Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations

ANONYMOUS AUTHOR(S)

The *conformance testing* of programming language implementations is crucial to support correct and consistent execution environments. Because manually maintaining conformance tests for real-world programming languages is cumbersome and labor-intensive, researchers have presented various ways to make conformance tests effective and efficient. One such approach is to use graph coverage, one of the most widely-used coverage criteria, to generate tests that reach different parts of a *mechanized language specification*. Since mechanized specifications use functions or inductive definitions to describe the semantics of language features, traditional graph coverage criteria for software work as they are. However, they may not produce high-quality conformance tests because language implementations often have specialized execution paths for different features, even when their semantics descriptions use the same functions. Traditional graph coverage may not distinguish test requirements of such language features, which degrades the quality of conformance testing. Similarly, it may not distinguish test requirements of different parts of the same language feature when their semantics descriptions use the same functions.

We present *feature-sensitive (FS) coverage* as a novel coverage criterion to generate high-quality conformance tests for language implementations. It is a general extension of graph coverage, refining conventional test requirements using the innermost enclosing language features. We also introduce *feature-call-path-sensitive (FCPS) coverage*, a variant of FS coverage, and extend both coverage criteria using the k-limiting approach. To evaluate the effectiveness of the new coverage criteria for language implementations, we apply them to a mechanized specification of JavaScript. We extend JEST, the state-of-the-art JavaScript conformance test synthesizer using coverage-guided mutational fuzzing, with various FS and FCPS coverage criteria. For the latest JavaScript language specification (ES13, 2022), our tool automatically synthesizes 237,981 conformance tests in 50 hours with five coverage criteria. We evaluated the conformance of eight mainstream JavaScript implementations (four engines and four transpilers) with the synthesized conformance tests and discovered bugs in all of them. The tool detected 143 distinct conformance bugs (42 in engines and 101 in transpilers), 85 of which were confirmed by the developers and 83 of which were newly discovered bugs.

Additional Key Words and Phrases: JavaScript, conformance test synthesis, feature-sensitive coverage, coverage-guided fuzzing,

ACM Reference Format:
Anonymous Author(s). 2023. Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations. In . ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The *conformance testing* of programming language implementations is essential to provide correct and consistent implementations of the language semantics. Many programming languages have multiple implementations rather than a single reference implementation. For example, Java uses a Java Virtual Machine (JVM) to compile Java programs into JVM bytecode and execute them. Developers are free to choose one of the existing JVM implementations, such as OpenJ9, GraalVM,

PLDI'23, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

HotSpot, Zulu, and Corretto. Python has the reference interpreter, CPython, in addition to diverse interpreters, including PyPy, Jython, and IronPython. Therefore, ensuring correct and consistent execution environments in different implementations of the same language becomes crucial. However, since manually maintaining conformance test suites for real-world programming languages is cumbersome and labor-intensive, only a small number of programming languages, such as JavaScript [18] and XML [58], provide their official conformance test suites. Thus, researchers have presented ways to test the conformance of multiple implementations using differential testing [34] for compilers [14, 37, 52, 59], interpreters [7, 60], virtual machines [12], and debuggers [31]. To make differential testing for language implementations effective, researchers have proposed various techniques to synthesize diverse programs, such as generation-based fuzzing [7, 31, 37, 59], mutation-based fuzzing [12, 52], and deep learning [14, 60].

Graph coverage [4] is one of the most widely-used coverage criteria in evaluating the quality of conformance tests. Higher coverage of a conformance test suite denotes that it covers more test requirements (TRs) of a given coverage criterion for language implementations. Graph coverage helps generate tests that reach uncovered parts of software; coverage-guided fuzzing (CGF) [35] improves mutation-based fuzzing by selecting mutation target programs using coverage information. It also helps avoid an excessive number of conformance tests; researchers have presented various test minimization techniques [61] to reduce the number of tests, and Wong et al. [56] present coverage-guided test minimization.

One approach to making high-quality conformance tests is to use graph coverage to generate tests for *mechanized language specifications*. While we can use code coverage rather than graph coverage to generate tests for "actual language implementations," it will lead to different coverage information for different implementations. On the contrary, graph coverage for mechanized language specifications will lead to uniform coverage information for multiple implementations. Various programming languages, such as OCaml [38], C [8], C++[47], Java [10], JavaScript [43], and POSIX shell [21], have mechanized specifications that formally describe their semantics using diverse metalanguages and frameworks, such as Ott [50], Skel [9], and the $\mathbb{K}$ framework [48]. ~~Since mechanized~~ Mechanized specifications use functions or inductive definitions to describe the semantics of language features~~, we can easily~~ . Thus, it is possible to convert them as directed graphs and adapt them to apply traditional graph coverage criteria for software~~work as they are~~. For example, K-Java [10] is a mechanized specification for Java defined with the $\mathbb{K}$ framework, which describes language semantics as a set of reduction rules. Consider a directed graph whose nodes are reduction rules and edges are their dependencies in K-Java. Then, we can measure the coverage of a test suite in the directed graph denoting K-Java based on whether each test covers the test requirements of a graph coverage criterion.

Challenges. However, graph coverage may not produce high-quality conformance tests for mechanized language specifications. Mechanized specifications are usually written in a modular way with helper functions. Such a modular definition has the advantages of preventing duplicated or similar definitions of language semantics, reducing the size of a mechanized specification, and enhancing its readability. ~~At the same time~~Despite its usefulness, reusing the same helper function for different parts may degrade the quality of conformance testing in two ways.

First, traditional graph coverage may not distinguish test requirements of different language features when their semantics descriptions use the same functions, degrading conformance testing quality. For example, consider a mechanized specification for JavaScript that represents the abstract algorithms described in the official language specification, ECMA-262 [16]. Here, most of the semantics for the addition and subtraction operators are defined using the same **EvaluateStringOrNumericBinaryExpression** algorithm as a helper function. If conformance tests for the addition

operator already cover the test requirements in the algorithm, most conformance tests for the subtraction operator are removed after the coverage-guided test minimization process. However, real-world JavaScript engines are highly optimized and often have specialized execution paths for different language features, even when their semantics descriptions use the same functions. Therefore, we need to test possible edge cases for the subtraction operator as well, even though similar edge cases for the addition operator are already tested.

Furthermore, it may not distinguish test requirements of different parts of the same language feature when their semantics descriptions use the same functions, degrading the quality of conformance testing. For example, consider the mechanized specification for JavaScript again. In JavaScript, the `String.prototype.normalize` built-in API normalizes a given string into a normalization form named by a given argument. The definition of the semantics for this built-in API feature uses the **ToString** algorithm as a helper function twice to represent conversions to strings for 1) **`this`** value and 2) the first argument of the API call. Assume that a conformance test suite already covers the test requirements in the **ToString** algorithm thanks to various values for **`this`** value. Then, there is no chance to generate new conformance tests that check edge cases of the conversion from the first argument to string when performing coverage-guided fuzzing.

This Work. To alleviate this problem, we introduce *feature-sensitive (FS) coverage*, a novel coverage criterion to generate high-quality conformance tests for programming language implementations. It is a general extension of graph coverage, refining test requirements using the innermost enclosing language features. FS coverage resolves the problem of sharing the same helper functions for the semantics of different language features. We also present a *feature-call-path-sensitive (FCPS) coverage*, a variant of FS coverage with feature-call-paths from language features to test requirements. FCPS coverage resolves the problem of sharing the same helper functions for the semantics of different parts of the same language feature. In addition, we extend both coverage criteria using the k-limiting approach as k-FS coverage and k-FCPS coverage. To evaluate the effectiveness of the new coverage criteria, we apply them to a real-world programming language, JavaScript. We select JavaScript as the evaluation target language because 1) it has the most up-to-date mechanized specification and 2) it has the official conformance test suite, Test262 [18]. We extend JEST [42], the state-of-the-art JavaScript conformance test synthesizer using coverage-guided mutational fuzzing, with various FS and FCPS coverage criteria. For the latest language specification (ES13, 2022), our tool automatically synthesizes 237,981 conformance tests in 50 hours with five coverage criteria. We evaluated the conformance of eight mainstream JavaScript implementations (four engines and four transpilers) with the synthesized conformance tests and discovered bugs in all of them. The tool detected 143 distinct conformance bugs (42 in engines and 101 in transpilers), 85 of which were confirmed by the developers and 83 of which were newly discovered bugs.

Contributions. We summarize our contributions as follows:

- We introduce novel *feature-sensitive (FS) coverage* to discriminate test requirements with the innermost enclosing language features to enhance the quality of conformance testing for programming language implementations. It can resolve the problem of sharing the same helper functions for the semantics of different language features.
- We also present *feature-call-path-sensitive (FCPS) coverage* as its variant with feature-call-paths from language features to test requirements to distinguish different parts in the semantics of the same language feature.
- We experimentally show that the new coverage criteria outperform the traditional coverage criteria in the context of conformance bug detection in eight mainstream JavaScript

AdditiveExpression[Yield, Await] :
MultiplicativeExpression[?Yield, ?Await]
AdditiveExpression[?Yield, ?Await] **+** *MultiplicativeExpression*[?Yield, ?Await]
AdditiveExpression[?Yield, ?Await] **-** *MultiplicativeExpression*[?Yield, ?Await]

Fig. 1. Syntax of *AdditiveExpression* in ES13

Syntax-directed operations (SDOs)

Evaluation of *AdditiveExpression* : *AdditiveExpression* **+** *MultiplicativeExpression*
1. Return ?[2] EvaluateStringOrNumericBinaryExpression[1](*AdditiveExpression*, **+**, *MultiplicativeExpression*).[3]

Evaluation of *AdditiveExpression* : *AdditiveExpression* **-** *MultiplicativeExpression*
1. Return ?[5] EvaluateStringOrNumericBinaryExpression[4](*AdditiveExpression*, **-**, *MultiplicativeExpression*).[6]

Fig. 2. Semantics of *AdditiveExpression* defined with two syntax-directed operations (SDOs) in ES13

implementations (four engines and four transpilers) with the latest ECMA-262 (ES13, 2022). The tool uncovered 83 brand-new bugs.

2 BACKGROUND AND MOTIVATION

In this section, we explain why traditional graph coverage may not produce high-quality conformance tests using JavaScript as an example language. We select JavaScript because its mechanized specifications are actively maintained, while most mechanized specifications of other languages are outdated. Since all the existing JavaScript mechanized specifications [19, 27, 39, 43] closely capture the abstract algorithms in ECMA-262 [16], we show how JavaScript mechanized specifications describe the JavaScript syntax and semantics using ECMA-262. Then, we explain the control-flow graph (CFG) of abstract algorithms in ECMA-262 and how to use the CFG in coverage-guided fuzzing. Finally, we demonstrate why a simple node coverage criterion cannot fully discriminate different semantics in different language features or even in the same language feature.

2.1 JavaScript Language Specification (ECMA-262)

Now, we explain how the latest version of ECAM-262 (ES13, 2022) describes the syntax and semantics of JavaScript language features with simple examples.

2.1.1 Syntax. ECMA-262 defines the JavaScript syntax with a variant of the extended Backus–Naur form (EBNF). It consists of syntactic productions defined with multiple alternatives; each alternative is a sequence of symbols. Unlike the original EBNF, its nonterminals are parametric with boolean arguments: ? denotes passing the argument as is, and + and ~ denote passing `true` and `false`, respectively. In addition, it supports various extensions, including context-sensitive symbols and conditional alternatives. For example, consider the following simple additive expression:

```
1 + 2
```
(1)

It computes the addition of two Number values, `1` and `2`. Figure 1 describes its syntax with the production of *AdditiveExpression*[1]. It requires two boolean parameters, Yield and Await, and consists of three alternatives. The second (or third) alternative consists of three symbols: a nonterminal *AdditiveExpression*, a terminal **+** (or **-**), and a nonterminal *MultiplicativeExpression*.

2.1.2 Semantics. ECMA-262 defines the JavaScript semantics using abstract algorithms, and there are three kinds of abstract algorithms:

[1] https://262.ecma-international.org/13.0/#prod-AdditiveExpression

Normal algorithms

EvaluateStringOrNumericBinaryExpression (*leftOperand*, *opText*, *rightOperand*)
[7]...
5. Return ?[9] ApplyStringOrNumericBinaryOperator[8](*lval*, *opText*, *rval*).[10]

ApplyStringOrNumericBinaryOperator (*lval*, *opText*, *rval*)
[11]...
3. Let *lnum* be ?[13] ToNumeric[12](*lval*).
4. Let *rnum* be ?[15] ToNumeric[14](*rval*).
5. If Type(*lnum*) is different from Type(*rnum*)[16], throw a **TypeError** exception.[17]
...[18]

ToNumeric (*value*)
[19]...
2. If Type(*primValue*) is BigInt[20], return *primValue*.[21]
...[22]

Fig. 3. Three normal algorithms transitively used in the semantics of *AdditiveExpression* in ES13

- Syntax-directed operations (SDOs) (e.g., **Evaluation** of *AdditiveExpression* **:** ··· in Figure 2)
- Normal algorithms (e.g., **ToNumeric** in Figure 3)
- Built-in methods (e.g., **Number** in Figure 4)

A *syntax-directed operation (SDO)* defines the semantics of each alternative in syntactic productions. It consists of its 1) target alternative, 2) name, 3) optional parameters, and 4) body. Each algorithm body is a pseudo-code consisting of well-organized steps written in a natural language, English. For example, two abstract algorithms in Figure 2 are SDOs whose target alternatives are the second and third alternatives of *AdditiveExpression* production for addition (+) and subtraction (-) operators, respectively. Their names are **Evaluation** with no optional parameters, and the bodies consist of a single step that invokes another normal algorithm **EvaluateStringOrNumericBinaryExpression**[2]. Note that the metavariables *AdditiveExpression* and *MultiplicativeExpression* in these SDOs store *abstract syntax trees (ASTs)* of the left-hand and right-hand sides of given additive expressions, respectively. For instance, if the first SDO in Figure 2 takes the additive expression in (1), the two metavariables store ASTs of two Number literals, 1 and 2, respectively. It means that abstract algorithms in ECMA-262 treat ASTs as values and can store them in variables or pass them as arguments of other algorithms. The "?" operator is a shorthand for the following sequence of steps to handle control flows:

1. If *argument* is an abrupt completion, return Completion(*argument*).
2. Else if *argument* is a Completion Record, set *argument* to *argument*.[[Value]].

where a completion record is *abrupt* when it represents exceptional control flows, such as `throw`, `return`, `break`, and `continue`. In other words, the "?" operator is a branch that checks whether given values are abrupt completions and directly returns them if so.

A *normal algorithm* is the primary form of an abstract algorithm defined by its 1) name, 2) parameters, and 3) body. It is commonly used as a helper function, and multiple normal algorithms are used when defining the semantics of language features. Hence, the semantics of different language features often share the same normal algorithms. For example, both SDOs in Figure 2 invoke the same normal algorithm **EvaluateStringOrNumericBinaryExpression** with different

[2]https://262.ecma-international.org/13.0/#sec-evaluatestringornumericbinaryexpression

Built-in methods

Number (*value*)
1. If *value* is present[23], then
   a. Let *prim* be ?[25] ToNumeric[24](*value*).
...[26]

Fig. 4. Built-in method **Number** in ES13

second arguments **+** and **−**, respectively. Then, they transitively invoke other normal algorithms, **ApplyStringOrNumericBinaryOperator**[3] and **ToNumeric**[4]. Thus, at least three normal algorithms are shared in the semantics of the addition and subtraction expressions.

JavaScript provides diverse built-in APIs as opaque functions, such as `Object.getPrototypeOf` and `Number.prototype.toString`. A *built-in method* defines the semantics of a built-in API. For example, Figure 4 describes the semantics of the `Number`[5] built-in API. Since its primary functionality is to convert a given JavaScript value to its corresponding Number value, it also uses the normal algorithm **ToNumeric** as a helper function.

2.1.3 Language Features. In JavaScript, a language feature is 1) a *syntactic feature* or 2) a *built-in API feature*. A syntactic feature is related to a specific JavaScript syntax consisting of an alternative in a syntactic production and its corresponding SDO. On the other hand, a built-in API feature is related to a built-in API instead of a specific syntax. For example, the `+` and `-` operators are syntactic features (f_{add} and f_{sub}) defined by the second and third alternatives of *AdditiveExpression* and their corresponding **Evaluation** SDOs. The **Number** built-method describes the semantics of the built-in `Number` API feature ($f_{\text{B:Number}}$).

2.2 Control Flow Graph (CFG) of ECMA-262

To define the coverage of a conformance test suite using graph coverage criteria, we need a directed graph of the JavaScript mechanized specification. CFG is the most common way to construct a directed graph from a mechanized language specification. In a CFG, a node denotes a sequence of instructions, and an edge indicates a control flow in the mechanized specification. An edge often has an annotation to represent a specific control flow, such as conditional branches (`#t` or `#f`) and function calls (`call`) and returns (`ret`).

For example, Figure 5 depicts a CFG of the abstract algorithms in Figures 2, 3, and 4. In this figure, circles (or diamonds) denote nodes (or branches), arrows denote edges, and boxes indicate algorithms. The labels inside nodes match the labels annotated in the algorithms in Figures 2, 3, and 4. Let us apply coverage-guided fuzzing [35] with a node coverage criterion in the CFG and assume that a simple JavaScript program, `1 + 2`, exists in the program pool. The program does not satisfy the condition in the branch labeled 20 because the left-hand and right-hand sides of `1 + 2` are both Number values rather than BigInt values. Thus, it does not cover the red node labeled 21. Now assume that another program, `3n + 4n`, is generated by mutating the previous program. Then, it covers the red node labeled 21 because it satisfies the condition in the branch labeled 20 with BigInt values on both sides of the `+` operator.

[3] https://262.ecma-international.org/13.0/#sec-applystringornumericbinaryoperator
[4] https://262.ecma-international.org/13.0/#sec-tonumeric
[5] https://262.ecma-international.org/13.0/#sec-number-constructor-number-value

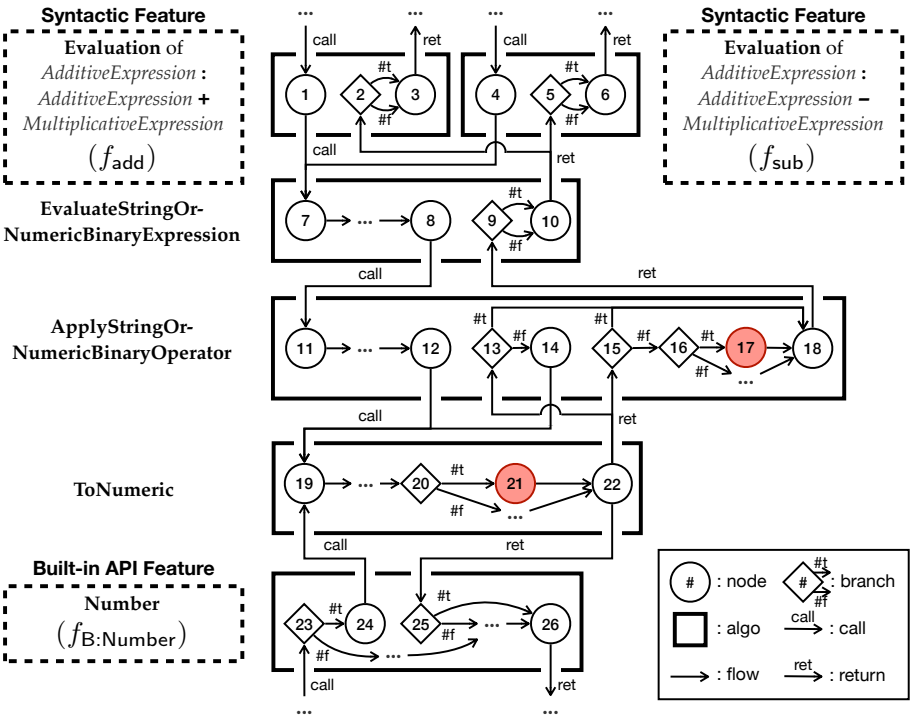


Fig. 5. Control-flow graph (CFG) of abstract algorithms in Figures 2, 3, and 4

2.3 Motivation

Unfortunately, a simple node coverage criterion in CFGs of mechanized specifications cannot fully discriminate different semantics in different language features or even in the same feature. We explain such cases with simple examples using the CFG in Figure 5.

2.3.1 Different Semantics in Different Language Features. The semantics of different language features may use the same abstract algorithms as helper functions. For example, the semantics of `+` and `-` operators transitively use **ApplyStringOrNumericBinaryOperator**. In the algorithm, the red node labeled 17 represents throwing **TypeError** exception. If the program pool contains a program `2n + 1`, it covers the red node labeled 17 because it has different types of numeric values, a BigInt `2n` and a Number `1`, as the left-hand and right-hand sides of the `+` operator. Similarly, another program `2n - 1` using the `-` operator covers the node. However, `2n - 1` will not be added to the program pool because the node labeled 17 is already covered by `2n + 1`, even though `2n - 1` may reveal a different implementation of the semantics. For a higher quality of conformance testing, a more fine-grained definition of graph coverage is necessary to discriminate `2n + 1` and `2n - 1`.

2.3.2 Different Semantics in the Same Language Feature. In addition, different parts in the semantics of the same language feature may use the same algorithm more than once. For example, the semantics of the `+` operator uses **ApplyStringOrNumericBinaryOperator**, and it invokes **ToNumeric** twice in the nodes labeled 12 and 14. Now, assume that the current program pool contains a program `2n + 1` again. Then, the red node labeled 21 is covered by the program `2n + 1` because the left-hand side is a BigInt `2n`. It means that another similar program `1 + 2n` would not be added to the program pool because the test requirement for the node labeled 21 is already covered by `2n + 1`. However,

`1 + 2n` is also a meaningful test case because it checks the edge case when the right-hand side of the + operator is a BigInt value.

In the remainder of the paper, we formally define a feature-sensitive coverage criterion and its variants to resolve the problems (Section 3). Then, we explain how to implement a conformance test synthesizer with feature-sensitive coverage criteria (Section 4). Finally, after evaluating feature-sensitive coverage criteria with mainstream JavaScript implementations (Section 5), we discuss related work (Section 6) and conclude (Section 7).

3 FEATURE-SENSITIVE COVERAGE CRITERIA

This section formulates a general definition of graph coverage for a directed graph and explains representative coverage criteria as examples. Then, we introduce *feature-sensitive (FS) coverage* criteria as general extensions of graph coverage criteria to discriminate semantics between different language features. Finally, we define *feature-call-path-sensitive (FCPS) coverage* criteria as variants of FS coverage criteria to distinguish different parts in the semantics of the same language feature.

3.1 Notations

First, we define notations used in the definition of graph coverage criteria. A *directed graph* $\mathbb{G} = (\mathbb{N}, \mathbb{N}_\iota, \mathbb{N}_f, \mathbb{E})$ consists of:

- a set of *nodes* $\mathbb{N}$
- a set of *initial nodes* $\mathbb{N}_\iota \subseteq \mathbb{N}$
- a set of *final nodes* $\mathbb{N}_f \subseteq \mathbb{N}$
- a set of *edges* $\mathbb{E} \subseteq \mathbb{N} \times \mathbb{N} \times (\mathbb{A} \uplus \{\bot\})$ with a set of *annotations* $\mathbb{A}$

The notation $n \xrightarrow{a} n'$ denotes an edge from a node n to a node n' with an annotation $a \in \mathbb{A}$. If an edge has an empty annotation $\bot$, we omit the annotation: $n \rightarrow n'$. In a given directed graph $\mathbb{G}$, a *path* $p \in \mathbb{P}_\mathbb{G}$ is a sequence of one or more nodes, where each pair of adjacent nodes is an edge:

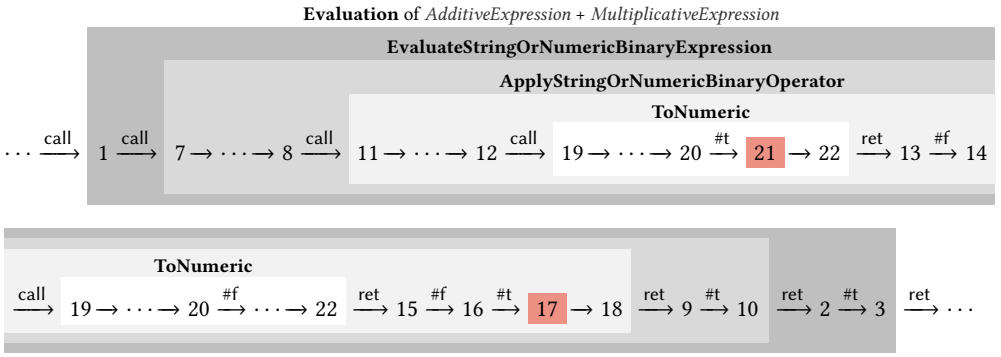
$$\mathbb{P}_\mathbb{G} = \{n_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{m-1}} n_m \mid \forall i < m.\ n_i \xrightarrow{a_i} n_{i+1} \in \mathbb{E} \wedge n_m \in \mathbb{N}\} \tag{2}$$

The length of a path is defined as $\|n_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{m-1}} n_m\| = m$. A path p is a *subpath* ($\sqsubseteq$) of a path p' when p is a subsequence of p'. We use the notation $\preceq$ for a prefix relation, and $\text{first}(p)$ and $\text{last}(p)$ denote the first and last nodes of the path p, respectively. A path p is a ~~*full*~~*test path* when it starts at an initial node and ends at a final node: $\text{first}(p) \in \mathbb{N}_\iota \wedge \text{last}(p) \in \mathbb{N}_f$. Then, $\text{path}_\mathbb{G} : \mathbb{T} \rightarrow \mathbb{P}_\mathbb{G}$ is a mapping from a *test* $t \in \mathbb{T}$ to a ~~full path in the graph~~ test path in $\mathbb{G}$, and we call $\text{path}_\mathbb{G}(t)$ the *execution path* of t.

Example. Consider the CFG $\mathbb{G}$ in Figure 5 and the following JavaScript programs as a test set $T \subseteq \mathbb{T}$:

$$T = \{\cdots, \underbrace{\texttt{2n + 1}}_{t_{\text{add}}}, \underbrace{\texttt{2n - 1}}_{t_{\text{sub}}}, \cdots\} \tag{3}$$

Then, $\text{path}_{\mathbb{G}}(t_{\text{add}})$, the execution path of t_{add}, is depeicted as follows:



And, $\text{path}_{\mathbb{G}}(t_{\text{sub}})$ is equal to $\text{path}_{\mathbb{G}}(t_{\text{add}})$ except for nodes labeled ~~4, 5, and 6~~ 1, 2, and 3 in the **Evaluation** SDO for ~~subtraction rather than 1, 2, and 3~~ addition replaced with nodes labeled 4, 5, and 6 in the **Evaluation** SDO for ~~addition~~subtraction. ~~The~~ Note that the following path p is a subpath of both $\text{path}_{\mathbb{G}}(t_{\text{add}})$ and $\text{path}_{\mathbb{G}}(t_{\text{sub}})$:

$$p = 22 \xrightarrow{\text{ret}} 15 \xrightarrow{\#\text{f}} 16 \xrightarrow{\#\text{t}} 17 \tag{4}$$

whose length is $\|p\| = 3$.

3.2 Graph Coverage Criteria

We formulate ~~graph coverage~~ *graph coverage* criteria by referring to their well-known definitions [4]. ~~For a given directed graph, we specify a *graph coverage* criterion~~ , and they are defined by 1) a set of *test requirements* and 2) a *cover relation* between paths and test requirements:

Definition 3.1 (Graph Coverage Criteria). A *graph coverage* criterion $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$ for a given directed graph $\mathbb{G}$ is defined with:

- a set of *test requirements (TRs)* $\mathbb{R}_{\mathbb{G}}$
- a *cover relation* $\overset{\text{cover}}{\sim} \subseteq \mathbb{P}_{\mathbb{G}} \times \mathbb{R}_{\mathbb{G}}$ between paths and TRs

In a specific graph coverage criterion $C_{\mathbb{G}}$, we say that a path p *covers* a TR $r \in \mathbb{R}_{\mathbb{G}}$ when $p \overset{\text{cover}}{\sim} r$. A test $t \in \mathbb{T}$ covers the TR r if there exists a prefix path p of its execution path that covers the TR:

$$t \overset{\text{cover}}{\sim} r \iff \exists p \in \mathbb{P}_{\mathbb{G}}.\ \text{s.t. } p \preceq \text{path}_{\mathbb{G}}(t) \wedge p \overset{\text{cover}}{\sim} r \tag{5}$$

A test set $T \subseteq \mathbb{T}$ *satisfies* ($\vdash$) the criterion $C_{\mathbb{G}}$ when it covers all ~~valid TRs:~~ feasible TRs:

$$T \vdash C_{\mathbb{G}} \iff \forall r \in \mathbb{R}_{\mathbb{G}}.\ r \text{ is valid}\text{is feasible} \Rightarrow \exists t \in T.\ \text{s.t. } t \overset{\text{cover}}{\sim} r \tag{6}$$

where a TR r is ~~*valid*~~*feasible* if there exists a possible test $t \in \mathbb{T}$ that covers r. If $T \vdash C_{\mathbb{G}} \Rightarrow T \vdash C'_{\mathbb{G}}$ for any test set T, we say that $C_{\mathbb{G}}$ *subsumes* $C'_{\mathbb{G}}$ and use the notation: $C_{\mathbb{G}} \rhd C'_{\mathbb{G}}$. The subsumption relation between graph coverage criteria is a ~~partial order~~preorder.

Definition 3.2 (Node Coverage). In a *node coverage* criterion $C^{\text{node}}_{\mathbb{G}}$,

- the set of **TRs** $\mathbb{R}_{\mathbb{G}}$ is a set of nodes: $\mathbb{R}_{\mathbb{G}} = \mathbb{N}$
- a path p **covers** a node n when it ends with the node n: $p \overset{\text{cover}}{\sim} n \iff \text{last}(p) = n$ [6]

[6] Another way to define node coverage is using a *visit relation* between paths and any nodes in the paths. However, we use a *cover relation* between paths and the last nodes in the paths because it is suitable for further extensions of graph coverage.

The *node coverage* criterion is the most common graph coverage criterion whose test requirements are nodes, and we can generalize it into *k-limiting path coverage* criteria using paths:

Definition 3.3 (k-Limiting Path Coverage). In a *k-limiting path coverage* criterion $C_{\mathbb{G}}^{k\text{-path}}$,

- the set of **TRs** $\mathbb{R}_{\mathbb{G}}$ is a set of paths whose lengths are bounded by k: $\mathbb{R}_{\mathbb{G}} = \{p \in \mathbb{P}_{\mathbb{G}} \mid \|p\| \leq k\}$
- a path p **covers** a path p' when their last nodes are equal and the path p' is a subpath of p: $p \overset{\text{cover}}{\sim} p' \iff \text{last}(p) = \text{last}(p') \wedge p' \sqsubseteq p$

Now, the node coverage criterion can be redefined as the 0-limiting path coverage criterion ($C_{\mathbb{G}}^{0\text{-path}} = C_{\mathbb{G}}^{\text{node}}$), and other graph coverage criteria match with k-limiting path coverages as well:

- The *edge coverage* criterion is $C_{\mathbb{G}}^{1\text{-path}}$
- The *edge-pair coverage* criterion is $C_{\mathbb{G}}^{2\text{-path}}$
- The *complete path coverage* criterion is $C_{\mathbb{G}}^{\infty\text{-path}}$

Note that k-limiting path coverage criteria utilize the inequality for path lengths $\|p\| \leq k$ rather than equality $\|p\| = k$. Thus, if $i \leq j$, the set of TRs in $C_{\mathbb{G}}^{i\text{-path}}$ is always a subset of that in $C_{\mathbb{G}}^{j\text{-path}}$, and $C_{\mathbb{G}}^{j\text{-path}}$ subsumes $C_{\mathbb{G}}^{i\text{-path}}$. The *branch coverage* criterion is a variant of the edge coverage criterion that treats only out-edges of conditional branches as TRs. It is possible to merge multiple coverage criteria by defining unions of their TRs and cover relations. For example, a *node-or-branch coverage* criterion is a merge of node and branch coverage criteria.

The complete path coverage criterion might have infinite TRs because of recursions and loop structures. To resolve this problem, Ammann and Offutt [4] have presented a *simple path coverage* criterion that considers only simple paths as TRs. A path $n_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{m-1}} n_m$ is *simple* if there are no duplicated nodes in the path, with the exception that the first and last nodes may be identical: $\forall i, j.\ n_i = n_j \Rightarrow (i = j \vee \{i, j\} = \{0, m\})$. In addition, they extend it to a *prime path coverage* criterion to reduce the number of TRs by filtering out duplicated simple paths, where a *prime path* is a maximal length simple path in the graph. However, even such advanced structural coverage criteria still need many TRs for the entire control-flow graphs. Hence, they are usually used for unit testing [32] in practice with intra-procedural control-flow graphs.

Example. Consider the CFG $\mathbb{G}$ in Figure 5 and the test set T in (3), including t_{add} and t_{sub}. If we measure the 3-limiting path coverage $C_{\mathbb{G}}^{3\text{-path}}$ for T, both the node labeled 17 and the path p in (4) are test requirements $\mathbb{R}_{\mathbb{G}}$. First, the prefix path, whose last node is 17, of $\text{path}_{\mathbb{G}}(t_{\text{add}})$ covers both TRs: the node labeled 17 and p. Thus, the test t_{add} for addition covers both of them. Similarly, the test t_{sub} for subtraction covers both of them for the same reason. Unfortunately, either t_{add} or t_{sub} might be removed in the program pool because they cover the same TRs, the node labeled 17 and the path p.

3.3 Feature-Sensitive (FS) Coverage Criteria

To alleviate the problem introduced in Section 2.3.1, we present *feature-sensitive (FS) coverage* criteria as general extensions of any graph coverages depending on the following three components:

- a given graph coverage criterion $C_{\mathbb{G}}$
- a set of *language features* $\mathbb{F}$
- a *feature mapping* $\text{feat} : \mathbb{N} \to \mathbb{F} \uplus \{\bot\}$, a partial mapping from nodes to language features

where $\text{feat}(n) = \bot$ means that there is no language feature for the node n.

We first define the *call-site stack* $p\,|_{\text{call}} \in \mathbb{N}^*$ of a path p as a sequence of nodes constructed by:

$$p\,|_{\text{call}} = \begin{cases} \epsilon & \text{if } p = n \\ [n_1, \cdots, n_m, \text{last}(p')] & \text{if } p = p' \xrightarrow{\text{call}} n \wedge p'\,|_{\text{call}} = [n_1, \cdots, n_m] \\ [n_1, \cdots, n_{m-1}] & \text{if } p = p' \xrightarrow{\text{ret}} n \wedge p'\,|_{\text{call}} = [n_1, \cdots, n_m] \\ p'\,|_{\text{call}} & \text{if } p = p' \xrightarrow{a} n \text{ where } a \notin \{\text{call}, \text{ret}\} \end{cases} \tag{7}$$

In other words, $p\,|_{\text{call}}$ keeps only call-sites not matched with return-sites in the path p. ~~A~~ ; a *call-site* is a node having a call edge ($\xrightarrow{\text{call}}$) as its out-edge, and a *return-site* is a node having a return edge ($\xrightarrow{\text{ret}}$) as its in-edge. Then, we define the *feature extractor* $\text{ext}_{\mathbb{F}} : \mathbb{P}_{\mathbb{G}}\,|_{\text{call}} \rightarrow \mathbb{F} \uplus \{\bot\}$ as a partial mapping from call-site stacks to the innermost enclosing language features $\mathbb{F}$:

$$\text{ext}_{\mathbb{F}}([n_1, \cdots, n_m]) = \begin{cases} f & \text{if } \exists i.\ \text{s.t. } \text{feat}(n_i) = f \wedge \forall j > i.\ \text{feat}(n_j) = \bot \\ \bot & \text{otherwise} \end{cases} \tag{8}$$

Similarly, $\text{ext}_{\mathbb{F}}(p\,|_{\text{call}}) = \bot$ means that there is no language feature for the path p.

Definition 3.4 (Feature-Sensitive (FS) Coverage Criteria). For a given graph coverage criterion $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *feature-sensitive (FS) coverage* criterion $C^{\text{FS}}_{\mathbb{G}} = (\mathbb{R}^{\text{FS}}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$ is defined as follows:

- the set of **feature-sensitive test requirements (FS-TRs)** $\mathbb{R}^{\text{FS}}_{\mathbb{G}}$ is a set of original TRs optionally ~~tagged~~ with language features: $\mathbb{R}^{\text{FS}}_{\mathbb{G}} = \mathbb{R}_{\mathbb{G}} \times (\mathbb{F} \uplus \{\bot\})$
- a path p **covers** a FS-TR (r, f) when p covers the original TR r and f is the innermost enclosing language feature of p: $p \overset{\text{cover}}{\sim} (r, f) \iff p \overset{\text{cover}}{\sim} r \wedge \text{ext}_{\mathbb{F}}(p\,|_{\text{call}}) = f$

Example. For the CFG $\mathbb{G}$ in Figure 5, assume that the feature mapping is as follows:

$$\text{feat}(n) = \begin{cases} f_{\text{add}} & \text{if } n \in \{1, 2, 3\} \\ f_{\text{sub}} & \text{if } n \in \{4, 5, 6\} \end{cases} \qquad \begin{matrix} f_{\text{B:Number}} & \text{if } n \in \{23, 24, 25, 26\} \\ \bot & \text{otherwise} \end{matrix} \tag{9}$$

Consider two tests t_{add} and t_{sub} in the test set T (3), and the following two prefix paths p_{add} and p_{sub}, whose last nodes are labeled 17, of their execution paths:

$$p_{\text{add}} : \cdots \xrightarrow{\text{call}} 1 \xrightarrow{\text{call}} \cdots \rightarrow 8 \xrightarrow{\text{call}} \cdots \rightarrow 12 \xrightarrow{\text{call}} \cdots \xrightarrow{\text{ret}} 13 \xrightarrow{\#\text{f}} 14 \xrightarrow{\text{call}} \cdots \xrightarrow{\text{ret}} 15 \xrightarrow{\#\text{f}} 16 \xrightarrow{\#\text{t}} 17$$
$$p_{\text{sub}} : \cdots \xrightarrow{\text{call}} 4 \xrightarrow{\text{call}} \cdots \rightarrow 8 \xrightarrow{\text{call}} \cdots \rightarrow 12 \xrightarrow{\text{call}} \cdots \xrightarrow{\text{ret}} 13 \xrightarrow{\#\text{f}} 14 \xrightarrow{\text{call}} \cdots \xrightarrow{\text{ret}} 15 \xrightarrow{\#\text{f}} 16 \xrightarrow{\#\text{t}} 17$$

First, the call-site stack of p_{add} is $p_{\text{add}}\,|_{\text{call}} = [\cdots, 1, 8]$ because other call-sites labeled 12 and 14 are removed by matched return-sites labeled 13 and 15, respectively. Since there is no feature mapping for the call-site labeled 8, the innermost enclosing feature of p_{add} is $\text{ext}_{\mathbb{F}}(p_{\text{add}}\,|_{\text{call}}) = \text{feat}(1) = f_{\text{add}}$. Hence, if we use a FS node coverage criterion $C^{\text{FS[node]}}_{\mathbb{G}}$, the path p_{add} covers a FS-TR $(17, f_{\text{add}})$, and the test t_{add} for addition covers it as well. In a similar way, we know that the innermost enclosing language feature of p_{sub} is $\text{ext}_{\mathbb{F}}(p_{\text{sub}}\,|_{\text{call}}) = \text{feat}(4) = f_{\text{sub}}$. It means that t_{sub} covers a new FS-TR $(17, f_{\text{sub}})$ instead of $(17, f_{\text{add}})$ and remains in the program pool.

In addition, we extend $\text{ext}_{\mathbb{F}}$ to apply the k-limiting approach to FS coverage criteria. The extended feature extractor $\text{ext}^k_{\mathbb{F}} : \mathbb{P}_{\mathbb{G}}\,|_{\text{call}} \rightarrow \mathbb{F}^{\leq k}$ collects at most k enclosing language features:

$$\text{ext}^k_{\mathbb{F}}([n_1, \cdots, n_m]) = \begin{cases} \epsilon & \text{if } k = 0 \vee m = 0 \\ \text{ext}^{k-1}_{\mathbb{F}}([n_1, \cdots, n_{m-1}]) \cdot f & \text{if } \text{feat}(n_m) = f \\ \text{ext}^k_{\mathbb{F}}([n_1, \cdots, n_{m-1}]) & \text{otherwise} \end{cases} \tag{10}$$

Definition 3.5 (k-Limiting Feature-Sensitive (k-FS) Coverage Criteria). For a given $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *k-limiting feature-sensitive (k-FS) coverage* criterion $C^{k\text{-FS}}_{\mathbb{G}} = (\mathbb{R}^{k\text{-FS}}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$ is defined as follows:

- the set of k-**feature-sensitive test requirements (k-FS-TRs)** $\mathbb{R}_{\mathbb{G}}^{k\text{-FS}}$ is a set of original TRs ~~tagged~~ with at most k language features: $\mathbb{R}_{\mathbb{G}}^{k\text{-FS}} = \mathbb{R}_{\mathbb{G}} \times \mathbb{F}^{\leq k}$
- a path p **covers** a k-FS-TR $(r, \overline{f})$ when p covers the original TR r and $\overline{f}$ is the k-most enclosing language features of p: $p \overset{\text{cover}}{\sim} (r, \overline{f}) \iff p \overset{\text{cover}}{\sim} r \wedge \text{ext}^k_{\mathbb{F}}(p|_{\text{call}}) = \overline{f}$

Example. Consider a JavaScript program `[] - (2n + 1)` as a test t with the graph in Figure 5. It throws a **TypeError** exception in the node labeled 17 during the execution of its sub-expression `2n + 1`. Let p be a prefix path, whose last node is labeled 17, of the execution path of t. Then, $\text{ext}^2_{\mathbb{F}}(p) = [f_{\text{sub}}, f_{\text{add}}]$ because the innermost enclosing feature is f_{add}, and the next enclosing one is f_{sub} for the path p. If we use 2-FS node coverage criterion $C_{\mathbb{G}}^{\text{2-FS[node]}}$, the set of 2-FS-TRs is $\mathbb{R}_{\mathbb{G}}^{\text{2-FS}} = (\mathbb{N}, \mathbb{F}^{\leq 2})$, and the test t covers a 2-FS-TR $(17, [f_{\text{sub}}, f_{\text{add}}])$.

The k-FS coverage criteria divide TRs using each combination of different language features. Especially, the k-FS coverage criteria with $k \geq 2$ are helpful to cover edge cases in JavaScript engines and transpilers because they are heavily optimized and handle even the same language features differently depending on which language features are used together. For example, a *destructuring pattern* [17] helps developers easily declare variables with the values stored in the properties of an object or an array. However, JavaScript engines and transpilers often have a different execution path to handle the pattern when it is declared in a `for-in/of` statement. Actually, the GraalJS [3] engine and the Babel [1] transpiler contain conformance bugs and crashing bugs, respectively, that are reproducible only with a combination of a destructuring pattern and a `for-in/of` statement.[7]

3.4 Feature-Call-Path-Sensitive (FCPS) Coverage Criteria

As explained in Section 2.3.2, a more fine-grained set of test requirements is necessary to distinguish different parts of the semantics in the same language feature. Thus, we define *feature-call-path-sensitive (FCPS) coverage* criteria as variants of FS coverage criteria. The core idea is to distinguish TRs using paths from the innermost enclosing language features. However, if we keep paths as they are, the number of TRs exponentially increases because of the path explosion caused by sequential branches. Hence, we abstract a path p to a corresponding *feature-call-path* $f_{\text{cp}} \in \mathbb{F}_{\text{cp}} = \mathbb{F} \times \mathbb{P}_{\mathbb{G}}|_{\text{call}}$, which consists of the innermost enclosing feature and a subsequence of the call-site stack $p|_{\text{call}}$ from the feature. We define the *feature-call-path extractor* $\text{ext}_{\mathbb{F}_{\text{cp}}} : \mathbb{P}_{\mathbb{G}}|_{\text{call}} \to (\mathbb{F}_{\text{cp}} \uplus \{\bot\})$:

$$\text{ext}_{\mathbb{F}_{\text{cp}}}([n_1, \cdots, n_m]) = \begin{cases} \bot & \text{if } m = 0 \\ (f, [n_m]) & \text{if } \text{feat}(n_m) = f \\ f_{\text{cp}} & \text{if } f_{\text{cp}} = \bot \\ (f, [n'_0, \cdots, n'_i]) & \text{if } f_{\text{cp}} = (f, [n'_0, \cdots, n'_{m'}]) \wedge \exists i. \text{ s.t. } n'_i = n_m \\ (f, \overline{n} \cdot n_m) & \text{if } f_{\text{cp}} = (f, \overline{n}) \end{cases} \tag{11}$$

where $f_{\text{cp}} = \text{ext}_{\mathbb{F}_{\text{cp}}}([n_1, \cdots, n_{m-1}])$. The algorithm starts with $\bot$, denoting no feature-call-path for p, because no enclosing feature exists in the beginning ($m = 0$). It then recursively keeps the call-sites in a given call-site stack $p|_{\text{call}}$. However, it refreshes the result when there exists a mapping from the current call-site to a language feature ($\text{feat}(n_m) = f$). It also removes cycles to prevent a possibly infinite length of feature-call-path and removes duplicated cases ($\exists i.$ s.t. $n'_i = n_m$).

Definition 3.6 (Feature-Call-Path-Sensitive (FCPS) Coverage Criteria). For a given $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *feature-call-path-sensitive (FCPS) coverage* criterion $C_{\mathbb{G}}^{\text{FCPS}} = (\mathbb{R}_{\mathbb{G}}^{\text{FCPS}}, \overset{\text{cover}}{\sim})$ is defined as follows:

- the set of **feature-call-path-sensitive test requirements (FCPS-TRs)** $\mathbb{R}_{\mathbb{G}}^{\text{FCPS}}$ is a set of original TRs optionally ~~tagged~~ with feature-call-paths: $\mathbb{R}_{\mathbb{G}}^{\text{FCPS}} = \mathbb{R}_{\mathbb{G}} \times (\mathbb{F}_{\text{cp}} \uplus \{\bot\})$

[7] We anonymized links of bug reports for double-blinded reviewing.

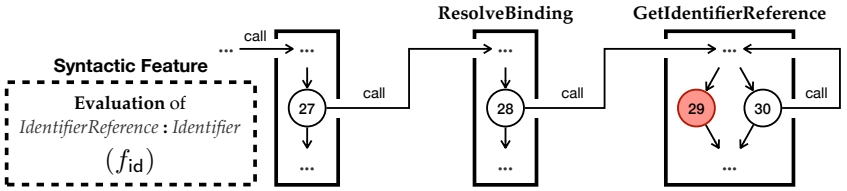


Fig. 6. Excerpt from the CFG of abstract algorithms for *IdentifierReference*

- a path p **covers** a FCPS-TR (r, f_{cp}) when p covers the original TR r and f_{cp} is the feature-call-path extracted from p: $p \overset{\text{cover}}{\sim} (r, f_{cp}) \iff p \overset{\text{cover}}{\sim} r \wedge \text{ext}_{\mathbb{F}_{cp}}(p|_{\text{call}}) = f_{cp}$

Example. We show two examples for FCPS node coverage criteria. First, consider the following two JavaScript programs as tests with the CFG in Figure 5:

$$t_0 = \texttt{2n + 1} \qquad t_1 = \texttt{1 + 2n} \tag{12}$$

If we use FS node coverage criterion $C_{\mathbb{G}}^{\text{FS[node]}}$, both tests t_0 and t_1 cover the same FS-TR $(21, f_{\text{add}})$, and one of them might be removed in the program pool. However, if we use FCPS node coverage criterion $C_{\mathbb{G}}^{\text{FCPS[node]}}$, t_0 and t_1 cover different FCPS-TRs $(21, (f_{\text{add}}, [1, 8, 12]))$ and $(21, (f_{\text{add}}, [1, 8, 14]))$, respectively. The other example is about the cycles in the call-site stacks with the graph in Figure 6. It depicts an excerpt from the CFG of abstract algorithms in ES13 transitively used in f_{id}, a syntactic feature defined by the first alternative of *IdentifierReference* and its **Evaluation** SDO. Assume that we do not remove cycles in the feature-call-paths $\mathbb{F}_{cp}$ during the extraction algorithm $\text{ext}_{\mathbb{F}_{cp}}$. Then, since the algorithm **GetIdentifierReference** contains a self-recursion, there exists an infinite number of possible feature-call-paths from f_{id} to the node labeled 29:

$$(f_{\text{id}}, [27, 28]) \qquad (f_{\text{id}}, [27, 28, 30]) \qquad (f_{\text{id}}, [27, 28, 30, 30]) \qquad \cdots \tag{13}$$

Thus, we remove cycles in feature-call-paths to resolve this issue, and there exists only two possible feature-call-paths: $(f_{\text{id}}, [27, 28])$ and $(f_{\text{id}}, [27, 28, 30])$.

Similar to the extension of FS coverage criteria to k-FS coverage criteria, we define k-FCPS coverage criteria by extending $\text{ext}_{\mathbb{F}_{cp}}$ into $\text{ext}^k_{\mathbb{F}_{cp}} : \mathbb{P}_{\mathbb{G}}|_{\text{call}} \to \mathbb{F}_{cp}^{\leq k}$:

$$\text{ext}^k_{\mathbb{F}_{cp}}([n_1, \cdots, n_m]) = \begin{cases} (\epsilon, \epsilon) & \text{if } k = 0 \vee m = 0 \\ (\overline{f} \cdot f, [n_m]) & \text{if } \text{feat}(n_m) = f \wedge \text{ext}^{k-1}_{\mathbb{F}_{cp}}([n_1, \cdots, n_{m-1}]) = (\overline{f}, \_) \\ \overline{f_{cp}} & \text{if } \overline{f_{cp}} = (\epsilon, \epsilon) \\ (\overline{f}, [n'_0, \cdots, n'_i]) & \text{if } \overline{f_{cp}} = (\overline{f}, [n'_0, \cdots, n'_{m'}]) \wedge \exists i. \text{ s.t. } n'_i = n_m \\ (\overline{f}, \overline{n} \cdot n_m) & \text{if } \overline{f_{cp}} = (\overline{f}, \overline{n}) \end{cases} \tag{14}$$

where $\mathbb{F}_{cp}^{\leq k} = \mathbb{F}^{\leq k} \times \mathbb{P}_{\mathbb{G}}|_{\text{call}}$ is a set of extended feature-call-paths, and $\overline{f_{cp}} = \text{ext}^k_{\mathbb{F}_{cp}}([n_1, \cdots, n_{m-1}])$.

Definition 3.7 (k-Limiting Feature-Call-Path-Sensitive (k-FCPS) Coverage Criteria). For a given $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *k-limiting feature-call-path-sensitive (k-FCPS) coverage* criterion $C_{\mathbb{G}}^{k\text{-FCPS}} = (\mathbb{R}_{\mathbb{G}}^{k\text{-FCPS}}, \overset{\text{cover}}{\sim})$ is defined as follows:

- the set of **k-feature-call-path-sensitive test requirements (k-FCPS-TRs)** $\mathbb{R}_{\mathbb{G}}^{k\text{-FCPS}}$ is a set of original TRs ~~tagged~~ with the extended feature-call-paths bounded by k: $\mathbb{R}_{\mathbb{G}}^{k\text{-FCPS}} = \mathbb{R}_{\mathbb{G}} \times \mathbb{F}_{cp}^{\leq k}$

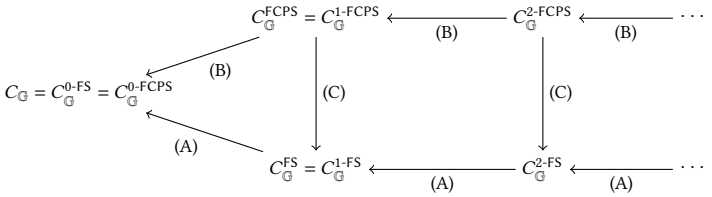


Fig. 7. Subsumption relations between k-FS and k-FCPS coverage criteria

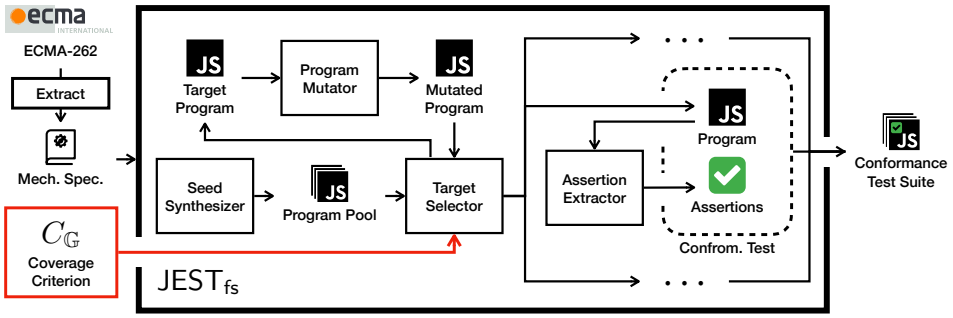


Fig. 8. Overall structure of JEST$_{\text{fs}}$

- a path p **covers** a k-FCPS-TR $(r, \overline{f_{\text{cp}}})$ when p covers the original TR r and $\overline{f_{\text{cp}}}$ is the extended feature-call-path extracted from p bounded by k:

$$p \overset{\text{cover}}{\sim} (r, \overline{f_{\text{cp}}}) \iff p \overset{\text{cover}}{\sim} r \wedge \text{ext}^{k}_{\mathbb{F}_{\text{cp}}}(p\,|_{\text{call}}) = \overline{f_{\text{cp}}}$$

We prove Theorem 3.9 for the subsumption relations between k-FS and k-FCPS coverage criteria. Figure 7 illustrates the relations using edges annotated with equations in Theorem 3.9.

Lemma 3.8. *Consider two graph coverage criteria $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{cover}{\sim})$ and $C'_{\mathbb{G}} = (\mathbb{R}'_{\mathbb{G}}, \overset{cover'}{\sim})$. If there exists a ~~valid~~ feasible TR $r \in \mathbb{R}_{\mathbb{G}}$ that satisfies the following condition for each ~~valid~~ feasible TR $r' \in \mathbb{R}'_{\mathbb{G}}$:*

$$\forall t \in \mathbb{T}.\ t \overset{cover}{\sim} r \Rightarrow t \overset{cover'}{\sim} r'. \tag{15}$$

Then, $C_{\mathbb{G}}$ subsumes $C'_{\mathbb{G}}$ ($C_{\mathbb{G}} \triangleright C'_{\mathbb{G}}$).

Proof. Assume $T \vdash C_{\mathbb{G}}$. For a given ~~valid~~ feasible TR $r' \in \mathbb{R}'_{\mathbb{G}}$, let $r \in \mathbb{R}_{\mathbb{G}}$ be the ~~valid~~ feasible TR satisfying (15). Then, there exists a test $t \in T$ such that $t \overset{\text{cover}}{\sim} r$ because r is ~~valid~~ feasible and $T \vdash C_{\mathbb{G}}$. Finally, $t \overset{\text{cover}}{\sim} r'$. □

Theorem 3.9 (Subsumption Relation). *For a given integer $k > 0$, the following three subsumption relations ($\triangleright$) between k-FS and k-FCPS coverage criteria satisfy:*

$$(A)\ C_{\mathbb{G}}^{k\text{-FS}} \triangleright C_{\mathbb{G}}^{(k-1)\text{-FS}} \qquad (B)\ C_{\mathbb{G}}^{k\text{-FCPS}} \triangleright C_{\mathbb{G}}^{(k-1)\text{-FCPS}} \qquad (C)\ C_{\mathbb{G}}^{k\text{-FCPS}} \triangleright C_{\mathbb{G}}^{k\text{-FS}}$$

Proof. We prove (A) using Lemma 3.8, and omit the other cases because their proofs are similar. Let $k > 0$. For a given ~~valid~~ feasible $(k-1)$-FS-TR $(r, \overline{f})$, there exists a test $t \in \mathbb{T}$ such that $t \overset{\text{cover}}{\sim} (r, \overline{f})$ because $(r, \overline{f})$ is ~~valid~~feasible. There exists a prefix path p of $\text{path}_{\mathbb{G}}(t)$ such that $p \overset{\text{cover}}{\sim} (r, \overline{f})$ ($\because$ (5)). Then, a k-FS-TR $(r, \text{ext}^{k}_{\mathbb{F}}(p\,|_{\text{call}}))$ satisfies the condition (15) because of the inductive definition of $\text{ext}^{k}_{\mathbb{F}}$ in (10). Finally, k-FS coverage criteria subsume $(k-1)$-FS coverage criteria. □

4 IMPLEMENTATION

This section introduces our JavaScript conformance test synthesizer supporting k-FS and k-FCPS coverage criteria, JEST_{fs}, and explains how it can detect conformance bugs in JavaScript implementations. Figure 8 illustrates its overall structure. JEST_{fs} takes 1) a mechanized specification extracted by ESMeta [2] and 2) a coverage criterion $C_{\mathbb{G}}$ and performs coverage-guided fuzzing [35] using the CFG of the mechanized specification. It uses the following four modules from JEST [42]:

- **Seed Synthesizer**: As the first step, Seed Synthesizer automatically synthesizes a set of JavaScript programs as the initial *program pool*. It uses the JavaScript syntax described in the language specification to cover diverse alternatives in syntactic productions. JEST_{fs} uses two existing synthesizers: 1) a non-recursive synthesizer and 2) a built-in synthesizer.
- **Target Selector**: To measure the coverage in the CFG, Target Selector extracts the execution path of each program in the pool by interpreting it using the abstract algorithms in the specification. While the baseline tool supports only a node-or-branch coverage criterion, we extend it to support k-FS and k-FCPS node-or-branch coverage criteria as well. If a program does not cover new TRs, it removes the program from the pool. Then, it selects a program in the pool as a mutation target that potentially increases the coverage or stops the iteration when the current status satisfies the termination condition.
- **Program Mutator**: To increase the coverage in the CFG, Program Mutator repeatedly tries to mutate a JavaScript program to a new one using mutation methods. JEST_{fs} uses five mutation methods: 1) random mutation, 2) nearest syntax tree mutation, 3) string substitution, 4) object substitution, and 5) statement insertion.
- **Assertion Extractor**: After the mutation iteration, Assertion Extractor automatically extracts seven kinds of assertions from each program in the pool. The assertions represent the expected final state of each program according to the semantics described in the specification. As a result, each pair of a program and the corresponding extracted assertions is a *conformance test* for JavaScript.

A synthesized conformance test consists of a JavaScript program and corresponding assertions. To check a JavaScript engine's conformance, it is enough to run the program in the test and assertions together using the target engine. If at least one assertion fails, the target engine has a conformance bug related to the test. To check a JavaScript transpiler's conformance, we should transpile the program in the test using the target transpiler. If the transpiler abnormally terminates, it has a conformance bug because programs in conformance tests are valid. Otherwise, we should run the transpiled program and assertions together using a trusted engine. If at least one assertion fails, the target transpiler has a conformance bug related to the test.

5 EVALUATION

This section evaluates feature-sensitive coverage criteria with the following research questions:

- **RQ1 (Conformance Bug Detection):** How many conformance bugs in JavaScript implementations are detected by synthesized conformance tests? (Section 5.1)
- **RQ2 (Effectiveness of k-FS Coverage Criteria):** Are higher k-FS coverage criteria more effective than lower k-FS coverage criteria in detecting conformance bugs? (Section 5.2)
- **RQ3 (Effectiveness of k-FCPS Coverage Criteria):** Are k-FCPS coverage criteria more effective than k-FS coverage criteria in detecting conformance bugs? (Section 5.3)
- **RQ4 (Comparison with Test262):** Can conformance tests synthesized by JEST_{fs} complement Test262, the official JavaScript conformance suite maintained manually? (Section 5.4)

We apply JEST_{fs} to the latest language specification (ES13, 2022) [16], which synthesized 237,981 conformance tests in 50 hours with five graph coverage criteria: 1) 0-FS, 2) 1-FS, 3) 2-FS, 4) 1-FCPS,

Table 1. Detected conformance bugs in JavaScript engines and transpilers

Kind	Name	Version	Release	# Detected Unique Bugs		
				# New	# Confirmed	# Reported
Engine	V8	v10.8.121	2022.10.06	0	0	4
	JSC	v615.1.10	2022.10.26	15	15	24
	GraalJS	v22.2.0	2022.07.26	9	9	10
	SpiderMonkey	v107.0b4	2022.10.24	1	3	4
	Total			**25**	**27**	**42**
Transpiler	Babel	v7.19.1	2022.09.15	30	30	35
	SWC	v1.3.10	2022.10.21	27	27	41
	Terser	v5.15.1	2022.10.05	1	1	18
	Obfuscator	v4.0.0	2022.02.15	0	0	7
	Total			**58**	**58**	**101**
Total				**83**	**85**	**143**

and 5) 2-FCPS node-or-branch coverage. We performed our experiments with five Ubuntu machines with a 4.0GHz Intel(R) Core(TM) i7-6700k and 32GB of RAM (Samsung DDR4 2133MHz 8GB*4).

Using the synthesized JavaScript conformance tests, we check the conformance of eight mainstream implementations listed in Table 1. We select them as evaluation targets because they support all the language features in ES13. V8, JSC, and SpiderMonkey are JavaScript engines used in web browsers, Google Chrome, Apple Safari, and Mozilla Firefox, respectively, and GraalJS is a JavaScript engine by Oracle. Babel and SWC are transpilers that desugar new language features into old ones, usually ES5.1 features, for legacy host environments. Terser is a code compressor that reduces code size, and Obfuscator obfuscates code to make it hard to understand and reverse-engineering. For the transpiler conformance check, we use V8 as the default engine to execute transpiled code with assertions. If a test fails on V8, we use another engine that passes the test; if a test fails on all engines, we do not use the test.

5.1 Conformance Bug Detection

Table 1 summarizes the conformance bugs detected by JEST_{fs} in all the evaluation targets. We manually inspected the failed conformance test cases, found 143 distinct conformance bugs, and reported them to the corresponding developers. As a result, 85 out of 143 bugs were officially confirmed, and 83 were newly discovered bugs. The other 47 reported bugs are still under review, or developers have not yet responded. Among 143 detected bugs, 42 are engine bugs, and 101 are transpiler bugs. We present two real-world bug examples.

Order of Execution. JavaScript engines must follow the execution order of each language feature described in the language specification. However, we found a bug[8] related to the execution order of the `delete` operation that causes the execution of originally unreachable code in GraalJS. For example, while the following code should return `false`, it throws an exception with `"ERR"` by executing the originally unreachable code inside the arrow function in GraalJS:

```
false && delete (() => { throw "ERR"; })(); // Expected: false
```

In addition, we detected another bug[9] related to the execution order of property reads in all target engines. ECMA-262 may consider changing the semantics according to the one used in most implementations.

[8]We anonymized links of bug reports for double-blinded reviewing.
[9]We anonymized links of bug reports for double-blinded reviewing.

Table 2. Comparison of synthesized conformance tests guided by five graph coverage criteria

Coverage Criteria $C_{\mathbb{G}}$	# Covered k-F(CP)S-TR (k)			# Syn. Test	# Bug
	# Node	# Branch	# Total		
0-FS node-or-branch (0-fs)	10.0	5.6	15.6	2,111	55
1-FS node-or-branch (1-fs)	79.3	45.7	125.0	6,766	83
2-FS node-or-branch (2-fs)	1,199.8	696.3	1,896.1	97,423	102
1-FCPS node-or-branch (1-fcps)	179.7	97.6	277.3	9,092	87
2-FCPS node-or-branch (2-fcps)	2,323.1	1,297.6	3,620.7	122,589	111

Asynchronous Function / Generator. One of the complex language features in JavaScript is asynchronous functions and generators introduced in ES6 (2015). We detected a bug[10] in SpiderMonkey that breaks the logic of asynchronous function calls. For example, the following code must return a rejected `Promise` object because a non-iterable value `undefined` is assigned to an array destructuring pattern `[]` in the **`async`** arrow function:

```
(async function ([]) {})(); // Expected: A rejected Promise object
```

However, it unexpectedly terminates with a **TypeError** exception in SpiderMonkey. A developer of SpiderMonkey explained it as follows:

> "The **`async`**-function spec was changed at some point [...]
> this is also not covered by test262."

5.2 Effectiveness of k-FS Coverage Criteria

Table 2 shows the result of conformance test synthesis via JEST_{fs} with five graph coverage criteria. Note that 0-FS node-or-branch coverage criterion is the same with the node-or-branch coverage criterion. To evaluate the effectiveness of k-FS coverage criteria, we compare the synthesized conformance tests guided by different k-FS node-or-branch coverage criteria (0-fs, 1-fs, and 2-fs in Table 2). The second to the fourth columns denote the numbers of covered k-FS- or k-FCPS-TRs for nodes (**# Node**), branches (**# Branch**), and both (**# Total**), respectively. The fifth and the sixth columns denote the numbers of synthesized conformance tests (**# Syn. Test**) and detected distinct bugs (**# Bug**), respectively.

The results show that higher k-FS coverage criteria are more effective than lower k-FS. On average, 8.03 (125.0K / 15.6K) 1-FS-TRs exist per each 0-FS-TR, and 15.17 (1,896.1K / 125.0K) 2-FS-TRs exist per each 1-FS-TR. It means that each node or branch is used in 8.03 different language features, and each language feature could be used in 15.17 other language features on average. For a more detailed information, we draw a histogram of the number of covered 1-FS-TRs (or 2-FS-TRs) per each covered 0-FS-TR (or 1-FS-TR) in Figures 9 (a) and (b). The largest number of covered 1-FS-TRs per each covered 0-FS-TR is 303 for a node in the **[[GetOwnProperty]]** algorithm. In other words, this algorithm is used in 303 different language features, because the semantics of many syntactic or built-in API features use this algorithm to access object properties. The largest number of covered 2-FS-TRs per each covered 1-FS-TR is 116 for a node whose innermost enclosing feature is the syntactic feature f_{id} for identifier references explained in Section 3.4. In other words, the syntactic feature f_{id} is used in 116 different language features, because identifier references can be used in diverse syntactic features, such as function names, destructuring patterns, and property definitions. The number of synthesized tests increased 3.21x (6,766 / 2,111) from 0-FS to 1-FS coverage criteria and 14.4x (97,423 / 6,766) from 1-FS to 2-FS coverage criteria. In addition, the number of detected unique bugs also increased when using higher k-FS node-or-branch coverage

[10]We anonymized links of bug reports for double-blinded reviewing.

criteria. The baseline with 0-FS coverage criterion detects 55 conformance bugs in engines and transpilers. The conformance tests synthesized with 1-FS coverage criterion detect 28 (83 - 55) more conformance bugs, and tests synthesized with 2-FS coverage criterion detect ~~4 (87~~ 19 (102 - 83) more bugs. Now, we present two bug examples that show the effectiveness of k-FS coverage criteria.

Empty Name Binding for `let` in `for`-Loop. JavaScript provides diverse shapes of `for`-loops as syntactic features defined with the *ForStatement* production. Among them, a `for`-loop with a `let`-binding is its third alternative. While it normally has one or more name bindings, we can pass an empty list of name bindings using an empty object destructuring pattern `{}`. However, Babel crashes when transpiling a `for`-loop with empty name bindings for `let`:[11]

```
for (let {} = 0; 0; ) ; // Expected: Normally terminates
```

Because the **CreatePerIterationEnvironment** algorithm that checks the empty name bindings is used for other language features, the tests synthesized with a 0-FS coverage criterion failed to detect this conformance bug. On the other hand, feature-sensitive coverage criteria can discriminate the usage of the empty name binding checking semantics in different language features. Thus, we successfully detected this conformance bug with 1-FS, 2-FS, 1-FCPS, and 2-FCPS coverage criteria.

Computed Property for `async` Method in `class`. JavaScript provides *computed properties* to allow defining property names using any expressions. For example, let's define an object using a computed property: `let x = { ["a"+"b"]() { return 42 } }`. Then, `x` is an object having a property `ab` that stores a function as a method of the object: `x.ab() === 42`. In addition, it also assigns the `name` property of the function as the property name: `x.ab.name === "ab"`. However, JSC does not follow this semantics when the computed property is used for an `async` method inside classes. For example, the following program checks whether the `name` property of the `async` method in the class `C` is `"f"`:[12]

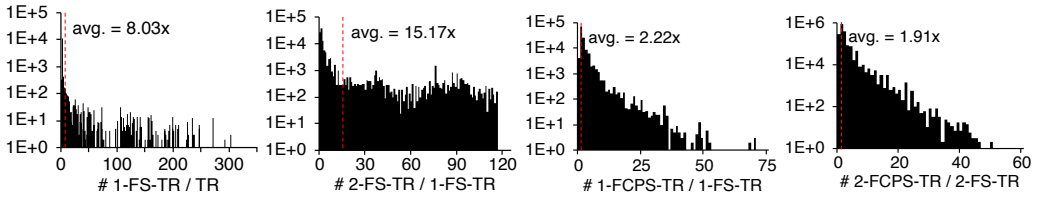
```
class C { async ["f"] () {} } // Expected: C.prototype.f.name === "f"
```

However, the `name` property is `"async"` instead of `"f"` in the JSC engine. Since it is a combination of a `class`, an `async` method, and a computed property, 0 or 1-FS coverage criteria may not keep it in the final program pool. On the other hand, 2-FS coverage criterion can discriminate it with other tests. If a conformance test covers 2-FS-TR consisting of two syntactic features *AsyncMethod* production with **PropMethod** SDO and *ComputedProeprtyName* production with **PropMethod** SDO, it can find this conformance bug. Our experiment successfully detected this conformance bug with tests synthesized with 2-FS and 2-FCPS coverage criteria.

5.3 Effectiveness of k-FCPS Coverage Criteria

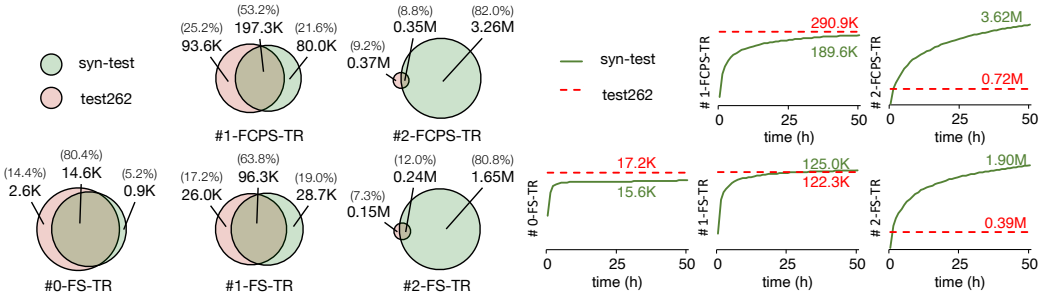
We also evaluate the effectiveness of k-FCPS coverage criteria compared to k-FS coverage criteria. According to Table 2, the number of covered 1-FCPS- and 2-FCPS-TRs are 277.3K and 3,620.7K, respectively. Thus, 2.22 (277.3K / 125.0K) 1-FCPS-TRs exist per each 1-FS-TR, and 1.91 (3,620.7K / 1,896.1K) 2-FCPS-TRs exist per each 2-FS-TR on average. It means that 2.22 and 1.91 feature-call-paths exist from the innermost language features to nodes or branches in each 1-FS-TR and 2-FS-TR, respectively. For a more detailed information, we also draw a histogram of the number of covered 1-FCPS-TRs (or 2-FCPS-TRs) per each covered 1-FS-TR (or 2-FS-TR) in Figures 9 (c) and (d). The largest number of covered 1-FCPS-TRs per 1-FS-TR is 70 for a node in the `Array.prototype.splice` built-in method. It is a powerful built-in API feature that changes the contents of an array by

[11] We anonymized links of bug reports for double-blinded reviewing.
[12] We anonymized links of bug reports for double-blinded reviewing.



(a) 1-FS-TRs vs base TR (b) 2-FS-TRs vs 1-FS-TR (c) 1-FCPS-TRs vs 1-FS-TR (d) 2-FCPS-TRs vs 2-FS-TR

Fig. 9. The histogram of numbers of k-FS or k-FCPS TRs per less sensitive k-FS or k-FCPS TR



(a) Covered TRs by synthesized tests and Test262 (b) Covered TRs over time

Fig. 10. Covered k-FS-TRs and k-FCPS-TRs for synthesized tests via JEST_{fs} and Test262

removing or replacing existing elements and/or adding new elements in place. Thus, its semantics is quite complex and uses diverse helper functions, and the number of possible feature-call-paths in this featuere is much larger than the others. The largest number of 2-FCPS-TRs per 2-FS-TR is 53 for a node whose innermost enclosing feature is a syntactic feature for **`yield`** expressions because it touches various helper functions for asynchronous behaviors. Because of the increased number of TRs, the number of synthesized tests also increased 1.34x (9,092 / 6,766) from 1-FS to 1-FCPS coverage criteria and 1.26x (122,589 / 97,423) from 2-FS to 2-FCPS coverage criteria. In addition, the number of detected unique bugs also increased when using k-FCPS coverage criteria than k-FS coverage criteria. The conformance tests synthesized with 1-FCPS and 2-FCPS coverage criteria detected 4 (87 - 83) and 9 (111 - 102) more conformance bugs than 1-FS and 2-FS coverage criteria, respectively. Now, we introduce a conformance bug example that show the effectiveness of k-FCPS coverage criteria compared to the k-FS coverage criteria.

`String.prototype.normalize`. The `String.prototype.normalize` built-in API normalizes a given string into the normalization form named by a given argument. For example, `"abc".normalize("NFC")` produces the NFC normalization form of `"abc"`. If an invalid name, such as an empty string `""`, is given as the argument, it should throw a **RangeError** exception. However, the following program noramlly terminates in GraalJS:

```
String.prototype.normalize.call(0, ""); // Expected: RangeError
```

As we discussed in Section 1, k-FS coverage criteria even with a high k value cannot detect this bug, while 1-FCPS and 2-FCPS coverage criteria can.

5.4 Comparison with Test262

We compare the coverage of automatically synthesized conformance tests with that of Test262, the official JavaScript conformance test suite. As described in Section 4, the baseline tool JEST relies on the mechanized specification extracted by ESMeta. Thus, we filter out conformance tests in Test262 that utilize language features not supported in the extracted mechanized specification. We use the conventional methodology in the literature [19, 39, 43] to remove inapplicable tests in Test262. Then, we measured the coverage of 23,910 applicable Test262 conformance tests with five k-FS and k-FCPS node-or-branch coverage criteria.

Figure 10 shows (~~b) venn~~ a) Venn diagrams of the numbers of covered k-FS-TRs and k-FCPS-TRs for the synthesized conformance tests (syn-test) via JEST_{fs} and applicable conformance tests in Test262 (test262) and (b) their changes over time. Without any feature-sensitive coverages, the coverage of synthesized tests is less than that of Test262, and only 5.2% (0.9K) 0-FS-TRs are newly covered by the synthesized tests. On the other hand, the numbers of k-FS-TRs covered by only synthesized tests increase when using higher k: 28.7K (19.0%) for 1-FS-TRs and 1.65M (80.8%) for 2-FS-TRs. In addition, the number of 1-FCPS-TRs (or 2-FCPS-TRs) covered by only synthesized tests is higher than the number of 1-FS-TRs (or 2-FS-TRs) covered by only synthesized tests: 80.0K (21.6%) for 1-FCPS-TRs and 3.26M (82.0%) for 2-FCPS-TRs. Figure 10(b) also shows that the coverage of Test262 is better than that of synthesized tests without any feature-sensitive coverages, but the coverage of synthesized tests outperforms that of Test262 with 2-FS-TRs and 2-FCPS-TRs. We believe that this is why JEST_{fs} successfully detected diverse new bugs in existing JavaScript implementations, even though they have been heavily tested using Test262 and various fuzzing techniques.

6 RELATED WORK

Coverage Criteria in Software Testing. Coverage criteria in software testing are essential in measuring the quality of test inputs. The most common coverage criteria are structural coverages in a given program's control-flow graph (CFG) [4, 13] also utilizing data-flow information [24] and type information [6]. On the other hand, model-based coverage [53] criteria consider specialized abstract behavior models and define the test requirements in the model. Such models include state transitions [5], autonomous driving systems [28], deep neural networks (DLLs) [33, 36, 46, 51, 57]. However, there are no specialized coverage criteria for programming language tools, such as compilers, interpreters, and transpilers. In this paper, we first presented feature-sensitive coverages as general extensions of graph coverages for programming language tools to discriminate test requirements based on enclosing language features or feature-call-paths.

Mechanized Specification. Researchers have presented mechanized specifications to formally describe the semantics of diverse programming languages, such as OCaml [38], C [8], C++[47], Java [10], and POSIX shell [21]. At the same time, general metalanguages or frameworks for mechanized language specifications have also emerge as. For example, Sewell et al. [50] presented Ott as a tool that compiles language semantics into proof assistant code for Coq, HOL, and Isabelle/HOL and supports a metalanguage used in defining language semantics as inference rules. The $\mathbb{K}$ framework [48] proposed a formalism for writing operational semantics and provides a derivation of verifiers directly from the semantics. Bodin et al. [9] developed a skeletal semantics framework in Coq for creating big-step semantics by focusing on the structure of the semantics.

For JavaScript, diverse mechanized specifications have been presented based on ECMA-262 [16]. KJS [39] utilizes the $\mathbb{K}$ framework, and Fragoso Santos et al. [19] presented a metalanguage, JSIL, for ES5.1. Researchers have used such mechanized specification in diverse fields: verification [19], symbolic execution [20], abstract interpretation-based static analysis [25, 26, 29, 44, 49], and double

debugger [11]. However, most JavaScript mechanized specifications focused on only ES5.1, released in 2011, and required manual description of the semantics. On the other hand, ESMeta supports a metalanguage IR_{ES} for the latest version of ECMA-262 and the automatic extraction of mechanized specification used in conformance test synthesis [42], specification type analysis [41], and static analyzer derivation [40]. Hence, we implemented JEST_{fs} based on ESMeta to synthesize conformance tests from the latest specification (ES13, 2022) with feature-sensitive coverages.

Conformance Testing for JavaScript. Diverse host environments support JavaScript engines for server-side programming, cross-platform desktop or mobile applications, PDF files, and even embedded systems. In addition, JavaScript transpilers become essential tools in the deployment process of JavaScript applications. Therefore, ensuring the conformance of engines and transpilers according to the language specification is crucial to consistent execution environments for JavaScript. The current solution is to maintain conformance tests by hand, and engine and transpiler developers commonly utilize Test262 [18], the official JavaScript conformance test suite. Researchers have focused on testing JavaScript engines to detect bugs using generation-based fuzzing [15, 22, 23, 45] and mutation-based fuzzing [22, 45, 54, 55]. In addition, they often utilize deep learning [30, 60] to generate JavaScript programs in advance and differential testing [7] to check the correctness of execution results. However, most existing techniques focused on detecting crashing bugs or security vulnerabilities rather than conformance bugs. While COMFORT [60] targets conformance bugs, it heavily relies on the results of differential testing instead of the language semantics described in ECMA-262. On the other hand, JEST [42] is the first tool that automatically synthesizes JavaScript conformance tests according to the language semantics described in the language specification. We implemented JEST_{fs} by augmenting it with k-FS and k-FCPS coverages and outperformed the ability of conformance bug detection of the JEST.

7 CONCLUSION

Conformance testing using graph coverage has been one of the most promising approaches to support correct and consistent implementations of programming language semantics. However, because language implementations often utilize different execution paths even for the same functionalities, traditional graph coverage does not produce high-quality conformance tests. In this paper, we present novel coverage criteria especially designed for language implementations: *feature-sensitive (FS) coverage* and *feature-call-path-sensitive (FCPS) coverage* by refining conventional test requirements using enclosing language features and call paths. Our experiments show that the new coverage criteria outperform the traditional coverage criteria in the context of conformance bug detection in real-world JavaScript implementations.

REFERENCES

[1] 2022. *Babel: An open-source JavaScript compiler mainly used to convert ES6+ code into a backwards compatible version of JavaScript.* https://babeljs.io/

[2] 2022. *ESMeta: An ECMAScript specification metalanguage used for automatically generating language-based tools.* https://github.com/es-meta/esmeta

[3] 2022. *GraalJS, An implementation of the JavaScript built on the GraalVM by Oracle Labs.* https://github.com/graalvm/graaljs

[4] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press.

[5] Cyrille Artho, Quentin Gros, Guillaume Rousset, Kazuaki Banzai, Lei Ma, Takashi Kitamura, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. 2017. Model-Based API Testing of Apache ZooKeeper. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).* https://doi.org/10.1109/ICST.2017.33

[6] Sora Bae, Joonyoung Park, and Sukyoung Ryu. 2017. Partition-Based Coverage Metrics and Type-Guided Search in Concolic Testing for JavaScript Applications. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE).* https://doi.org/10.1109/FormaliSE.2017.10

[7] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3548606.3560624

[8] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[9] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal Semantics and Their Interpretations. In *Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3290357

[10] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676982

[11] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference (WWW)*. https://doi.org/10.1145/3184558.3185969

[12] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00127

[13] John Joseph Chilenski and Steven P Miller. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 9, 5 (1994), 193–200.

[14] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3213846.3213848

[15] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2021.24224

[16] ECMA International. 2022. *ECMA-262, 13th edition, ECMAScript ®2022 Language Specification.* https://262.ecma-international.org/13.0/

[17] ECMA International. 2022. *ECMA-262, 13th edition, ECMAScript ®2022 Language Specification: 13.15.5 Destructuring Assignment.* https://262.ecma-international.org/13.0/#sec-destructuring-assignment

[18] ECMA International. 2022. *Test262: A conformanece test suite for the latest draft of ECMAScript.* https://github.com/tc39/test262

[19] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienundefined, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1–33. https://doi.org/10.1145/3158138

[20] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1–31. https://doi.org/10.1145/3290379

[21] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3371111

[22] Han, HyungSeok and Oh, DongHyeon and Cha, Sang Kil. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2019.23263

[23] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-augmented fuzzing for JavaScript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2229–2242. https://doi.org/10.1145/3460120.3484823

[24] PM Herman. 1976. A Data Flow Analysis Approach to Program Testing. *Australian Computer Journal* 8, 3 (1976), 92–96.

[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*. https://doi.org/10.1007/978-3-642-03237-0_17

[26] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2635868.2635904

[27] Adam Khayam, Louis Noizet, and Alan Schmitt. 2022. A Faithful Description of ECMAScript Algorithms. In *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming (PPDP)*. https://doi.org/10.1145/3551357.3551381

[28] Thomas Laurent, Stefan Klikovits, Paolo Arcaini, Fuyuki Ishikawa, and Anthony Ventresque. 2022. Parameter Coverage for Testing of Autonomous Driving Systems Under Uncertainty. *ACM Trans. Softw. Eng. Methodol.* (jul 2022). https://doi.org/10.1145/3550270

[29] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of 19th International Workshop on Foundations of Object-Oriented Languages (FOOL)*.

[30] Lee, Suyoung and Han, HyungSeok and Cha, Sang Kil and Son, Sooel. 2020. Montage: A Neural Network Language {Model-Guided}{JavaScript} Engine Fuzzer. In *Proceedings of 29th USENIX Security Symposium*. 2613–2630.

[31] Daniel Lehmann and Michael Pradel. 2018. Feedback-Directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3236024.3236037

[32] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*. https://doi.org/10.1109/ICSTW.2009.30

[33] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/3238147.3238202

[34] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[35] Michal Zalewski. 2007. *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/

[36] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of International Conference on Machine Learning (ICML)*.

[37] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: Differential Testing for Embedded Compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. https://doi.org/10.1145/2998392.2998397

[38] Scott Owens. 2008. A Sound Semantics for OCaml Light. In *European Symposium on Programming (ESOP)*. https://dl.acm.org/doi/10.1007/978-3-540-78739-6_1

[39] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://dl.acm.org/doi/10.1145/2737924.2737991

[40] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications Using Meta-Level Static Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3540250.3549097

[41] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE51524.2021.9678781

[42] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 13–24. https://doi.org/10.1109/ICSE43902.2021.00015

[43] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

[44] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. https://doi.org/10.1109/ICSE-C.2017.4

[45] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[46] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132785

[47] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103656.2103718

[48] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[49] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic Determinacy Analysis. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2499370.2462168

[50] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, et al. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of functional programming* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

[51] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ASE)*. https://doi.org/10.1145/3238147.3238172

[52] Haoxin Tu, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang. 2022. Detecting C++ Compiler Front-End Bugs via Grammar Mutation and Differential Testing. *IEEE Transactions on Reliability* (2022), 1–15. https://doi.org/10.1109/TR.2022.3171220

[53] Mark Utting and Bruno Legeard. 2010. *Practical Model-based Testing: A Tools Approach*. Elsevier.

[54] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security – ESORICS 2016*. https://doi.org/10.1007/978-3-319-45744-4_29

[55] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00081

[56] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. 1997. A Study of Effective Regression Testing in Practice. In *Proceedings The Eighth International Symposium on Software Reliability Engineering (ISSRE)*. 264–274. https://doi.org/10.1109/ISSRE.1997.630875

[57] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3293882.3330579

[58] XMLTestSuite. 2011. *Extensible Markup Language (XML) Conformance Test Suites*. https://www.w3.org/XML/Test/

[59] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993532

[60] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454054

[61] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritisation: A Survey. *Software Testing, Verification, and Reliability* 22, 2 (March 2012), 67–120. https://doi.org/10.1002/stvr.430