Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations

JIHYEOK PARK, Korea University, South Korea
DONGJUN YOUN, KAIST, South Korea
KANGUK LEE, KAIST, South Korea
SUKYOUNG RYU, KAIST, South Korea

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

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

CCS Concepts: • Software and its engineering → Software testing and debugging.

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

ACM Reference Format:

Authors’ addresses: Jihyeok Park, jihyeok_park@korea.ac.kr, Korea University, Seoul, South Korea; Dongjun Youn, f52985@kaist.ac.kr, KAIST, Daejeon, South Korea; Kanguk Lee, p51lee@kaist.ac.kr, KAIST, Daejeon, South Korea; Sukyoung Ryu, sryu.cs@kaist.ac.kr, KAIST, Daejeon, South Korea.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2473-1421/2023/6-ART126
https://doi.org/10.1145/3591240
1 INTRODUCTION

The conformance testing of programming language implementations is essential to provide correct and consistent implementations of the language semantics. Many programming languages have multiple implementations rather than a single reference implementation. For example, Java uses a Java Virtual Machine (JVM) to compile Java programs into JVM bytecode and execute them. Developers are free to choose one of the existing JVM implementations, such as OpenJ9, GraalVM, HotSpot, Zulu, and Corretto. Python has the reference interpreter, CPython, in addition to diverse interpreters, including PyPy, Jython, and IronPython. Therefore, ensuring correct and consistent execution environments in different implementations of the same language becomes crucial. However, since manually maintaining conformance test suites for real-world programming languages is cumbersome and labor-intensive, only a small number of programming languages, such as JavaScript [ECMA International 2022b] and XML [XMLTestSuite 2011], provide their official conformance test suites. Thus, researchers have presented ways to test the conformance of multiple implementations using differential testing [McKeeman 1998] for compilers [Cummins et al. 2018; Ofenbeck et al. 2016; Tu et al. 2022; Yang et al. 2011], interpreters [Bernhard et al. 2022; Ye et al. 2021], virtual machines [Chen et al. 2019], and debuggers [Lehmann and Pradel 2018]. To make differential testing for language implementations effective, researchers have proposed various techniques to synthesize diverse programs, such as generation-based fuzzing [Bernhard et al. 2022; Lehmann and Pradel 2018; Ofenbeck et al. 2016; Yang et al. 2011], mutation-based fuzzing [Chen et al. 2019; Tu et al. 2022], and deep learning [Cummins et al. 2018; Ye et al. 2021].

Graph Coverage. Graph coverage [Ammann and Offutt 2008] is one of the most widely-used coverage criteria in evaluating the quality of conformance tests. Higher coverage of a conformance test suite denotes that it covers more test requirements (TRs) of a given coverage criterion for language implementations. Graph coverage helps generate tests that reach uncovered parts of software; coverage-guided fuzzing (CGF) [Michal Zalewski 2007] improves mutation-based fuzzing by selecting mutation target programs using coverage information. It also helps avoid an excessive number of conformance tests; researchers have presented various test minimization techniques [Yoo and Harman 2012] to reduce the number of tests, and Wong et al. [1997] present coverage-guided test minimization.

Graph Coverage for Mechanized Language Specifications. One approach to making high-quality conformance tests is to use graph coverage to generate tests for mechanized language specifications. While we can use code coverage to generate tests for “actual language implementations,” it leads to different coverage information for different implementations. On the contrary, graph coverage for mechanized language specifications leads to uniform coverage for multiple implementations. Various programming languages, such as OCaml [Owens 2008], C [Blazy and Leroy 2009], C++ [Ramananandro et al. 2012], Java [Bogdanas and Roşu 2015], JavaScript [Park et al. 2020a], and POSIX shell [Greenberg and Blatt 2019], have mechanized specifications that describe their semantics using diverse metalanguages and frameworks, such as Ott [Sewell et al. 2010], Skel [Bodin et al. 2019], and the K framework [Rosu and Serbanută 2010]. Mechanized specifications use functions or inductive definitions to describe the semantics of language features. Thus, it is possible to convert them as directed graphs and adapt them to apply traditional graph coverage criteria for software. For example, K-Java [Bogdanas and Roşu 2015] is a mechanized specification for Java defined with the K framework, which describes language semantics as a set of reduction rules. Consider a directed graph whose nodes are reduction rules and edges are their dependencies in K-Java. Then, we can measure the coverage of a test suite in the directed graph denoting K-Java based on whether each test covers the test requirements of a graph coverage criterion.
Challenges. However, graph coverage may not produce high-quality conformance tests for mechanized language specifications. Mechanized specifications are usually written in a modular way with helper functions. Such a modular definition has the advantages of preventing duplicated or similar definitions of language semantics, reducing the size of a mechanized specification, and enhancing its readability. Despite its usefulness, reusing the same helper function for different parts may degrade the quality of conformance testing in two ways.

First, traditional graph coverage may not distinguish test requirements of different language features when their semantics descriptions use the same functions, degrading conformance testing quality. For example, consider a mechanized specification for JavaScript that represents the abstract algorithms described in the official language specification, ECMA-262 [ECMA International 2022a]. Here, most of the semantics for the addition and subtraction operators are defined using the same `EvaluateStringOrNumericBinaryExpression` algorithm as a helper function. If conformance tests for the addition operator already cover the test requirements in the algorithm, most conformance tests for the subtraction operator are removed after the coverage-guided test minimization process. However, real-world JavaScript engines are highly optimized and often have specialized execution paths for different language features, even when their semantics descriptions use the same functions. Therefore, we need to test possible edge cases for the subtraction operator as well, even though similar edge cases for the addition operator are already tested.

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

This Work. To alleviate this problem, we introduce feature-sensitive (FS) coverage, a novel coverage criterion to generate high-quality conformance tests for programming language implementations. It is a general extension of graph coverage, refining test requirements using the innermost enclosing language features. FS coverage resolves the problem of sharing the same helper functions for the semantics of different language features. We also present a feature-call-path-sensitive (FCPS) coverage, a variant of FS coverage with feature-call-paths from language features to test requirements. FCPS coverage resolves the problem of sharing the same helper functions for the semantics of different parts of the same language feature. In addition, we extend both coverage criteria using the $k$-limiting approach as $k$-FS coverage and $k$-FCPS coverage. To evaluate the effectiveness of the new coverage criteria, we apply them to a real-world programming language, JavaScript. We select JavaScript as the evaluation target language because 1) it has the most up-to-date mechanized specification and 2) it has the official conformance test suite, Test262 [ECMA International 2022b]. We extend JEST [Park et al. 2021b], the state-of-the-art JavaScript conformance test synthesizer using coverage-guided mutational fuzzing, with various FS and FCPS coverage criteria. For the latest language specification (ES13, 2022), our tool automatically synthesizes 237,981 conformance tests in 50 hours with five coverage criteria. We evaluated the conformance of eight mainstream JavaScript implementations (four engines and four transpilers) with the synthesized conformance tests and discovered bugs in all of them. The tool detected 143 distinct conformance bugs, 85 of which were confirmed by the developers and 83 of which were newly discovered bugs.
AdditiveExpression := [Yield, Await] :
  MultiplicativeExpression[?Yield, ?Await]

Fig. 1. Syntax of AdditiveExpression in ES13

Contributions. We summarize our contributions as follows:

- We introduce novel feature-sensitive (FS) coverage to discriminate test requirements with the innermost enclosing language features to enhance the quality of conformance testing for programming language implementations. It can resolve the problem of sharing the same helper functions for the semantics of different language features.

- We also present feature-call-path-sensitive (FCPS) coverage as its variant to distinguish different parts in the semantics of the same language feature.

- We experimentally show that the new coverage criteria outperform the traditional coverage criteria in the context of conformance bug detection in eight mainstream JavaScript implementations (four engines and four transpilers) with the latest ECMA-262 (ES13, 2022). The tool uncovered 83 brand-new bugs.

2 BACKGROUND AND MOTIVATION

In this section, we explain why traditional graph coverage may not produce high-quality conformance tests using JavaScript as an example language. We select JavaScript because its mechanized specifications are actively maintained, while most mechanized specifications of other languages are outdated. Since all the existing JavaScript mechanized specifications [Fragoso Santos et al. 2018; Khayam et al. 2022; Park et al. 2015, 2020a] closely capture the abstract algorithms in ECMA-262 [ECMA International 2022a], we show how JavaScript mechanized specifications describe the JavaScript syntax and semantics using ECMA-262. Then, we explain the control-flow graph (CFG) of abstract algorithms in ECMA-262 and how to use the CFG in coverage-guided fuzzing. Finally, we demonstrate why a simple node coverage criterion cannot fully discriminate different semantics in different language features or even in the same language feature.

2.1 JavaScript Language Specification (ECMA-262)

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

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

\[
1 + 2
\]

(1)

It computes the addition of two Number values, 1 and 2. Figure 1 describes its syntax with the production of AdditiveExpression. 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.

\[\text{https://262.ecma-international.org/13.0/#prod-AdditiveExpression}\]
Feature-Sensitive Coverage for Conformance Testing

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

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

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

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

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

A normal algorithm is the primary form of an abstract algorithm defined by its 1) name, 2) parameters, and 3) body. It is commonly used as a helper function, and multiple normal algorithms are used when defining the semantics of language features. Hence, the semantics of different language features often share the same normal algorithms. For example, both SDOs in Figure 2 invoke the same normal algorithm EvaluateStringOrNumericBinaryExpression with different second arguments + and −, respectively. Then, they transitively invoke other normal algorithms, ApplyStringOrNumericBinaryOperator and ToNumeric. Thus, at least three normal algorithms are shared in the semantics of the addition and subtraction expressions.

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

EvaluateStringOrNumericBinaryExpression (leftOperand, opText, rightOperand)

1. If (leftOperand is present) then
2. Let lval be \( ? \) ToNumeric(leftOperand).
3. Let rnum be \( ? \) ToNumeric(rightOperand).
4. If Type(lval) is different from Type(rnum), throw a TypeError exception.
5. Return \( ? \) ApplyStringOrNumericBinaryOperator(lval, opText, rnum).

ApplyStringOrNumericBinaryOperator (lval, opText, rval)

1. Let lnum be \( ? \) ToNumeric(lval).
2. Let rnum be \( ? \) ToNumeric(rval).
3. If Type(lnum) is different from Type(rnum), throw a TypeError exception.
4. Return \( ? \) ApplyStringOrNumericBinaryOperator(lnum, opText, rnum).

ToNumeric (value)

1. If value is present, then
2. Let prim be \( ? \) ToNumeric(value).
3. Return prim.

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

Built-in methods

Number (value)

1. If value is present, then
2. Let prim be \( ? \) ToNumeric(value).
3. Return prim.

Fig. 4. Built-in method Number in ES13

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

2.2 Control Flow Graph (CFG) of ECMA-262

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

For example, Figure 5 depicts a CFG of the abstract algorithms in Figures 2, 3, and 4. In this figure, circles (or diamonds) denote nodes (or branches), arrows denote edges, and boxes indicate algorithms. The labels inside nodes match the labels annotated in the algorithms in Figures 2, 3, and 4. Let us apply coverage-guided fuzzing [Michal Zalewski 2007] with a node coverage criterion in the CFG and assume that a simple JavaScript program, \( 1 + 2 \), exists in the program pool. The program does not satisfy the condition in the branch labeled 20 because the left-hand and right-hand sides of \( 1 + 2 \) are both Number values rather than BigInt values. Thus, it does not cover the red node labeled 21. Now assume that another program, \( 3n + 4n \), is generated by mutating the previous program. Then, it covers the red node labeled 21 because it satisfies the condition in the branch labeled 20 with BigInt values on both sides of the + operator.
2.3 Motivation

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

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

2.3.2 Different Semantics in the Same Language Feature. In addition, different parts in the semantics of the same language feature may use the same algorithm more than once. For example, the semantics of the + operator uses ApplyStringOrNumericBinaryOperator, and it invokes ToNumeric twice in the nodes labeled 12 and 14. Now, assume that the current program pool contains a program $2n + 1$ again. Then, the red node labeled 21 is covered by the program $2n + 1$ because the left-hand side is a BigInt $2n$. It means that another similar program $1 + 2n$ would not be added to the program pool because the test requirement for the node labeled 21 is already covered by $2n + 1$. However, $1 + 2n$ is also a meaningful test case because it checks the edge case when the right-hand side of the + operator is a BigInt value.
In the remainder of the paper, we formally define a feature-sensitive coverage criterion and its variants to resolve the problems (Section 3). Then, we explain how to implement a conformance test synthesizer with feature-sensitive coverage criteria (Section 4). Finally, after evaluating feature-sensitive coverage criteria with mainstream JavaScript implementations (Section 5), we discuss related work (Section 6) and conclude (Section 7).

3 FEATURE-SENSITIVE COVERAGE CRITERIA

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

3.1 Notations

First, we define notations used in the definition of graph coverage criteria. A directed graph \( G = (N, N_i, N_f, E) \) consists of:

- \( N \): a set of nodes
- \( N_i \subseteq N \): a set of initial nodes
- \( N_f \subseteq N \): a set of final nodes
- \( E \subseteq N \times N \times (A \cup \{⊥\}) \): a set of edges optionally annotated with annotations \( A \).

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

\[
P_G = \{n_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{m-1}} n_m \mid \forall i < m. n_i \xrightarrow{a_i} n_{i+1} \in E \land n_m \in N\}
\]

(2)

The length of a path is defined as \( \|n_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{m-1}} n_m\| = m \). A path \( p \) is a subpath (\( \subseteq \)) of a path \( p' \) when \( p \) is a subsequence of \( p' \). We use the notation \( \preceq \) for a prefix relation, and first(\( p \)) and 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: first(\( p \)) \in \( N_i \) and last(\( p \)) \in \( N_f \). Then, \( \text{path}_G : T \rightarrow P_G \) is a mapping from a test path \( t \) in \( T \) to a test path in \( G \), and we call \( \text{path}_G(t) \) the execution path of \( t \).

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

\[
T = \{\cdots, 2n + 1, 2n - 1, \cdots\}
\]

(3)

Then, \( \text{path}_G(t_{\text{add}}) \), the execution path of \( t_{\text{add}} \), is depicted as follows:

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

\[
p = 22 \xrightarrow{\text{ret}} 15 \xrightarrow{\text{if}} 16 \xrightarrow{\text{if}} 17
\]

whose length is \( ||p|| = 3 \).

### 3.2 Graph Coverage Criteria

We formulate graph coverage criteria by referring to their well-known definitions \[\text{[Ammann and Offutt 2008]},\] and they are defined by 1) a set of test requirements and 2) a cover relation between paths and test requirements:

**Definition 3.1 (Graph Coverage Criteria).** A graph coverage criterion \( C_G = (\mathcal{R}_G, \sim) \) for a given directed graph \( G \) is defined with:

- a set of test requirements (TRs) \( \mathcal{R}_G \)
- a cover relation \( \sim \subseteq \mathcal{P}_G \times \mathcal{R}_G \) between paths and TRs

In a specific graph coverage criterion \( C_G \), we say that a path \( p \) covers a TR \( r \in \mathcal{R}_G \) when \( p^\text{cover} r \). A test \( t \in \mathcal{T} \) covers the TR \( r \) if there exists a prefix path \( p \) of its execution path that covers the TR:

\[
t^\text{cover} r \iff \exists p \in \mathcal{P}_G. \text{ s.t. } p \leq \text{path}_G(t) \wedge p^\text{cover} r
\]

A test set \( T \subseteq \mathcal{T} \) satisfies (\( \triangleright \)) the criterion \( C_G \) when it covers all feasible TRs:

\[
T \triangleright C_G \iff \forall r \in \mathcal{R}_G. \text{ r is feasible } \Rightarrow \exists t \in \mathcal{T}. \text{ s.t. } t^\text{cover} r
\]

where a TR \( r \) is feasible if there exists a possible test \( t \in \mathcal{T} \) that covers \( r \). If \( T \triangleright C_G \Rightarrow T \triangleright C'_{G} \) for any test set \( T \), we say that \( C_G \) subsumes \( C'_{G} \) and use the notation: \( C_G \triangleright C'_{G} \). The subsumption relation between graph coverage criteria is a preorder.

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

- the set of TRs \( \mathcal{R}_G \) is a set of nodes: \( \mathcal{R}_G = \mathbb{N} \)
- a path \( p \) covers a node \( n \) when it ends with the node \( n \): \( p^\text{cover} n \iff \text{last}(p) = n^{6} \)

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

**Definition 3.3 (k-Limiting Path Coverage).** In a \( k \)-limiting path coverage criterion \( C^\text{\text{\text{-path}}}_G \),

- the set of TRs \( \mathcal{R}_G \) is a set of paths whose lengths are bounded by \( k \): \( \mathcal{R}_G = \{ p \in \mathcal{P}_G \mid ||p|| \leq k \} \)
- a path \( p \) covers a path \( p' \) when their last nodes are equal and the path \( p' \) is a subpath of \( p \):

\[
p^\text{cover} p' \iff \text{last}(p) = \text{last}(p') \wedge p' \subseteq p
\]

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

- The edge coverage criterion is \( C^\text{1-path}_G \)
- The edge-pair coverage criterion is \( C^\text{2-path}_G \)
- The complete path coverage criterion is \( C^\text{\text{-path}}_G \)

---

6Another 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.
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 $C^i_{G}$ is always a subset of that in $C^j_{G}$, and $C^i_{G}$ subsumes $C^j_{G}$. 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, Ammann and Offutt [2008] 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 [Li et al. 2009] in practice with intra-procedural control-flow graphs.

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

### 3.3 Feature-Sensitive (FS) Coverage Criteria

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

- a given graph coverage criterion $C_G$
- a set of language features $\mathcal{F}$
- a feature mapping $\text{feat} : \mathbb{N} \rightarrow \mathcal{F} \cup \{\bot\}$, a partial mapping from nodes to language features where $\text{feat}(n) = \bot$ means that there is no language feature for the node $n$.

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

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

In other words, $p\mid_{\text{call}}$ 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 $\text{ext}_F : \mathcal{P}_G \mid_{\text{call}} \rightarrow \mathcal{F} \cup \{\bot\}$ as a partial mapping from call-site stacks to the innermost enclosing language features $\mathcal{F}$:

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

(8)

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

**Definition 3.4 (Feature-Sensitive (FS) Coverage Criteria).** For a given graph coverage criterion $C_G = (R_G, \sim)$, the feature-sensitive (FS) coverage criterion $C^F_G = (R^F_G, \sim)$ is defined as follows:

- the set of feature-sensitive test requirements (FS-TRs) $R^F_G$ is a set of original TRs optionally with language features: $R^F_G = R_G \times (\mathcal{F} \cup \{\bot\})$
- a path $p$ covers a FS-TR $(r, f)$ when $p$ covers the original TR $r$ and $f$ is the innermost enclosing language feature of $p$: $p \sim (r, f) \iff p \sim r \wedge \text{ext}_F(p\mid_{\text{call}}) = f$
**Example.** For the CFG \( G \) in Figure 5, assume that the feature mapping is as follows:

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

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

\[
\begin{align*}
  p_{\text{add}} : & \text{ call } 1 \text{ call } 8 \text{ call } 12 \text{ ret } 13 \text{ ret } 15 \rightarrow 17 \\
  p_{\text{sub}} : & \text{ call } 4 \text{ call } 8 \text{ call } 12 \text{ ret } 13 \text{ ret } 15 \rightarrow 17
\end{align*}
\]

First, the call-site stack of \( p_{\text{add}} \) is \( p_{\text{add}} |_{\text{call}} = [\cdot \cdot \cdot , 1, 8] \) 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 \( p_{\text{add}} \) is \( \text{ext}(p_{\text{add}} |_{\text{call}}) = \text{feat}(1) = f_{\text{add}} \).

Hence, if we use a FS node coverage criterion \( C^F_{G}[\text{node}] \), the path \( p_{\text{add}} \) covers a FS-TR \((17, f_{\text{add}})\), and the test \( t_{\text{add}} \) for addition covers it as well. In a similar way, we know that the innermost enclosing language feature of \( p_{\text{sub}} \) is \( \text{ext}(p_{\text{sub}} |_{\text{call}}) = \text{feat}(4) = f_{\text{sub}} \) instead of \((17, f_{\text{add}})\) and remains in the program pool.

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

\[
\text{ext}^k_G([n_1, \ldots, n_m]) = \begin{cases} 
  \epsilon & \text{if } k = 0 \lor m = 0 \\
  \text{ext}^{k-1}_G([n_1, \ldots, n_{m-1}]) \cdot f & \text{if } \text{feat}(n_m) = f \\
  \text{ext}^{k}_G([n_1, \ldots, n_{m-1}]) & \text{otherwise}
\end{cases}
\]

\[ (10) \]

**Definition 3.5 (k-Limiting Feature-Sensitive (k-FS) Coverage Criteria).** For a given \( C_G = (\mathbb{R}_G, \sim) \), the \( k \)-limiting feature-sensitive (k-FS) coverage criterion \( C^k_{G}[\text{node}] \) is defined as follows:

- the set of \textit{k-feature-sensitive test requirements} (k-FS-TRs) \( \mathbb{R}^k_{G}[\text{node}] \) is a set of original TRs with at most \( k \) language features: \( \mathbb{R}^k_{G}[\text{node}] = \mathbb{R}_G \times \mathbb{P}^{\leq k} \)

- a path \( p \) \textit{covers} a k-FS-TR \((r, f)\) when \( p \) covers the original TR \( r \) and \( f \) is the \( k \)-most enclosing language features of \( p \):

\[
\text{ext}^k_G(p |_{\text{call}}) = f
\]

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

The k-FS coverage criteria divide TRs using each combination of different language features. Especially, the k-FS coverage criteria with \( k \geq 2 \) are helpful to cover edge cases in JavaScript engines and transpilers because they are heavily optimized and handle even the same language features differently depending on which language optimized are used together. For example, a \textit{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 \texttt{for-in/of} statement. Actually, the GraalJS engine and the Babel transpiler contain conformance bugs and crashing bugs, respectively, that are reproducible only with a combination of a destructuring pattern and a \texttt{for-in/of} statement.\(^8\)

\(^7\)https://262.ecma-international.org/13.0/#sec-destructuring-assignment

\(^8\)https://github.com/oracle/graaljs/issues/656 and https://github.com/babel/babel/issues/15100
3.4 Feature-Call-Path-Sensitive (FCPS) Coverage Criteria

As explained in Section 2.3.2, a more fine-grained set of test requirements is necessary to distinguish different parts of the semantics in the same language feature. Thus, we define feature-call-path-sensitive (FCPS) coverage criteria as variants of FS coverage criteria. The core idea is to distinguish TRs using paths from paths that 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 \( f_p \in F_p \times P_G \mid \text{call} \), which consists of the innermost enclosing feature and a subsequence of the call-site stack \( p \mid \text{call} \) from the feature. We define the feature-call-path extractor \( \text{ext}_{F_p} : P_G \mid \text{call} \to (F_p \cup \{ \bot \}) \):

\[
\text{ext}_{F_p}([n_1, \ldots, n_m]) = \begin{cases} 
\bot & \text{if } m = 0 \\
(f, [n_m]) & \text{if } \text{feat}(n_m) = f \\
f_{cp} & \text{if } f_{cp} = \bot \\
(f, [n_0', \ldots, n'_i]) & \text{if } f_{cp} = (f, [n_0', \ldots, n'_i]) \land \exists i. \text{ s.t. } n_i' = n_m \\
(f, n \cdot n_m) & \text{if } f_{cp} = (f, n) 
\end{cases}
\]

where \( f_{cp} = \text{ext}_{F_p}([n_1, \ldots, n_{m-1}]) \). 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 \( p \mid \text{call} \). 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 \((\exists i. \text{ s.t. } n_i' = n_m)\).

**Definition 3.6 (Feature-Call-Path-Sensitive (FCPS) Coverage Criteria).** For a given \( C_G = (\mathbb{R}_G, \sim_\text{FCPS}) \), the feature-call-path-sensitive (FCPS) coverage criterion \( C_G^{\text{FCPS}} = (\mathbb{R}_G, \sim_\text{FCPS}) \) is defined as follows:

- the set of **feature-call-path-sensitive test requirements (FCPS-TRs)** \( \mathbb{R}_G^{\text{FCPS}} \) is a set of original TRs optionally with feature-call-paths: \( \mathbb{R}_G^{\text{FCPS}} = \mathbb{R}_G \times (F_p \cup \{ \bot \}) \)
- a path \( p \) **covers** a FCPS-TR \((r, f_{cp})\) when \( p \) covers the original TR \( r \) and \( f_{cp} \) is the feature-call-path extracted from \( p \): \( p \sim (r, f_{cp}) \iff p \sim r \land \text{ext}_{cp}(p \mid \text{call}) = f_{cp} \)

**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 5. If we use FS node coverage criterion \( C_G^{\text{FS[node]}} \), both tests \( t_0 \) and \( t_1 \) cover the same FS-TR \((21, f_{add})\), and one of them might be removed in the program pool. However, if we use FCPS node coverage criterion \( C_G^{\text{FCPS[O]}}, t_0 \) and \( t_1 \) cover different FCPS-TRs \((21, (f_{add}, [1, 8, 12])) \) and \((21, (f_{add}, [1, 8, 14])) \), respectively. The other example is about the cycles in the call-site stacks with the graph in Figure 6. It depicts an excerpt from the CFG of abstract algorithms in ES13 transitively used in \( f_{id} \), a syntactic feature defined by the first alternative of \( \text{IdentifierReference} \) and its Evaluation SDO. Assume that we do not remove cycles in the feature-call-paths \( F_p \) during the extraction algorithm \( \text{ext}_{F_p} \). Then, since the algorithm \( \text{IdentifierReference} \) contains a self-recursion, there exists an infinite number of possible feature-call-paths from \( f_{id} \) to the node labeled 29: \((f_{id}, [27, 28])\), \((f_{id}, [27, 28, 30])\), \((f_{id}, [27, 28, 30, 30])\), and so on. Thus, we remove cycles in feature-call-paths to resolve this issue, and there exists only two possible feature-call-paths: \((f_{id}, [27, 28])\) and \((f_{id}, [27, 28, 30])\).
Fig. 7. Subsumption relations between k-FS and k-FCPS coverage criteria

Similar to the extension of FS coverage criteria to k-FS coverage criteria, we define k-FCPS coverage criteria by extending \( \text{ext}_{\text{cp}} \) into \( \text{ext}_{\text{cp}}^k \):  

\[
\text{ext}_{\text{cp}}^k ([n_1, \ldots, n_m]) = \begin{cases} 
(\varepsilon, \varepsilon) & \text{if } k = 0 \lor m = 0 \\
(\bar{f} \cdot f, [n_m]) & \text{if } \text{feat}(n_m) = f \land \text{ext}_{\text{cp}}^{k-1}([n_1, \ldots, n_{m-1}]) = (\bar{f}, \_)
\end{cases}
\]

where \( \mathbb{P}_{\leq k} = \mathbb{P}_k \times \mathbb{P}_{\text{call}} \) is a set of extended feature-call-paths, and \( \bar{f}_{\text{cp}} = \text{ext}_{\text{cp}}^k ([n_1, \ldots, n_{m-1}]) \).

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

- the set of k-feature-call-path-sensitive test requirements (k-FCPS-TRs) \( \mathbb{R}_G^k \) is a set of original TRs with the extended feature-call-paths bounded by \( k: \mathbb{R}_G^k = \mathbb{R}_G \times \mathbb{P}_{\leq k} \)
- a path \( p \) covers a k-FCPS-TR \( (r, \bar{f}_{\text{cp}}) \) when \( p \) covers the original TR \( r \) and \( \bar{f}_{\text{cp}} \) is the extended feature-call-path extracted from \( p \) bounded by \( k \):

\[
p \sim (r, \bar{f}_{\text{cp}}) \iff p \sim r \land \text{ext}_{\text{cp}}^k (p |_{\text{call}}) = \bar{f}_{\text{cp}}
\]

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

**Lemma 3.8.** Consider two graph coverage criteria \( C_G = (\mathbb{R}_G, \sim) \) and \( C'_G = (\mathbb{R}'_G, \sim') \). If there exists a feasible TR \( r \in \mathbb{R}_G \) that satisfies the following condition for each feasible TR \( r' \in \mathbb{R}'_G \):

\[
\forall t \in T, t \sim r \Rightarrow t \sim' r'.
\]

Then, \( C_G \) subsumes \( C'_G \) (\( C_G \succ C'_G \)).

**Proof.** Assume \( T \vdash C_G \). For a given feasible TR \( r' \in \mathbb{R}'_G \), let \( r \in \mathbb{R}_G \) be the feasible TR satisfying (13). Then, there exists a test \( t \in T \) such that \( t \sim r \) because \( r \) is feasible and \( T \vdash C_G \). Finally, \( t \sim' r' \).  

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

\[
(A) C_G^{k,\text{FS}} \succ C_G^{(k-1),\text{FS}} \quad (B) C_G^{k,\text{FCPS}} \succ C_G^{(k-1),\text{FCPS}} \quad (C) C_G^{k,\text{FCPS}} \succ C_G^{k,\text{FS}}
\]

**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 \( (r, \bar{f}) \), there exists a test \( t \in T \) such that \( t \sim (r, \bar{f}) \) because \((r, \bar{f})\) is feasible. There exists a prefix path \( p \) of path_\!(\mathbb{R}_G)\!(t) \) such that \( p \sim (r, \bar{f}) \) (\( \vdash (5) \)). Then, a k-FS-TR \( (r, \text{ext}_G^k (p |_{\text{call}})) \) satisfies the condition (13) because of the inductive definition of \( \text{ext}_G^k \) in (10). Finally, k-FS coverage criteria subsume \((k-1)\)-FS coverage criteria.  

4 IMPLEMENTATION

This section introduces JESTfs, our JavaScript conformance test synthesizer supporting k-FS and k-FCPS coverage criteria, and explains how it can detect conformance bugs in JavaScript implementations. Figure 8 illustrates its overall structure. JESTfs takes 1) a mechanized specification extracted by ESMeta\textsuperscript{9} and 2) a coverage criterion $C_G$ and performs coverage-guided fuzzing using the CFG of the mechanized specification. It uses the following four modules from JEST [Park et al. 2021b]:

- **Seed Synthesizer**: As the first step, Seed Synthesizer automatically synthesizes a set of JavaScript programs as the initial program pool. It uses the JavaScript syntax described in the language specification to cover diverse alternatives in syntactic productions. JESTfs uses two existing synthesizers: 1) a non-recursive synthesizer and 2) a built-in synthesizer.

- **Target Selector**: To measure the coverage in the CFG, Target Selector extracts the execution path of each program in the pool by interpreting it using the abstract algorithms in the specification. While the baseline tool supports only a node-or-branch coverage criterion, we extend it to support k-FS and k-FCPS node-or-branch coverage criteria as well. If a program does not cover new TRs, it removes the program from the pool. Then, it selects a program in the pool as a mutation target that potentially increases the coverage or stops the iteration when the current status satisfies the termination condition.

- **Program Mutator**: To increase the coverage in the CFG, Program Mutator repeatedly tries to mutate a JavaScript program to a new one using mutation methods. JESTfs uses five mutation methods: 1) random mutation, 2) nearest syntax tree mutation, 3) string substitution, 4) object substitution, and 5) statement insertion.

- **Assertion Extractor**: After the mutation iteration, Assertion Extractor automatically extracts seven kinds of assertions from each program in the pool. The assertions represent the expected final state of each program according to the semantics described in the specification. As a result, each pair of a program and the corresponding extracted assertions is a conformance test for JavaScript.

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

\textsuperscript{9}https://github.com/es-meta/esmeta
Table 1. Detected conformance bugs in JavaScript engines and transpilers

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

5 EVALUATION

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

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

We apply JESTfs to the latest language specification (ES13, 2022) [ECMA International 2022a], 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.

5.1 Conformance Bug Detection

Table 1 summarizes the conformance bugs detected by JESTfs in all the evaluation targets. We manually inspected the failed conformance test cases, found 143 distinct conformance bugs, and reported them to the corresponding developers. As a result, 85 out of 143 bugs were officially confirmed, and 83 were newly discovered bugs. The other 47 reported bugs are still under review, or developers have not yet responded. Among 143 detected bugs, 42 are engine bugs, and 101 are transpiler bugs. We present two real-world bug examples.
Table 2. Comparison of synthesized conformance tests guided by five graph coverage criteria

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th># Covered k-F(CPS)-TR (k)</th>
<th># Syn. Test</th>
<th># Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Node</td>
<td># Branch</td>
<td># Total</td>
</tr>
<tr>
<td>0-FS node-or-branch (0-fs)</td>
<td>10.0</td>
<td>5.6</td>
<td>15.6</td>
</tr>
<tr>
<td>1-FS node-or-branch (1-fs)</td>
<td>79.3</td>
<td>45.7</td>
<td>125.0</td>
</tr>
<tr>
<td>2-FS node-or-branch (2-fs)</td>
<td>1,199.8</td>
<td>696.3</td>
<td>1,896.1</td>
</tr>
<tr>
<td>1-FCPS node-or-branch (1-fcps)</td>
<td>179.7</td>
<td>97.6</td>
<td>277.3</td>
</tr>
<tr>
<td>2-FCPS node-or-branch (2-fcps)</td>
<td>2,323.1</td>
<td>1,297.6</td>
<td>3,620.7</td>
</tr>
</tbody>
</table>

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

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

In addition, we detected another bug 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.

**Asynchronous Function / Generator.** One of the complex language features in JavaScript is asynchronous functions and generators introduced in ES6 (2015). We detected a bug 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:

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

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

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

### 5.2 Effectiveness of k-FS Coverage Criteria

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

The results show that higher k-FS coverage criteria are more effective than lower k-FS. On average, 8.03 (125.0K / 15.6K) 1-FS-TRs exist per each 0-FS-TR, and 15.17 (1,896.1K / 125.0K) 2-FS-TRs exist per each 1-FS-TR. It means that each node or branch is used in 8.03 different language features, and each language feature could be used in 15.17 other language features on average. For a more detailed information, we draw a histogram of the number of covered 1-FS-TRs (or 2-FS-TRs).

10[https://github.com/oracle/graaljs/issues/671](https://github.com/oracle/graaljs/issues/671)
11[https://github.com/tc39/ecma262/issues/2659](https://github.com/tc39/ecma262/issues/2659)
12[https://bugzilla.mozilla.org/show_bug.cgi?id=1799288](https://bugzilla.mozilla.mozilla.org/show_bug.cgi?id=1799288)
per each covered 0-FS-TR (or 1-FS-TR) in Figures 9 (a) and (b). The largest number of covered 1-FS-TRs per each covered 0-FS-TR is 303 for a node in the [[GetOwnProperty]] algorithm. In other words, this algorithm is used in 303 different language features, because the semantics of many syntactic or built-in API features use this algorithm to access object properties. The largest number of covered 2-FS-TRs per each covered 1-FS-TR is 116 for a node in the [[Comprehension]] algorithm. In other words, this algorithm 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.

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

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

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

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

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

However, the name property is "async" instead of "f" in the JSC engine. Since it is a combination of a class, an async method, and a computed property, 0 or 1-FS coverage criteria may not keep it in the final program pool. On the other hand, 2-FS coverage criterion can discriminate it with other tests. If a conformance test covers 2-FS-TR consisting of two syntactic features AsyncMethod production with PropMethodName 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.

---

13https://github.com/babel/babel/issues/15100
14https://github.com/babel/babel/issues/15100

5.3 Effectiveness of $k$-FCPS Coverage Criteria

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 9 (c) and (d). The largest number of covered 1-FCPS-TRs per 1-FS-TR is 70 for a node in the Array.prototype.splice built-in method. It is a powerful built-in API feature that changes the contents of an array 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. Now, we introduce a conformance bug that show the effectiveness of $k$-FCPS coverage criteria compared to the $k$-FS coverage criteria.

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

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

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

5.4 Comparison with Test262

We compare the coverage of automatically synthesized conformance tests with that of Test262, the official JavaScript conformance test suite. As described in Section 4, the baseline tool JEST relies on the mechanized specification extracted by ESMeta. Thus, we filter out conformance tests in Test262 that utilize language features not supported in the extracted mechanized specification. We use the conventional methodology in the literature [Fragoso Santos et al. 2018; Park et al. 2015, 2020a] to remove inapplicable tests in Test262. Then, we measured the coverage of 23,910 applicable Test262 conformance tests with five $k$-FS and $k$-FCPS node-or-branch coverage criteria.
Figure 10 shows (a) Venn diagrams of the numbers of covered $k$-FS-TRs and $k$-FCPS-TRs for the synthesized conformance tests (syn-test) via JEST$_{fs}$ and applicable conformance tests in Test262 (test262) and (b) their changes over time. Without any feature-sensitive coverages, the coverage of synthesized tests is less than that of Test262, and only 5.2% (0.9K) 0-FS-TRs are newly covered by the synthesized tests. On the other hand, the numbers of $k$-FS-TRs covered by only synthesized tests increase when using higher $k$: 28.7K (19.0%) for 1-FS-TRs and 1.65M (80.8%) for 2-FS-TRs. In addition, the number of 1-FCPS-TRs (or 2-FCPS-TRs) covered by only synthesized tests is higher than the number of 1-FS-TRs (or 2-FS-TRs) covered by only synthesized tests: 80.0K (21.6%) for 1-FCPS-TRs and 3.26M (82.0%) for 2-FCPS-TRs. Figure 10(b) also shows that the coverage of Test262 is better than that of synthesized tests without any feature-sensitive coverages, but the coverage of synthesized tests outperforms that of Test262 with 2-FS-TRs and 2-FCPS-TRs.

6 RELATED WORK

Coverage Criteria in Software Testing. Coverage criteria in software testing are essential in measuring the quality of test inputs. The most common coverage criteria are structural coverages in a given program’s control-flow graph (CFG) [Ammann and Offutt 2008; Chilenski and Miller 1994] also utilizing data-flow information [Herman 1976] and type information [Bae et al. 2017]. On the other hand, model-based coverage [Utting and Legeard 2010] criteria consider specialized abstract behavior models and define the test requirements in the model. Such models include state transitions [Artho et al. 2017], autonomous driving systems [Laurent et al. 2022], deep neural networks (DLNs) [Ma et al. 2018; Odena et al. 2019; Pei et al. 2017; Sun et al. 2018; Xie et al. 2019]. However, there are no specialized coverage criteria for programming language tools, such as compilers, interpreters, and transpilers. In this paper, we first presented feature-sensitive coverages as general extensions of graph coverages for programming language tools to discriminate test requirements based on enclosing language features or feature-call-paths.

Mechanized Specification. Researchers have presented mechanized specifications to formally describe the semantics of diverse programming languages, such as OCaml [Owens 2008], C [Blazy and Leroy 2009], C++ [Ramananandro et al. 2012], Java [Bogdan and Rosu 2015], and POSIX shell [Greenberg and Blatt 2019]. At the same time, general metalanguages or frameworks for mechanized language specifications have also emerge as. For example, Sewell et al. [2010] 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 $\mathcal{K}$ framework [Rosu and Serbanuta 2010] proposed a formalism for writing operational semantics and provides a derivation of verifiers directly from the semantics. Bodin et al. [2019] developed a skeletal semantics framework in Coq for creating big-step semantics by focusing on the structure of the semantics.
For JavaScript, diverse mechanized specifications have been presented based on ECMA-262 [ECMA International 2022a]. KJS [Park et al. 2015] utilizes the K framework, and Fragoso Santos et al. [2018] presented a metalanguage, JSIL, for ES5.1. Researchers have used such mechanized specification in diverse fields: verification [Fragoso Santos et al. 2018], symbolic execution [Fragoso Santos et al. 2019], abstract interpretation-based static analysis [Jensen et al. 2009; Kashyap et al. 2014; Lee et al. 2012; Park et al. 2017; Schäfer et al. 2013], and double debugger [Charguéraud et al. 2018]. However, most JavaScript mechanized specifications focused on only ES5.1, released in 2011, and required manual description of the semantics. On the other hand, ESMeta supports a metalanguage IRES for the latest version of ECMA-262 and the automatic extraction of mechanized specification used in conformance test synthesis [Park et al. 2021b], specification type analysis [Park et al. 2021a], and static analyzer derivation [Park et al. 2022]. Hence, we implemented JESTfs based on ESMeta to synthesize conformance tests from the latest version (ES13, 2022) 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 [ECMA International 2022b], the official JavaScript conformance test suite. Researchers have focused on testing JavaScript engines to detect bugs using generation-based fuzzing [Dinh et al. 2021; Han, HyungSeok and Oh, DongHyeon and Cha, Sang Kil 2019; He et al. 2021] and mutation-based fuzzing [Park et al. 2020b; Veggalam et al. 2016; Wang et al. 2019]. In addition, they often utilize deep learning [Lee, Suyoung and Han, HyungSeok and Cha, Sang Kil and Son, Sooel 2020; Ye et al. 2021] to generate JavaScript programs in advance and differential testing [Bernhard et al. 2022] to check the correctness of execution results. However, most existing techniques focused on detecting crashing bugs or security vulnerabilities rather than conformance bugs. While COMFORT [Ye et al. 2021] targets conformance bugs, it heavily relies on the results of differential testing instead of the language semantics. On the other hand, JEST [Park et al. 2021b] is the first tool that automatically synthesizes JavaScript conformance tests according to the language semantics described in the language specification. We implemented JESTfs by augmenting it with k-FS and k-FCPS coverages and outperformed the ability of conformance bug detection of the JEST.

7 CONCLUSION

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

ACKNOWLEDGMENTS

This research was supported by National Research Foundation of Korea (NRF) (2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460 and 2023-2020-0-01819), and Samsung Electronics Co., Ltd (G01210570).
DATA-AVAILABILITY STATEMENT

The source code of JESTfs and the dataset of our study are publicly available at [Park et al. 2023]. The latest version of the artifact is maintained as an open-source project as a GitHub repository at https://github.com/jestfs/jestfs.

REFERENCES


Received 2022-11-10; accepted 2023-03-31