Selective Feature-Sensitive Coverage for Conformance Testing of Programming Languages

KANGUK LEE, KAIST, South Korea
SEUNGHWAN KIM, Korea University, South Korea
JIHYEOK PARK*, Korea University, South Korea
SUKYOUNG RYU*, KAIST, South Korea

The *conformance testing* of programming language implementations is crucial to support correct and consistent execution environments. Researchers often use coverage information in the *mechanized language specification* to generate or check the quality of conformance tests. Since specifications use inductive definitions to describe the semantics of language features, traditional graph coverage criteria for software can still be applied. 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, degrading conformance testing quality. Similarly, it may not distinguish test requirements of different parts of the same language feature if their semantics use the same functions.

We introduce *feature-sensitive (FS) coverage* as a novel coverage criterion for generating high-quality conformance tests for language implementations. The core idea is to enhance traditional graph coverage by incorporating the innermost enclosing language features. To further improve the quality of conformance tests, we extend this approach to *feature-call-path-sensitive (FCPS)* coverage and its k-limiting variant. To assess the effectiveness of the new coverage criteria, we apply them to the mechanized JavaScript specification for ES13 and extend JEST, a state-of-the-art JavaScript conformance test synthesizer. Using five coverage criteria, our tool synthesizes 237,981 conformance tests for ES13 in 50 hours. The tool detected 157 conformance bugs (45 in engines and 112 in transpilers), 139 confirmed by the developers, and 136 newly discovered bugs. However, a higher k value may lead to an excessive number of tests. To resolve this issue, we present a *selective* FS/FCPS that utilizes the *key feature stacks* for the target language implementation. The selective approach for transpiler conformance testing with ECMA-262 for ES15 successfully reduces the number of synthesized tests by 70.7% on average. Still, it detects 53 bugs compared to 57 bugs detected by the non-selective approach.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Semantics**; *General programming languages*; *Dynamic analysis*.

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

ACM Reference Format:
Kanguk Lee, Seunghwan Kim, Jihyeok Park, and Sukyoung Ryu. 2026. Selective Feature-Sensitive Coverage for Conformance Testing of Programming Languages. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (September 2026), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding author

Authors' Contact Information: Kanguk Lee, KAIST, School of Computing, Daejeon, South Korea, p51lee@kaist.ac.kr; Seunghwan Kim, Korea University, Department of Computer Science and Engineering, Seoul, South Korea, tmdghks12@korea.ac.kr; Jihyeok Park, Korea University, Department of Computer Science and Engineering, Seoul, South Korea, jihyeok_park@korea.ac.kr; Sukyoung Ryu, KAIST, School of Computing, Daejeon, South Korea, sryu.cs@kaist.ac.kr.

Manuscript submitted to ACM

1 Introduction

The *conformance testing* of programming languages is essential for providing correct and consistent implementations. In particular, it becomes more critical when multiple implementations exist for a programming language. For example, JavaScript has multiple engines, including V8, SpiderMonkey, and JavaScriptCore. Python supports CPython as the reference interpreter and other interpreters as well, such as PyPy, Jython, and IronPython. In such cases, the language specification is the only source of truth for language semantics. The conformance tests are used to check whether the implementations conform to the semantics described in the language specification. For example, Test262 [13] is the official conformance test suite for JavaScript maintained by the Technical Committee 39 (TC39). It is widely used by JavaScript engine developers to check whether their implementations conform to the official JavaScript language specification, ECMA-262 [12].

The quality of conformance tests is often evaluated using their coverage for the *mechanized specification* of the target language. A mechanized specification [5, 7, 17, 35, 40, 46] is an executable and formal specification of the language semantics, which is usually written in a metalanguage or framework, such as $\mathbb{K}$ [47], Ott [49], and Skel [6]. We can measure the quality of a test suite by measuring its coverage for the mechanized specification using diverse *graph coverage* [2] criteria defined over the control-flow graph (CFG) of the specification. Higher coverage of the test suite denotes that it covers more test requirements (TRs) of the language semantics, which is a good indicator of the quality of the test suite. While we can measure the code coverage of a test suite for "actual language implementations", it leads to different coverage for different implementations. On the contrary, coverage for "language specifications" leads to uniform coverage because they are based on the same language semantics. In addition, the coverage for specifications can be used to reduce the number of tests in a test suite using test minimization techniques [55, 59].

1.1 Problem: Limitation of Coverage for Language Specification

However, traditional graph coverage criteria may not produce high-quality conformance tests for language specifications because of a modular definition of the semantics. Language semantics are often defined using numerous helper functions, which may themselves rely on other helper functions. While such a modular definition reduces the size of the mechanized specification and enhances its readability, it may degrade the quality of conformance testing because of the overlapped TRs in different or the same language features.

Overlapped TRs in Different Language Features. First, traditional graph coverage may not distinguish TRs of different language features when their semantics 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 [12]. Here, most of the semantics for the addition and subtraction operators are defined using the same **EvaluateStringOrNumericBinaryExpression** algorithm. If conformance tests for the addition operator already cover the TRs 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 though their semantics share the same parts in the specification. For example, V8 engine has specialized execution path for addition with a complex finite machine to handle both numeric and string addition, while it uses a generic execution path for subtraction. Therefore, we need to test possible edge cases for subtraction as well, although similar ones for addition are already tested.

Overlapped TRs in the Same Language Feature. Furthermore, it may not distinguish TRs 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 TRs 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.

1.2 Our Solution: Feature-Sensitive Coverage

To alleviate this problem, we introduce *feature-sensitive (FS) coverage*, an extended coverage criterion to generate high-quality conformance tests for programming language implementations. It is a general extension of graph coverage, refining TRs 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 TRs. FCPS coverage resolves the problem of sharing the same helper functions for the semantics of different parts in the same language feature. 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 conformance test synthesis task for a real-world programming language. We select JavaScript as the evaluation target because 1) it has the most up-to-date mechanized specification and 2) it has the official conformance test suite, Test262 [13]. We extend JEST [39], the state-of-the-art JavaScript conformance test synthesizer using coverage-guided mutational fuzzing, with various FS and FCPS coverage criteria. For the language specification ECMA-262 for ES13, our tool automatically synthesizes 237,981 conformance tests in 50 hours with five coverage criteria. We evaluated the synthesized conformance tests with eight mainstream JavaScript implementations (four engines and four transpilers) and discovered bugs in all of them. The tool detected 157 conformance bugs, 139 confirmed by the developers and 136 newly discovered bugs.

Our contributions are as follows:

- We introduce novel *feature-sensitive (FS) coverage* to discriminate TRs with the innermost enclosing language features to enhance the quality of conformance testing for programming language implementations.
- We extend it to *feature-call-path-sensitive (FCPS) coverage* to further improve the conformance testing quality by considering the call-paths from language features to TRs.
- We experimentally show that our new coverage criteria outperform the traditional coverage criteria in the context of conformance bug detection with eight mainstream JavaScript implementations (four engines and four transpilers) with the ECMA-262 for ES13. The tool uncovered 136 brand-new bugs.

This article extends a previously published conference paper [42]. Specifically, we update the status of conformance bugs detected by our tool listed in Table 1 and introduce *selective* FS/FCPS coverage criteria. For k-FS or k-FCPS coverage, a higher k value leads to an excessive number of tests during conformance test synthesis due to the exponential growth of TRs. The selective k-FS/FCPS coverage criteria address this issue by reducing the number of tests while maintaining their quality, achieved by identifying *key feature stacks* for the target language implementation. While any method

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

Fig. 1. Syntax of *AdditiveExpression* in ECMA-262 for ES13.

can be used to extract key feature stacks for the selective criteria, we propose a way for transpilers based on online learning of *triggering-features*. To evaluate effectiveness, we compare conformance testing results between selective and non-selective 2-FCPS coverage. Note that we use the mechanized specification for ES15 instead of ES13 in the extended experiments because the ESMeta now targets the latest specification for ES15.

In the remainder, we explain background and motivation for our work (§2). Then, we formally define our feature-sensitive coverage criteria and its variants (§3) and its selective approach (§4). Finally, we detail the implementation of a conformance test synthesizer using our new coverage criteria and selective approach (§5), evaluate it on real-world implementations (§6), discuss threats to validity (§7), compare with other related work (§8), and conclude (§9).

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 [15, 23, 36, 40] closely capture the abstract algorithms in ECMA-262 [12], 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 parts in different language features or even in the same language feature.

2.1 JavaScript Language Specification (ECMA-262)

This section first explains how ECMA-262 describes the JavaScript syntax and semantics using simple examples. Then, we define JavaScript language features used in the remainder of the paper.

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 a simple additive expression: `1 + 2` 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*.

[1]https://tc39.es/ecma262/2022/#prod-AdditiveExpression

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

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

Built-in methods

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

... •26

Fig. 2. Semantics of 1) the syntax of + and – operators (top) and 2) the `Number` built-in API (bottom) that shares the same normal algorithms (middle) in ECMA-262 for ES13.

Semantics. ECMA-262 defines the JavaScript semantics using abstract algorithms, and there are three kinds of abstract algorithms in Figure 2:

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

A *syntax-directed operation* (SDO) defines the semantics of each syntactic production alternative. It consists of 1) target alternative, 2) name, 3) optional parameters, and 4) body. The body is pseudo-code in English, detailing well-structured steps. For example, the two abstract algorithms in Figure 2 are SDOs for the second and third alternatives of the *AdditiveExpression* production about addition (+) and subtraction (-) operators, respectively. Both are named **Evaluation** without optional parameters, and their bodies invoke the **EvaluateStringOrNumericBinaryExpression** [2] algorithm. JavaScript abstract syntax trees (ASTs) are values, with *AdditiveExpression* and *MultiplicativeExpression* metavariables storing ASTs of the left- and right-hand sides. For instance, given `1 + 2`, the metavariables hold ASTs for the Number literals `1` and `2`. A *completion record* represents evaluation results for control flow, while abrupt completions represent

[2]https://tc39.es/ecma262/2022/#sec-evaluatestringornumericbinaryexpression

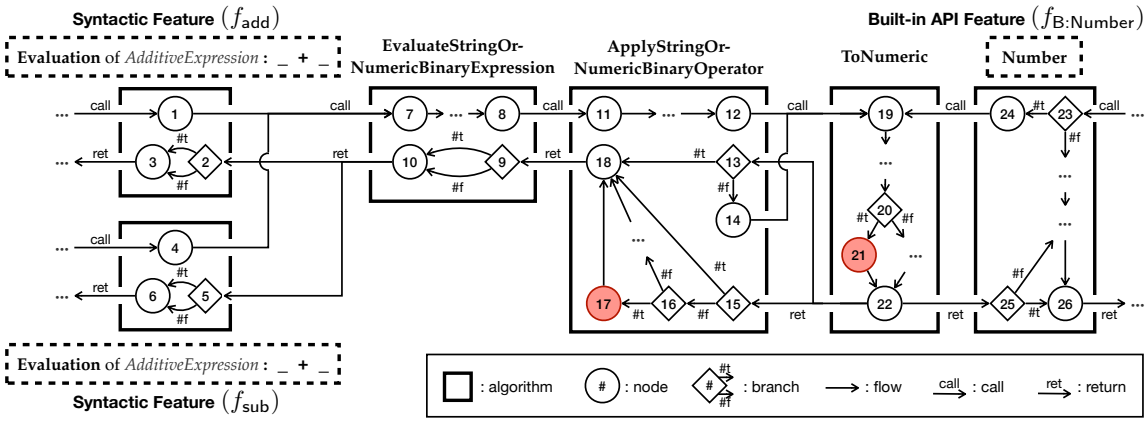


Fig. 3. Control-flow graph (CFG) of abstract algorithms in Figure 2.

exceptional flows. The "?" operator checks for abrupt completions, returning them immediately if found, or otherwise returning the value in the normal completion record. Thus, "?" implicitly branches and captures semantic edge cases.

A *normal algorithm* is the primary form of an abstract algorithm defined by its 1) name, 2) parameters, and 3) body. These opaque functions are provided by the JavaScript engine, not implemented in JavaScript. For example, both SDOs in Figure 2 use the same normal algorithm **EvaluateStringOrNumericBinaryExpression** and transitively use other ones, such as **ApplyStringOrNumericBinaryOperator**[3] and **ToNumeric**[4]. Thus, at least three normal algorithms are shared in the semantics of the + and - operators.

A *built-in method* defines the semantics of a JavaScript built-in API. For example, `Number.prototype.toString` or `Object.getPrototypeOf` are representative built-in APIs. They are opaque functions that are not directly implemented in JavaScript but are provided by the JavaScript engine. For example, the bottom of Figure 2 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.

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 **Evaluation** SDOs. The **Number** built-in method describes the semantics of the built-in `Number` API feature ($f_{B:Number}$).

2.2 Control Flow Graph (CFG) of ECMA-262

To define conformance test suite coverage using graph coverage criteria, a directed graph of the JavaScript mechanized specification is needed. The most common form is a *control-flow graph* (CFG), where nodes represent instruction sequences and edges represent control flows.

[3]https://tc39.es/ecma262/2022/#sec-applystringornumericbinaryoperator
[4]https://tc39.es/ecma262/2022/#sec-tonumeric
[5]https://tc39.es/ecma262/2022/#sec-number-constructor-number-value

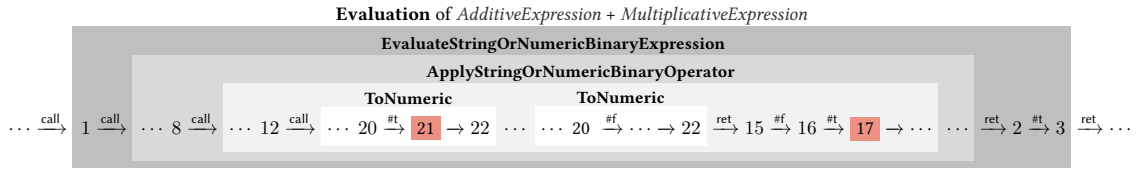


Fig. 4. The execution path of the test (JavaScript program) t_{add} = `2n + 1` according to the CFG in Figure 3.

For example, Figure 3 depicts a CFG of the abstract algorithms in Figure 2. Thick boxes indicate algorithms, circles (or diamonds) represent nodes (or branches), and arrows denote edges with optional annotations for branches (`#t` or `#f`) and function calls (`call`) and returns (`ret`). The labels inside nodes match the labels annotated in the algorithms in Figure 2. Let us apply coverage-guided fuzzing [33] 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 cover the red node 21 because it does not satisfy the condition in the branch 20 because the left- and right-hand sides of `1 + 2` are both Number values rather than BigInt values. However, if another program, `3n + 4n`, is generated during the fuzzing process, it covers the red node 21 because it satisfies the condition in the branch 20.

2.3 Motivation

Unfortunately, a node coverage criterion for CFGs of mechanized specifications cannot fully discriminate different parts in different language features or even in the same feature. We explain such cases with examples using the CFG in Figure 3.

Overlapped TRs 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 17 represents throwing **TypeError** exception. If the program pool contains a program `2n + 1`, it covers the red node 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 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`.

Overlapped TRs 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 12 and 14. Now, assume that the current program pool contains a program `2n + 1` again. Then, the red node 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 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.

3 Feature-Sensitive Coverage

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 semantics parts of the same language feature.

3.1 Notations

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

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

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

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 $\mathsf{first}(p)$ and $\mathsf{last}(p)$ denote the first and last nodes of the path p, respectively. A path p is a *test path* when it starts at an initial node and ends at a final node: $\mathsf{first}(p) \in \mathbb{N}_\iota \wedge \mathsf{last}(p) \in \mathbb{N}_f$. Then, $\mathsf{path}_\mathbb{G} : \mathbb{T} \rightarrow \mathbb{P}_\mathbb{G}$ is a mapping from a *test* $t \in \mathbb{T}$ to a test path in $\mathbb{G}$, and we call $\mathsf{path}_\mathbb{G}(t)$ the *execution path* of t.

Example. Consider the CFG $\mathbb{G}$ in Figure 3 and the following JS programs as a test set $T \subseteq \mathbb{T}$:

$$T = \{\cdots, \quad t_{\mathsf{add}} = \texttt{2n + 1}, \quad t_{\mathsf{sub}} = \texttt{2n - 1}, \quad \cdots\} \tag{1}$$

Figure 4 shows the execution path $\mathsf{path}_\mathbb{G}(t_{\mathsf{add}})$ of the test t_{add}, and $\mathsf{path}_\mathbb{G}(t_{\mathsf{sub}})$ is equal to this path except for nodes 1, 2, and 3 in the **Evaluation** SDO for addition replaced with nodes 4, 5, and 6 in the **Evaluation** SDO for subtraction. The following path p is a subpath of both $\mathsf{path}_\mathbb{G}(t_{\mathsf{add}})$ and $\mathsf{path}_\mathbb{G}(t_{\mathsf{sub}})$ and its length is $\|p\| = 3$:

$$p = 22 \xrightarrow{\mathsf{ret}} 15 \xrightarrow{\#\mathsf{f}} 16 \xrightarrow{\#\mathsf{t}} 17, \tag{2}$$

3.2 Graph Coverage

We formulate *graph coverage* criteria based on [2]:

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

- a set of *test requirements (TRs)* $\mathbb{R}_\mathbb{G}$
- a *cover relation* $\overset{\mathsf{cover}}{\sim} \subseteq \mathbb{P}_\mathbb{G} \times \mathbb{R}_\mathbb{G}$ 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 r if there exists a prefix path p of its execution path that covers it:

$$t \overset{\text{cover}}{\sim} r \iff \exists p \in \mathbb{P}_{\mathbb{G}}.\ (p \preceq \text{path}_{\mathbb{G}}(t) \wedge p \overset{\text{cover}}{\sim} r) \tag{3}$$

A test set $T \subseteq \mathbb{T}$ *satisfies* a criterion $C_{\mathbb{G}}$ when it covers all feasible TRs:

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

where a TR r is *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 $C_{\mathbb{G}}$ *subsumes* $C'_{\mathbb{G}}$ and use the notation: $C_{\mathbb{G}} \rhd C'_{\mathbb{G}}$. The subsumption relation between graph coverage criteria is a preorder.

Definition 3.2 (Node Coverage $C^{\text{node}}_{\mathbb{G}}$)**.** The node coverage criterion $C^{\text{node}}_{\mathbb{G}} = (\mathbb{N}, \overset{\text{cover}}{\sim})$ for $\mathbb{G}$ is defined with:

- 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:[6]

$$p \overset{\text{cover}}{\sim} n \iff \text{last}(p) = n$$

The *node coverage* criterion is the most common graph coverage criterion whose TRs 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 $C^{k\text{-path}}_{\mathbb{G}}$,

- 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 ($C^{0\text{-path}}_{\mathbb{G}} = C^{\text{node}}_{\mathbb{G}}$), and other graph coverage criteria match with k-limiting path coverages:

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

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 $C^{i\text{-path}}_{\mathbb{G}}$ is always a subset of that in $C^{j\text{-path}}_{\mathbb{G}}$, and $C^{j\text{-path}}_{\mathbb{G}}$ subsumes $C^{i\text{-path}}_{\mathbb{G}}$. 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, [2] have presented a *simple path coverage* criterion or a *prime path coverage* criterion. 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 [29] in practice with intra-procedural control-flow graphs.

[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.

Example. Consider the CFG $\mathbb{G}$ in Figure 3 and the test set T in (1). If we measure the 3-limiting path coverage $C_{\mathbb{G}}^{\text{3-path}}$ for T, both the node labeled 17 and the path p in (2) 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

To alleviate the problem introduced in §2.3, 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. A feature mapping feat can be constructed by looking up the language features in the semantics, $\text{feat}(n) = f$ if the node n is in the language feature f's semantic definition. Each semantic node is created under at most one such function, and the feature mapping *feat* is therefore single-valued and unambiguous by construction. For example, for the CFG $\mathbb{G}$ in Figure 3, we define the feature mapping feat as in Equation (4).

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

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

In other words, $p\,|_{\text{call}}$ keeps only call-sites not matched with return-sites in the path p; 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* $\text{ext}_{\mathbb{F}} : \mathbb{P}_{\mathbb{G}}\,|_{\text{call}} \to \mathbb{F} \uplus \{\bot\}$ as a partial mapping from call-site stacks to the innermost enclosing language features $\mathbb{F}$:

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

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)**.** For a given graph coverage criterion $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *feature-sensitive (FS) coverage* criterion $C_{\mathbb{G}}^{\text{FS}} = (\mathbb{R}_{\mathbb{G}}^{\text{FS}}, \overset{\text{cover}}{\sim})$ is defined with:

- the set of **feature-sensitive test requirements (FS-TRs)** $\mathbb{R}_{\mathbb{G}}^{\text{FS}}$ is a set of TRs optionally with language features:

$$\mathbb{R}_{\mathbb{G}}^{\text{FS}} = \mathbb{R}_{\mathbb{G}} \times (\mathbb{F} \uplus \{\bot\})$$

- a path p **covers** a FS-TR (r, f) when p covers the 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 3, assume that the feature mapping is as follows:

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

Consider two tests t_{add} and t_{sub} in the test set T (1), and the following two prefix paths p_{add} and p_{sub}, whose last nodes are labeled 17, of their execution paths:

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

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

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 $C_{\mathbb{G}}^{\text{FS[node]}}$, 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, 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 $\text{ext}_{\mathbb{F}}^{k} : \mathbb{P}_{\mathbb{G}}\,|_{\text{call}} \rightarrow \mathbb{F}^{\leq k}$ collects *feature stacks* with at most k enclosing language features:

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

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

- the set of **k-feature-sensitive test requirements (k-FS-TRs)** $\mathbb{R}_{\mathbb{G}}^{k\text{-FS}}$ is a set of TRs with at most k 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 TR r and $\overline{f}$ is the k-most enclosing language features of p:

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

Example. Consider a JavaScript program `[] - (2n+1)` as a test t with the graph in Figure 3. 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, $\text{ext}_{\mathbb{F}}^{2}(p) = [f_{\text{sub}}, f_{\text{add}}]$ 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. In particular, 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*[7] 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 engine and the Babel transpiler contain conformance bugs[8] and crashing bugs[9], respectively, that are reproducible only with a combination of a destructuring pattern and a `for-in/of` statement.

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

As explained in §2.3, 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* $\text{ext}_{\mathbb{F}_{\text{cp}}} : \mathbb{P}_{\mathbb{G}}|_{\text{call}} \to (\mathbb{F}_{\text{cp}} \uplus \{\bot\})$:

$$\text{ext}_{\mathbb{F}_{\text{cp}}}([n_1, \cdots, n_m]) = \begin{cases} \bot & \text{if } m = 0 \\ (f, [n_m]) & \text{if } \text{feat}(n_m) = f \\ f_{\text{cp}} & \text{if } f_{\text{cp}} = \bot \\ (f, [n'_0, \cdots, n'_i]) & \text{if } f_{\text{cp}} = (f, [n'_0, \cdots, n'_{m'}]) \wedge \exists i.\ n'_i = n_m \\ (f, \overline{n} \cdot n_m) & \text{if } f_{\text{cp}} = (f, \overline{n}) \end{cases}$$

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. $n'_i = n_m$). In our evaluation environment, this cycle removal reduced the number of FCPS TRs by approximately 6%.

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

- the set of **feature-call-path-sensitive test requirements (FCPS-TRs)** $\mathbb{R}^{\text{FCPS}}_{\mathbb{G}}$ is a set of TRs with optional feature-call-paths:
$$\mathbb{R}^{\text{FCPS}}_{\mathbb{G}} = \mathbb{R}_{\mathbb{G}} \times (\mathbb{F}_{\text{cp}} \uplus \{\bot\})$$
- a path p **covers** a FCPS-TR (r, f_{cp}) when p covers the TR r and f_{cp} is the feature-call-path extracted from p:
$$p \overset{\text{cover}}{\sim} (r, f_{\text{cp}}) \iff p \overset{\text{cover}}{\sim} r \wedge \text{ext}_{\mathbb{F}_{\text{cp}}}(p|_{\text{call}}) = f_{\text{cp}}$$

Example. We show two examples for FCPS node coverage criteria. First, consider the two JS programs, t_0 = `2n + 1` and t_1 = `1 + 2n`, as tests with the CFG in Figure 3. If we use FS node coverage criterion $C^{\text{FS[node]}}_{\mathbb{G}}$, 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

[7] https://tc39.es/ecma262/2022/#sec-destructuring-assignment
[8] https://github.com/oracle/graaljs/issues/656
[9] https://github.com/babel/babel/issues/15100

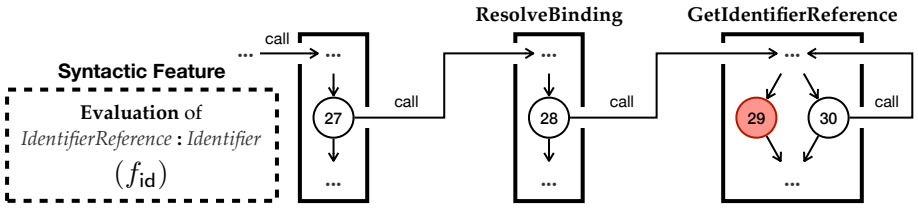


Fig. 5. CFG of abstract algorithms for *IdentifierReference*.

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 5. It depicts an excerpt from the CFG of abstract algorithms in ECMA-262 for 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 $\mathbb{F}_{\text{cp}}$ during the extraction algorithm $\text{ext}_{\mathbb{F}_{\text{cp}}}$. 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: $(f_{\text{id}}, [27, 28])$, $(f_{\text{id}}, [27, 28, 30])$, $(f_{\text{id}}, [27, 28, 30, 30])$, and so on. 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 $\text{ext}_{\mathbb{F}_{\text{cp}}}$ into $\text{ext}^k_{\mathbb{F}_{\text{cp}}} : \mathbb{P}_{\mathbb{G}}|_{\text{call}} \to \mathbb{F}^{\leq k}_{\text{cp}}$:

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

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

Definition 3.7 (k-Limiting Feature-Call-Path-Sensitive (k-FCPS) Coverage)**.** 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 with

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

$$\mathbb{R}_{\mathbb{G}}^{k\text{-FCPS}} = \mathbb{R}_{\mathbb{G}} \times \mathbb{F}^{\leq k}_{\text{cp}}$$

- a path p **covers** a k-FCPS-TR $(r, \overline{f_{\text{cp}}})$ when p covers the 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 6 illustrates the relations using edges annotated with equations in Theorem 3.9.

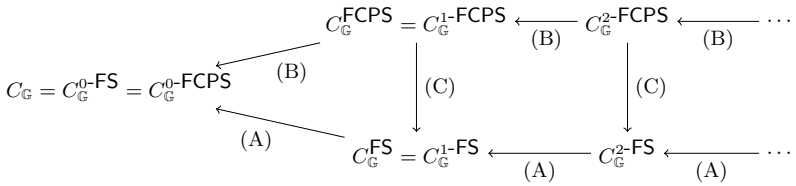


Fig. 6. Subsumption relations between k-FS/FCPS coverage.

Lemma 3.8. *$C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{cover}{\sim})$ and $C'_{\mathbb{G}} = (\mathbb{R}'_{\mathbb{G}}, \overset{cover'}{\sim})$ are given graph coverage criteria. Then, $C_{\mathbb{G}}$ subsumes $C'_{\mathbb{G}}$ (i.e., $C_{\mathbb{G}} \rhd C'_{\mathbb{G}}$) if there exists a feasible TR $r \in \mathbb{R}_{\mathbb{G}}$ that satisfies the following condition for each feasible TR $r' \in \mathbb{R}'_{\mathbb{G}}$:*

$$\forall t \in \mathbb{T}.\ t \overset{cover}{\sim} r \Rightarrow t \overset{cover'}{\sim} r'. \tag{5}$$

Proof. Assume $T \vdash C_{\mathbb{G}}$. For a given feasible TR $r' \in \mathbb{R}'_{\mathbb{G}}$, let $r \in \mathbb{R}_{\mathbb{G}}$ be the feasible TR satisfying (5). Then, there exists a test $t \in T$ such that $t \overset{cover}{\sim} r$ because r is feasible and $T \vdash C_{\mathbb{G}}$. Finally, $t \overset{cover}{\sim} r'$. □

Theorem 3.9 (Subsumption Relation)**.** *For a given integer $k > 0$, the following three subsumption relations ($\rhd$) satisfy:*

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

Proof. We prove (A) using Lemma 3.8, and omit the other cases because their proofs are similar. Let $k > 0$. For a given feasible $(k-1)$-FS-TR $(r, \overline{f})$, there exists a test $t \in \mathbb{T}$ such that $t \overset{cover}{\sim} (r, \overline{f})$ because $(r, \overline{f})$ is feasible. There exists a prefix path p of $\mathrm{path}_{\mathbb{G}}(t)$ such that $p \overset{cover}{\sim} (r, \overline{f})$ ($\because$ (3)). Then, a k-FS-TR $(r, \mathrm{ext}^{k}_{\mathbb{F}}(p|_{\mathsf{call}}))$ satisfies the condition (5) because of the inductive definition of $\mathrm{ext}^{k}_{\mathbb{F}}$ in (3.3). Finally, k-FS coverage criteria subsume $(k-1)$-FS coverage criteria. □

4 Selective Feature-Sensitive Coverage

While k-FS and k-FCPS coverage criteria can improve the quality of conformance testing, a high k value leads to an exponential growth in test requirements and generates an infeasible number of test cases. We observe that not all feature stacks are equally significant for a given language implementation. For example, Babel does not transpile **`let`** declarations if the target environment supports ES6+ features, and we do not need to test the transpilation of **`let`** declarations in this case. It means that we can reduce the number of test requirements and test cases while preserving the effectiveness of the coverage criteria by focusing on the key feature stacks. To achieve this, we introduce *selective* variants of the feature-sensitive coverage criteria.

4.1 Selective Feature-Sensitive (sFS) Coverage

We first introduce *k-selective feature-sensitive (k-sFS) coverage criteria*, defined with a given set of *key feature stacks* $\mathcal{F} \subseteq \mathbb{F}^*$ that is specified externally (e.g., by users) for a target language implementation. Note that $\mathcal{F}$ should have at least the empty feature stack ($\epsilon \in \mathcal{F}$) to represent feature-insensitive test requirements. The *feature trimmer*

$\mathsf{trim}^k_{\mathcal{F}} : \mathbb{F}^{\leq k} \to \mathcal{F}^{\leq k}$ is a function that recursively trims outermost features to key feature stacks:

$$\mathsf{trim}^k_{\mathcal{F}}([f_1, \cdots, f_m]) = \begin{cases} \epsilon & \text{if } \ k = 0 \vee m = 0 \\ [f_1, \cdots, f_m] & \text{if } \ [f_1, \cdots, f_m] \in \mathcal{F}^{\leq k} \\ \mathsf{trim}^k_{\mathcal{F}}([f_2, \cdots, f_m]) & \text{otherwise} \end{cases}$$

Note that $\mathcal{F}^{\leq k}$ is the set of key feature stacks with a bounded length of k:

$$\mathcal{F}^{\leq k} = \left\{ \overline{f} \;\middle|\; \overline{f} \in \mathcal{F} \wedge \left|\overline{f}\right| \leq k \right\}.$$

We define k-sFS coverage using the feature trimmer $\mathsf{trim}^k_{\mathcal{F}}$.

Definition 4.1 (k-Selective Feature-Sensitive (k-sFS) Coverage)**.** For a given $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *$k$-selective feature-sensitive (k-sFS) coverage* criterion $C^{k\text{-sFS}}_{\mathbb{G}} = (\mathbb{R}^{k\text{-sFS}}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$ is defined with:

- the set of **k-selective-feature-sensitive test requirements (k-sFS-TRs)** $\mathbb{R}^{k\text{-sFS}}_{\mathbb{G}}$ is a set of TRs with the key feature stacks bounded by k:
$$\mathbb{R}^{k\text{-sFS}}_{\mathbb{G}} = \mathbb{R}_{\mathbb{G}} \times \mathcal{F}^{\leq k}$$
- a path p **covers** a k-sFS-TR $(r, \overline{f})$ when p covers the TR r and $\overline{f}$ is the trimmed feature stack extracted from p bounded by k:
$$p \overset{\text{cover}}{\sim} (r, \overline{f}) \iff p \overset{\text{cover}}{\sim} r \ \wedge \mathsf{trim}^k_{\mathcal{F}} \circ \mathsf{ext}^k_{\mathbb{F}}(p|_{\mathsf{call}}) = \overline{f}$$

Similarly, we define *k-selective feature-call-path-sensitive (k-sFCPS) coverage criteria* using the feature trimmer $\mathsf{trim}^k_{\mathcal{F}}$.

Definition 4.2 (k-Selective-Feature-Call-Path-Sensitive (k-sFCPS) Coverage)**.** For a given $C_{\mathbb{G}} = (\mathbb{R}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$, the *$k$-selective feature-call-path-sensitive (k-sFCPS) coverage* criterion $C^{k\text{-sFCPS}}_{\mathbb{G}} = (\mathbb{R}^{k\text{-sFCPS}}_{\mathbb{G}}, \overset{\text{cover}}{\sim})$ is defined with:

- the set of **k-selective-feature-call-path-sensitive test requirements (k-sFCPS-TRs)** $\mathbb{R}^{k\text{-sFCPS}}_{\mathbb{G}}$ is a set of TRs with the key extended feature-call-paths bounded by k:
$$\mathbb{R}^{k\text{-sFCPS}}_{\mathbb{G}} = \mathbb{R}_{\mathbb{G}} \times \mathcal{F}^{\leq k}_{\mathsf{cp}}$$
where $\mathcal{F}^{\leq k}_{\mathsf{cp}} = \mathcal{F}^{\leq k} \times \mathbb{P}_{\mathbb{G}}|_{\mathsf{call}}$
- a path p **covers** a k-sFCPS-TR $(r, \overline{f_{\mathsf{cp}}})$ when p covers the TR r and $\overline{f_{\mathsf{cp}}}$ is the trimmed feature-call-path extracted from p bounded by k:
$$p \overset{\text{cover}}{\sim} (r, \overline{f_{\mathsf{cp}}}) \iff p \overset{\text{cover}}{\sim} r \ \wedge \ \mathsf{ext}^k_{\mathbb{F}_{\mathsf{cp}}}(p|_{\mathsf{call}}) = (\overline{f}, \overline{n}) \ \wedge \ (\mathsf{trim}^k_{\mathcal{F}}(\overline{f}), \overline{n}) = \overline{f_{\mathsf{cp}}}$$

Example. Consider the following JavaScript programs as tests with the graph in Figure 3:

t_1 `= [] + (2n + 1)` t_2 `= [] - (2n + 1)` t_3 `= [] + (2n - 1)` t_4 `= [] - (2n - 1)`

and the prefix paths p_i of t_i, whose last nodes are all 17. Then, their enclosing feature stacks with a bound of 2 are:

$$\mathsf{ext}^2_{\mathbb{F}}(p_1) = [f_{\mathsf{add}}, f_{\mathsf{add}}] \qquad \mathsf{ext}^2_{\mathbb{F}}(p_2) = [f_{\mathsf{sub}}, f_{\mathsf{add}}] \qquad \mathsf{ext}^2_{\mathbb{F}}(p_3) = [f_{\mathsf{add}}, f_{\mathsf{sub}}] \qquad \mathsf{ext}^2_{\mathbb{F}}(p_4) = [f_{\mathsf{sub}}, f_{\mathsf{sub}}]$$

Now, suppose we use 2-sFS coverage with the following set of key feature stacks to focus on 1) addition enclosed by subtraction ($[f_{\mathsf{sub}}, f_{\mathsf{add}}]$ and 2) subtraction ($[f_{\mathsf{sub}}]$):

$$\mathcal{F} = \{[f_{\mathsf{sub}}, f_{\mathsf{add}}], [f_{\mathsf{sub}}]\}.$$

Then, the enclosing features are trimmed via $\text{trim}^2_{\mathcal{F}}$ as follows:

$$\text{trim}^2_{\mathcal{F}}([f_{\text{add}}, f_{\text{add}}]) = \epsilon \quad \text{trim}^2_{\mathcal{F}}([f_{\text{sub}}, f_{\text{add}}]) = [f_{\text{sub}}, f_{\text{add}}] \quad \text{trim}^2_{\mathcal{F}}([f_{\text{add}}, f_{\text{sub}}]) = [f_{\text{sub}}] \quad \text{trim}^2_{\mathcal{F}}([f_{\text{sub}}, f_{\text{sub}}]) = [f_{\text{sub}}]$$

and the tests cover the following 2-sFS-TRs:

$$t_1 \overset{\text{cover}}{\sim} (17, \epsilon) \qquad t_2 \overset{\text{cover}}{\sim} (17, [f_{\text{sub}}, f_{\text{add}}]) \qquad t_3 \overset{\text{cover}}{\sim} (17, [f_{\text{sub}}]) \qquad t_4 \overset{\text{cover}}{\sim} (17, [f_{\text{sub}}])$$

Theorem 4.3 (Subsumption Relation)**.** *For a given integer $k > 0$ and a set of key feature stacks $\mathcal{F}^{\leq k}$, the following five subsumption relations ($\triangleright$) satisfy:*

$$\textit{(A)}\ C^{k\text{-sFS}}_{\mathbb{G}} \triangleright C^{(k-1)\text{-sFS}}_{\mathbb{G}} \qquad \textit{(B)}\ C^{k\text{-sFCPS}}_{\mathbb{G}} \triangleright C^{(k-1)\text{-sFCPS}}_{\mathbb{G}} \qquad \textit{(C)}\ C^{k\text{-FS}}_{\mathbb{G}} \triangleright C^{k\text{-sFS}}_{\mathbb{G}}$$
$$\textit{(D)}\ C^{k\text{-FCPS}}_{\mathbb{G}} \triangleright C^{k\text{-sFCPS}}_{\mathbb{G}} \qquad \textit{(E)}\ C^{k\text{-sFCPS}}_{\mathbb{G}} \triangleright C^{k\text{-sFS}}_{\mathbb{G}}$$

Proof. These cases follow similarly from Theorem 3.9. □

4.2 Statistical Key Feature Stack Selection for Transpilers

As an example of key feature stack selection, we introduce a method to select key feature stacks for JS-to-JS *transpilers* using statistical information from the test generation process. JavaScript transpilers are only activated when certain features are present in the input program, and such features differ across transpilers. This observation motivates the key idea to 1) keep track of the *triggering-feature set* $F \subseteq \mathbb{F}$ during the test generation process and 2) determine the *key feature stacks* $\mathcal{F} \subseteq \mathbb{F}^*$ based on the triggering-features.

Collecting Triggering-Feature Set F. Let $T \subseteq \mathbb{T}$ be a finite set of generated tests, and $T_f \subseteq T$ be a subset of tests whose execution paths contain at least one node n corresponding to a feature $f \in \mathbb{F}$ (i.e., $\text{feat}(n) = f$). Then, the set T_f is further divided into two parts:

$$T^+_f = \left\{ t \in T_f \;\middle|\; t \text{ triggers the transpilation} \right\} \qquad T^-_f = \left\{ t \in T_f \;\middle|\; t \text{ does not trigger the transpilation} \right\}$$

If nearly all tests in T_f belong to T^+_f, then any test having f is likely to trigger the transpilation. Based on this intuition, we define the *triggering-score* $\psi(f)$ of each feature $f \in \mathbb{F}$:

Definition 4.4 (Triggering-Score ψ)**.** The *triggering-score* $\psi(f) \in [0, 1]$ for a feature $f \in \mathbb{F}$ is defined as the proportion of tests in T_f that trigger the transpilation process:

$$\psi(f) = \frac{\|T^+_f\|}{\|T_f\|}.$$

We use this triggering-score to collect triggering-feature set F with a predefined 1) upper threshold θ_u and 2) lower threshold θ_l. If $\psi(f)$ exceeds the θ_u, then f is added to F. If $\psi(f)$ falls below the θ_l, then f is removed from F. Since the set of generated tests T is continuously updated throughout the test generation, the target feature set F is also updated dynamically. This process ensures that F remains relevant to the transpilation process as the test generation progresses. Threshold values can be configured based on the specific needs of the transpiler implementation. In our evaluation, we use $\theta_u = 0.999$ to only keep features that almost always (i.e., 99.9% of the time) trigger the transpilation process. We use $\theta_l = 0.995$ slightly lower than θ_u to mitigate possible fluctuations in the triggering-feature set while the gathered data set is still small.

The values of θ_u and θ_l are selected based on an empirical comparison of three settings: (1) $\theta_u = 0.99$ and $\theta_l = 0.95$, (2) $\theta_u = 0.999$ and $\theta_l = 0.995$, and (3) $\theta_u = \theta_l = 1$. More relaxed thresholds will increase the number of selected features,

leading to more test requirements and test cases. Importantly, relaxing the thresholds does not reduce bug detection capability, as previously selected features remain included; instead, it monotonically increases test requirements and test cases. In contrast, setting the thresholds to the strict extreme $\theta_u = \theta_l = 1$ removes features whose triggering behavior has any exceptional edge cases, which can exclude valuable test requirements and lead to a noticeable reduction in detected bugs. Therefore, this empirical comparison helps us find the least relaxed threshold that balances test size reduction and bug detection. We do not explore adaptive thresholds that dynamically adjust based on feature discovery or test generation progress. While such schemes may further improve the balance between test size and bug detection, they would require additional implementation-specific information to assess whether the current triggering feature set is overly large or overly restricted. Therefore, we leave the exploration of such online feedback mechanisms for future work.

On average, setting (1) results in 46,270 tests and detects 17.0 bugs, Setting (2) results in 37,432 tests and detects 16.3 bugs, offering a reasonable balance between test size reduction and bug detection. Setting (3) results in smaller test size of 30,118 but detects only 13.7 bugs on average, a significant drop compared to the first two settings. Among the three transpilers (Babel, SWC, and Terser), Terser shows more significant drop in bug detection than the others when using setting (3). This implies that Terser has more exceptional edge-cases where a feature does not always trigger the transpilation, and using stricter thresholds harms its bug detection capability. Based on these results, we concluded that setting (2) is the most effective and selected it for the evaluation.

Example. Consider a language feature f_{add}. Suppose we have generated $10,000$ tests, among which $2,000$ tests contain the feature f_{add}; i.e., $|T| = 10,000$ and $\left|T_{f_{\text{add}}}\right| = 2,000$. Out of the $2,000$ tests, suppose $1,999$ tests trigger the transpilation; i.e., $\left|T^{+}_{f_{\text{add}}}\right| = 1,999$ and $\left|T^{-}_{f_{\text{add}}}\right| = 1$. Then, the triggering-score of the feature f_{add} is:

$$\psi(f_{\text{add}}) = \frac{1,999}{2,000} = 0.9995 > \theta_u = 0.999$$

Since the triggering-score exceeds the upper threshold θ_u, we add the feature f_{add} to the triggering-feature set F:

$$F = F \cup \{f_{\text{add}}\}$$

After generating more tests, suppose the updated triggering-score of the feature f_{add} becomes:

$$\psi(f_{\text{add}}) = \frac{3,900}{4,000} = 0.975 < \theta_l = 0.995$$

Since the triggering-score falls below the lower threshold θ_l, we remove the feature f_{add} from the triggering-feature set F:

$$F = F \setminus \{f_{\text{add}}\}$$

Identifying Key Feature Stacks $\mathcal{F}$. Given the triggering-feature set F, we identify key feature stacks $\mathcal{F}$ by considering the enclosing feature stacks whose outermost feature belongs to F. We observe that a transpiler triggers transpilation process when a specific feature is present, but it follows a different transpilation pass depending on how the internal components of that feature are structured. For example, static blocks of classes is a triggering-feature in Babel whose target version is ES6. However, Babel applies different transpilation passes depending on the internal components of the static block. A transpiler bug is only triggered when the internal components are lexical variable

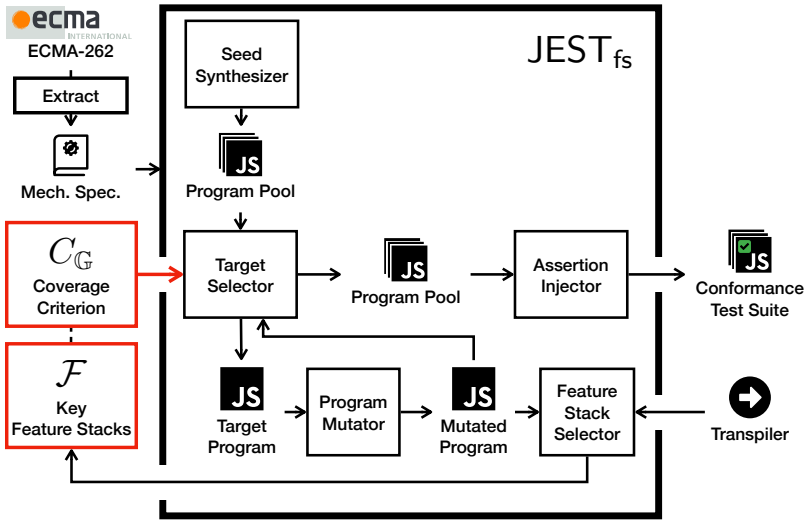


Fig. 7. Overall structure of JEST$_{fs}$ with a (selective) k-FS/FCPS coverage criterion.

declarations.[10] Thus, we define the set of key feature stacks $\mathcal{F}$ as follows:

$$\mathcal{F} = \left\{ \overline{f} \in \mathbb{F}^* \;\middle|\; \overline{f} = [f_1, \cdots, f_m] \wedge f_1 \in F \right\}$$

Example. Consider the triggering-feature set as:

$$F = \{f_{\text{sub}}\}$$

indicating that the transpiler is triggered by the subtraction operation. Following our definition, the key feature stacks are:

$$\mathcal{F} = \{[f_{\text{sub}}], [f_{\text{sub}}, f_{\text{add}}], [f_{\text{sub}}, f_{\text{sub}}], [f_{\text{sub}}, f_{\text{add}}, f_{\text{add}}], \ldots\}$$

As a result, test requirements where the outermost enclosing feature is f_{sub} are all treated distinctly. If we use the 2-sFS coverage with $\mathcal{F}$, the TRs are divided into different 2-sFS-TRs with the following key feature stacks with a bound of 2:

$$\mathcal{F}^{\leq 2} = \{[f_{\text{sub}}], [f_{\text{sub}}, f_{\text{add}}], [f_{\text{sub}}, f_{\text{sub}}]\}$$

This outcome aligns with our initial intuition that all inner features enclosed by the triggering-feature f_{sub} should be treated distinctly, as the transpiler behavior may vary depending on the internal components of the triggering-feature.

5 Implementation

This section introduces JEST$_{fs}$, our JavaScript conformance test synthesizer supporting (selective) k-FS and k-FCPS coverage criteria, and explains how it can detect conformance bugs in JavaScript implementations. Figure 7 illustrates its overall structure. JEST$_{fs}$ takes 1) a mechanized specification extracted by ESMeta [1] and 2) a coverage criterion $C_{\mathbb{G}}$ and performs coverage-guided fuzzing using the CFG of the mechanized specification. To support k-(s)FS and k-(s)FCPS coverage criteria, JEST$_{fs}$ extends four modules from JEST [39] and adds an optional module called the **Feature Stack Extractor** for selective k-FS/FCPS coverage criteria with a given transpiler.

[10] https://github.com/babel/babel/issues/15099

- **Seed Synthesizer**: As the first step, it 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, it 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, it 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 Injector**: After the mutation iteration, it automatically injects 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.
- **Feature Stack Extractor**: It is an optional module for selective k-FS/FCPS coverage criteria with a given transpiler. During the mutation iteration, it checks whether a mutated program triggers transpilation and updates key feature stacks $\mathcal{F}$ by keeping track of the triggering-feature set F. As explained in §4, we use $\theta_u = 0.999$ and $\theta_l = 0.995$ as the upper and lower thresholds for the selective k-FS/FCPS coverage criteria, respectively.

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.

6 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? (§6.1)
- **RQ2 (Effectiveness of k-FS Coverage):** Are higher k-FS coverage criteria more effective than lower k-FS coverage criteria in detecting conformance bugs? (§6.2)
- **RQ3 (Effectiveness of k-FCPS Coverage):** Are k-FCPS coverage criteria more effective than k-FS coverage criteria in detecting conformance bugs? (§6.3)
- **RQ4 (Comparison with Test262):** Can synthesized conformance tests complement Test262, the official JavaScript conformance suite maintained manually? (§6.4)

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	3	3	4
	JSC	v615.1.10	2022.10.26	26	26	26
	GraalJS	v22.2.0	2022.07.26	11	11	11
	SpiderMonkey	v107.0b4	2022.10.24	2	4	4
	Total			**42**	**44**	**45**
Transpiler	Babel	v7.19.1	2022.09.15	37	37	39
	SWC	v1.3.10	2022.10.21	37	37	47
	Terser	v5.15.1	2022.10.05	19	19	19
	Obfuscator	v4.0.0	2022.02.15	1	2	7
	Total			**94**	**95**	**112**
Total				**136**	**139**	**157**

- **RQ5 (Effectiveness of Selective Coverage):** Do selective coverage criteria effectively reduce the number of synthesized conformance tests while maintaining the effectiveness of conformance bug detection? (§6.5)
- **RQ6 (Memory and Runtime Overheads):** What is the memory and runtime overhead of using feature-sensitive coverage criteria? (§6.6)

We apply JEST_{fs} to the JavaScript language specification for ES13 [12], 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, 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.

To summarize the evaluation results: the synthesized conformance tests detected 157 conformance bugs across major JavaScript engines and transpilers, including 139 that were officially confirmed by developers. Increasing the feature-sensitivity level of coverage (k-FS) led to progressively better bug detection, with 55, 83, and 102 bugs detected using 0-FS, 1-FS, and 2-FS, respectively. The feature-call-path-sensitive (FCPS) variant further improved effectiveness, detecting 87 and 111 bugs using 1-FCPS and 2-FCPS coverage, respectively. The synthesized tests also achieved significantly higher specification coverage than the official Test262 suite, covering many specification behaviors that Test262 misses. Finally, the selective coverage strategy reduced the number of generated tests by more than 70% while preserving 92.5% of bug-finding effectiveness, demonstrating its practical utility.

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

6.1 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 157 unique conformance bugs (45 in engines and 112 in transpilers), and reported them to the corresponding developers. As a result, 139 out of 157 bugs were officially confirmed, and 136 were newly discovered. The other 18 reported bugs are still under review, or developers have not yet responded. We present two real-world bug examples.

Example 1. JavaScript engines must follow the execution order of language features described in the language specification. However, we found a bug[11] related to the execution order of the `delete` operation that causes the execution of originally unreachable code in GraalJS. For example, while the following should return `false`, it throws an exception with `"ERR"` by executing unreachable code inside the arrow function in GraalJS:

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

In addition, we detected another bug[12] 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.

Example 2. One of the complex language features in JavaScript is asynchronous functions and generators introduced in ES6 (2015). We detected a bug[13] 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 said:

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

6.2 Effectiveness of k-FS Coverage

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-

[11] https://github.com/oracle/graaljs/issues/671
[12] https://github.com/tc39/ecma262/issues/2659
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=1799288

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 unique 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 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 8 (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 §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 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 19 (102 - 83) more bugs. Now, we present two bug examples that show the effectiveness of k-FS coverage criteria.

Example 3. JavaScript provides diverse `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`:[14]

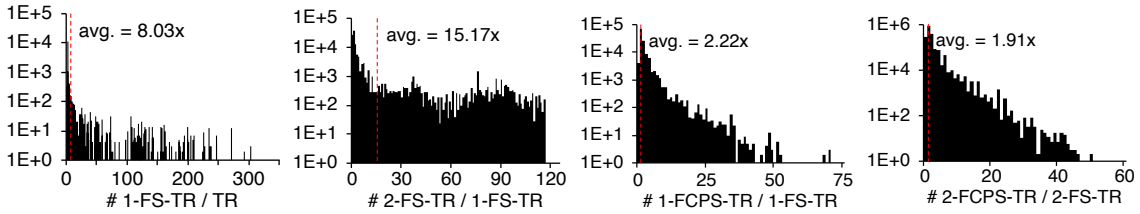
```
for (let {} = 0; 0; ) ; // Expected: Normally terminates
```

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.

Example 4. 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"`:[15]

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

[14] https://github.com/babel/babel/issues/15100
[15] https://bugs.webkit.org/show_bug.cgi?id=247725



(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. 8. The histogram of numbers of k-FS or k-FCPS TRs per less sensitive k-FS or k-FCPS TR

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 *ComputedPropertyName* 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.

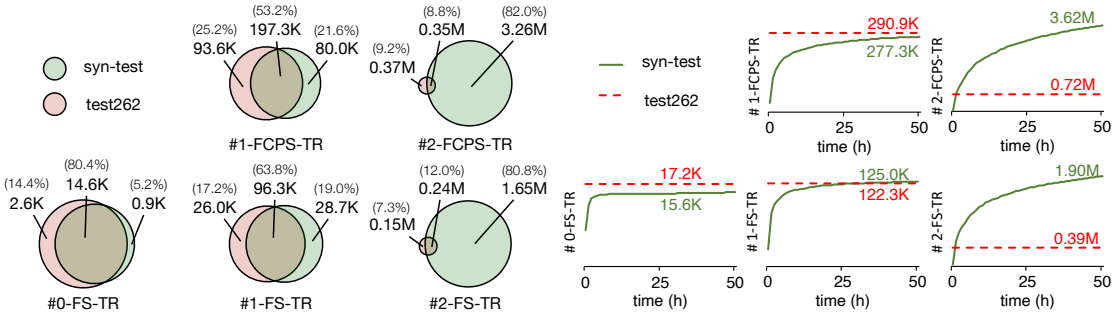
6.3 Effectiveness of k-FCPS Coverage

We 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. 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 8 (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 having quite complex semantics. Thus, the number of possible feature-call-paths in this feature 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. We introduce a conformance bug that shows the effectiveness of k-FCPS coverage criteria compared to the k-FS coverage criteria.

Example 5. 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 (e.g., `""`) is given as the argument, it should throw a **RangeError** exception. However, the following program normally terminates in GraalJS:

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

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



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

Fig. 9. Covered k-FS-TRs and k-FCPS-TRs for synthesized tests via JEST_{fs} and Test262

6.4 Comparison with Test262

We compare the coverage of synthesized conformance tests with that of Test262, the official conformance test suite. As described in §5, the baseline tool JEST relies on the mechanized specification extracted by ESMeta. Thus, we filter out tests in Test262 that utilize language features not supported in the mechanized specification. We use the methodology in the literature [15, 36, 40] to remove inapplicable tests in Test262. Then, we measured the coverage of 23,910 applicable tests with five k-FS/FCPS node-or-branch coverage.

Figure 9 shows (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 9(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 Test262 with 2-F(CP)S-TRs.

6.5 Effectiveness of Selective Coverage

This section is an extended part from the conference paper [42] and evaluates the effectiveness of selective coverage criteria described in §4 that utilize statistical information to select key feature stacks for transpilers. We use the mechanized specification for ES15 instead of ES13 in the extended experiments because the ESMeta now targets the latest specification for ES15. Additionally, selective coverage criteria are meaningless if transpilers transform most of the JavaScript programs. For example, Obfuscator obfuscates all programs, and Babel and SWC transpile most programs if their target versions are ES5.1. Thus, we use the same versions of transpilers listed in Table 1 as evaluation targets but exclude Obfuscator and change target language versions of Babel and SWC to ES6 instead of ES5.1. All configurations in this section were evaluated on the same machine, and our analysis focuses solely on relative comparisons between coverage criteria. To compare the non-selective and selective coverage criteria, we perform the conformance test synthesis using a non-selective (2-FCPS) criterion and a selective (2-sFCPS) criterion in 50 hours for each transpiler. We

Table 3. Statistics of conformance test synthesis with non-selective (2-FCPS) and selective (2-sFCPS) coverage for transpilers.

Transpiler	$C_{\mathbb{G}}$	# Feature	# Feature Stack	# TR	# Syn. Test	# Bug	Transpiled
Babel (ES6)	2-FCPS	1,179	1,391.22K	3,845.6K	127,868	18	34.67%
	2-sFCPS	231	272.58K	1,047.4K	41,540	18	78.42%
	Δ	0.196x	0.196x	0.272x	0.325x	1.000x	2.262x
SWC (ES6)	2-FCPS	1,179	1,391.22K	3,845.6K	127,868	11	28.70%
	2-sFCPS	228	269.04K	897.8K	32,368	11	93.43%
	Δ	0.193x	0.193x	0.233x	0.253x	1.000x	3.256x
Terser	2-FCPS	1,179	1,391.22K	3,845.6K	127,868	24	68.06%
	2-sFCPS	184	217.12K	967.7K	38,390	20	94.66%
	Δ	0.156x	0.156x	0.252x	0.300x	0.833x	1.391x
Average	2-FCPS	1,179	1,391.22K	3,845.6K	127,868	17.7	43.81%
	2-sFCPS	214	252.91K	971.6K	37,432	16.3	88.83%
	Δ	0.182x	0.182x	0.252x	0.293x	0.925x	2.028x

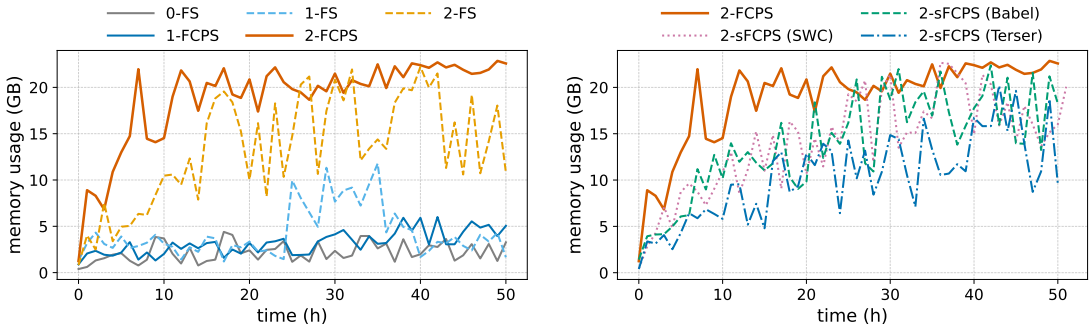
target 2-FCPS coverage criterion because it generates the largest number of synthesized conformance tests among the five graph coverage criteria.

Table 3 shows the comparison of the conformance test synthesis results. First two columns indicate the transpiler and the coverage criteria, respectively. The remaining columns provide various statistics of the synthesized conformance tests, including the number of triggering-features (**# Feature**), the number of key feature stacks (**# Feature Stack**), the number of test requirements (**# TR**), the number of synthesized conformance tests (**# Syn. Test**), the number of detected unique bugs (**# Bug**), and the ratio of tests transpiled by the transpiler relative to the total synthesized tests (**Transpiled**). While there is no concept of triggering-features or key feature stacks in the non-selective coverage criteria, we count all touched features and feature stacks as triggering-features and key feature stacks, respectively, to compare them with the selective coverage criteria. The delta (Δ) row for each transpiler is calculated by dividing the value of 2-FCPS by that of 2-sFCPS for each metric. The average row provides the average values of the three transpilers.

The results show that selective coverage criteria can reduce the number of test requirements or synthesized tests while maintaining the effectiveness of conformance bug detection. The number of test requirements decreased 0.252x on average, and the number of synthesized tests decreased 0.293x on average from 2-FCPS to 2-sFCPS coverage criteria. This decrease is because selective coverage criteria can focus on key feature stacks that are more likely to trigger conformance bugs. The selective approach collects 214 triggering-features on average and identifies 252.91K key feature stacks, which are 18.2% and 18.2% of the touched features and feature stacks, respectively. However, the effectiveness of conformance bug detection is not significantly affected by the selective coverage criteria. The synthesized tests with the 2-FCPS coverage criterion detect a total of 46 transpiler bugs, and the tests with the 2-sFCPS coverage criterion detect 43 bugs, which is 92.5% of the total bugs.

6.6 Memory and Runtime Overheads

We measure the memory and runtime overheads introduced by feature-sensitive coverage criteria during conformance test synthesis. All experiments in this section were conducted on a machine equipped with 4.5 GHz AMD Ryzen 9 7950X processor and 128 GB of RAM.



(a) Non-Selective Coverage Criteria

(b) Selective Coverage Criteria

Fig. 10. Memory usage of conformance test synthesis with different coverage criteria

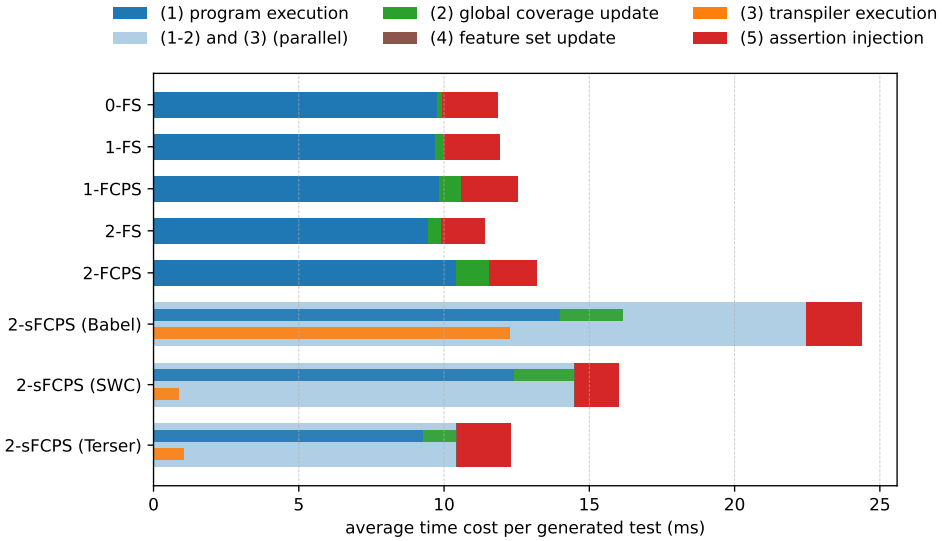


Fig. 11. Runtime of conformance test synthesis with different coverage criteria

Figure 10 shows the memory usage over time for (a) non-selective and (b) selective coverage criteria. In the non-selective case (Figure 10a), increasing feature sensitivity from 0-FS to 1-FS and adding call-path sensitivity at the same level (1-FCPS) has only a slight additional cost. In contrast, increasing k from 1-FS to 2-FS causes a significant jump, indicating that the number of tracked test requirements (TRs) grows more sharply with higher feature depth than with added feature-call-path sensitivity.

In the selective case (Figure 10b), memory usage is noticeably reduced across all targets. Among transpilers, Terser, which has a smaller triggering feature set (184 features), consumes less memory than Babel and SWC (231 and 228 features, respectively), highlighting that selective coverage scales with the size of the target feature set. We note that a

transient spike in the 1-FS line around the 30-hour is likely due to delayed garbage collection and does not reflect the behavior of the coverage criteria.

To measure runtime overheads, we break down the average per-test time into five steps:

(1) *Mutated program execution*, which collects coverage information.
(2) *Global coverage update*, which merges the new coverage into the global coverage information.
(3) *Transpiler execution* (selective only), to determine triggered features.
(4) *Feature set update* (selective only), which updates the target feature set.
(5) *Assertion injection*, which inserts assertions to convert the mutant into a test.

In selective coverage criteria, steps (1) and (2) run in parallel with step (3), sharing system resources.

Figure 11 presents the average runtime of each step across coverage criteria. For non-selective coverage, step (1) dominates the cost per test. Notably, adding feature-call-path sensitivity to 1-FS (i.e., 1-FCPS) increases the time for step (2) more than increasing the feature depth from $k = 1$ to $k = 2$ (0.34 ms to 0.76 ms vs. 0.34 ms to 0.47 ms), suggesting that indexing call-path-sensitive coverage incurs more overhead in this step than indexing additional feature-depth alone.

Interestingly, the time spent in step (5), assertion injection, decreases by 20% when moving from 1-FS to 2-FS or from 1-FCPS to 2-FCPS. This is likely because higher feature sensitivity, generating more diverse and complex programs, leads to larger portion of programs that throw exceptions, for which only simple assertions checking for exceptions are injected. Since more complex assertions like equality checks are only generated for normally terminating programs, this reduces the average time spent in assertion injection.

For selective coverage criteria, runtime for step (1) and (2) is higher than in non-selective cases for Babel and SWC, likely due to shared system load from parallel transpiler execution. In contrast, Terser's lighter transpilation results in similar step (1) and (2) times between selective and non-selective cases, also not causing any additional delays from parallel execution. For step (3), transpiler execution time for Babel (12.3 ms) is significantly higher than for SWC (0.9 ms) and Terser (1.1 ms), likely due to Babel's more extensive transformations. SWC exhibits particularly fast transpilation; however, its aggressively parallel execution leads to increased resource contention, which slows down steps (1) and (2) when executed concurrently. Feature set update (step 4) is negligible in runtime cost (less than 0.02 ms) across all transpilers.

Note that the average runtime of overlapping parallel steps (step (1)/(2) vs. step (3)) may exceed the maximum of either individually. This occurs because averages are computed across multiple tests, where different steps may dominate on a per-test basis.

7 Threats to Validity

Evaluation Target. Although we primarily use JavaScript as our target language, the underlying principles of FS/FCPS coverage criteria are language-agnostic. ECMAScript specifications are relatively modular and algorithmic, which simplifies identifying language features and their semantic boundaries. For languages defined using operational semantics, FS/FCPS coverage can be applied by treating inference rules or semantic functions corresponding to syntactic constructs as features. Similarly, for denotational semantics, language features can be derived from the semantic clauses associated with each syntactic form. For languages with macro systems or non-hierarchical syntactic constructs, FS coverage would require incorporating macro expansion or preprocessing phases into the mechanized semantics; in such cases, macro definitions or expansion rules could be treated as language features. These potential extensions suggest that the applicability of FS/FCPS coverage primarily depends on the availability of a mechanized specification with

identifiable semantic components, rather than on the specific structure of ECMAScript. Given that diverse programming languages, including OCaml [35], C [5], Java [7], WebAssembly [60], and POSIX shell [17], provide mechanized specifications, we expect our coverage criteria to be applicable to a wide range of programming languages. Instead, for target implementations, we use eight mainstream JavaScript engines and transpilers to show the general effectiveness of our coverage criteria for conformance testing.

Generalizability of Selective Coverage. While we propose and evaluate a way to extract a triggering-feature set from JS-to-JS *transpilers* for selective FS/FCPS coverage, we currently lack an evaluation on other language implementations (e.g., JS-to-C transpilers, engines, interpreters). These implementations do not have the same obvious trigger-based activations as transpilers. For example, a JavaScript engine will attempt to execute all input code without skipping or altering language features. This difference makes it challenging to directly apply the same black-box key feature stack extraction method used for transpilers to other types of implementations. That said, this limitation is not inherent to the concept of selective coverage. In principle, one could redefine what a *trigger* means for engines by choosing a suitable internal event or condition as the trigger signal. For instance, with proper gray-box instrumentation of an engine, a user-defined trigger could be an event such as the activation of the JIT compiler or the use of a specialized optimization path. We did not explore this direction in the current work due to complexity and tooling constraints, and we leave a systematic investigation of such instrumentation-based selective coverage for future work. Also, our evaluation of selective coverage focuses on 2-FCPS, which is the most effective configuration for bug detection and generates the largest number of test requirements. The selective approach applies identically to both k-FS and k-FCPS, and thus generalizes to both coverage criteria. However, we do not separately evaluate selective FS, as it provides limited additional insight beyond selective FCPS relative to the cost of additional experiments. Furthermore, we do not evaluate $k \geq 3$ due to the exponential growth of test requirements and the resulting resource constraints.

Threshold Sensitivity. The choice of upper θ_u and lower θ_l thresholds affects the trade-off between test reduction and bug retention capabilities of the selective coverage criteria. As explained in §4.2, we use $\theta_u = 0.999$ and $\theta_l = 0.995$ in our evaluation. The results show that these thresholds are effective because they allow us to retain 92.5% of the bugs and reduce the number of tests by 72.4% on average. However, the optimal choice of thresholds is not universal and may vary depending on the target language implementation and the conformance testing goal.

Coverage Calculation Overhead. If we use FS/FCPS coverage criteria, we need to calculate the innermost enclosing language features for each test case. Additionally, we need to collect the key feature stacks for selective coverage. While it introduces overhead in the test synthesis process, it is negligible compared to the overall test synthesis time.

Conformance Test Synthesis Time. As shown in Figure 9b, our test synthesis process using FS/FCPS coverage takes up to 50 hours per language version. While this may seem expensive, we argue that the cost is acceptable in practice because the mechanized language specification evolves much more slowly than language implementations. In most cases, a complete test regeneration is only needed when the specification is updated, typically on an annual cycle (e.g., ECMA-262 editions). Even for internal drafts or staging versions, specification-level changes tend to occur at a granularity of days rather than hours, making our approach viable for maintaining conformance suites in real-world settings.

For the selective approach, which does rely on implementation-specific behavior, we acknowledge that its reuse window may be shorter. However, triggering features tend to remain stable over most development cycles, enabling the generated tests to remain useful over time.

8 Related Work

Coverage Criteria in Software Testing. Coverage criteria in software testing are essential in measuring the quality of test inputs. The most common ones are structural coverage criteria in a given program's control-flow graph (CFG) [2, 10], data-flow information [20], or types [4]. On the other hand, model-based coverage [52] criteria consider specialized abstract behavior models and define the TRs in the model. Such models include state transitions [3], autonomous driving systems [24], deep neural networks [31, 34, 44, 50, 56]. However, there are no specialized coverage criteria for programming language implementations. In this paper, we first presented feature-sensitive criteria as general extensions of graph coverage criteria for PL implementations to discriminate TRs 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 [35], C [5], C++[46], Java [7], and POSIX shell [17]. At the same time, general metalanguages or frameworks for mechanized language specifications have emerged as well. For example, [49] presented Ott as a tool that compiles language semantics into proof assistant code and supports a metalanguage used in defining language semantics as inference rules. The $\mathbb{K}$ framework [47] proposed a formalism for writing operational semantics and provides a derivation of verifiers directly from the semantics. [6] developed a skeletal semantics framework in Coq for creating big-step semantics based on the structure of semantics.

For JavaScript, diverse mechanized specifications have been presented based on ECMA-262 [12]. KJS [36] utilizes the $\mathbb{K}$ framework, and [15] presented a metalanguage, JSIL, for ES5.1. Researchers have used them in diverse fields: verification [15], symbolic execution [16], abstract interpretation-based static analysis [21, 22, 26, 41, 48], and double debugger [8]. However, most of them focused on only ES5.1, released in 2011, and required manual description of the semantics. On the other hand, ESMeta supports a metalanguage $\mathsf{IR}_{\mathsf{ES}}$ for the latest version of ECMA-262 and the automatic extraction of mechanized specification used in conformance test synthesis [39], specification type analysis [38], and static analyzer derivation [37]. Hence, we implemented $\mathsf{JEST}_{\mathsf{fs}}$ based on ESMeta to synthesize conformance tests from the latest version of ECMA-262 with feature-sensitive coverages.

Conformance Testing for JavaScript. Diverse host environments support JavaScript engines, and even 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 [13], the official JavaScript conformance test suite. Researchers have focused on testing engines to detect bugs using generation-based fuzzing [11, 18, 19], mutation-based fuzzing [43, 53, 54], deep learning [27, 58], and differential testing [28, 32, 51]. However, most existing techniques focused on detecting crashing bugs or security vulnerabilities rather than conformance bugs. While Comfort [58] targets conformance bugs, it heavily relies on differential testing instead of the semantics. On the other hand, JEST [39] is the first tool that synthesizes JavaScript conformance tests according to the semantics described in ECMA-262. We implemented $\mathsf{JEST}_{\mathsf{fs}}$ by augmenting it with (selective) k-FS/FCPS coverage criteria and outperformed the ability of bug detection of the JEST.

Testing Language Implementations. Beyond JavaScript, a substantial work has investigated testing techniques for language implementations. A widely adopted approach is differential testing, which executes the same program across multiple implementations and detects behavioral differences. For example, Csmith and its successors generate

random C programs to expose miscompilation bugs in compilers [30, 57]. More recent work combines differential testing with fuzzing and coverage guidance, including coverage-directed approaches for Java Virtual Machines that guide test generation using runtime code coverage of a reference JVM implementation [9], graybox fuzzing of C compilers and analyzers [14], and interpreter-guided differential testing of JIT compilers [45]. Related work based on metamorphic testing generates semantics-preserving program variants to detect inconsistencies within a single compiler or runtime [25]. While these approaches have been effective in discovering crashes and miscompilations, they generally lack coverage criteria defined directly over formal language semantics, which limits their ability to systematically detect semantics-level conformance bugs.

9 Conclusion

Conformance testing using graph coverage has been one of the most promising approaches to support correct and consistent programming language implementations. However, since language implementations often utilize different execution paths even for the same language features, traditional graph coverage does not produce high-quality conformance tests. In this paper, we present novel coverage criteria designed for language implementations: *feature-sensitive (FS)* and *feature-call-path-sensitive (FCPS)* coverage by refining given test requirements using enclosing language features and feature-call-paths. Our experiments show that our new criteria outperform the traditional criteria in the context of conformance bug detection in real-world JavaScript implementations. In addition, we present *selective* FS/FCPS coverage criteria to reduce the number of tests while preserving the test quality by identifying *key feature stacks* for the target language implementation.

9.1 Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) (RS-2021-NR060080, RS-2022-NR070102, and RS-2024-00344597) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) (RS-2024-00337703, RS-2024-00440780) grants funded by the Korea government (MSIT).

References

[1] 2022. *ESMeta: An ECMAScript specification metalanguage used for automatically generating language-based tools.* https://github.com/es-meta/esmeta

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

[3] 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).* doi:10.1109/ICST.2017.33

[4] 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).* doi:10.1109/FormaliSE.2017.10

[5] 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. doi:10.1007/s10817-009-9148-3

[6] 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.* doi:10.1145/3290357

[7] 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).* doi:10.1145/2676726.2676982

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

[9] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16).* Association for Computing Machinery, New York, NY, USA, 85–99. doi:10.1145/2908080.2908095

[10] 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.

[11] 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)*. doi:10.14722/ndss.2021.24224

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

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

[14] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. doi:10.1145/3597926.3598130

[15] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, 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. doi:10.1145/3158138

[16] 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. doi:10.1145/3290379

[17] 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*. doi:10.1145/3371111

[18] 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)*. doi:10.14722/ndss.2019.23263

[19] 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. doi:10.1145/3460120.3484823

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

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

[22] 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)*. doi:10.1145/2635868.2635904

[23] 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)*. doi:10.1145/3551357.3551381

[24] 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). doi:10.1145/3550270

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[26] 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)*.

[27] 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.

[28] 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)*. doi:10.1145/3236024.3236037

[29] 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)*. doi:10.1109/ICSTW.2009.30

[30] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. doi:10.1145/3428264

[31] 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)*. doi:10.1145/3238147.3238202

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

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

[34] 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)*.

[35] 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

[36] 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

[37] 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)*. doi:10.1145/3540250.3549097

[38] 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)*. doi:10.1109/ASE51524.2021.9678781

[39] 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. doi:10.1109/ICSE43902.2021.00015

[40] 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. doi:10.1145/3324884.3416632

[41] 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)*. doi:10.1109/ICSE-C.2017.4

[42] Jihyeok Park, Dongjun Youn, Kanguk Lee, and Sukyoung Ryu. 2023. Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations. *Proc. ACM Program. Lang.* 7, PLDI, Article 126 (jun 2023), 23 pages. doi:10.1145/3591240

[43] 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. doi:10.1109/SP40000.2020.00067

[44] 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)*. doi:10.1145/3132747.3132785

[45] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-guided differential JIT compiler unit testing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 981–992. doi:10.1145/3519939.3523457

[46] 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)*. doi:10.1145/2103656.2103718

[47] 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. doi:10.1016/j.jlap.2010.03.012

[48] 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)*. doi:10.1145/2499370.2462168

[49] 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. doi:10.1017/S0956796809990293

[50] 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)*. doi:10.1145/3238147.3238172

[51] 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. doi:10.1109/TR.2022.3171220

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

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

[54] 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)*. doi:10.1109/ICSE.2019.00081

[55] 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*. 264–274. doi:10.1109/ISSRE.1997.630875

[56] 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)*. doi:10.1145/3293882.3330579

[57] 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)*. doi:10.1145/1993498.1993532

[58] 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)*. doi:10.1145/3453483.3454054

[59] 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. doi:10.1002/stvr.430

[60] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (June 2024), 26 pages. doi:10.1145/3656440