stored in the variable `x`. However, because of a typo in the **Evaluation** algorithm for such update expressions in ES11, the behavior of the program is not defined in ES11. To represent this situation in the conformance test, we tag `Abort` in the comment as follows:

```
// Abort
var x = 42; x++;
```

3) Variable Values: We inject assertions that compare the values of variables with expected values. To focus on variables introduced by tests, we do not check the values of pre-defined variables like built-in objects. For numbers, we distinguish `-0` from `+0` using division by zero because `1/-0` and `1/+0` produce negative and positive infinity values, respectively. The following example checks whether the value of `x` is `3`:

```
var x = 1 + 2;
$assert.sameValue(x, 3);
```

4) Object Values: To check the equality of object values, we keep a representative path for each object. If the injector meets an object for the first time, it keeps the current path of the object as its representative path and injects assertions for the properties of the object. Otherwise, the injector adds assertions to compare the values of the objects with the current path and the representative path. In the following example:

```
var x = {}, y = {}, z = { p: x, q: y };
$assert.sameValue(z.p, x);
$assert.sameValue(z.q, y);
```

because the injector meets two different new objects stored in `x` and `y`, it keeps the paths `x` and `y`. Then, the object stored in `z` is also a new object but its properties `z.p` and `z.q` store already visited objects values. Thus, the injector inserts two assertions that check whether `z.p` and `x` have the same object value and `z.q` and `y` as well. To handle built-in objects, we store all the paths of built-in objects in advance.

5) Object Properties: Checking object properties involves checking four attributes for each property. We implement a helper `$verifyProperty` to check the attributes of each property for each object. For example, the following code checks the attributes of the property of `x.p`:

```
var x = { p: 42 };
$verifyProperty(x, "p", {
    value: 42.0,      writable: true,
    enumerable: true, configurable: true
});
```

6) Property Keys: Since ECMAScript 2015 (ES6), the specification defines orders between property keys in objects. We check the order of property keys by `Reflect.ownKeys`, which takes an object and returns an array of the object's property keys. We implement a helper `$assert.compareArray` that takes two arrays and compares their lengths and contents. For example, the following program checks the property keys and their order of the object in `x`:

```
var x = {[Symbol.match]: 0, p: 0, 3: 0, q: 0, 1: 0}
$assert.compareArray(
    Reflect.ownKeys(x),
    ["1", "3", "p", "q", Symbol.match]
);
```

7) Internal Methods and Slots: While internal methods and slots of JavaScript objects are generally inaccessible by users, the names in the following are accessible by indirect getters:

Name	Indirect Getter
[[Prototype]]	`Object.getPrototypeOf(x)`
[[Extensible]]	`Object.isExtensible(x)`
[[Call]]	`typeof f === "function"`
[[Construct]]	`Reflect.construct(function(){},[],x)`

The internal slot [[Prototype]] represents the prototype object of an object, which is available by a built-in function `Object.getPrototypeOf`. The internal slot [[Extensible]] is also available by a built-in function `Object.isExtensible`. The internal methods [[Call]] and [[Construct]] represent whether a given object is a function and a constructor, respectively. Because the methods are not JavaScript values, we simply check their existence using helpers `$assert.callable` and `$assert.constructable`. For [[Call]], we use the `typeof` operator because it returns `"function"` if and only if a given value is an object with the [[Call]] method. For [[Construct]] method, we use the `Reflect.construct` built-in function that checks the existence of the [[Construct]] methods and invokes it. To avoid invoking [[Construct]] unintentionally, we call `Reflect.construct` with a dummy function `function(){}` as its first argument and a given object as its third argument. For example, the following code shows how the injector injects assertions for internal methods and slots:

```
function f() {}
$assert.sameValue(Object.getPrototypeOf(f),
                  Function.prototype);
$assert.sameValue(Object.isExtensible(x), true);
$assert.callable(f);
$assert.constructable(f);
```

E. Bug Localizer

The bug detection and localization phase uses the execution results of given conformance tests on multiple JavaScript engines. If a small number of engines fail in running a specific conformance test, the engines may have bugs causing the test failure. If most engines fail for a test, the test may be incorrect, which implies a bug in the specification.

When we have a set of failed test cases that may contain bugs of an engine or a specification, we classify the test cases using their failure messages and give ranks between possible buggy program elements to localize the bug. We use Spectrum Based Fault Localization (SBFL) [18], which is a ranking technique based on likelihood of being faulty for each program element. We use the following formula called $ER1_b$, which is one of the best SBFL formulae theoretically analyzed by Xie et al. [20]:

$$n_{ef} - \frac{n_{ep}}{n_{ep} + n_{np} + 1}$$

where n_{ef}, n_{ep} , n_{nf}, and n_{np} represent the number of test cases; subscripts $_e$ and $_n$ respectively denote whether a test case touches a relevant program element or not, and subscripts $_f$ and $_p$ respectively denote whether the test case is failed or passed.

We use abstract algorithms of ECMAScript as program elements used for SBFL. To improve the localization accuracy, we use method-level aggregation [21]. It first calculates SBFL scores for algorithm steps and aggregates them up to algorithm-level using the highest score among those from steps of each algorithm.

IV. Evaluation

To evaluate JEST that performs N+1-version differential testing of JavaScript engines and its specification, we applied the tool to four JavaScript engines that fully support modern JavaScript features and the latest specification, ECMAScript 2020 (ES11, 2020). Our experiments use the following four JavaScript engines, all of which support ES11:

- **V8(v8.3)[1]:** An open-source high-performance engine for JavaScript and WebAssembly developed by Google [14]
- **GraalJS(v20.1.0)[2]:** A JavaScript implementation built on GraalVM [15], which is a Java Virtual Machine (JVM) based on HotSpot/OpenJDK developed by Oracle
- **QuickJS(2020-04-12)[3]:** A small and embedded JavaScript engine developed by Fabrice Bellard and Charlie Gordon [16]
- **Moddable XS(v10.3.0)[4]:** A JavaScript engine at the center of the Moddable SDK [17], which is a combination of development tools and runtime software to create applications for micro-controllers

To extract a mechanized specification from ECMAScript, we utilize the tool JISET, which is a JavaScript IR-based semantics extraction toolchain, to automatically generate a JavaScript interpreter from ECMAScript. To focus on the core semantics of JavaScript, we consider only the semantics of strict mode JavaScript code that pass syntax checking including the EarlyError rules. To filter out JavaScript code that are not strict or fail syntax checking, we utilize the syntax checker of the most reliable JavaScript engine, V8. We performed our experiments on a machine equipped with 4.0GHz Intel(R) Core(TM) i7-6700k and 32GB of RAM (Samsung DDR4 2133MHz 8GB*4). We evaluated JEST with the following four research questions:

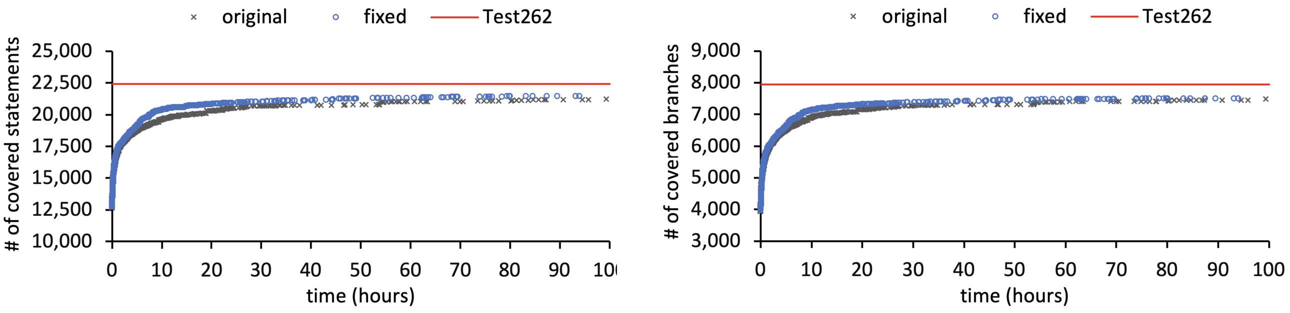
- **RQ1 (Coverage of Generated Tests)** Is the semantics coverage of the tests generated by JEST comparable to that of Test262, the official conformance test suite for ECMAScript, which is manually written?
- **RQ2 (Accuracy of Bug Localization)** Does JEST localize bug locations accurately?
- **RQ3 (Bug Detection in JavaScript Engines)** How many bugs of four JavaScript engines does JEST detect?
- **RQ4 (Bug Detection in ECMAScript)** How many bugs of ES11 does JEST detect?

[1] https://v8.dev/

[2] https://github.com/graalvm/graaljs#current-status

[3] https://bellard.org/quickjs/

[4] https://blog.moddable.com/blog/xs10/



(a) Statement coverage

(b) Branch coverage

Fig. 4: The semantics coverage changes during the test generation phase

TABLE I: Number of generated programs and covered branches of mutation methods

Mutation Method	Program	Branch (Avg.)
Nearest Syntax Tree Mutation	459	1,230(2.68)
Random Mutation	337	1,153(3.42)
Statement Insertion	209	650(3.11)
Object Substitution	169	491(2.91)
String Substitution	3	3(1.00)
Total	1,177	3,527(3.00)

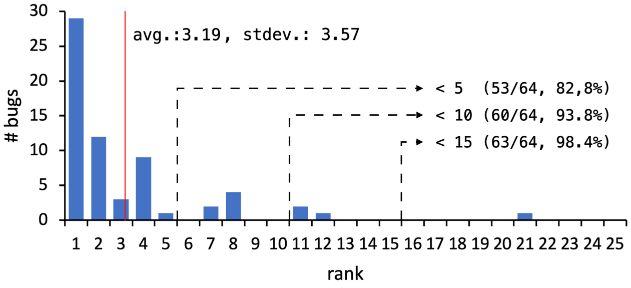


Fig. 5: Ranks of algorithms that caused the bugs detected by JEST

A. *Coverage of Generated Tests*

JEST generates the seed programs via Seed Synthesizer, which synthesizes 1,125 JavaScript programs in about 10 seconds and covers 97.78% (397/406) of reachable alternatives in the syntax productions of ES11. Among them, we filtered out 602 programs that do not increase the semantics coverage and started the mutation iteration with 519 programs. Figure 4 shows the change of semantics coverage of the program pool during the iterative process in 100 hours. The left and right graphs present the statement and branch coverages, respectively, and the top red line denotes the coverage of Test262. We generated conformance tests two times before and after fixing bugs detected by JEST because the specification bugs affected the semantics coverage. In each graph, dark gray X marks and blue O marks denote the semantics coverage of generated tests before and after fixing bugs. The semantics that we target in ES11 consists of 1,550 algorithms with 24,495 statements and 9,596 branches. For the statement coverage, Test262 covers 22,440 (91.61%) statements. The initial program pool covers 12,768 (52.12%) statements and the final program pool covers 21,230 (86.67%) and 21,482 (87.70%) statements before and after fixing bugs, respectively. For the branch coverage, Test262 covers 7,956 (82.91%) branches. The initial program pool covers 3,987 (41.55%) branches and the final program pool covers 7,480 (77.95%) and 7,514 (78.30%) branches before and after fixing bugs, respectively.

Table I shows the number of synthesized programs and covered branches for each mutation method during the test generation phase. In total, JEST successfully synthesize 1,177 new programs that cover 3,527 more branches than the initial program pool. Among five mutation methods, the nearest syntax tree mutation is the most contributed method (459 programs and 1,230 covered branches) and the least one is the string substitution (3 programs and 3 covered branches). On average, 3.00 branches are covered by a new program.

Finally, JEST generates 1,700 JavaScript programs and their average number of lines is 2.01. After injecting assertions, their average number of lines becomes 8.45. Compared to Test262, the number of generated tests are much smaller and their number of lines are also shorter than those of tests in Test262. Test262 provides 16,251 tests for the same range of semantics and their average number of lines is 49.67.

B. *Accuracy of Bug Localization*

To detect more bugs using more diverse programs, we repeated the conformance test generation phase for ten times. We executed the generated conformance tests on four JavaScript engines to find bugs in the engines and the specification. After inferring locations of the bugs in the engines or the specification based on the majority of the execution results, we manually checked whether the bugs are indeed in the engines or the specification. The following table shows that our method works well:

# Failed Engines	1	2	3	4	Total	Average
Engine Bugs	38	6	0	0	44	1.14
Specification Bugs	0	0	10	17	27	3.63

For engine bugs, the average number of engine failures is 1.14 while the average number of failed engines for specification bugs is 3.63. As we expected, when most engines fail for a test, the specification may have a bug.

Based on the results of conformance tests on four JavaScript engines, we localized the specification or engine bugs on the

TABLE III: Specification bugs in ECMAScript 2020 (ES11) detected by JEST

Name	Feature	#	Description	Assertion	Known	Created	Resolved	Existed
ES11-1	Function	12	Wrong order between property keys for functions	Key	O	2019-02-07	2020-04-11	429 days
ES11-2	Function	8	Missing property `name` for anonymous functions	Key	O	2015-06-01	2020-04-11	1,776 days
ES11-3	Loop	1	Returning iterator objects instead of iterator records in **ForIn/OfHeadEvaluation** for **`for-in`** loops	Exc	O	2017-10-17	2020-04-30	926 days
ES11-4	Expression	4	Using the wrong variable `oldvalue` instead of `oldValue` in **Evaluation** of *UpdateExpression*	Abort	O	2019-09-27	2020-04-23	209 days
ES11-5	Expression	1	Unhandling abrupt completion in **Abstract Equality Comparison**	Exc	O	2015-06-01	2020-04-28	1,793 days
ES11-6	Object	1	Unhandling abrupt completion in **Evaluation** of *PropertyDefinition* for object literals	Exc	X	2019-02-07	TBD	TBD

TABLE II: The number of engine bugs detected by JEST

Engines	Exc	Abort	Var	Obj	Desc	Key	In	Total
V8	0	0	0	0	0	2	0	2
GraalJS	6	0	0	0	2	8	0	16
QuickJS	3	0	1	0	0	2	0	6
Moddable XS	12	0	0	0	3	5	0	20
Total	21	0	1	0	5	17	0	44

semantics of ES11. Among 71 bugs, we excluded 7 syntax bugs and localized only 64 semantics bugs. Figure 5 shows the ranks of algorithms that caused the semantics bugs. The average rank is 3.19, and 82.8% of the algorithms causing the bugs are ranked less than 5, 93.8% less than 10, and 98.4% less than 15. Note that the location of one bug is ranked 21 because of the limitation of SBFL; its localization accuracy becomes low for a small number of failed test cases.

C. Bug Detection in JavaScript Engines

From four JavaScript engines, JEST detected 44 bugs: 2 from V8, 16 from GraalJS, 6 from QuickJS, and 20 from Moddable XS. Table II presents how many bugs for each assertion are detected for each engine. We injected seven kinds of assertions: exceptions (Exc), aborts (Abort), variable values (Var), object values (Obj), object properties (Desc), property keys (Key), and internal methods and slots (In). The effectiveness of bug finding is different for different assertions. The Exc and Key assertions detected engine bugs the most; out of 44 bugs, the former detected 21 bugs and the latter detected 17 bugs. Desc and Var detected 5 and 1 bugs, respectively, but the other assertions did not detect any engine bugs.

The most reliable JavaScript engine is V8 because JEST found only two bugs and the bugs are due to specification bugs in ES11. Because V8 strictly follows the semantics of functions described in ES11, it also implemented wrong semantics that led to ES11-1 and ES11-2 listed in Table III. The V8 team confirmed the bugs and fixed them.

We detected 16 engine bugs in GraalJS and one of them caused an engine crash. When we apply the prefix increment operator for `undefined` as `++undefined`, GraalJS throws `java.lang.IllegalStateException`. Because it crashes the engine, developers even cannot catch the exception as follows:

```
try { ++undefined; } catch(e) { }
```

The GraalJS team has been fixing the bugs we reported and asked whether we plan to publish the conformance test suite, because the tests generated by JEST detected many semantics bugs that were not detected by other conformance tests: *"Right now, we are running Test262 and the V8 and Nashorn unit test suites in our CI for every change, it might make sense to add your suite as well."*

In QuickJS, JEST detected 6 engine bugs, most of which are due to corner cases of the function semantics. For example, the following code should throw a `ReferenceError` exception:

```
function f (... { x = x }) { return x; } f()
```

because the variable `x` is not yet initialized when it tries to read the right-hand side of `x = x`. However, since QuickJS assumes that the initial value of `x` is `undefined`, the function call `f()` returns `undefined`. The QuickJS team confirmed our bug reports and it has been fixing the bugs.

JEST found the most bugs in Moddable XS; it detected 20 bugs for various language features such as optional chains, `Number.prototype.toString`, iterators of `Map` and `Set`, and complex assignment patterns. Among them, optional chains are newly introduced in ES11, which shows that our approach is applicable to finding bugs in new language features. We reported all the bugs found, and the Moddable XS team has been fixing them. They showed interests in using our test suite: *"As you know, it is difficult to verify changes because the language specification is so big. Test262, as great a resource as it is, is not definitive."*

D. Bug Detection in ECMAScript

From the latest ECMAScript ES11, JEST detected 27 specification bugs. Table III summarizes the bugs categorized by their root causes. Among them, five categories (ES11-1 to ES11-5) were already reported and fixed in the current draft of the next ECMAScript but ES11-6 was never reported before. We reported it to TC39; they confirmed it and they will fix it in the next version, ECMAScript 2021 (ES12).

ES11-1 contains 12 bugs; it is due to a wrong order between property keys of all kinds of function values such as **`async`** and generator functions, arrow functions, and classes. For example, if we define a class declaration with a name `A` (**`class`** `A {}`), three properties are defined in the function stored in the variable `A`: `length` with a number value `0`, `prototype` with an object, and `name` with a string `"A"`. The problem is the different order of their keys because of the wrong order of their creation. From ECMAScript 2015 (ES6), the order between property keys is no more implementation-dependent but it is related to the creation order of properties. While the order of property keys in the class `A` should be `[length, prototype, name]` according to the semantics of ES11, the order is `[length, name, prototype]` in three engines except V8. We found that it was already reported as a specification bug; we reported it to V8 and they fixed it. This bug was created on February 7, 2019 and TC39 fixed it on April 11, 2020; the bug lasted for 429 days.

ES11-2 contains 8 bugs that are due to the missing property `name` of anonymous functions. Until ES5.1, anonymous functions, such as an identity arrow function `x => x`, had their own property `name` with an empty string `""`. While ES6 removed the `name` property from anonymous functions, three engines except V8 still create the `name` property in anonymous functions. We also found that it was reported as a specification bug and reported it to V8, and it will be fixed in V8.

The bug in ES11-3 comes from the misunderstanding of the term "iterator object" and "iterator record". The algorithm **ForIn/OfHeadEvaluation** should return an iterator record, which is an implicit record containing only internal slots. However, In ES11, it returns an iterator object, which is a JavaScript object with some properties related to iteration. It causes a `TypeError` exception when executing the code `for(var x in {});` according to ES11 but all engines execute the code normally without any exceptions. This bug was resolved by TC39 on April 30, 2020.

ES11-4 contains four bugs caused by a typo for the variable in the semantics of four different update expressions: `x++`, `x--`, `++x`, and `--x`. In each **Evaluation** of four kinds of *UpdateExpression*, there exists a typo `oldvalue` in step 3 instead of `oldValue` declared in step 2. JEST could not execute the code `x++` using the semantics of ES11 because of the typo. For this case, we directly pass the code to Bug Localizer to test whether the code is executable in real-world engines and to localize the bug. Of course, four JavaScript engines executed the update expressions without any issues and this bug was resolved by TC39 on April 23, 2020.

Two bugs in ES11-5 and ES11-6 are caused by unhandling of abrupt completions in abstract equality comparison and property definitions of object literals, respectively. The bug in ES11-5 was confirmed by TC39 and was fixed on April 28, 2020. The bug in ES11-6 was a genuine one, and we reported it and received a confirmation from TC39 on August 18, 2020. The bug will be fixed in the next version, ES12.

V. Related Work

Our technique is related to three research fields: differential testing, fuzzing, and fault localization.

Differential Testing: Differential testing [10] utilizes multiple implementations as cross-referencing oracles to find semantics bugs. Researchers applied this technique to various applications domain such as Java Virtual Machine (JVM) implementations [22], SSL/TLS certification validation logic [23]–[25], web applications [26], and binary lifters [27]. Moreover, NEZHA [23] introduces a guided differential testing tool with the concept of δ-diversity to efficiently find semantics bugs. However, they have a fundamental limitation that they cannot test specifications; they use only cross-referencing oracles and target potential bugs in implementations. Our N+1-version differential testing extends the idea of differential testing with not only N different implementations but also a mechanized specification to test both of them. In addition, our approach automatically generates conformance tests directly from the specification.

Fuzzing: Fuzzing is a software testing technique for detecting security vulnerabilities by generating [28]–[30] or mutating [31]–[33] test inputs. For JavaScript [34] engines, Patrice et al. [35] presented white-box fuzzing using the JavaScript grammar, Han et al. [36] presented CodeAlchemist that generates JavaScript code snippets based on semantics-aware assembly, Wang et al. [37] presented Superion using Grammar-aware greybox fuzzing, Park et al. [38] presented DIE using aspect-preserving mutation, and Lee et al. [39] presented Montage using neural network language models (NNLMs). While they focus on finding security vulnerabilities rather than semantics bugs, our N+1-version differential testing focuses on finding semantics bugs by comparing multiple implementations with the mechanized specification, which was automatically extracted from ECMAScript by JISET. Note that JEST can also localize not only specification bugs in ECMAScript but also bugs in JavaScript engines indirectly using the bug locations in ECMAScript.

Fault Localization: To localize detected bugs in ECMAScript, we used Spectrum Based Fault Localization (SBFL) [18], which is a ranking technique based on likelihood of being faulty for each program element. Tarantula [40], [41] was the first tool that supports SBFL with a simple formula and researchers have developed many formulae [42]–[45] to increase the accuracy of bug localization. Sohn and Yoo [21] introduced a novel approach for fault localization using code and change metrics via learning of SBFL formulae. While we utilize a specific formula $ER1_b$ introduced by Xie et al. [20], we believe that it is possible to improve the accuracy of bug localization by using more advanced SBFL techniques.

VI. Conclusion

The development of modern programming languages follows the continuous integration (CI) and continuous deployment (CD) approach to instantly support fast changing user demands. Such continuous development makes it difficult to find semantics bugs in both the language specification and its various implementations. To alleviate this problem, we present N+1-version differential testing, which is the first technique to test both implementations and its specification in tandem. We actualized our approach for the JavaScript programming language via JEST, using four modern JavaScript engines and the latest version of ECMAScript (ES11, 2020). It automatically generated 1,700 JavaScript programs with 97.78% of syntax coverage and 87.70% of semantics coverage on ES11. JEST injected assertions to the generated JavaScript programs to convert them as conformance tests. We executed generated conformance tests on four engines that support ES11: V8, GraalJS, QuickJS, and Moddable XS. Using the execution results, we found 44 engine bugs (16 for GraalJS, 6 for QuickJS, 20 for Moddable XS, and 2 for V8) and 27 specification bugs. All the bugs were confirmed by TC39, the committee of ECMAScript, and the corresponding engine teams, and they will be fixed in the specification and the engines. We believe that JEST takes the first step towards co-evolution of software specifications, tests, and their implementations for CI/CD.

Acknowledgements

This work was supported by National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

References

[1] (2012) Mark Zuckerberg's Letter to Investors: 'The Hacker Way'. [Online]. Available: https://www.wired.com/2012/02/zuck-letter/

[2] (2020) What is CI/CD? Continuous integration and continuous delivery explained. [Online]. Available: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

[3] (2020) Node.js - A JavaScript runtime built on Chrome's V8 JavaScript engine. [Online]. Available: https://nodejs.org/

[4] (2020) Moddable - Tools to create open IoT products using standard JavaScript on low cast microcontrollers. [Online]. Available: https://www.moddable.com/

[5] (2020) Espruino - JavaScript for Microcontrollers. [Online]. Available: https://www.espruino.com/

[6] (2020) Tessel 2 - a robust IoT and robotics development platform. [Online]. Available: https://tessel.io/

[7] (2015) Standard ECMA-262 6th Edition ECMAScript 2015 Language Specification. [Online]. Available: https://ecma-international.org/ecma-262/6.0/

[8] Solidity. (2019) Official solidity documentation. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.7/

[9] S. Hwang and S. Ryu, "Gap between theory and practice: An empirical study of security patches in solidity," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 2020.

[10] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[11] J. Park, J. Park, S. An, and S. Ryu, "JISET: Javascript ir-based semantics extraction toolchain," in *Proceedings of ACM International Conference on Automated Software Engineering*, 2020.

[12] H. Nguyen, "Automatic extraction of x86 formal semantics from its natural language description," *Information Science*, 2018.

[13] A. V. Vu and M. Ogawa, "Formal semantics extraction from natural language specifications for arm," in *International Symposium on Formal Methods*. Springer, 2019, pp. 465–483.

[14] (2020) Google's open source high-performance JavaScript and WebAssembly engine, written in C++. [Online]. Available: https://v8.dev/

[15] (2020) A high performance implementation of the JavaScript programming language. Built on the GraalVM by Oracle Labs. [Online]. Available: https://github.com/graalvm/graaljs

[16] (2020) A small and embeddable Javascript engine by Fabrice Bellard and Charlie Gordon. [Online]. Available: https://bellard.org/quickjs/

[17] (2020) The JavaScript engine at the center of the Moddable SDK. [Online]. Available: https://github.com/Moddable-OpenSource/moddable

[18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[19] B. McKenzie, "Generating strings at random from a context free grammar," Department of Computer Science, University of Canterbury, Tech. Rep. TR-COSC 10/97, 1997.

[20] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–40, 2013.

[21] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[22] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[23] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *Proceedings of IEEE Symposium on Security and Privacy*, 2017, pp. 615–632.

[24] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 793–804.

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 38–49.

[26] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 263–274.

[27] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of ACM International Conference on Automated Software Engineering*, 2017, pp. 353–364.

[28] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[29] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 445–458.

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[31] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[32] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 861–875.

[33] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 511–522.

[34] A. Wirfs-Brock and B. Eich, "Javascript: the first 20 years," in *Proceedings of the ACM on Programming Languages*, vol. 4, 2020, pp. 1–189.

[35] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[36] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[37] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[38] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1629–1642.

[39] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," 2020.

[40] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.

[41] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[42] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005, pp. 99–104.

[43] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 662–664.

[44] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[45] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 449–456.