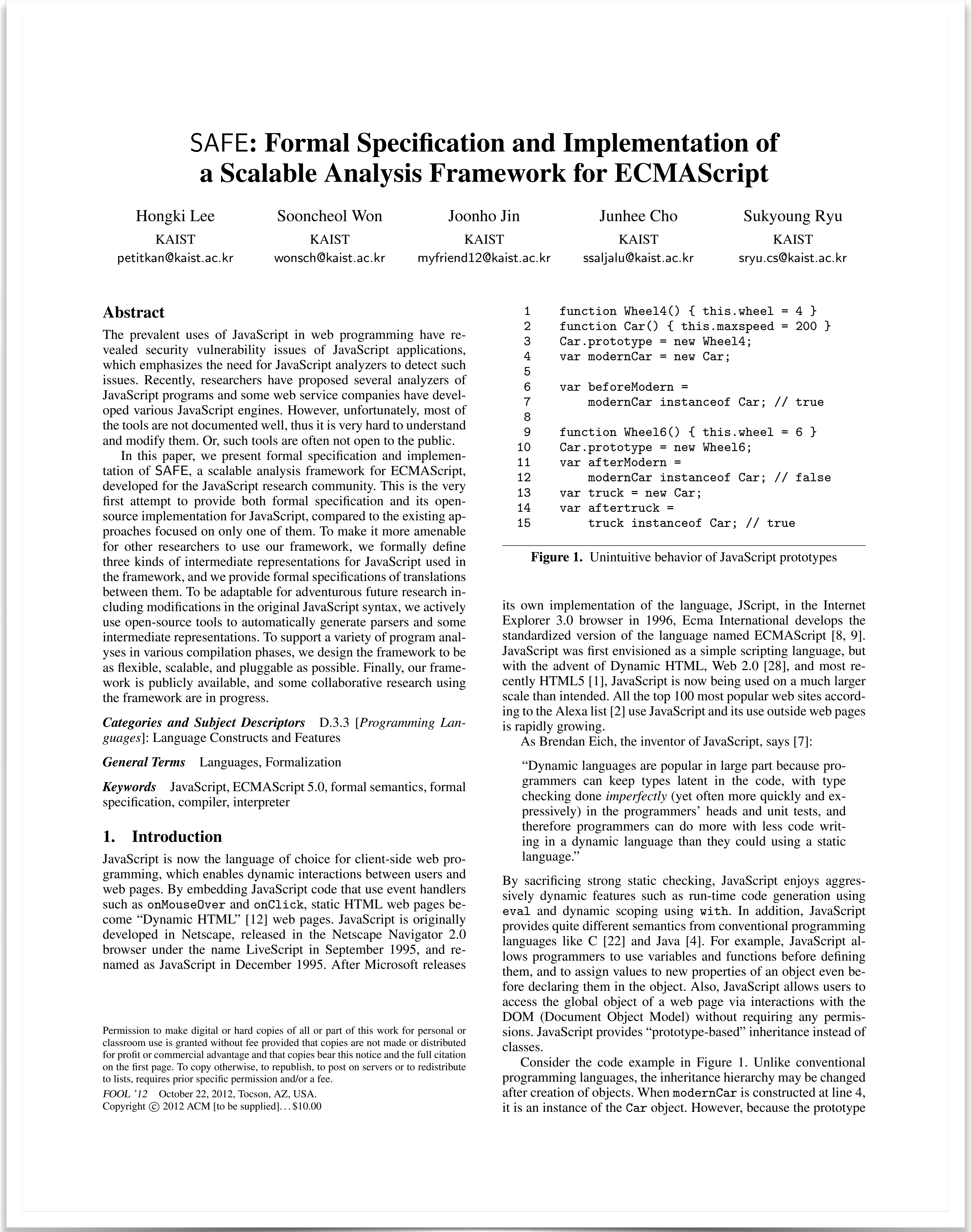
JavaScript Static Analysis for Evolving Language Specifications

Talk @ Agoric

Jihyeok Park

PLRG @ KAIST

January 5, 2022



SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript

Hongki Lee, KAIST, petitkan@kaist.ac.kr
Sooncheol Won, KAIST, wonsch@kaist.ac.kr
Joonho Jin, KAIST, myfriend12@kaist.ac.kr
Junhee Cho, KAIST, ssaljalu@kaist.ac.kr
Sukyoung Ryu, KAIST, sryu.cs@kaist.ac.kr

Abstract

The prevalent uses of JavaScript in web programming have revealed security vulnerability issues of JavaScript applications, which emphasizes the need for JavaScript analyzers to detect such issues. Recently, researchers have proposed several analyzers of JavaScript programs and some web service companies have developed various JavaScript engines. However, unfortunately, most of the tools are not documented well, thus it is very hard to understand and modify them. Or, such tools are often not open to the public.

In this paper, we present formal specification and implementation of SAFE, a scalable analysis framework for ECMAScript, developed for the JavaScript research community. This is the very first attempt to provide both formal specification and its open-source implementation for JavaScript, compared to the existing approaches focused on only one of them. To make it more amenable for other researchers to use our framework, we formally define three kinds of intermediate representations for JavaScript used in the framework, and we provide formal specifications of translations between them. To be adaptable for adventurous future research including modifications in the original JavaScript syntax, we actively use open-source tools to automatically generate parsers and some intermediate representations. To support a variety of program analyses in various compilation phases, we design the framework to be as flexible, scalable, and pluggable as possible. Finally, our framework is publicly available, and some collaborative research using the framework are in progress.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Formalization

Keywords JavaScript, ECMAScript 5.0, formal semantics, formal specification, compiler, interpreter

1. Introduction

JavaScript is now the language of choice for client-side web programming, which enables dynamic interactions between users and web pages. By embedding JavaScript code that use event handlers such as `onMouseOver` and `onClick`, static HTML web pages become "Dynamic HTML" [12] web pages. JavaScript is originally developed in Netscape, released in the Netscape Navigator 2.0 browser under the name LiveScript in September 1995, and renamed as JavaScript in December 1995. After Microsoft releases its own implementation of the language, JScript, in the Internet Explorer 3.0 browser in 1996, Ecma International develops the standardized version of the language named ECMAScript [8, 9]. JavaScript was first envisioned as a simple scripting language, but with the advent of Dynamic HTML, Web 2.0 [28], and most recently HTML5 [1], JavaScript is now being used on a much larger scale than intended. All the top 100 most popular web sites according to the Alexa list [2] use JavaScript and its use outside web pages is rapidly growing.

```
1   function Wheel4() { this.wheel = 4 }
2   function Car() { this.maxspeed = 200 }
3   Car.prototype = new Wheel4;
4   var modernCar = new Car;
5
6   var beforeModern =
7       modernCar instanceof Car; // true
8
9   function Wheel6() { this.wheel = 6 }
10  Car.prototype = new Wheel6;
11  var afterModern =
12      modernCar instanceof Car; // false
13  var truck = new Car;
14  var aftertruck =
15      truck instanceof Car; // true
```

Figure 1. Unintuitive behavior of JavaScript prototypes

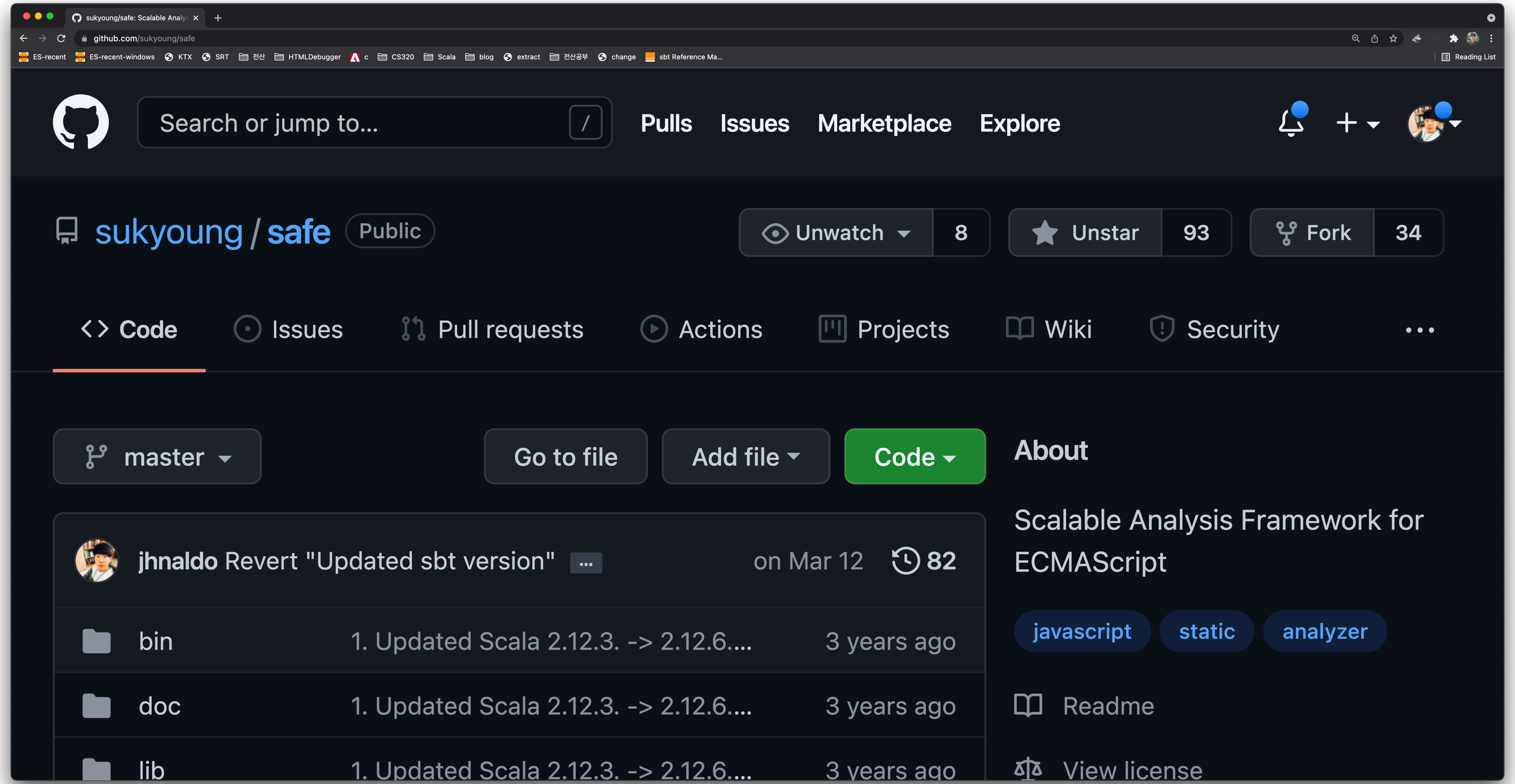
As Brendan Eich, the inventor of JavaScript, says [7]:

> "Dynamic languages are popular in large part because programmers can keep types latent in the code, with type checking done *imperfectly* (yet often more quickly and expressively) in the programmers' heads and unit tests, and therefore programmers can do more with less code writing in a dynamic language than they could using a static language."

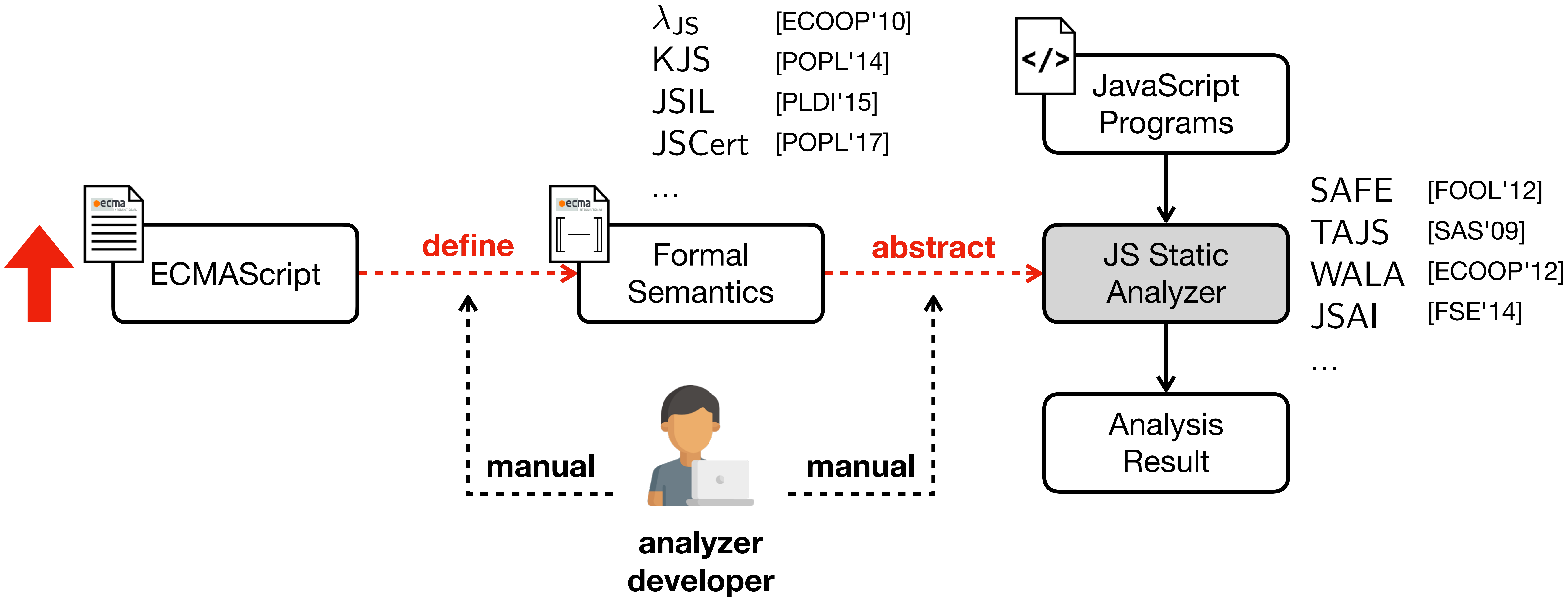
By sacrificing strong static checking, JavaScript enjoys aggressively dynamic features such as run-time code generation using `eval` and dynamic scoping using `with`. In addition, JavaScript provides quite different semantics from conventional programming languages like C [22] and Java [4]. For example, JavaScript allows programmers to use variables and functions before defining them, and to assign values to new properties of an object even before declaring them in the object. Also, JavaScript allows users to access the global object of a web page via interactions with the DOM (Document Object Model) without requiring any permissions. JavaScript provides "prototype-based" inheritance instead of classes.

Consider the code example in Figure 1. Unlike conventional programming languages, the inheritance hierarchy may be changed after creation of objects. When `modernCar` is constructed at line 4, it is an instance of the `Car` object. However, because the prototype

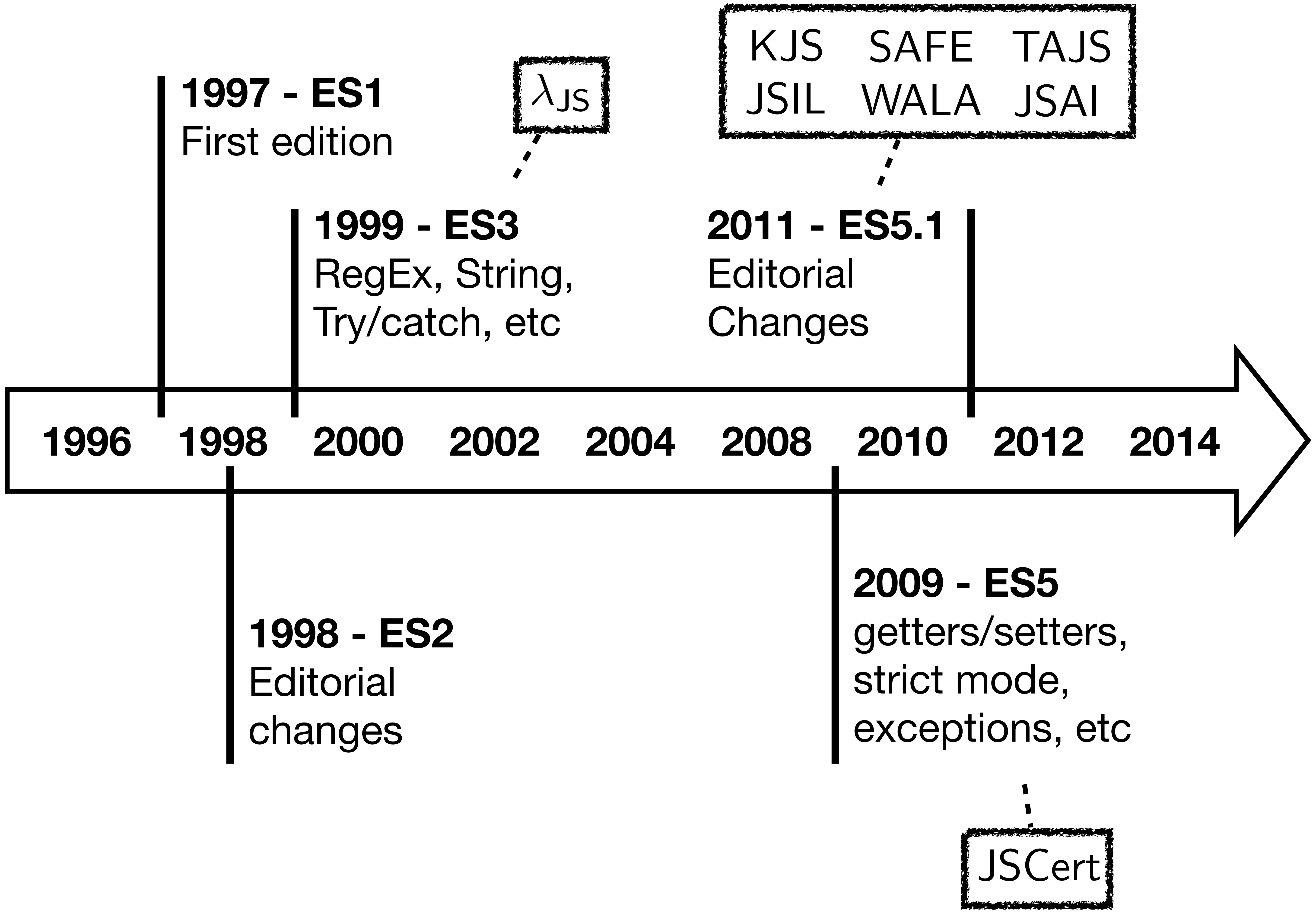
FOOL '12 October 22, 2012, Tocson, AZ, USA.



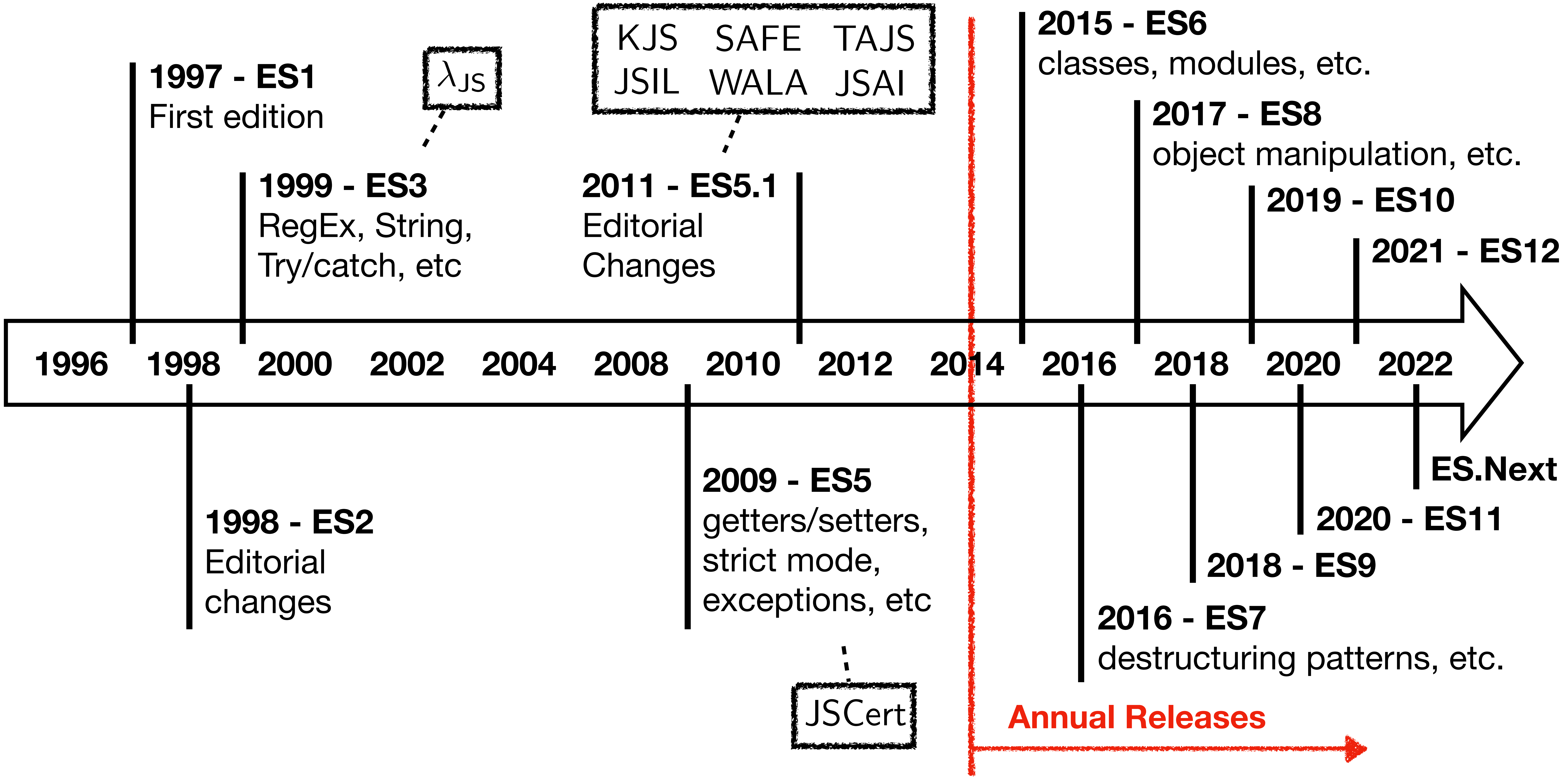
Problem: Manual JavaScript Static Analyzer



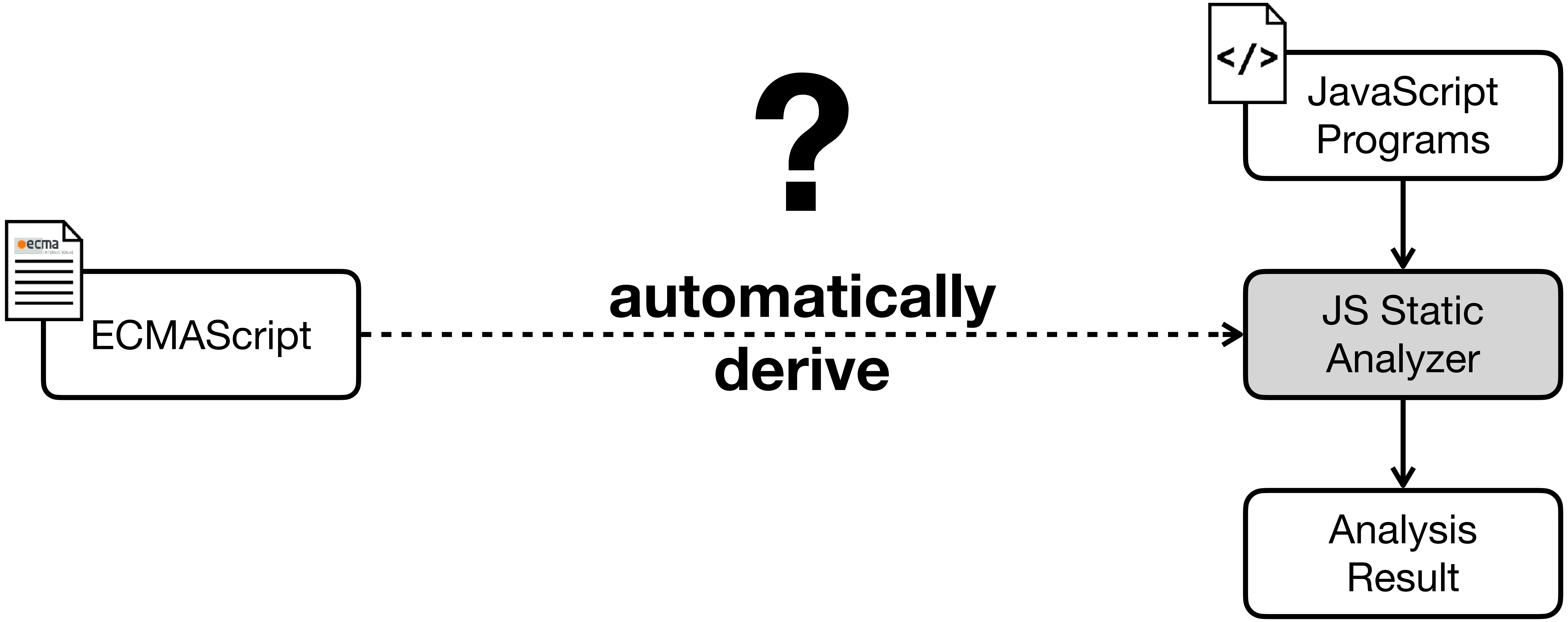
Problem: Fast Evolving JavaScript



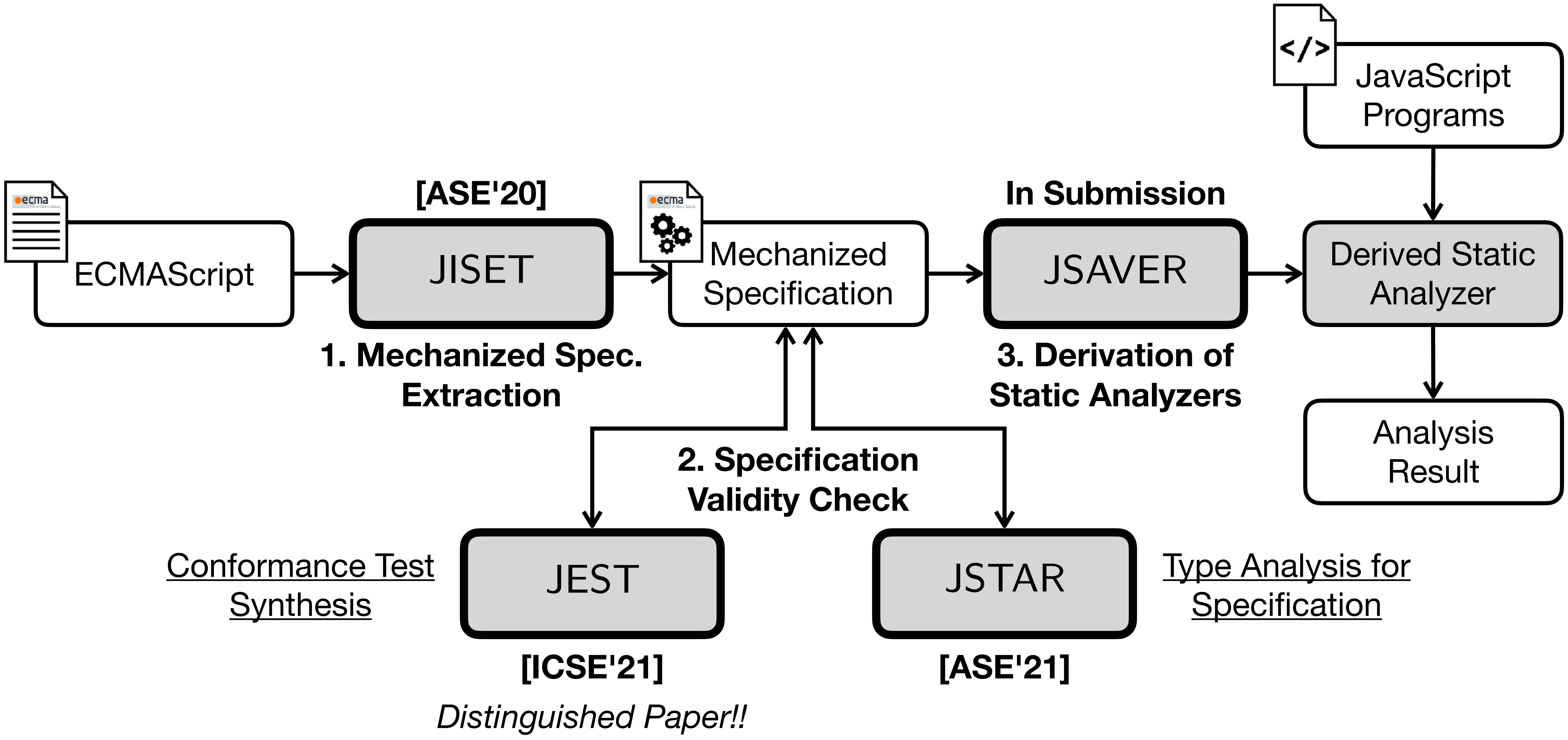
Problem: Fast Evolving JavaScript



Main Idea: Deriving Static Analyzer from Spec.



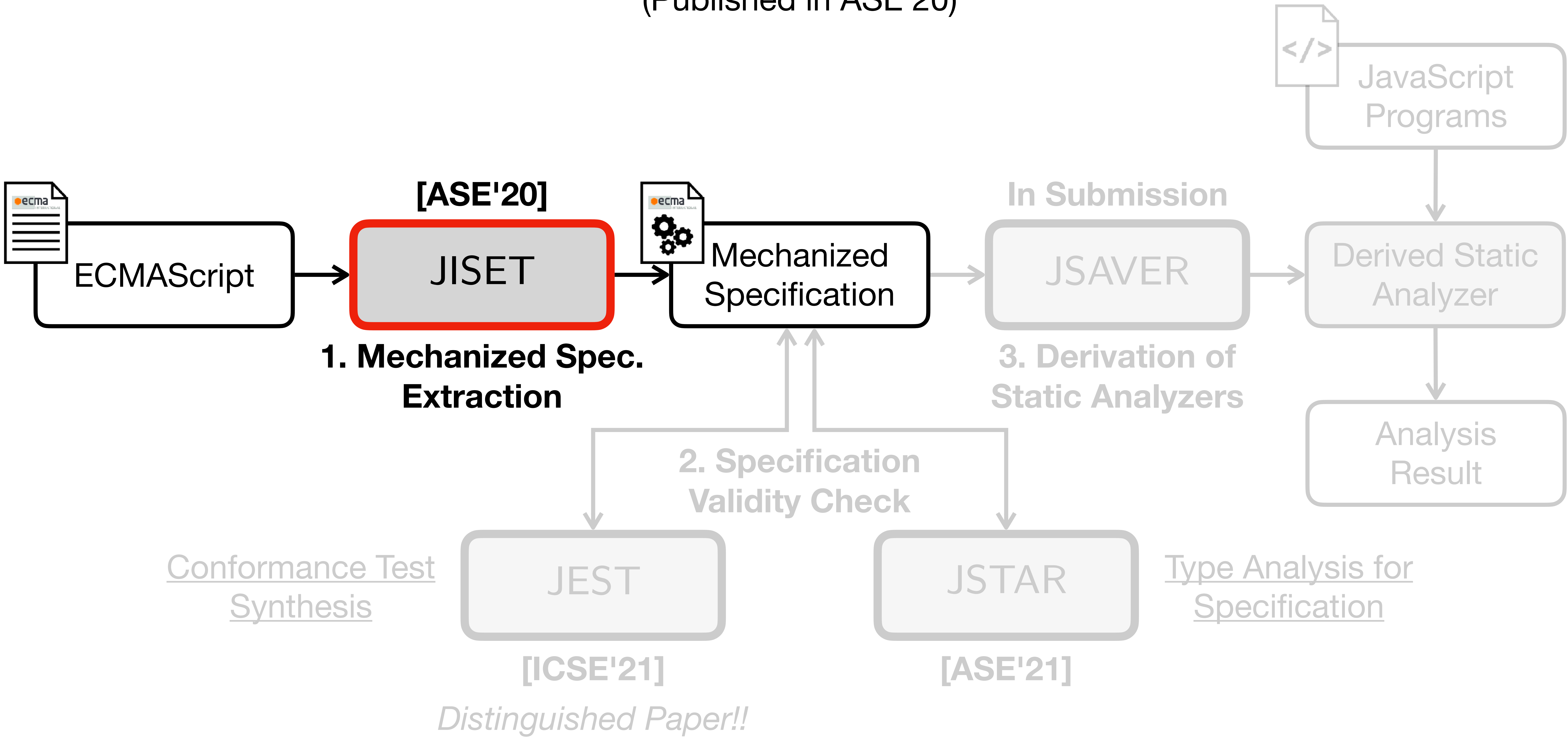
Overall Structure



JISET: JavaScript IR-based Semantics Extraction Toolchain

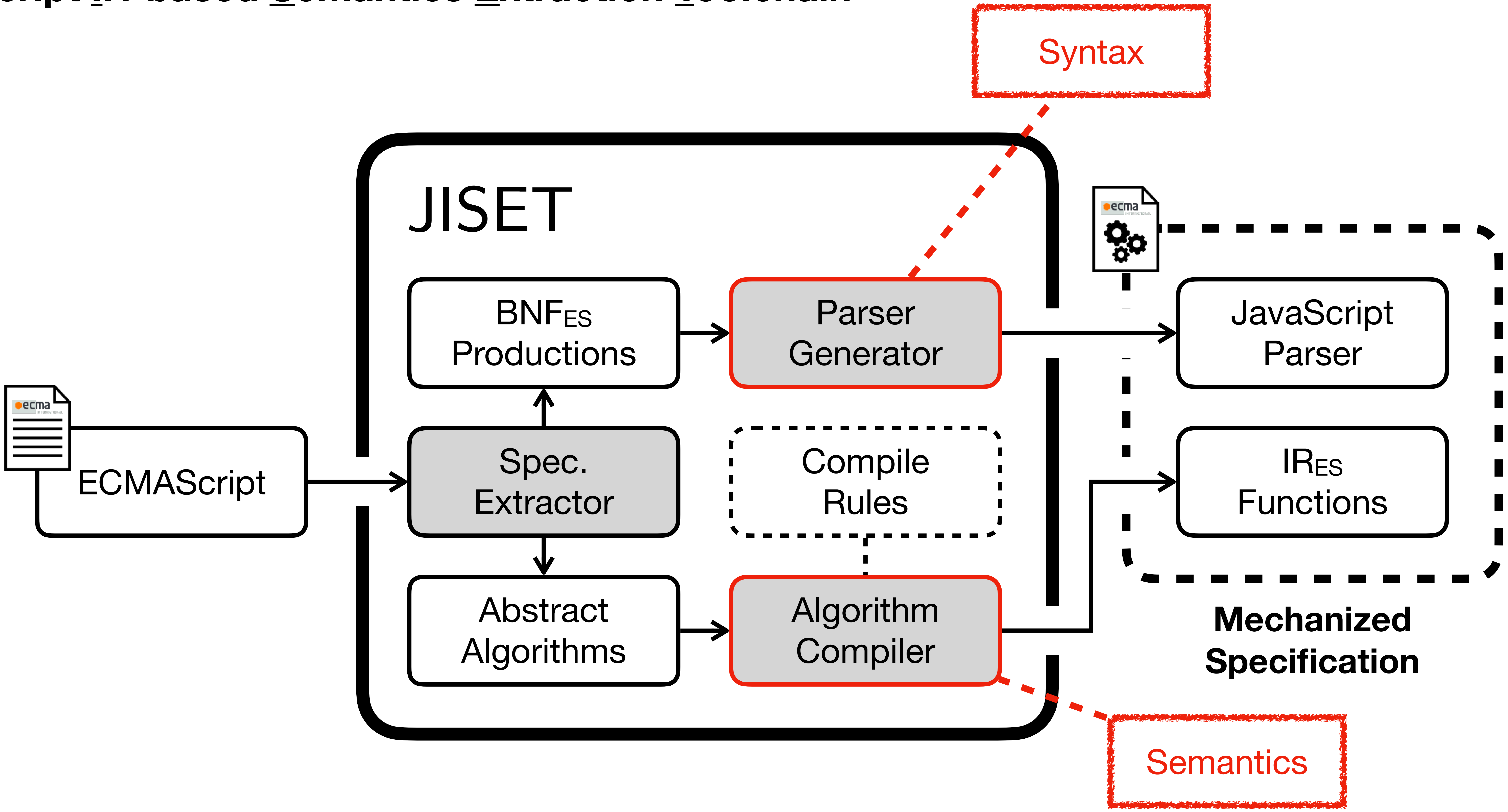
Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu

(Published in ASE'20)



JISET [ASE'20]

JavaScript IR-based Semantics Extraction Toolchain



JISET - Parser Generator (Syntax)

ArrayLiteral[Yield, Await] :

[*Elision*opt]

[*ElementList*[?Yield, ?Await]]

[*ElementList*[?Yield, ?Await] , *Elision*opt]

Parsing Expression Grammar
(+ Lookahead Parsing)

```
val ArrayLiteral: List[Boolean] => LAParser[T] = memo {
  case List(Yield, Await) =>
  "[" ~ opt(Elision) ~ "]"                            ^^ ArrayLiteral0 |
  "[" ~ ElementList(Yield,Await) ~ "]" ^^ ArrayLiteral1 |
  "[" ~ ElementList(Yield,Await) ~ ","
      ~ opt(Elision)~ "]"                             ^^ ArrayLiteral2
}
```

(POPL'04) Bryan Ford, "Parsing Expression Grammars: A Recognition-based Syntactic Foundation"

JISET - Algorithm Compiler (Semantics)

13.2.5.2 Runtime Semantics: Evaluation

ArrayLiteral : `[` *ElementList* `,` *Elision*$_{opt}$ `]`

1. Let *array* be ! ArrayCreate(0).
2. Let *nextIndex* be the result of performing ArrayAccumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*nextIndex*).
4. If *Elision* is present, then
   a. Let *len* be the result of performing ArrayAccumulation for *Elision* with arguments *array* and *nextIndex*.
   b. ReturnIfAbrupt(*len*).
5. Return *array*.

118 Compile Rules for Steps in Abstract Algorithms

```
syntax def ArrayLiteral[2].Evaluation(
  this, ElementList, Elision
) {
  let array = [! (ArrayCreate 0)]
  let nextIndex = (ElementList.ArrayAccumulation array 0)
  [? nextIndex]
  if (! (= Elision absent)) {
    let len = (Elision.ArrayAccumulation array nextIndex)
    [? len]
  }
  return array
}
```

Parsing rules **Conversion Rules**

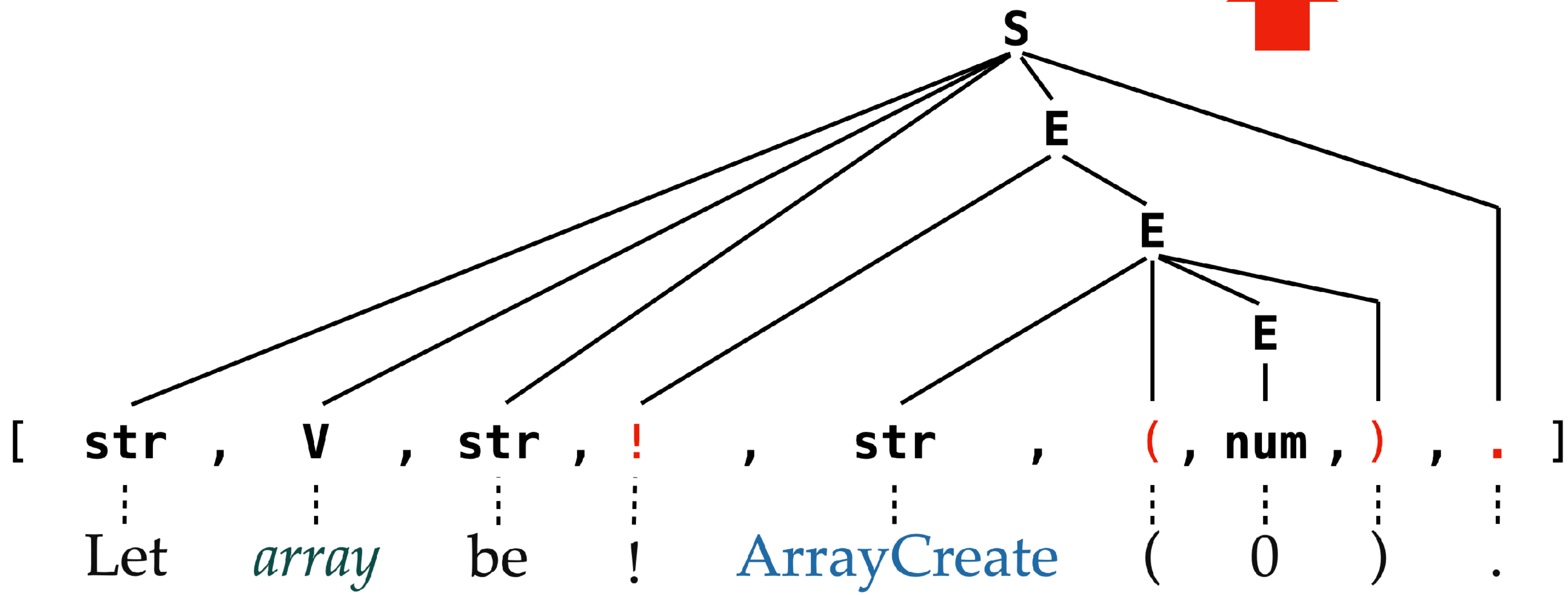
```
S = // statements
  Let ~ V ~ be ~ E ~ .  ^^ ILet

E = // expressions
  ! E                   ^^ EAbruptCheck |
  str ~ ( ~ E ~ )       ^^ ECall        |
  num                   ^^ _.toDouble
```

Simplified compile rules

```
let array = ! (ArrayCreate 0)
```

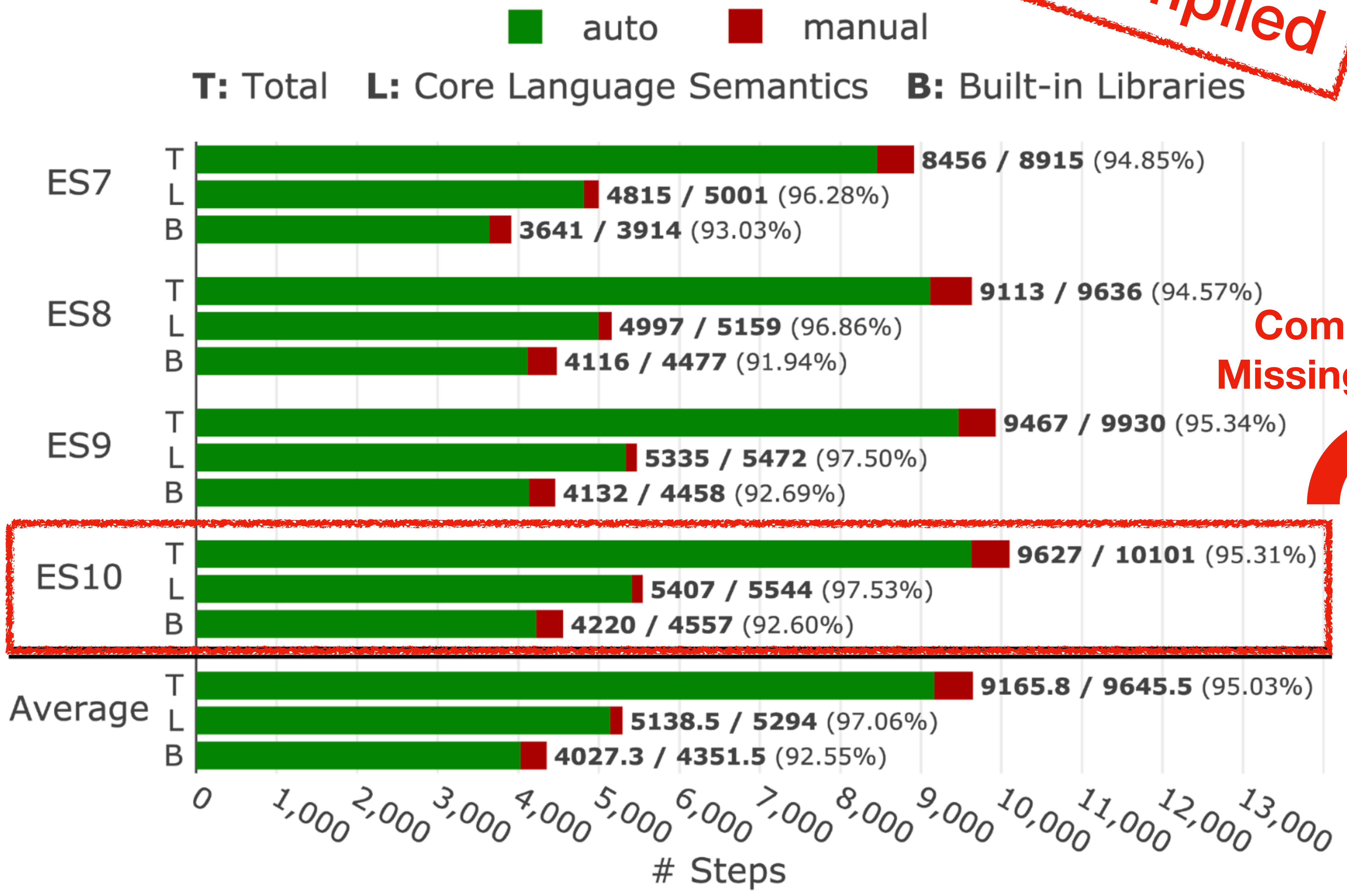
```
ILet(array, EAbruptCheck(
  ECall("ArrayCreate", 0)))
```



JISET - Evaluation

≈ 95% Compiled

Passed All Tests



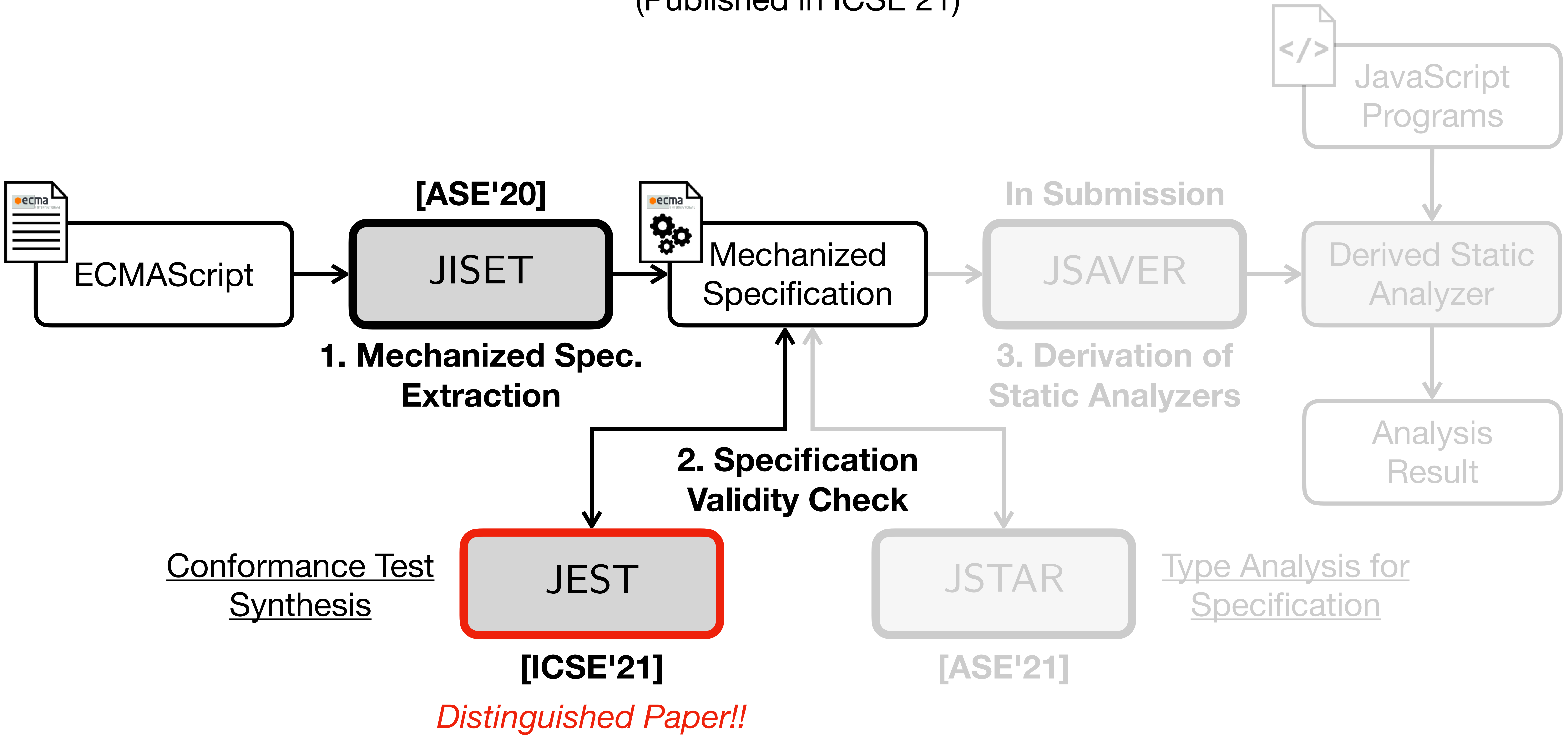
Complete Missing Parts

- **Test262** (Official Conformance Tests)
  - 18,064 applicable tests
- **Parsing tests**
  - Passed all 18,064 tests
- **Evaluation Tests**
  - Passed all 18,064 tests

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

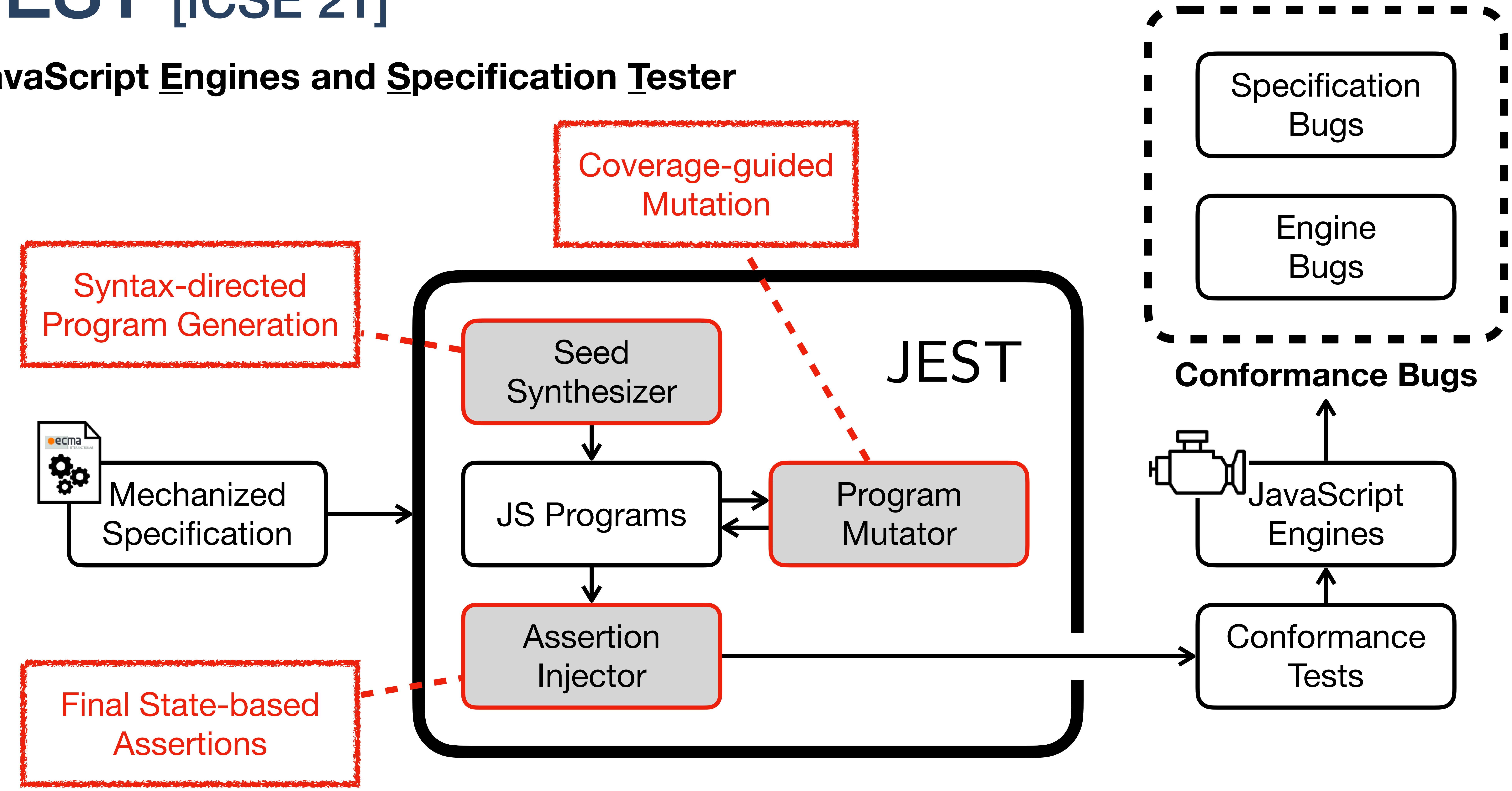
Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu

(Published in ICSE'21)



JEST [ICSE'21]

JavaScript Engines and Specification Tester



JEST - Assertion Injector (7 Kinds)

```
  var x = 1 + 2;

+ $assert.sameValue(x, 3);
```

JEST - Assertion Injector (7 Kinds)

1. **Exceptions (`Exc`)**

```
+ // Throw
  let x = 42;
  function x() {};
```

2. **Aborts (`Abort`)**

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

3. **Variable Values (`Var`)**

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

4. **Object Values (`Obj`)**

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

JEST - Assertion Injector (7 Kinds)

5. **Object Properties** (`Desc`)

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

---

6. **Property Keys** (`Key`)

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

---

7. **Internal Methods and Slots** (`In`)

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

JEST - Evaluation

- **JEST successfully synthesized 1,700 conformance tests from ES11**

44 Bugs in Engines

TABLE II: The number of engine bugs detected by JEST

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

27 Bugs in Spec.

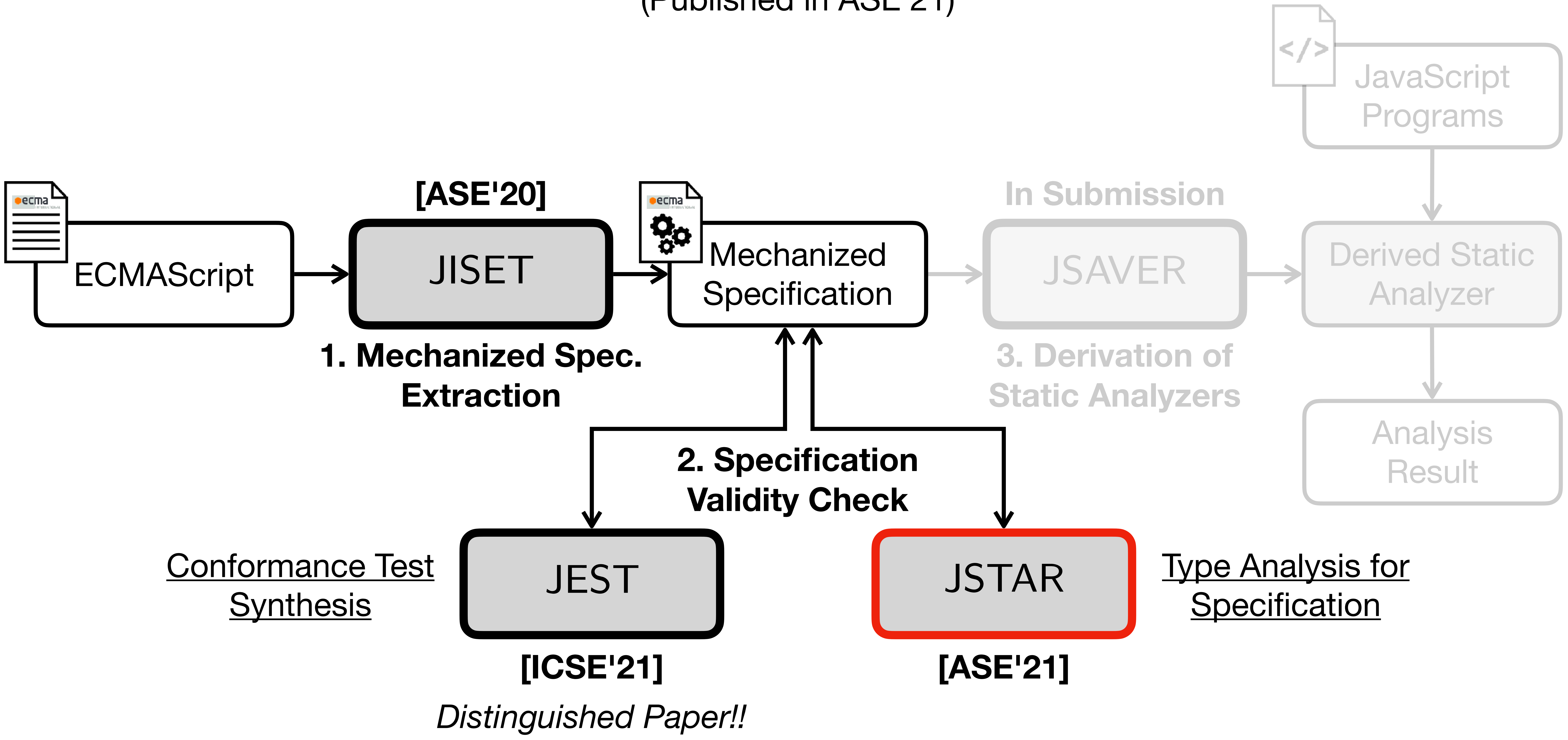
TABLE III: Specification bugs in ECMAScript 2020 (ES11) detected by JEST

Name	Feature	#	Assertion	Known	Created	Resolved	Existed
ES11-1	Function	12	Key	O	2019-02-07	2020-04-11	429 days
ES11-2	Function	8	Key	O	2015-06-01	2020-04-11	1,776 days
ES11-3	Loop	1	Exc	O	2017-10-17	2020-04-30	926 days
ES11-4	Expression	4	Abort	O	2019-09-27	2020-04-23	209 days
ES11-5	Expression	1	Exc	O	2015-06-01	2020-04-28	1,793 days
ES11-6	Object	1	Exc	X	2019-02-07	2020-11-05	637 days

JSTAR: JavaScript Specification Type Analyzer using Refinement

Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu

(Published in ASE'21)



JSTAR - Types in Specification

20.3.2.28 Math.round ($\boxed{x}$) `x: (String ∨ Boolean ∨ Number ∨ Object ∨ ...)`

1. Let $\boxed{n}$ be ? ToNumber(x). `n: (Number) ∧ ToNumber(x): (Number ∨ Exception)`
2. If n is an integral Number, return n.
3. If $\boxed{x < 0.5}$ and $\boxed{x > 0}$, return **+0**.
4. If $\boxed{x < 0}$ and $\boxed{x \geq \text{-}0.5}$, return **-0**.

...

Type Mismatch for numeric operator `>`

```
Math.round(true)  = ???
Math.round(false) = ???
```

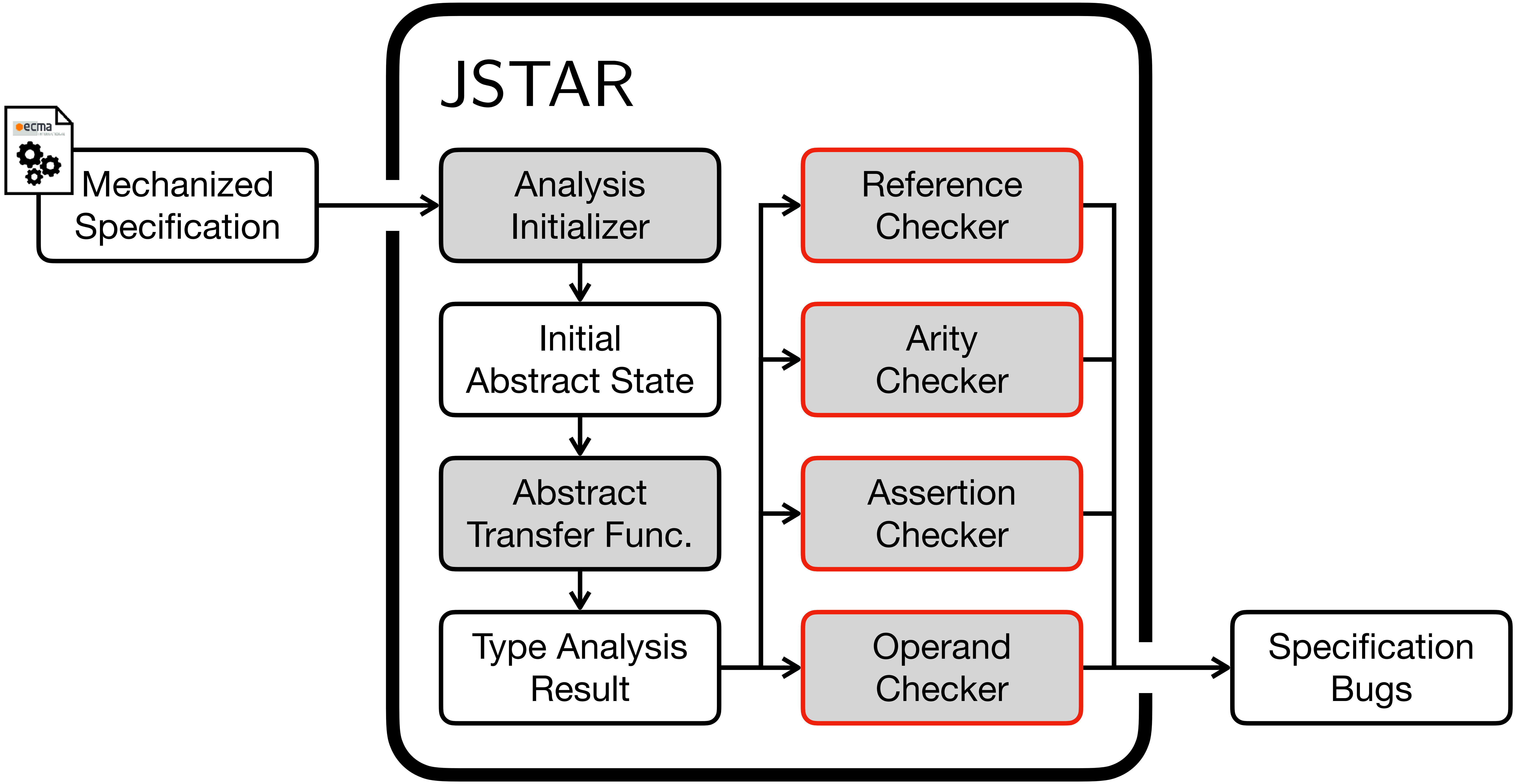
3. If $n < 0.5$ and $n > 0$, return **+0**.
4. If $n < 0$ and $n \geq \text{-}0.5$, return **-0**.

```
Math.round(true)  = 1
Math.round(false) = 0
```

https://github.com/tc39/ecma262/tree/575149cfd77aebcf3a129e165bd89e14caafc31c

JSTAR [ASE'21]

JavaScript Specification Type Analyzer using Refinement



JSTAR - Evaluation

- **Type Analysis for 864 versions of ECMAScript**

93 bugs detected

Checker	Bug Kind	Precision = (# True Bugs) / (# Detected Bugs)					
		no-refine		refine		Δ	
Reference	UnknownVar	62 / 106	17 / 60	63 / 78	17 / 31	+1 / -28	/ -29
	DuplicatedVar		45 / 46		46 / 47		+1 / +1
Arity	MissingParam	4 / 4	4 / 4	4 / 4	4 / 4	/	/
Assertion	Assertion	4 / 56	4 / 56	4 / 31	4 / 31	/ -25	/ -25
Operand	NoNumber	22 / 113	2 / 65	22 / 44	2 / 6	/ -69	/ -59
	Abrupt		20 / 48		20 / 38		/ -10
Total		92 / 279 (33.0%)		93 / 157 (59.2%)		+1 / -122 (+26.3%)	

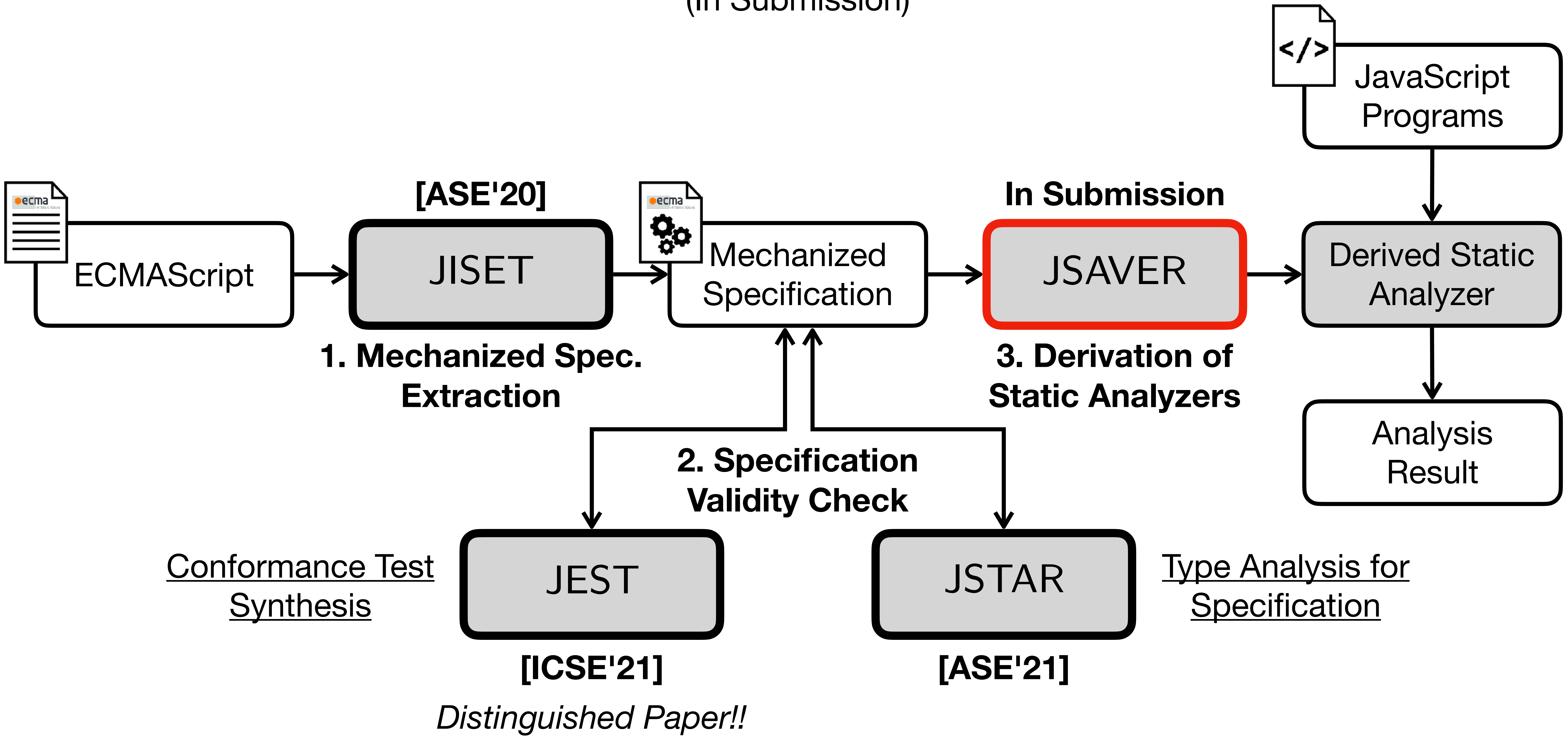
14 Bugs in Spec.

Name	Feature	#	Checker	Created	Life Span
ES12-1	Switch	3	Reference	2015-09-22	1,996 days
ES12-2	Try	3	Reference	2015-09-22	1,996 days
ES12-3	Arguments	1	Reference	2015-09-22	1,996 days
ES12-4	Array	2	Reference	2015-09-22	1,996 days
ES12-5	Async	1	Reference	2015-09-22	1,996 days
ES12-6	Class	1	Reference	2015-09-22	1,996 days
ES12-7	Branch	1	Reference	2015-09-22	1,996 days
ES12-8	Arguments	2	Operand	2015-12-16	1,910 days

Automatically Deriving JavaScript Static Analyzers from Language Specifications

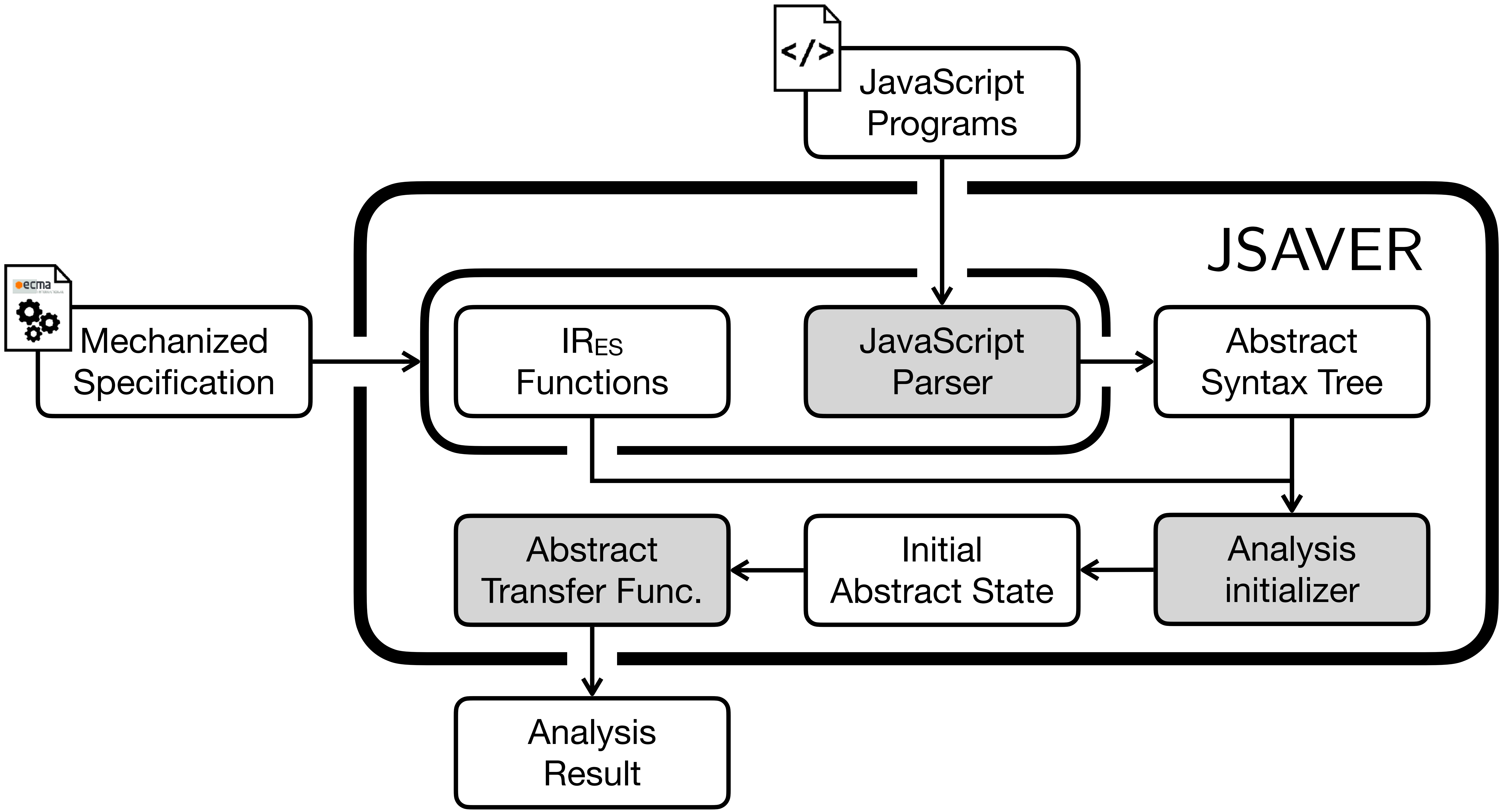
Jihyeok Park, Seungmin An, and Sukyoung Ryu

(In Submission)

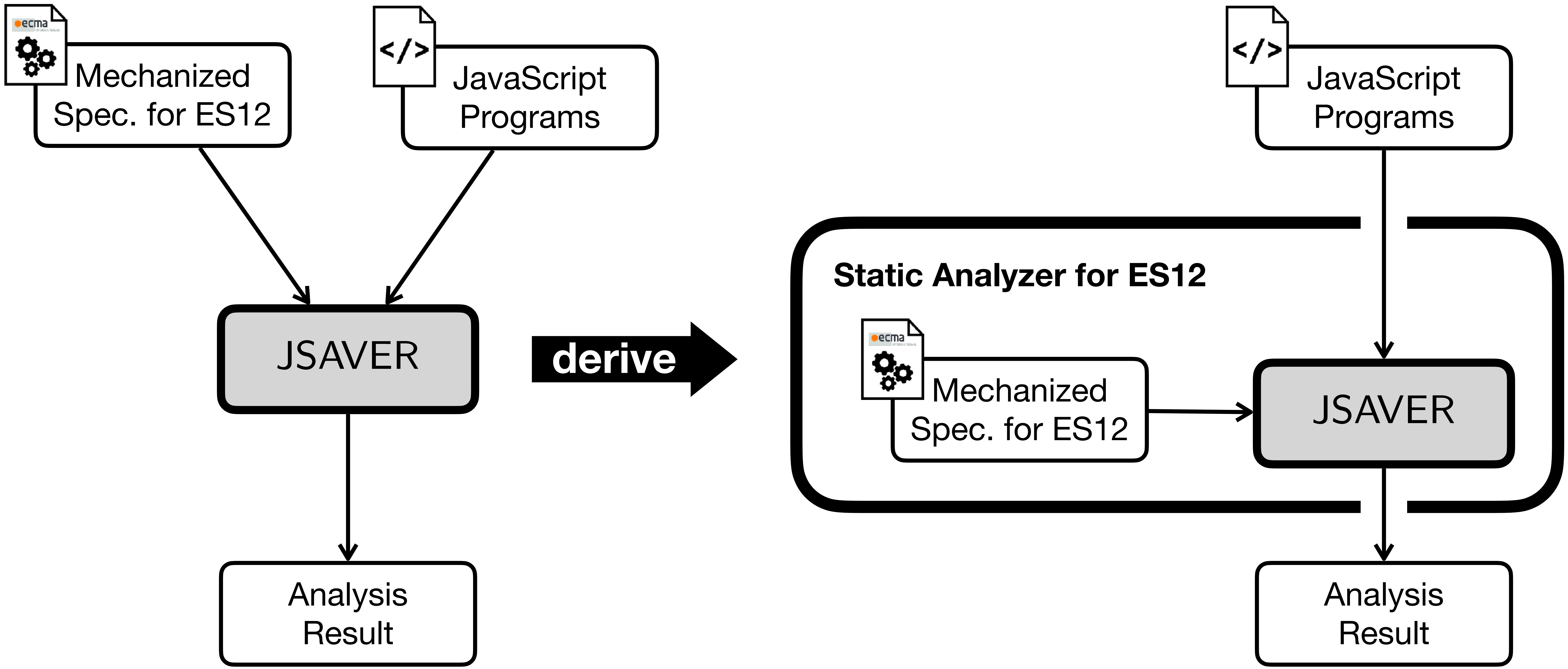


JSAVER [In Submission]

JavaScript Static Analyzer via ECMAScript Representation

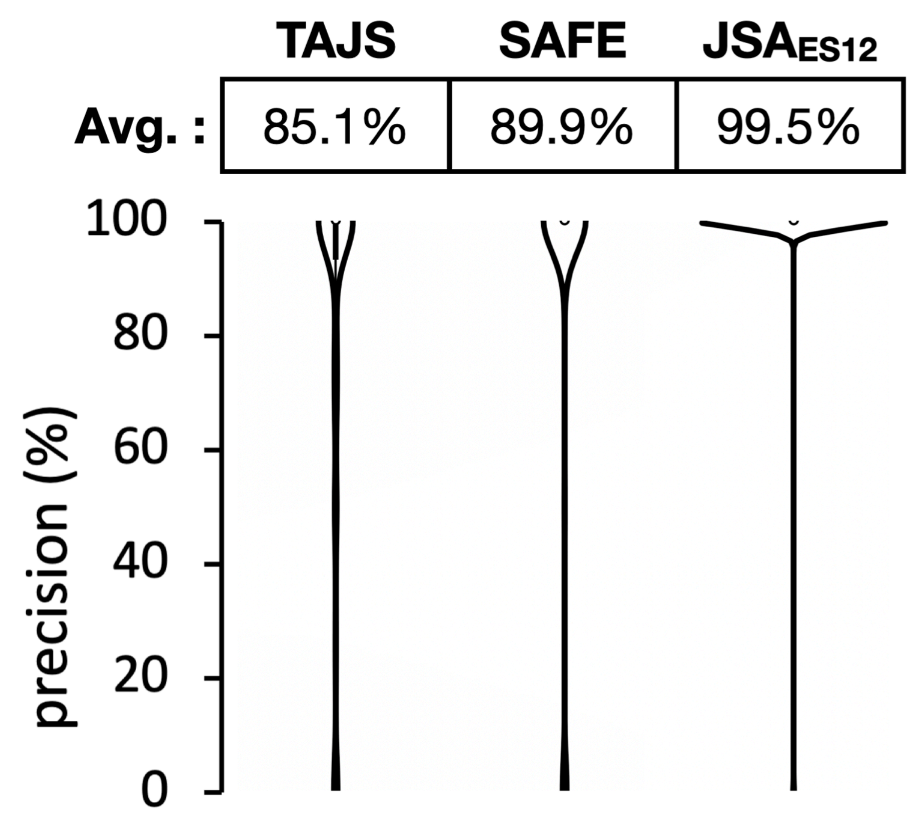


JSAVER - Static Analyzer Derivation

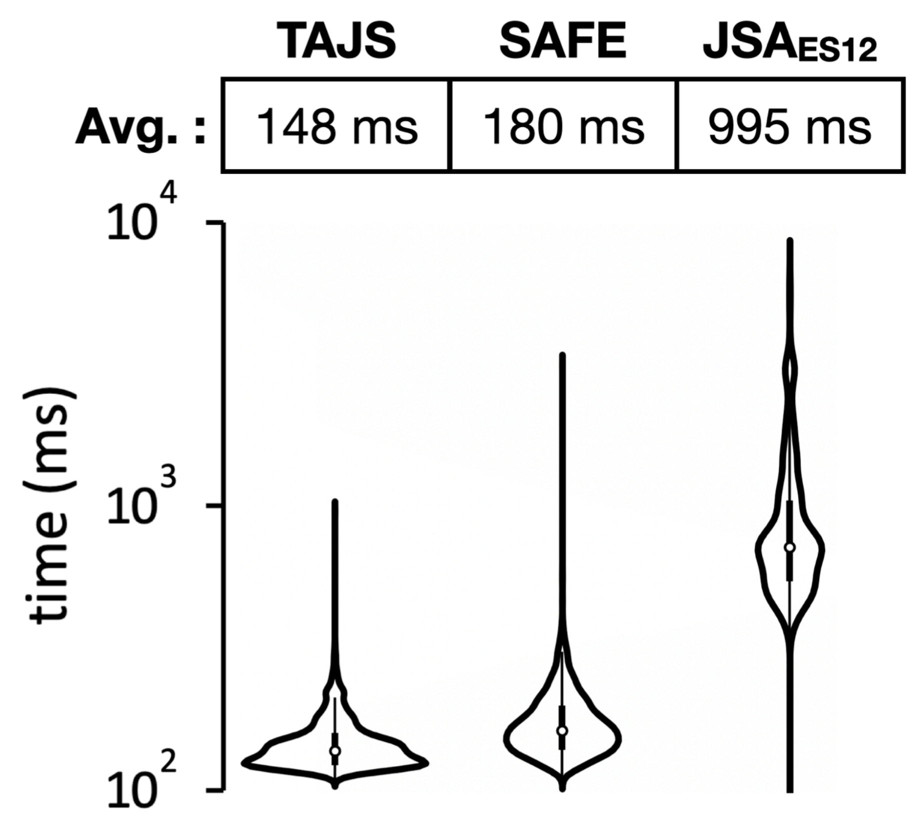


JSAVER - Evaluation

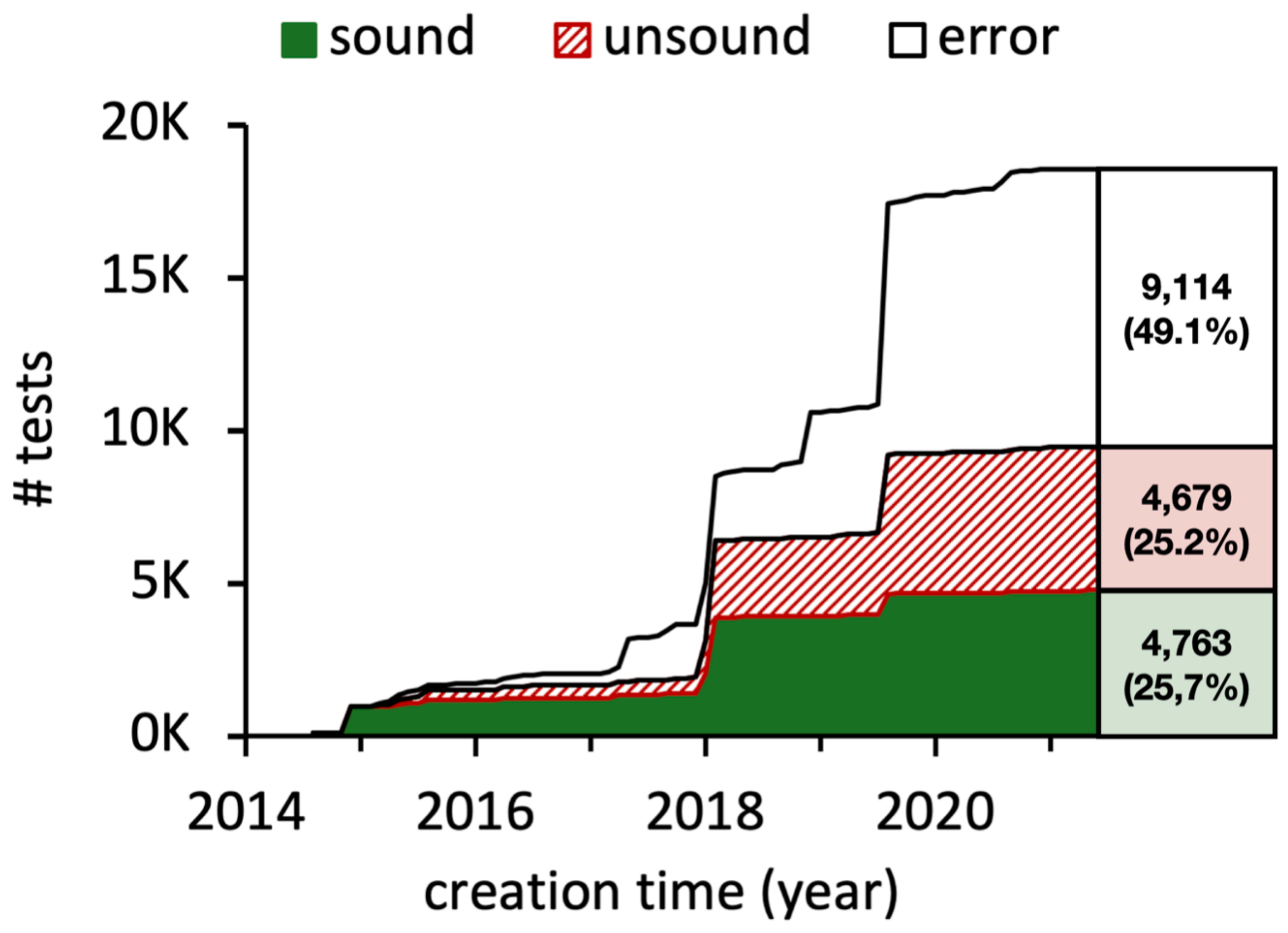
- **JSA$_{ES12}$ - A derived analyzer from ES12**
- **Evaluation with 18,556 Test262 tests**



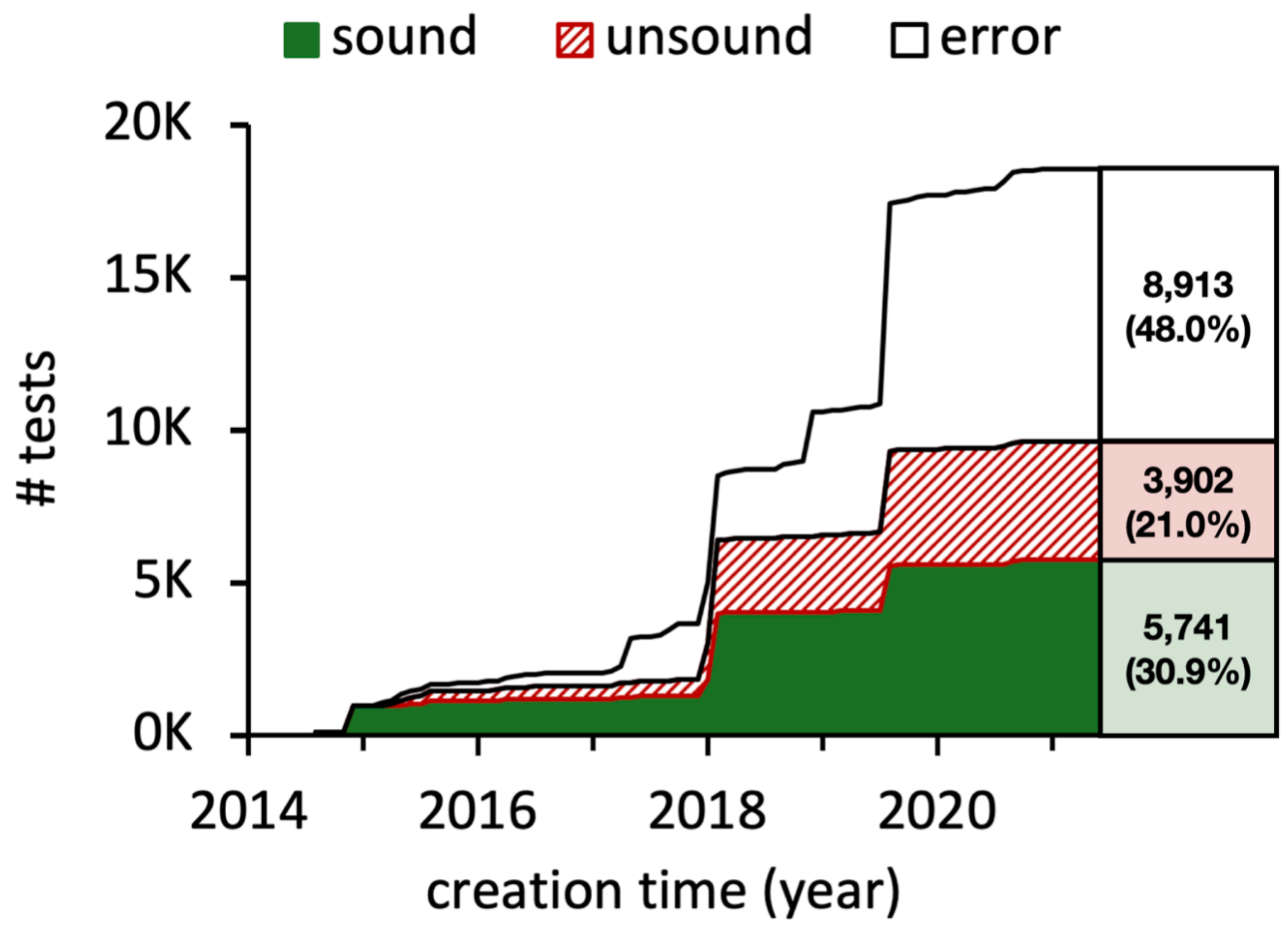
(a) The analysis precision



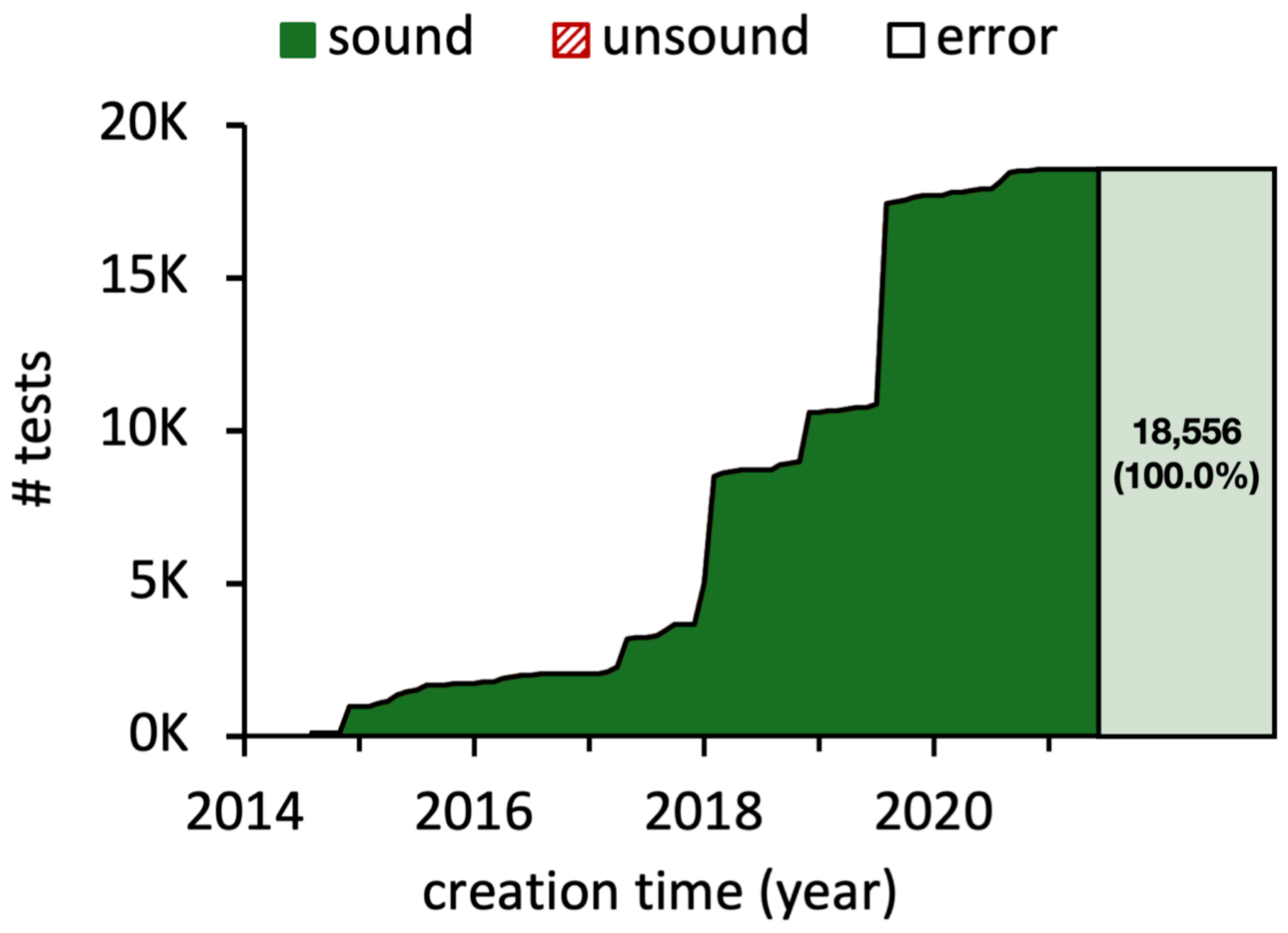
(b) The analysis performance



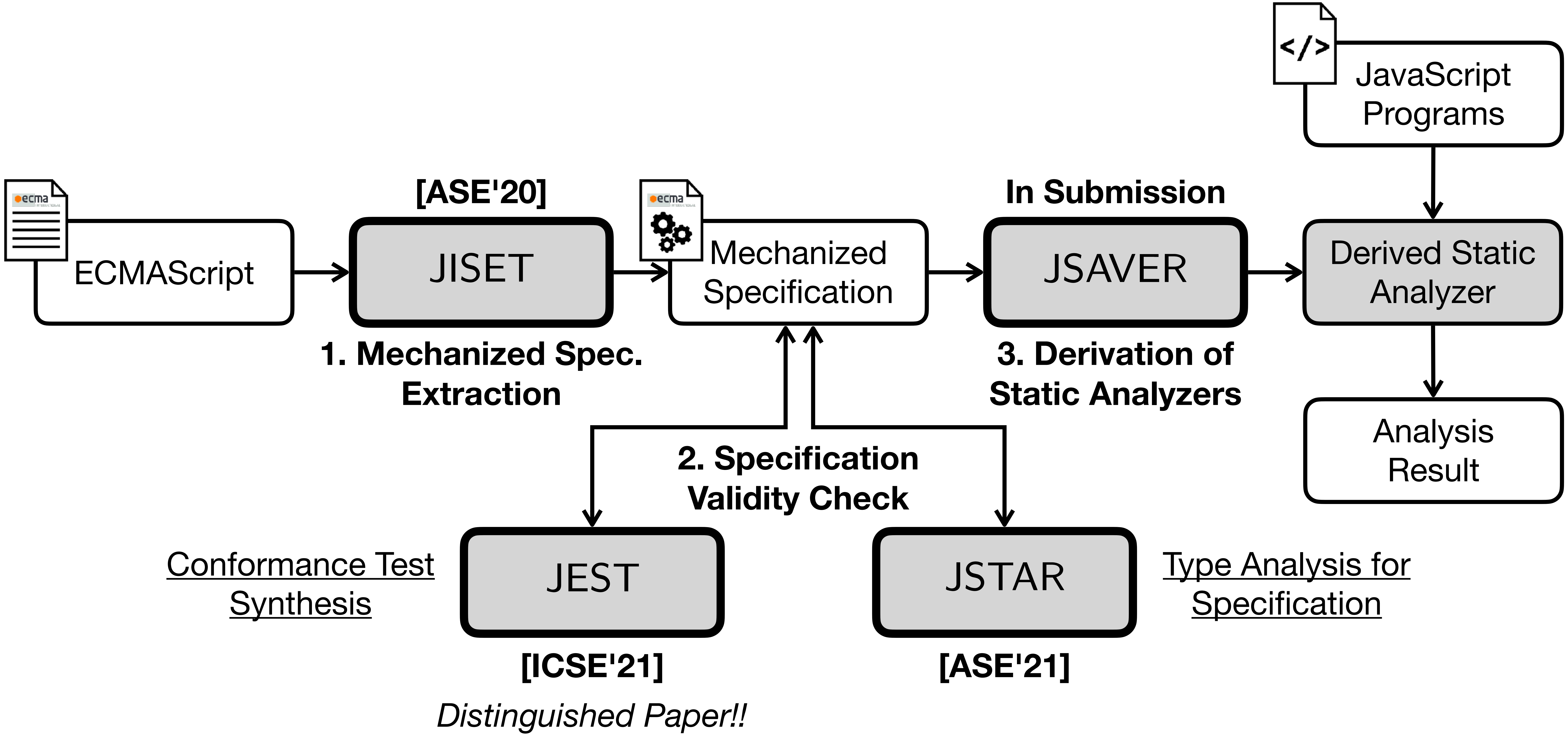
(a) Analysis results of TAJS



(b) Analysis results of SAFE



(c) Analysis results of JSA$_{ES12}$


JavaScript Programs
[ASE'20]
In Submission
ECMAScript
JISET
Mechanized Specification
JSAVER
Derived Static Analyzer
1. Mechanized Spec. Extraction
3. Derivation of Static Analyzers
Analysis Result
2. Specification Validity Check
Conformance Test Synthesis
JEST
JSTAR
Type Analysis for Specification
[ICSE'21]
[ASE'21]
Distinguished Paper!!