Automatically Generating Precise Oracles from Structured Natural Language Specifications

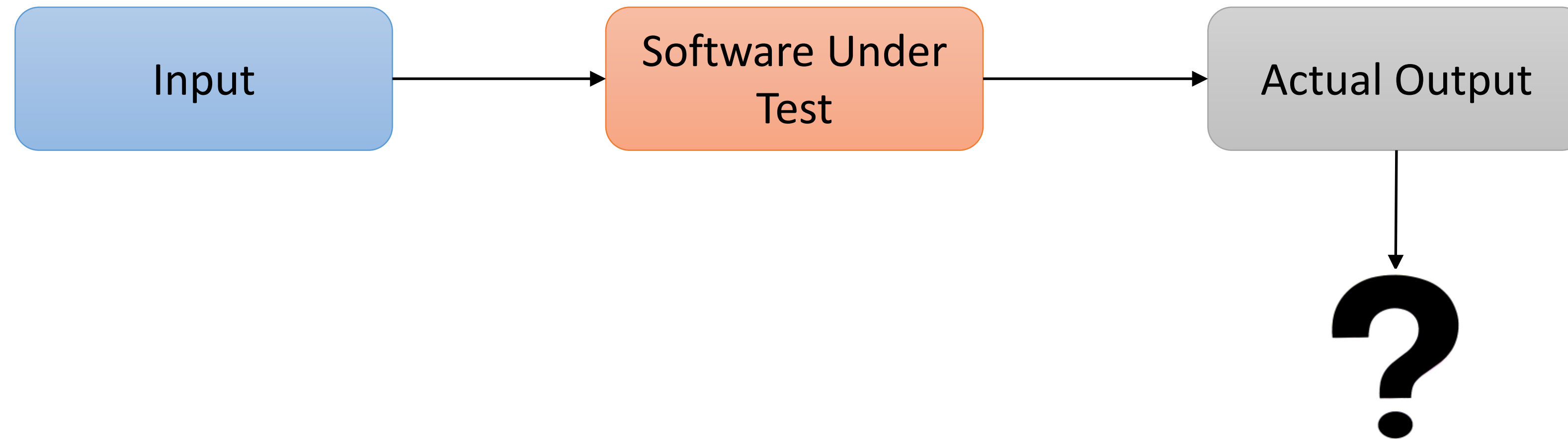
Manish Motwani

UMassAmherst

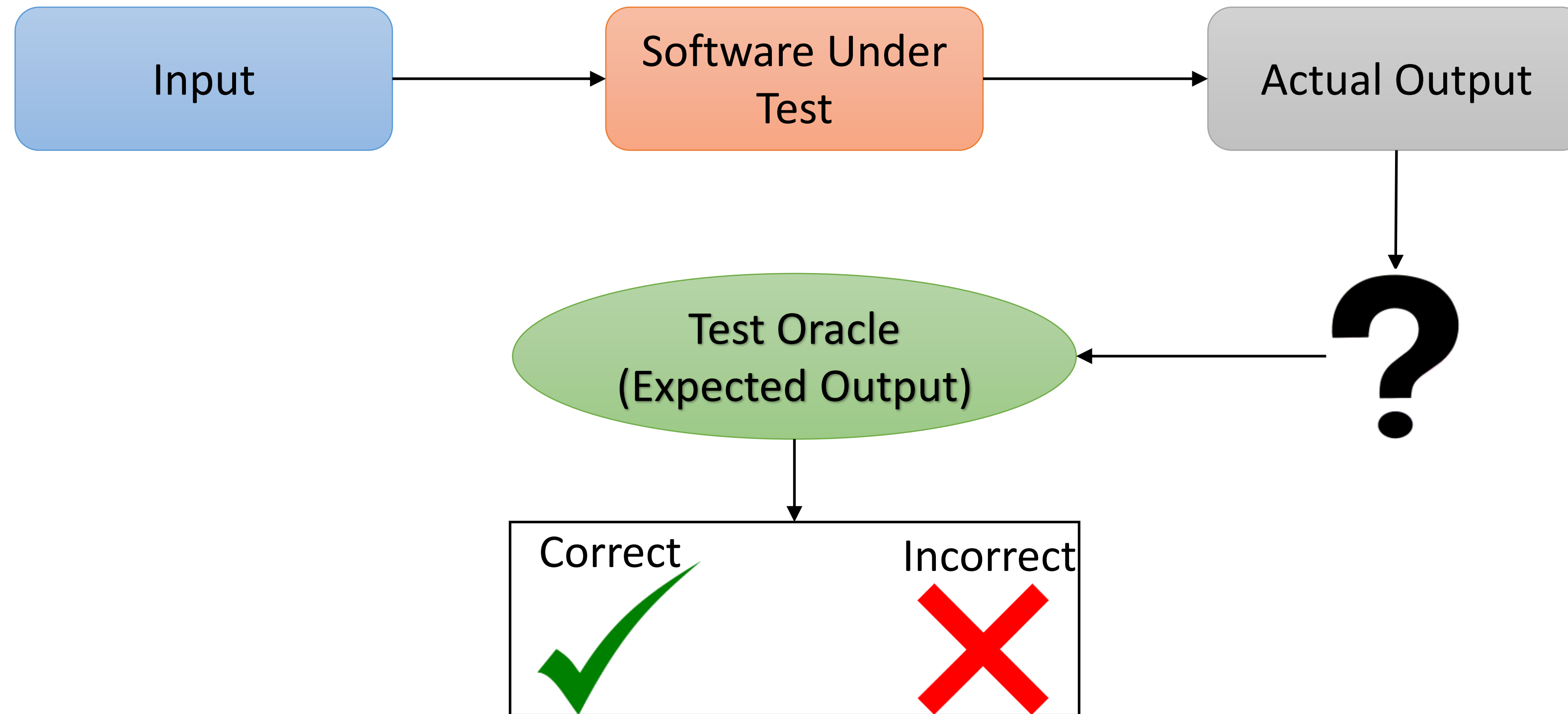
http://swami.cs.umass.edu

Yuriy Brun

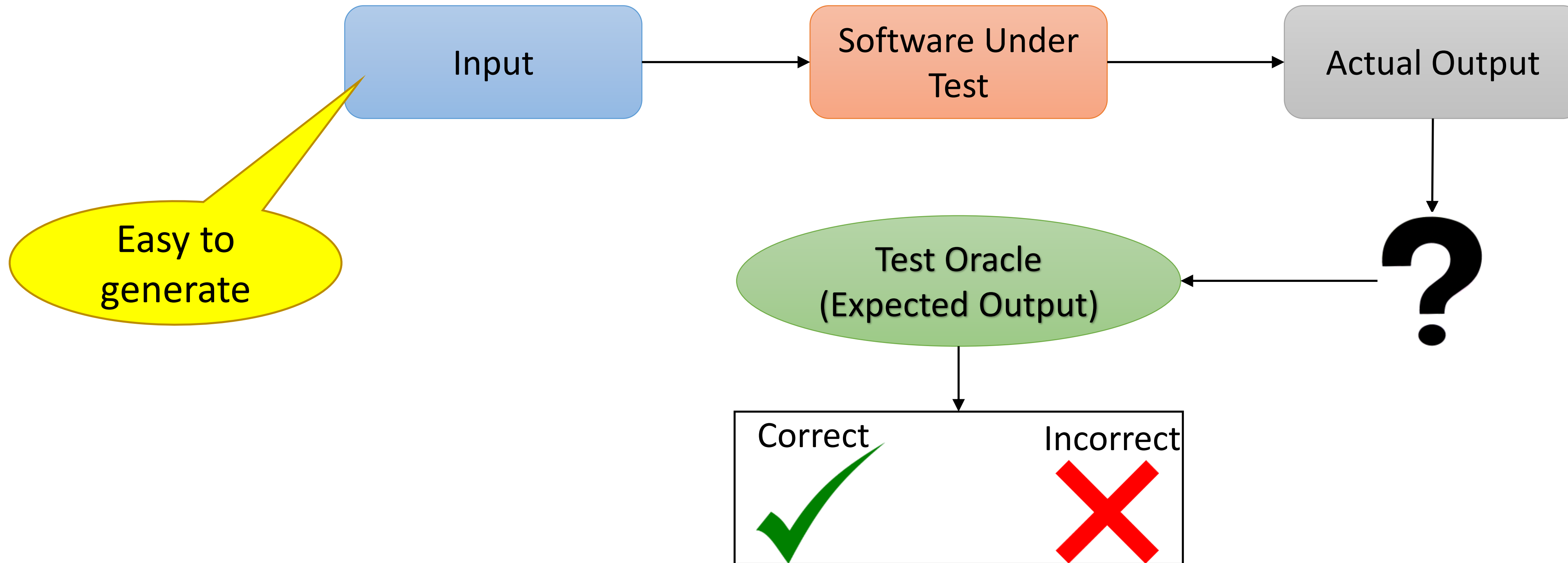
The Test Oracle Problem



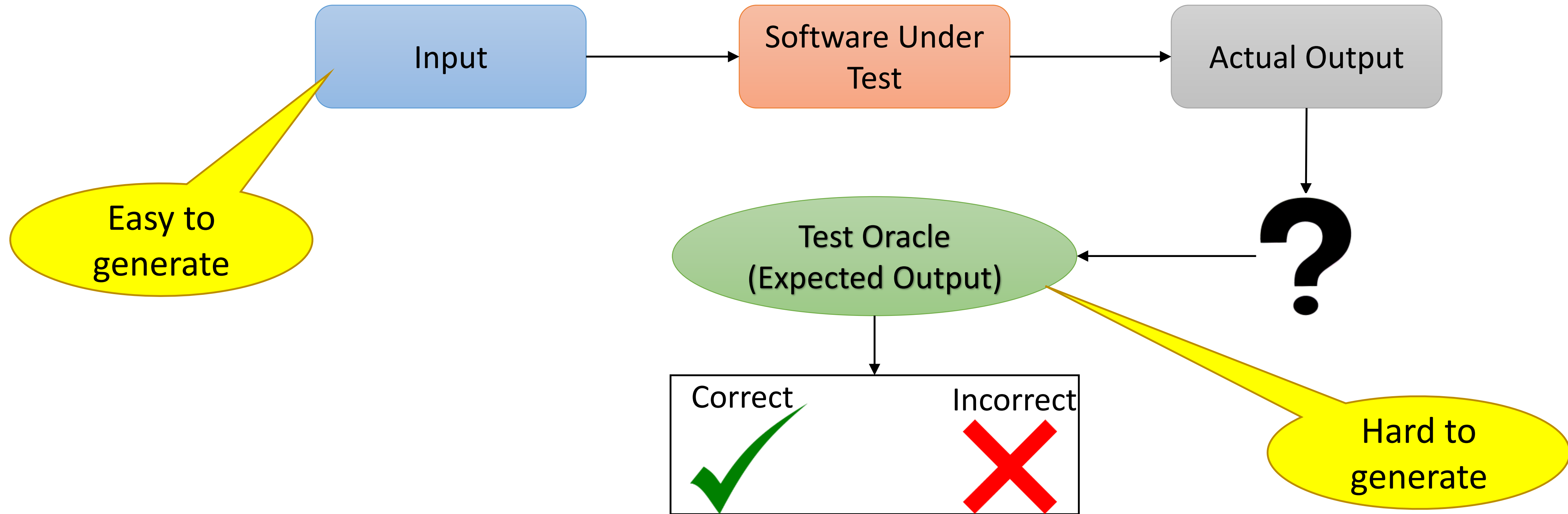
The Test Oracle Problem



The Test Oracle Problem



The Test Oracle Problem



Our Solution - Swami

Structured Informal Specification

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to `"Array"`. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the `length` property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Our Solution - Swami

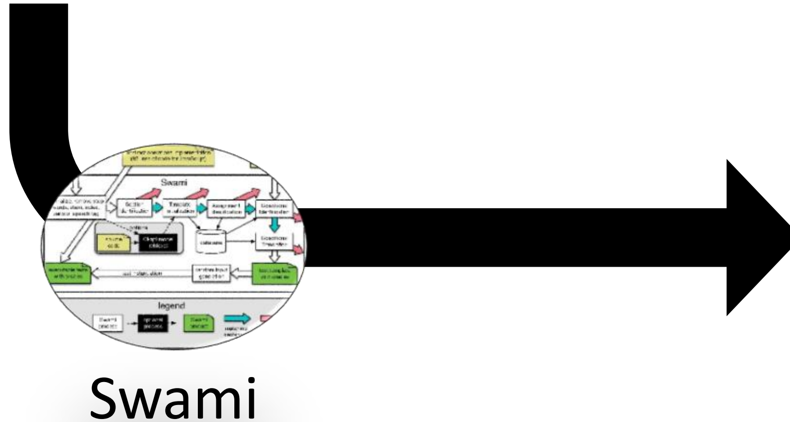
Structured Informal Specification

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to **`"Array"`**. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the **`length`** property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.



Executable Test

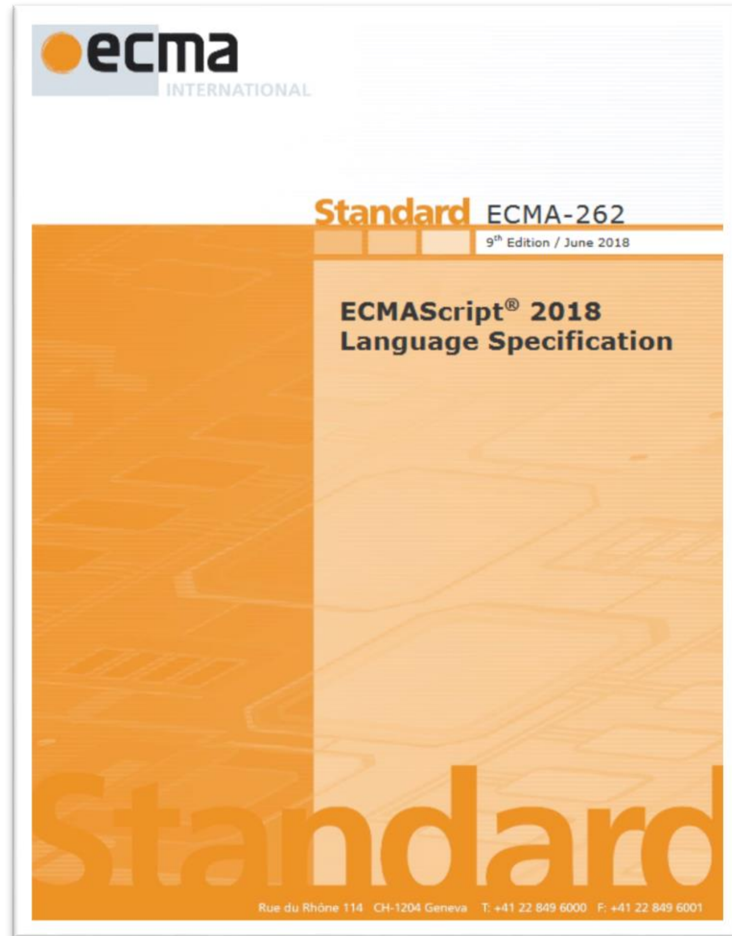
```
/*TEST TEMPLATE WITH ORACLE*/

function test_array_len( len ){
      if ( ToUint32(len)!=len) {
         try{
              var output = new Array ( len );
              return;
         }catch(e){
              assert.strictEqual(true, (e instanceof RangeError));
              return;
         }
      }
}

/*TEST INPUTS*/

test_array_len(1.1825863363010669e+308);
test_array_len(null);
test_array_len(-747);
test_array_len(368);
…
```

Test oracle

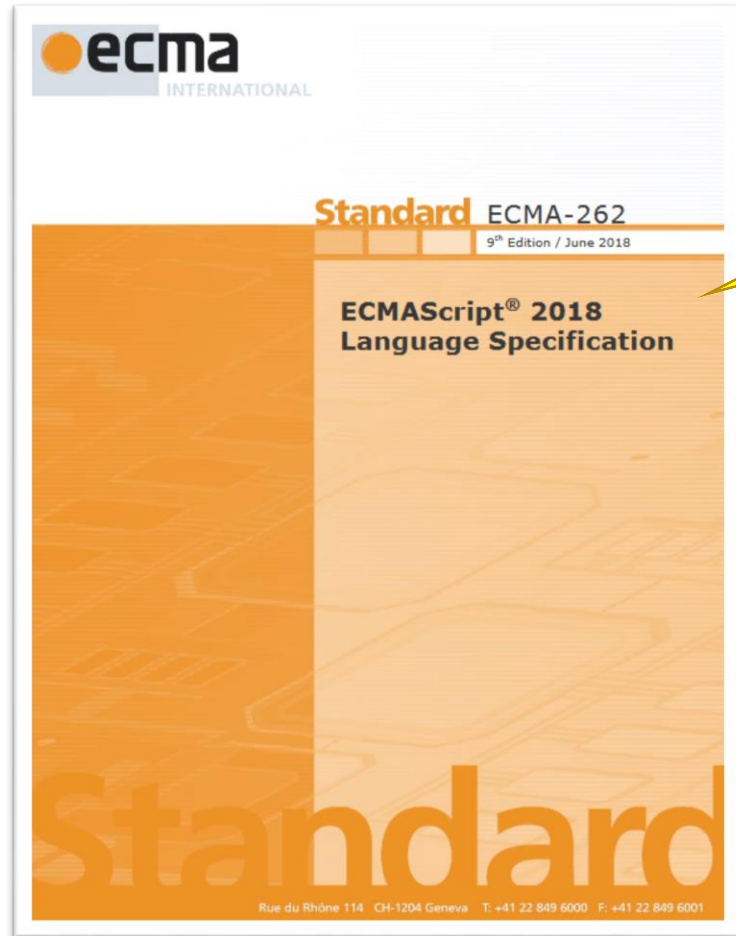
Test inputs

http://swami.cs.umass.edu

Why JavaScript specifications?

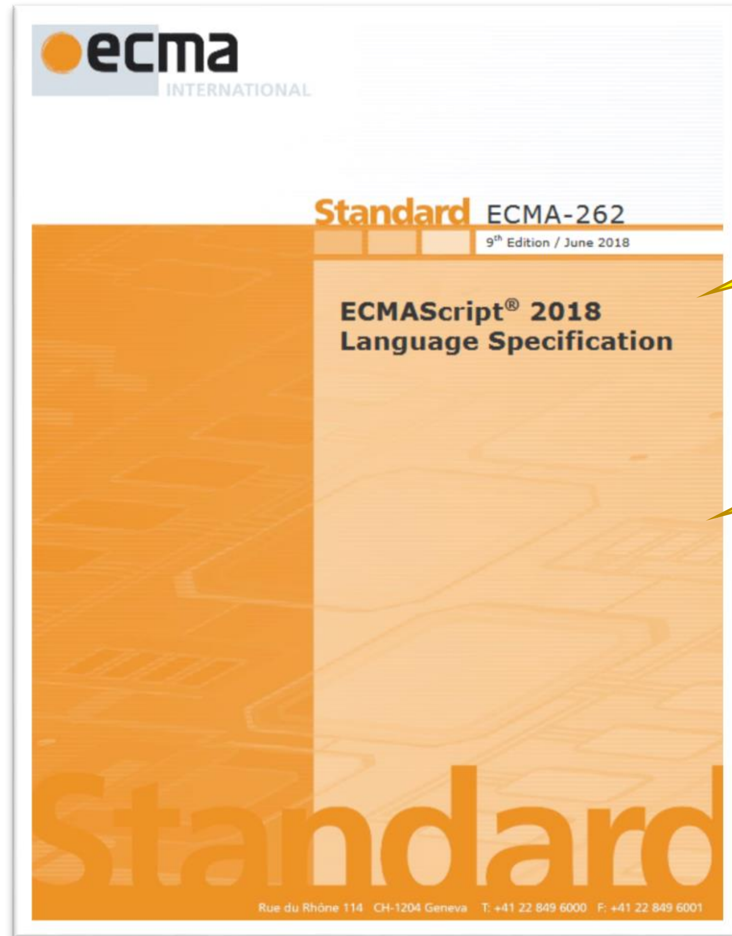


Why JavaScript specifications?



Does not get deprecated

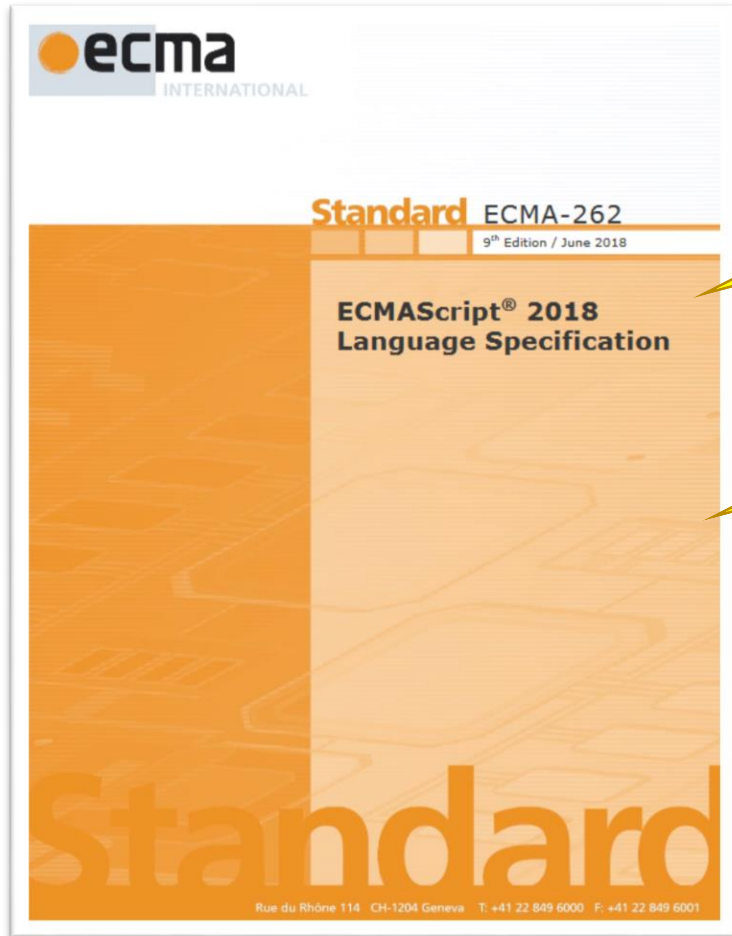
Why JavaScript specifications?



Does not get deprecated

Less ambiguous

Why JavaScript specifications?

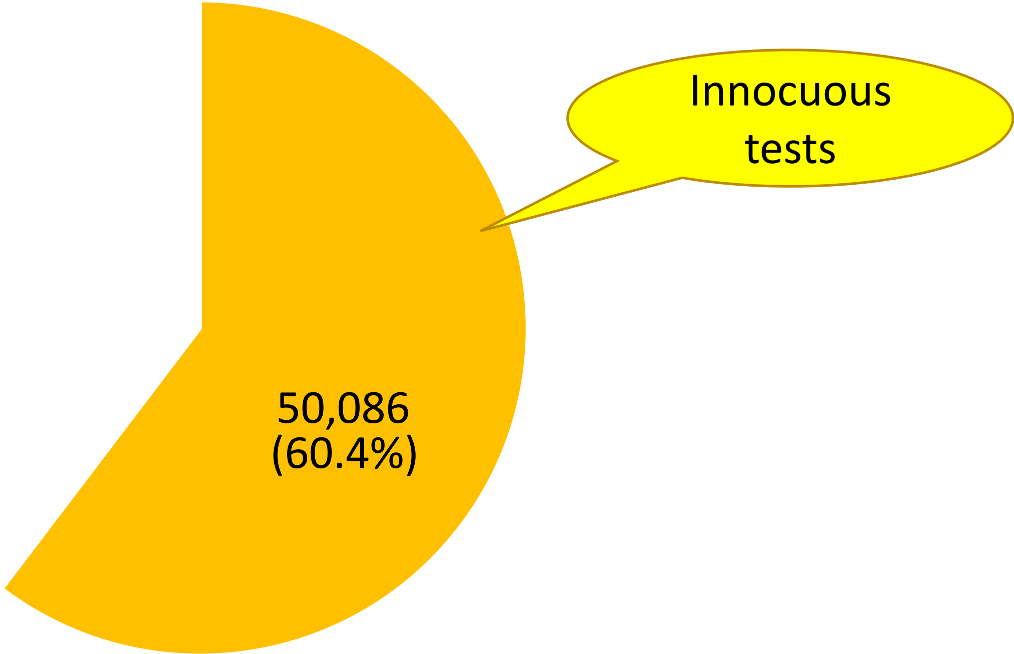


Does not get deprecated

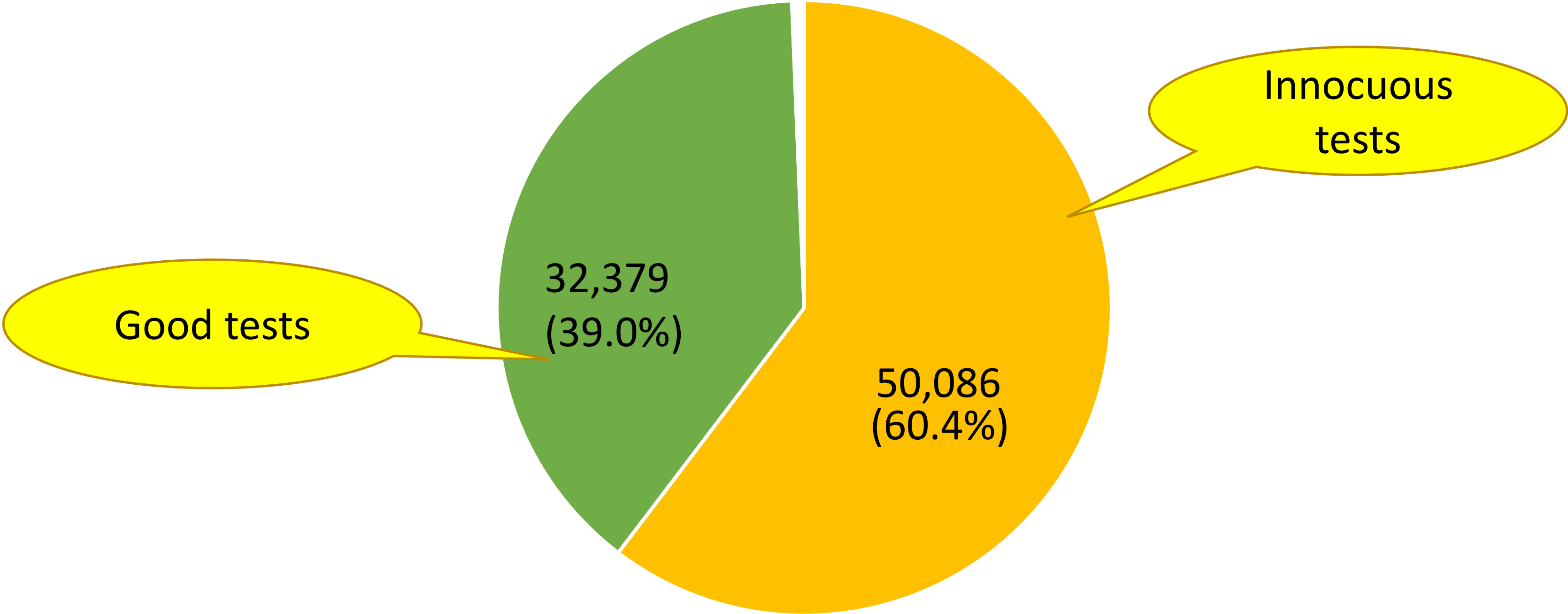
Less ambiguous

Multiple real-world projects adhere to the spec

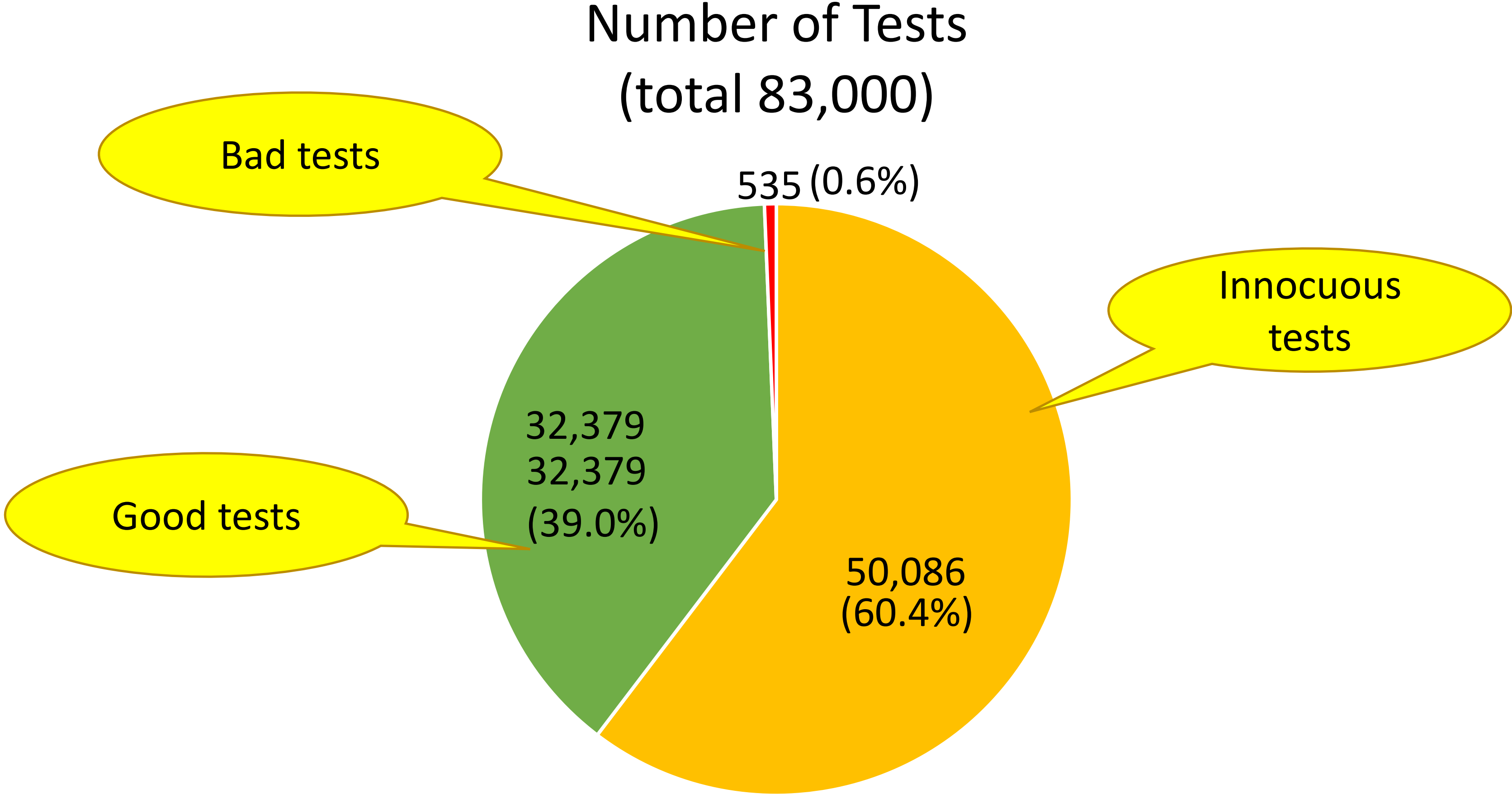
Swami-generated tests are precise to the specification



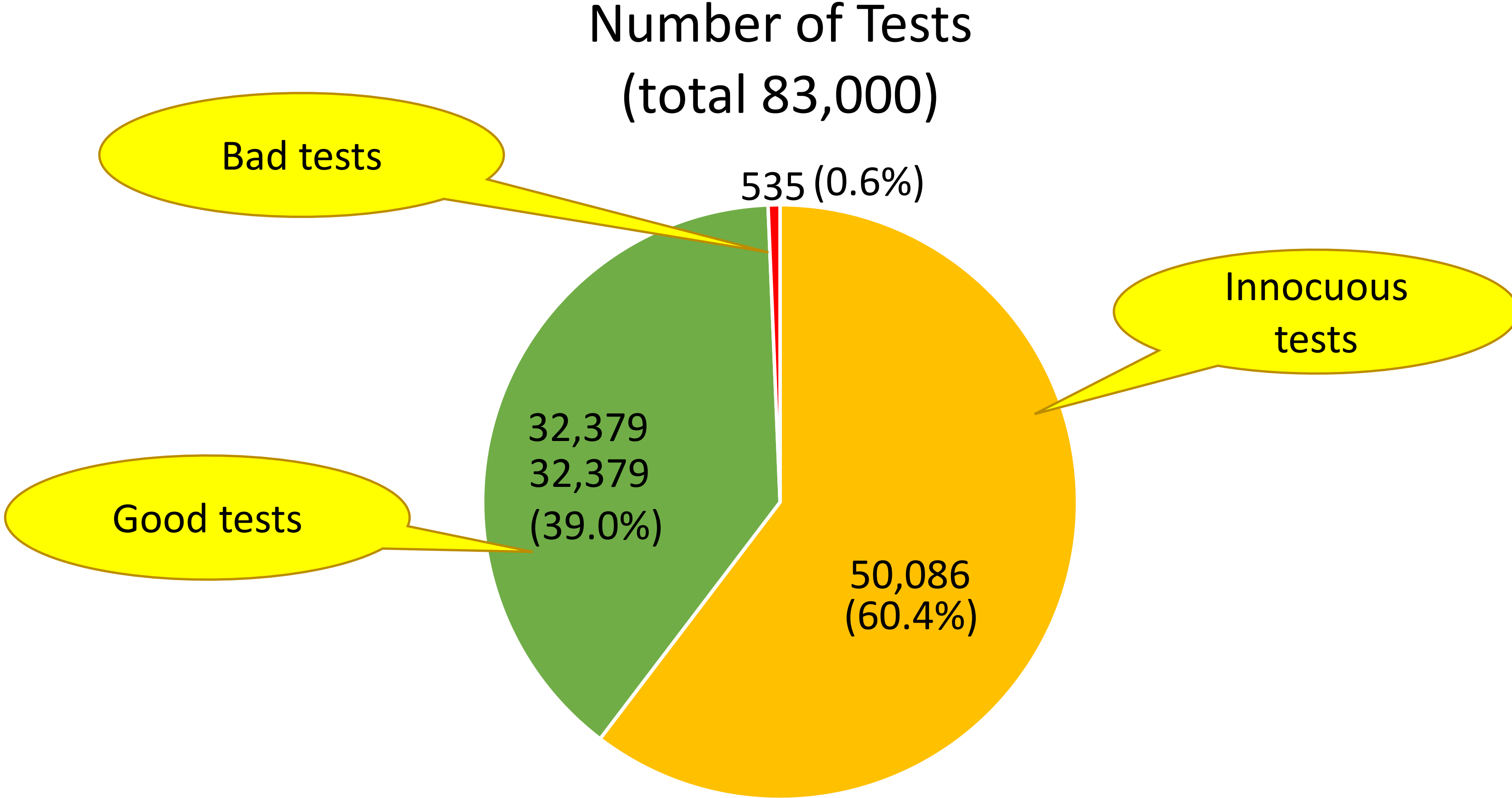
Swami-generated tests are precise to the specification



Swami-generated tests are precise to the specification



Swami-generated tests are precise to the specification



Of the non-innocuous tests, 98.4% are Good and only 1.6% are Bad

Swami covers more code and identifies features and bugs missed by developer-written tests

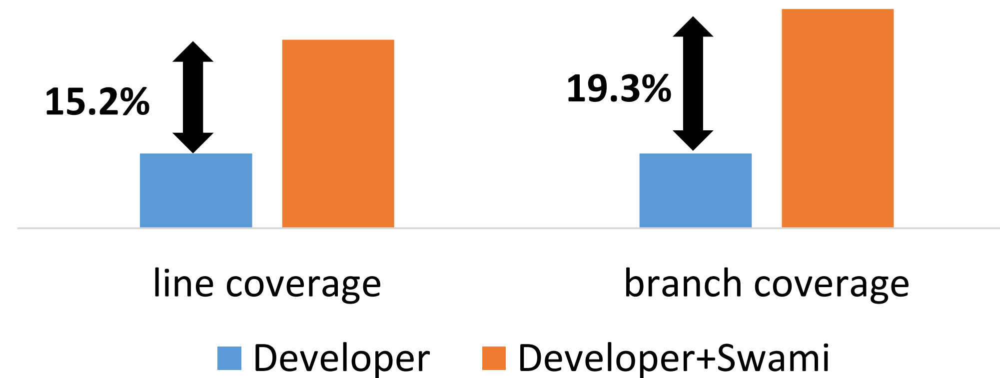
Missing Features / Bugs

- 15 missing features in Rhino
- 1 unknown bug in Rhino and Node.js
- 18 semantic disambiguities in JavaScript specification

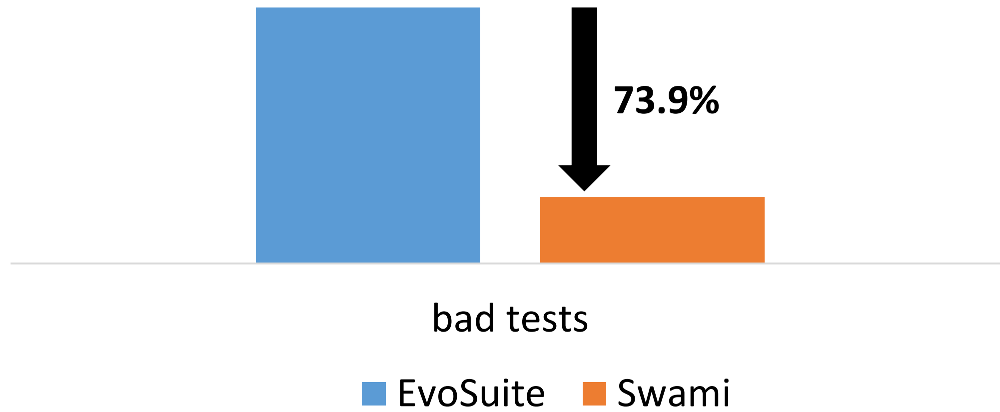
Swami covers more code and identifies features and bugs missed by developer-written tests

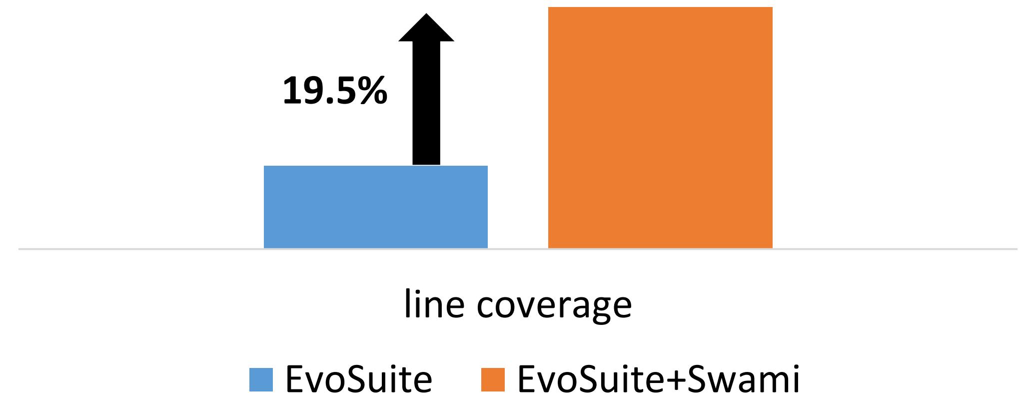
Missing Features / Bugs

- 15 missing features in Rhino
- 1 unknown bug in Rhino and Node.js
- 18 semantic disambiguities in JavaScript specification

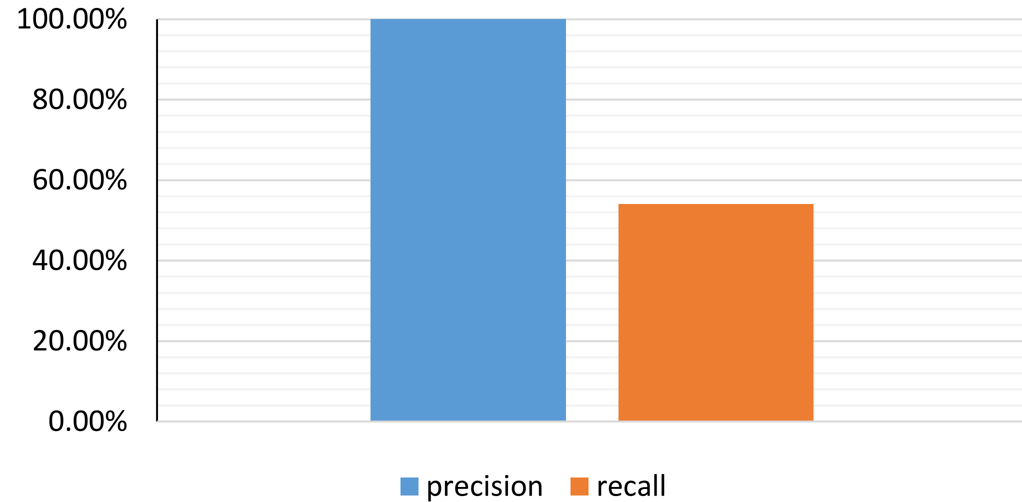


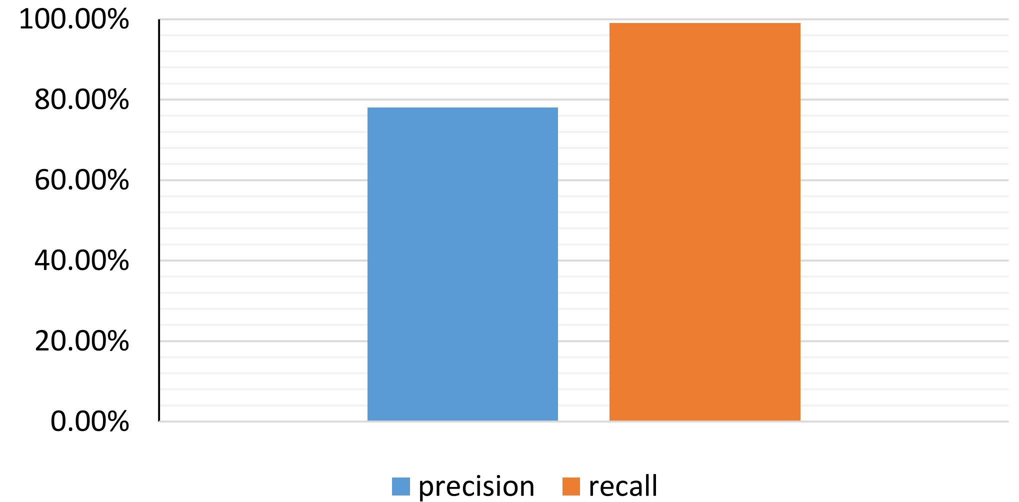
Swami generates fewer false alarms and covers code missed by EvoSuite





Swami identifies the specifications that encode testable behavior precisely





Why is it hard to derive oracles from informal specifications?

Table of Contents

Introduction
1 Scope
2 Conformance
3 Normative References
4 Overview
4.1 Web Scripting
4.2 ECMAScript Overview
4.2.1 Objects
4.2.2 The Strict Variant of ECMAScri
4.3 Terms and Definitions

5 Notational Conventions
5.1 Syntactic and Lexical Grammars
5.1.1 Context-Free Grammars
5.1.2 The Lexical and RegExp Grammars
5.1.3 The Numeric String Grammar
5.1.4 The Syntactic Grammar
5.1.5 Grammar Notation
5.2 Algorithm Conventions
5.2.1 Abstract Operations
5.2.2 Syntax-Directed Operations
5.2.3 Runtime Semantics
5.2.3.1 Implicit Completion Values

22 Indexed Collections
22.1 Array Objects
22.1.1 The Array Constructor
22.1.1.1 Array ( )
22.1.1.2 Array (*len*)
22.1.1.3 Array (...*items*)
22.1.2 Properties of the Array Constructor
22.1.2.1 Array.from (*items* [, *mapfn* [, *thisArg*]])
22.1.2.2 Array.isArray (*arg*)
22.1.2.3 Array.of (...*items*)
22.1.2.4 Array.prototype
22.1.2.5 get Array [@@species]
22.1.3 Properties of the Array Prototype Object

Why is it hard to derive oracles from informal specifications?

Encode testable behavior

Table of Contents

Introduction ✗
1 Scope ✗
2 Conformance ✗
3 Normative References ✗
4 Overview
4.1 Web Scripting ✗
4.2 ECMAScript Overview
4.2.1 Objects ✗
4.2.2 The Strict Variant of ECMAScri ✗
4.3 Terms and Definitions ✗

5 Notational Conventions
5.1 Syntactic and Lexical Grammars
5.1.1 Context-Free Grammars ✗
5.1.2 The Lexical and RegExp Grammars ✗
5.1.3 The Numeric String Grammar ✗
5.1.4 The Syntactic Grammar ✗
5.1.5 Grammar Notation ✗
5.2 Algorithm Conventions
5.2.1 Abstract Operations ✗
5.2.2 Syntax-Directed Operations ✗
5.2.3 Runtime Semantics
5.2.3.1 Implicit Completion Values ✗

22 Indexed Collections
22.1 Array Objects
22.1.1 The Array Constructor
22.1.1.1 Array ( ) ✓
22.1.1.2 Array (*len*) ✓
22.1.1.3 Array (...*items*) ✓
22.1.2 Properties of the Array Constructor
22.1.2.1 Array.from (*items* [, *mapfn* [, *thisArg*]]) ✓
22.1.2.2 Array.isArray (*arg*) ✓
22.1.2.3 Array.of (...*items*) ✓
22.1.2.4 Array.prototype ✗
22.1.2.5 get Array [@@species] ✗
22.1.3 Properties of the Array Prototype Object

Why is it hard to derive oracles from informal specifications?

Encode testable behavior

Abstract Operations

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to **`"Array"`**. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the **`length`** property of the newly constructed object is set to **`1`** and the **`0`** property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Why is it hard to derive oracles from informal specifications?

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to `"Array"`. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the `length` property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Why is it hard to derive oracles from informal specifications?

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to `"Array"`. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the `length` property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Why is it hard to derive oracles from informal specifications?

15.4.4.2 Array.prototype.toString ( )

When the `toString` method is called, the following steps are taken:

1. Let *array* be the result of calling ToObject on the **this** value.
2. Let *func* be the result of calling the [[Get]] internal method of *array* with argument `"join"`.
3. If IsCallable(*func*) is **false**, then let *func* be the standard built-in method Object.prototype.toString (15.2.4.2).
4. Return the result of calling the [[Call]] internal method of *func* providing *array* as the **this** value and an empty arguments list.

NOTE The `toString` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `toString` function can be applied successfully to a host object is implementation-dependent.

Why is it hard to derive oracles from informal specifications?

Encode testable behavior | Abstract Operations | Implicit Operations | Oracles embedded in Conditionals | Assignments using local variables | Ambiguous and Deprecated

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to `"Array"`. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the `length` property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

Related work: What can the state-of-the-art tools do?

Encode testable behavior	Abstract Operations	Implicit Operations	Oracles embedded in Conditionals	Assignments using local variables	Ambiguous and Deprecated

- **EvoSuite[1], Randoop[2]**
  - Cannot derive oracles from natural language specifications
  - Generated tests cannot identify missing features
- **Jdoctor[3] , Toradocu[4], @tComment[5]**
  - Closely tied to JavaDoc (use tags, e.g., @params, @throws) and Randoop, hence may not generalize

1. Fraser et al. TSE 2013, 2. Pacheco et al. ICSE 2007, 3. Blasi et al. ISSTA 2018, 4. Goffi et al. ISSTA 2016 , 5. Tan et al. ICST 2012

Related work: What can the state-of-the-art tools do?

Encode testable behavior | Abstract Operations | Implicit Operations | Oracles embedded in Conditionals | Assignments using local variables | Ambiguous and Deprecated

- **EvoSuite[1], Randoop[2]**
  - Cannot derive oracles from natural language specifications
  - Generated tests cannot identify missing features
- **Jdoctor[3] , Toradocu[4], @tComment[5]**
  - Closely tied to JavaDoc (use tags, e.g., @params, @throws) and Randoop, hence may not generalize

State-of-the-art tools are not capable of deriving test oracles from informal specifications that exists independent of the source code.

1. Fraser et al. TSE 2013, 2. Pacheco et al. ICSE 2007, 3. Blasi et al. ISSTA 2018, 4. Goffi et al. ISSTA 2016 , 5. Tan et al. ICST 2012

What kind of oracles exist in informal specifications?

Vague oracles for **common inputs**

11.6.3 Applying the Additive Operators to Numbers

The + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The − operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 binary double-precision arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sum of two infinities of opposite sign is **NaN**.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two negative zeroes is **−0**. The sum of two positive zeroes, or of two zeroes of opposite sign, is **+0**.

Concrete oracles for **uncommon inputs**

What kind of oracles exist in informal specifications?

Vague oracles for **common inputs**

11.6.3 Applying the Additive Operators to Numbers

The + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The − operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 binary double-precision arithmetic:

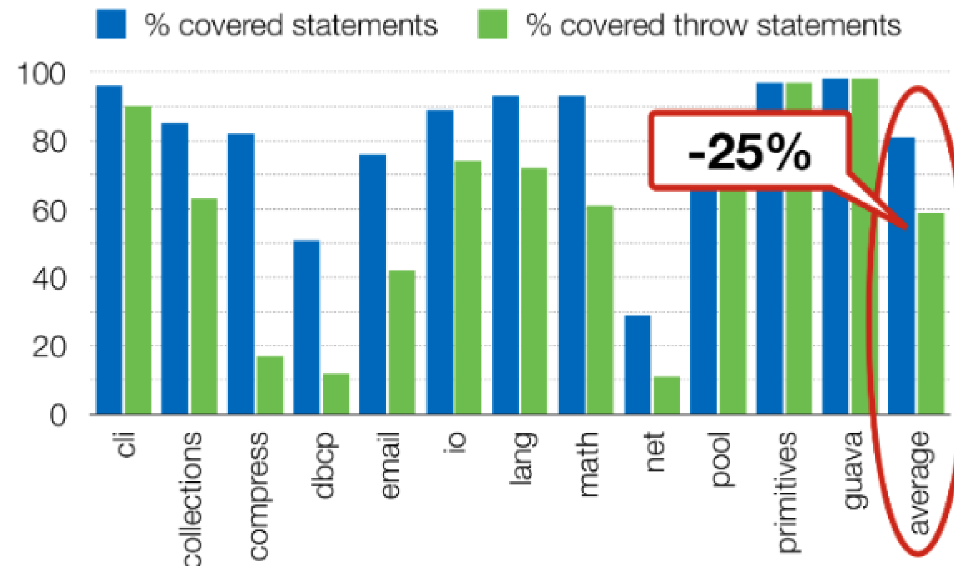
- If either operand is **NaN**, the result is **NaN**.
- The sum of two infinities of opposite sign is **NaN**.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- … opposite sign, is

Concrete oracles for **uncommon inputs**

Informal specifications typically contain oracles for **Exceptions** and **Boundary conditions**.

Is it useful to generate tests only for Exceptions and Boundary conditions?

- 10 popular, well-tested open source libraries
- The coverage of throw statements is usually significantly lower than overall coverage, in two cases below 50%

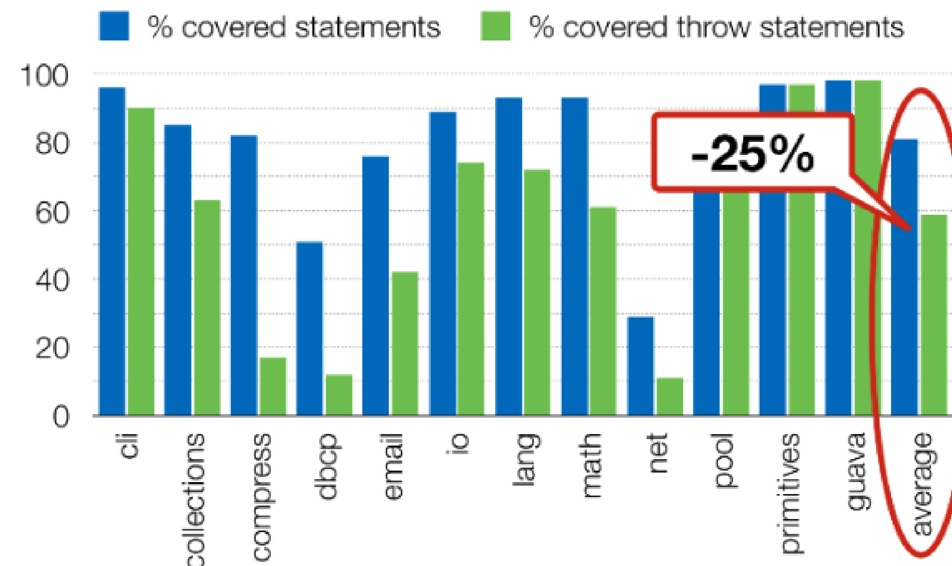


Source: Goffi, Alberto, et al. “Automatic generation of oracles for exceptional behaviors.” ISSTA, 2016.

Is it useful to generate tests only for Exceptions and Boundary conditions?

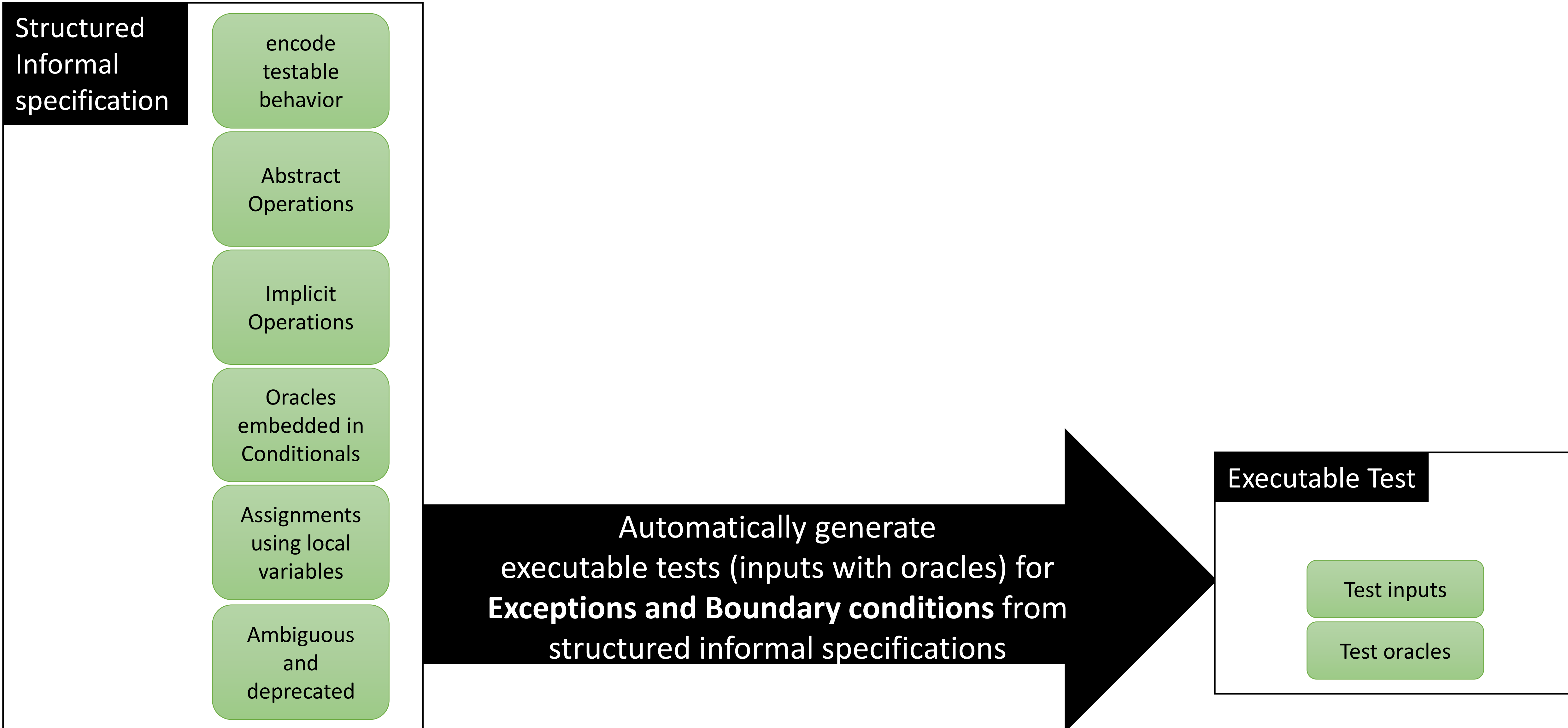
- 10 popular, well-tested open source libraries
- The coverage of throw statements is usually significantly lower than overall coverage, in two cases below 50%

Exceptions are under-tested by the developers



Source: Goffi, Alberto, et al. "Automatic generation of oracles for exceptional behaviors." ISSTA, 2016.

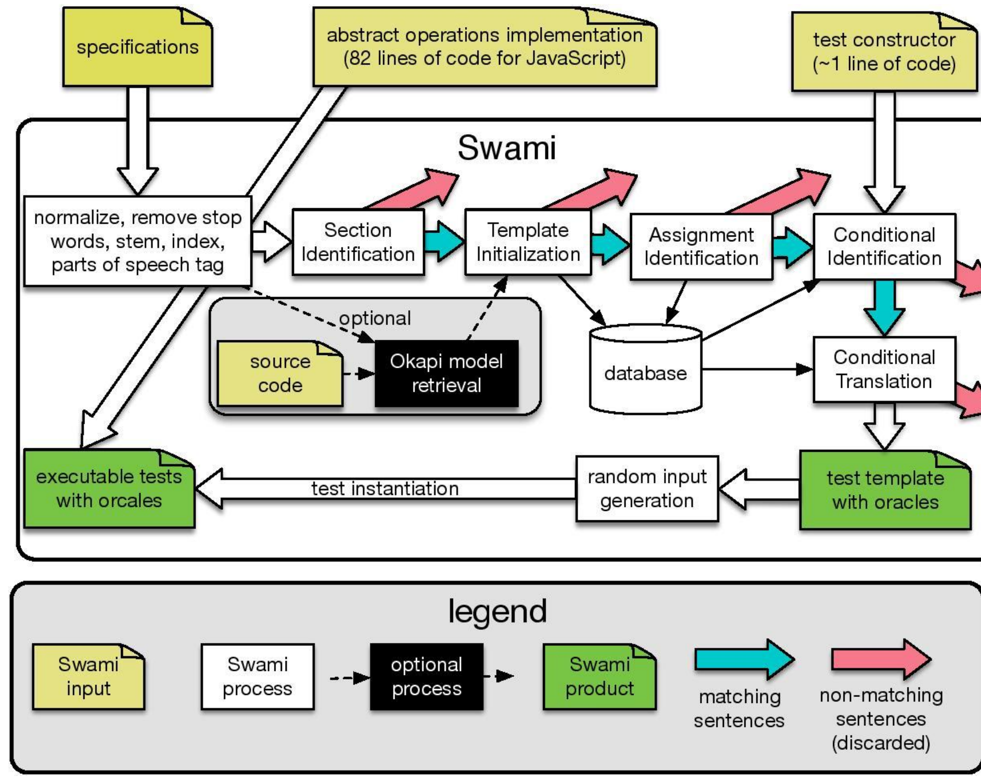
Goal of this work



Swami

Structured Informal specification

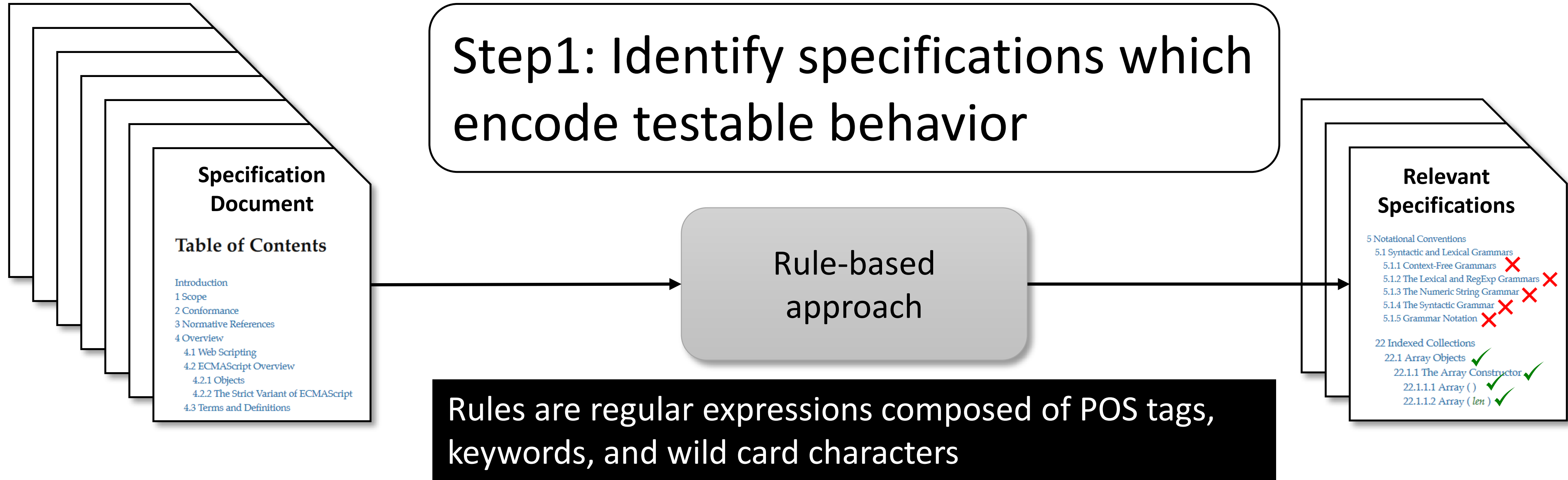
- encode testable behavior
- Abstract Operations
- Implicit Operations
- Oracles embedded in Conditionals
- Assignments using local variables
- Ambiguous and deprecated

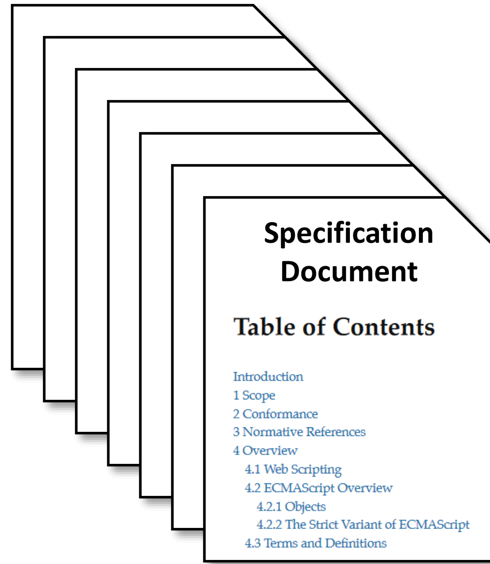


Automatically generate executable tests (inputs with oracles) for **Exceptions and Boundary conditions** from structured informal specifications

Executable Test

- Test inputs
- Test oracles

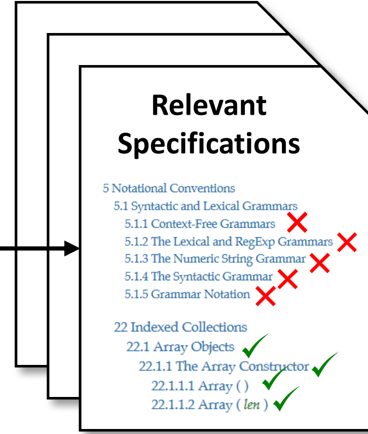

Step1: Identify specifications which encode testable behavior
Specification Document
Table of Contents
Introduction
1 Scope
2 Conformance
3 Normative References
4 Overview
4.1 Web Scripting
4.2 ECMAScript Overview
4.2.1 Objects
4.2.2 The Strict Variant of ECMAScript
4.3 Terms and Definitions
Rule-based approach
Relevant Specifications
5 Notational Conventions
5.1 Syntactic and Lexical Grammars
5.1.1 Context-Free Grammars
5.1.2 The Lexical and RegExp Grammars
5.1.3 The Numeric String Grammar
5.1.4 The Syntactic Grammar
5.1.5 Grammar Notation
22 Indexed Collections
22.1 Array Objects
22.1.1 The Array Constructor
22.1.1.1 Array ( )
22.1.1.2 Array (len)
Rules are regular expressions composed of POS tags, keywords, and wild card characters



Step1: Identify specifications which encode testable behavior

Rule-based approach

Heading RE: [CD new* NN LRB NN.* RRB]
Body RE: [If .* return .*] [if .* throw .* exception]



Step1: Identify specifications which encode testable behavior

Specification Document

Table of Contents

Introduction
1 Scope
2 Conformance
3 Normative References
4 Overview
4.1 Web Scripting
4.2 ECMAScript Overview
4.2.1 Objects
4.2.2 The Strict Variant of ECMAScript
4.3 Terms and Definitions

Rule-based approach

Relevant Specifications

5 Notational Conventions
5.1 Syntactic and Lexical Grammars
5.1.1 Context-Free Grammars ✗
5.1.2 The Lexical and RegExp Grammars ✗
5.1.3 The Numeric String Grammar ✗
5.1.4 The Syntactic Grammar ✗
5.1.5 Grammar Notation ✗
22 Indexed Collections
22.1 Array Objects ✓
22.1.1 The Array Constructor ✓
22.1.1.1 Array ( ) ✓
22.1.1.2 Array (*len*) ✓

when the format of specification document is unknown

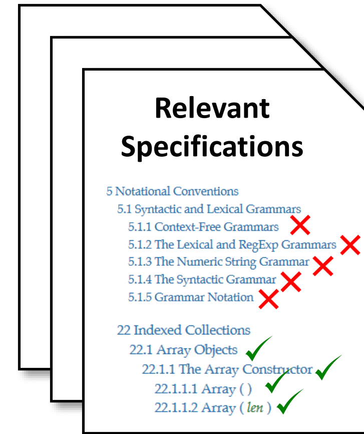
Source code

Information Retrieval-based approach

OKAPI model

section ID	matched Java class	similarity score
15.4.2.2	`ScriptRuntime.java`	0.37
15.4.2.2	`Interpreter.java`	0.31
15.4.2.2	`BaseFunction.java`	0.25
15.4.2.2	`ScriptableObject.java`	0.24
15.4.2.2	`NativeArray.java`	0.21

Example specification encoding testable behavior

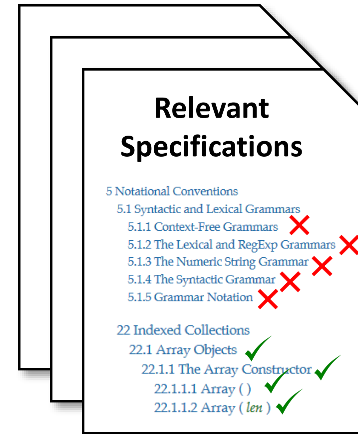


21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Example specification encoding testable behavior



21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Header RE:
CD new* NN LRB NN.* RRB

Body RE:
If .* throw .* exception

Body RE:
If .* return .*

Step2: Extract **method signature** from specification heading and initialize Test Template

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step2: Extract **method signature** from specification heading and initialize Test Template

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).

```
function test_< method name >(thisObj,<[ method args ]>) {}
```

3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.

```
function test_string_prototype_startswith(thisObj,searchString,position) {}
```

Initialized Test Template

11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step2: Extract **method signature** from specification heading and initialize Test Template

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).

```
function test_< method name >(thisObj,<[ method args ]>) {}
```

3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.

```
function test_string_prototype_startswith(thisObj,searchString,position) {}
```

Initialized Test Template

sequence of

```
new String(thisObj).startsWith(searchString, position);
```

Method invocation code

12. Otherwise, return **false**.

Step3: Identify and parse **Assignments** to store the local variables and their values

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step3: Identify and parse **Assignments** to store the local variables and their values

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

Step4: Identify and parse **Conditionals** to populate the conditional templates

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

Boundary Condition

```
if (<condition>) {
      var output = <method invocation>;
      <test constructor>(output, <expected output>);
      return;
}
```

3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

Boundary Condition

```
if (<condition>) {
      var output = <method invocation>;
      <test constructor>(output, <expected output>);
      return;
}
```

3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step pro...
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Exception oracle

Boundary condition oracle

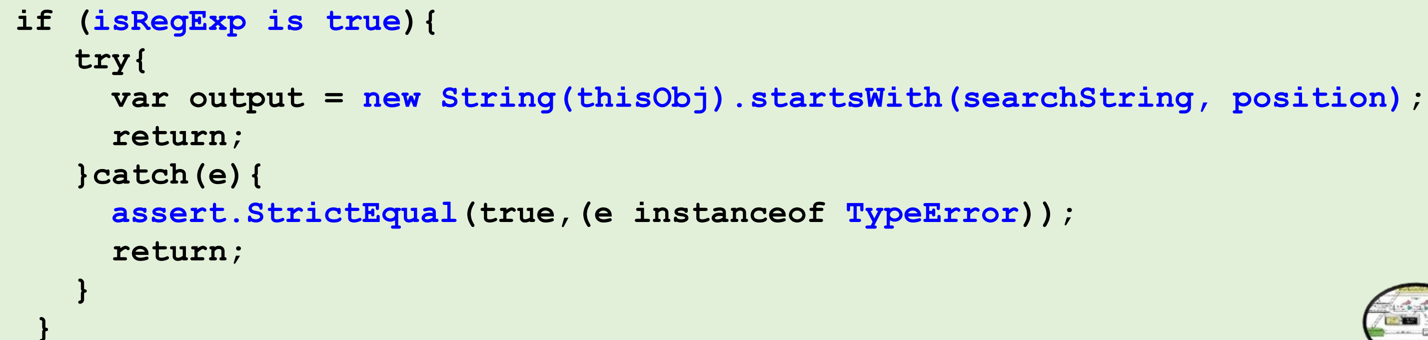
Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

4. If *isRegExp* is **true**, throw a **TypeError** exception.

Exception oracle



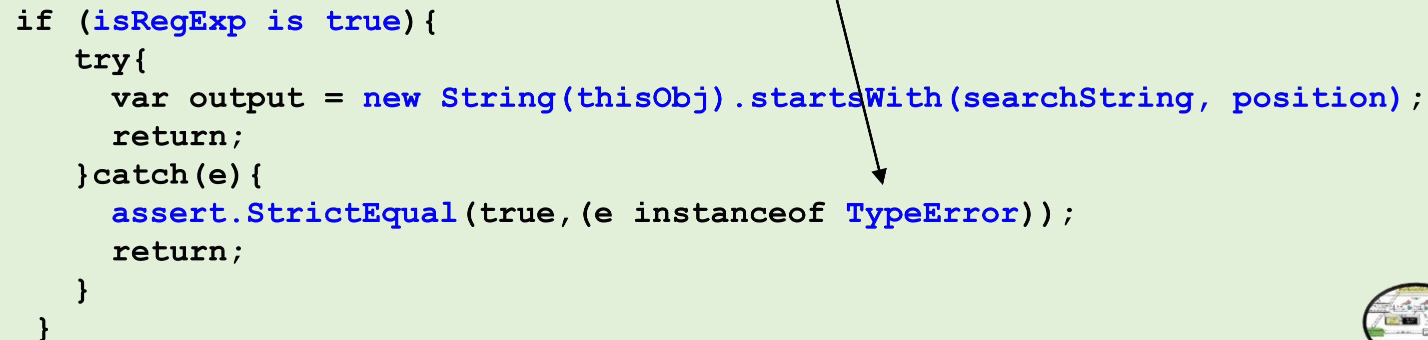
Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

4. If *isRegExp* is **true**, throw a **TypeError** exception.

Exception oracle



```
if (isRegExp is true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

From step2

4. If *isRegExp* is **true**, throw a **TypeError** exception.

Exception oracle

```
if (isRegExp is true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

From step2

4. If *isRegExp* is **true**, throw a **TypeError** exception.

Exception oracle

Input by developer

```
if (isRegExp is true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```

Step4: Identify and parse **Conditionals** to populate the conditional templates

Boundary Condition

```
if (<condition>) {
    var output = <method invocation>;
    <test constructor>(output, <expected output>);
    return;
}
```

21.1.3.20 String.prototype.startsWith (*searchString* [, *po*

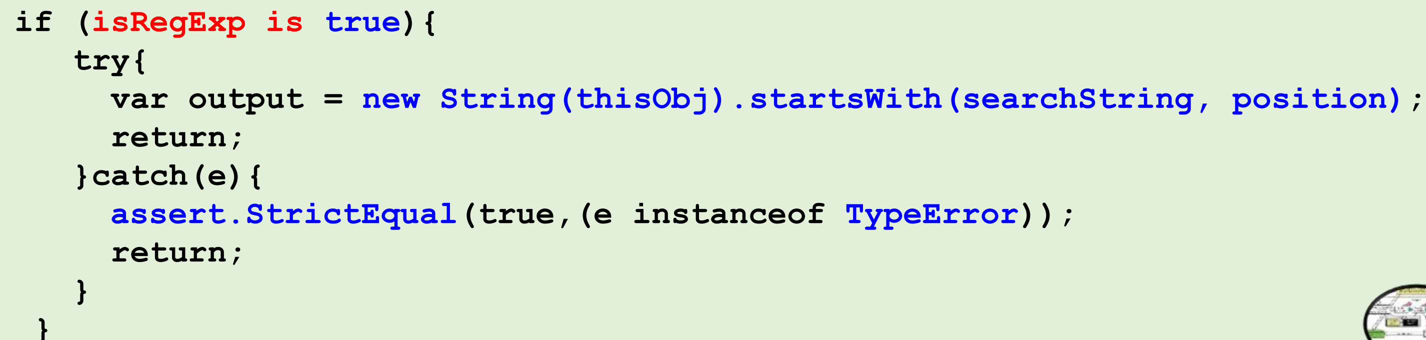
The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *sta*... ength *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

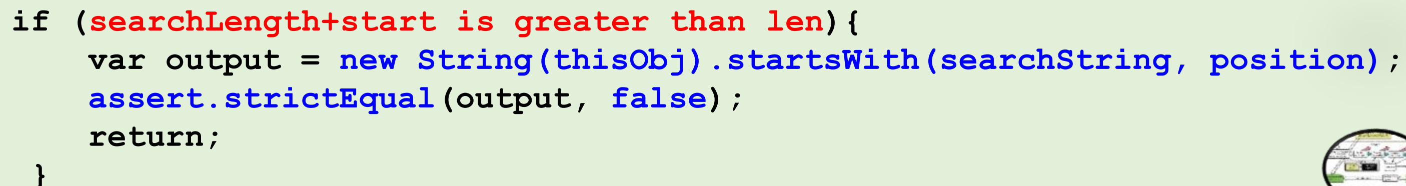
Boundary condition oracle

```
if (searchLength+start is greater than len){
    var output = new String(thisObj).startsWith(searchString, position);
    assert.strictEqual(output, false);
    return;
}
```

Step5: Recursively substitute **local variables** and **implicit operations**



```
if (isRegExp is true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```



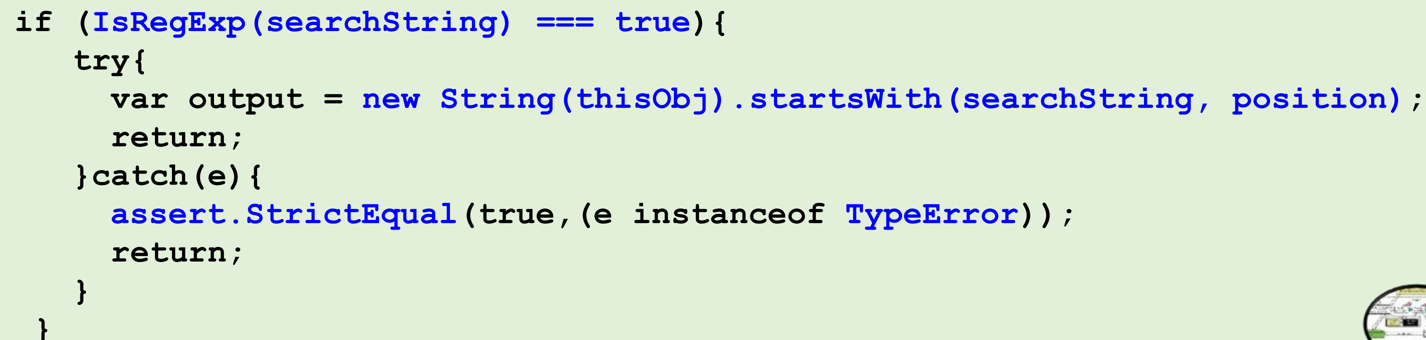
```
  if (searchLength+start is greater than len){
     var output = new String(thisObj).startsWith(searchString, position);
     assert.strictEqual(output, false);
     return;
   }
```

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

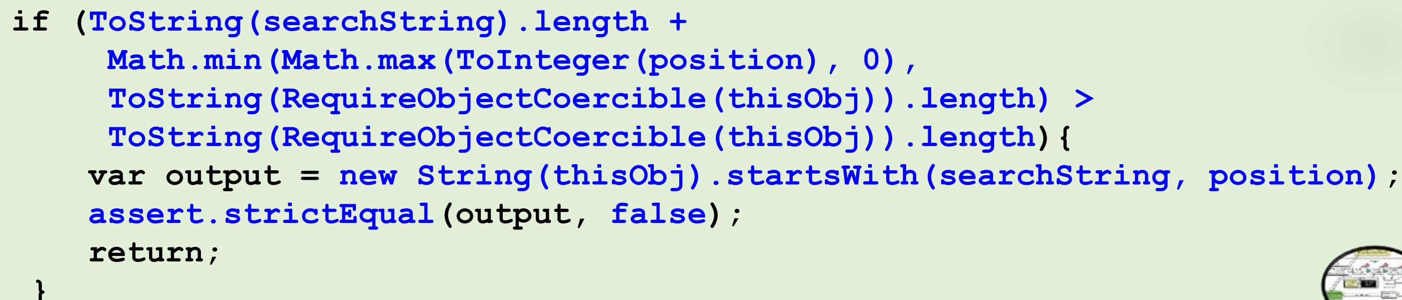
Method Arguments:

thisObj
searchString
position

Step5: Recursively substitute **local variables** and **implicit operations**



```
if (IsRegExp(searchString) === true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```



```
  if (ToString(searchString).length +
      Math.min(Math.max(ToInteger(position), 0),
      ToString(RequireObjectCoercible(thisObj)).length) >
      ToString(RequireObjectCoercible(thisObj)).length){
     var output = new String(thisObj).startsWith(searchString, position);
     assert.strictEqual(output, false);
     return;
  }
```

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

Method Arguments:

thisObj
searchString
position

Step6: Add conditionals to the initialized test template and check if it compiles

```
function test_string_prototype_startswith(thisObj,searchString,position) {

  if (IsRegExp(searchString) === true){
     try{
       var output = new String(thisObj).startsWith(searchString, position);
       return;
     }catch(e){
       assert.StrictEqual(true,(e instanceof TypeError));
       return;
     }
   }









}
```

Implement **Abstract Operations** (100 lines JS code)

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
…
```

```
function test_string_prototype_startswith(thisObj,searchString,position) {

   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }

}
```

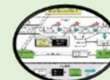
Implement **Abstract Operations** (100 lines JS code)

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
...
```

Abstract Operations

```
function test_string_prototype_startswith(thisObj,searchString,position) {
   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }

     if (ToString(searchString).length +
         Math.min(Math.max(ToInteger(position), 0),
         ToString(RequireObjectCoercible(thisObj)).length) >
         ToString(RequireObjectCoercible(thisObj)).length){
        var output = new String(thisObj).startsWith(searchString, position);
        assert.strictEqual(output, false);
        return;
      }

}
```

Test Template encoding Oracles



Step7: Instantiating test template by generating test inputs using random input generation

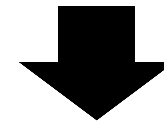
```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
…
```

Abstract Operations

```
function test_string_prototype_startswith(thisObj,searchString,position) {
  if (IsRegExp(searchString) === true){
     try{
       var output = new String(thisObj).startsWith(searchString, position);
       return;
     }catch(e){
       assert.StrictEqual(true,(e instanceof TypeError));
       return;
     }
  }

   if (ToString(searchString).length +
       Math.min(Math.max(ToInteger(position), 0),
       ToString(RequireObjectCoercible(thisObj)).length) >
       ToString(RequireObjectCoercible(thisObj)).length){
      var output = new String(thisObj).startsWith(searchString, position);
      assert.strictEqual(output, false);
      return;
   }

}
```

Test Template encoding Oracles

- **Total number of inputs**: 3
- **Heuristic**: String method => *thisObj* should be a valid string
- **Number of test inputs to be generated**: 1000



```
test_string_prototype_startswith("Y3I9", "E0RS6GU078", 894);
test_string_prototype_startswith("T82LL6", 572, false);
test_string_prototype_startswith("XU6W0", "J3A", Infinity);
test_string_prototype_startswith("W5E74X0R", null, NaN);
...
```

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

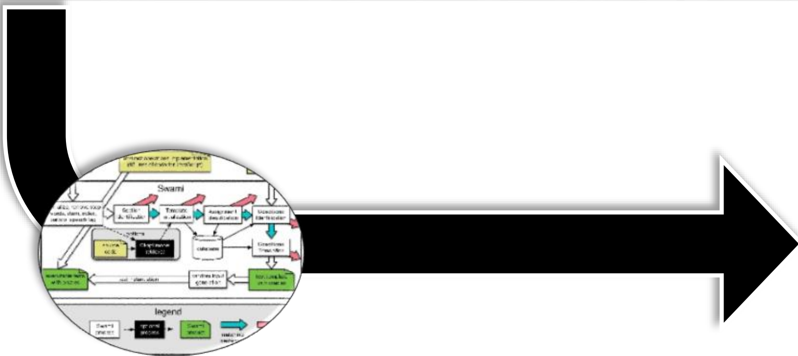
The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

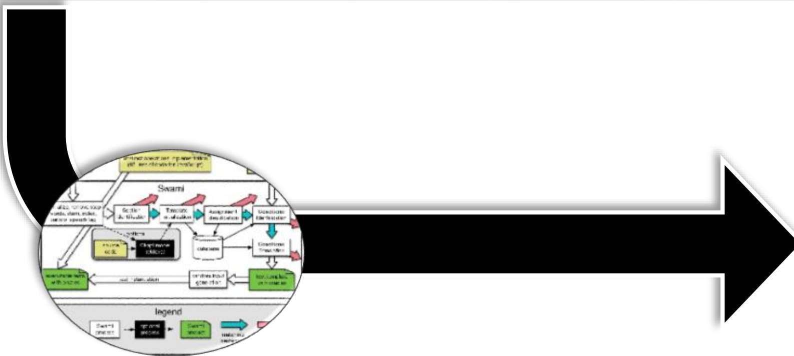


Swami

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefi**
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of len *searchStr*, return **true**.
12. Otherwise, return **false**.



Executable Test with Oracles

```
function IsRegExp(argument){
    return (argument instanceof RegExp);
}
```

Abstract Operations

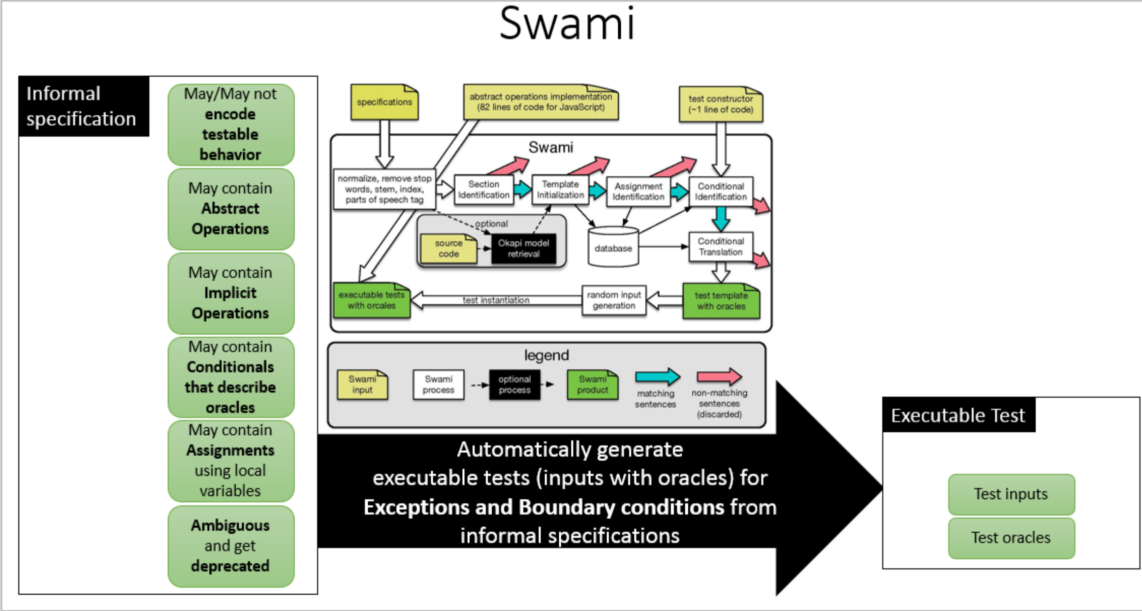
```
function test_string_prototype_startswith(thisObj,searchString,position) {

  if (IsRegExp(searchString) === true){
    try{
      var output = new String(thisObj).startsWith(searchString, position);
      return;
    }catch(e){
      assert.StrictEqual(true,(e instanceof TypeError));
      return;
    }
  }

   if (ToString(searchString).length +
       Math.min(Math.max(ToInteger(position), 0),
       ToString(RequireObjectCoercible(thisObj)).length) >
       ToString(RequireObjectCoercible(thisObj)).length){
     var output = new String(thisObj).startsWith(searchString, position);
     assert.strictEqual(output, false);
     return;
   }

}
```

Test Template encoding Oracles

```
test_string_prototype_startswith("Y3I9", "E0RS6GU078", 894);
test_string_prototype_startswith("T82LL6", 572, false);
test_string_prototype_startswith("XU6W0", "J3A", Infinity);
test_string_prototype_startswith("W5E74X0R", null, NaN);
...
```

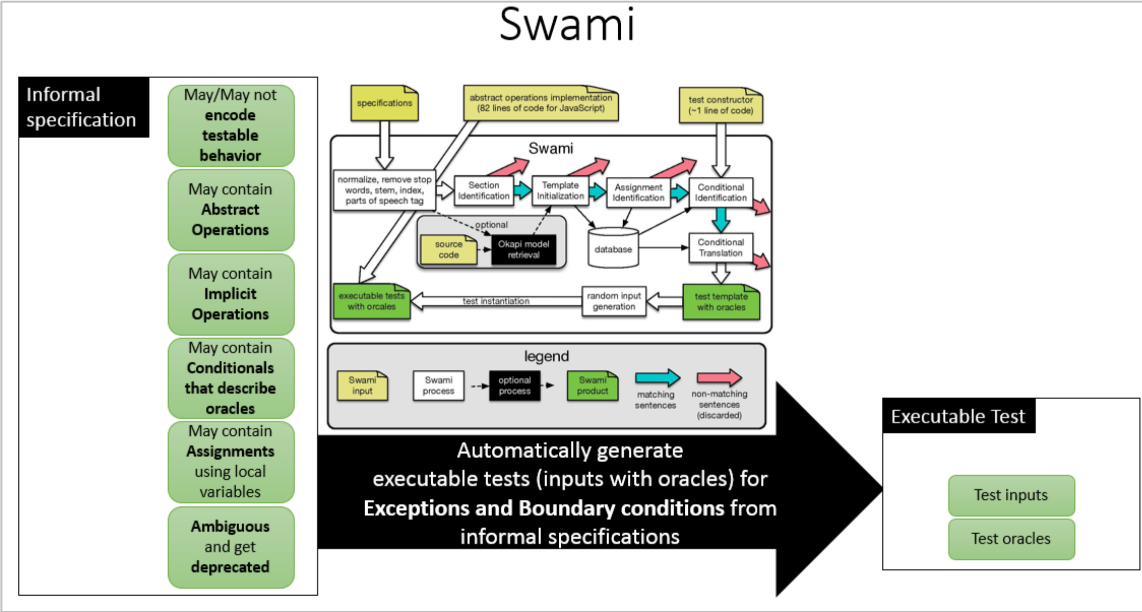
Test Inputs

Contributions

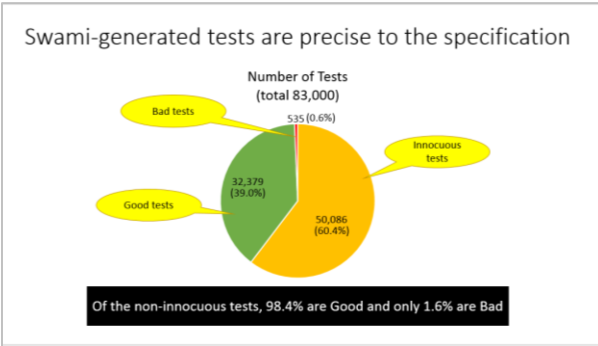


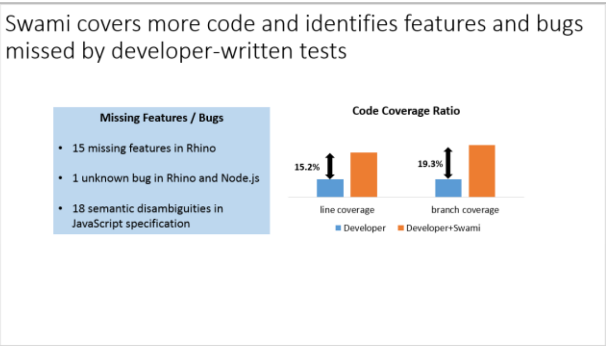
http://swami.cs.umass.edu

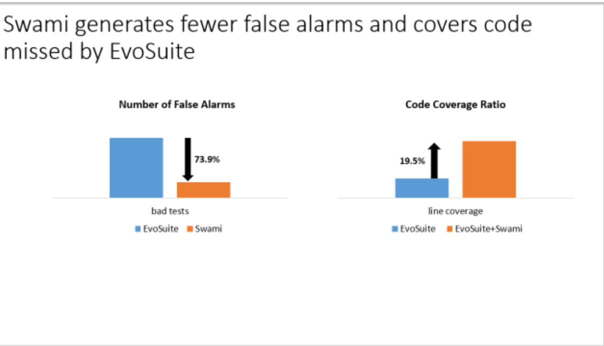
Contributions

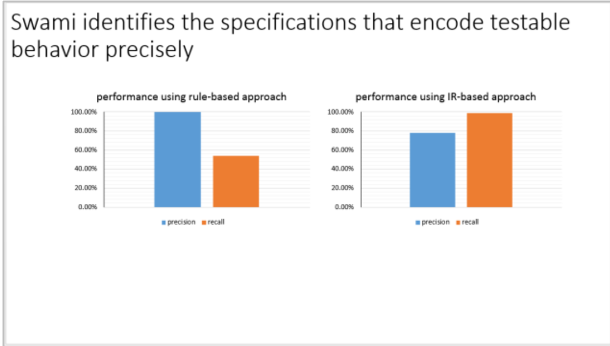


http://swami.cs.umass.edu

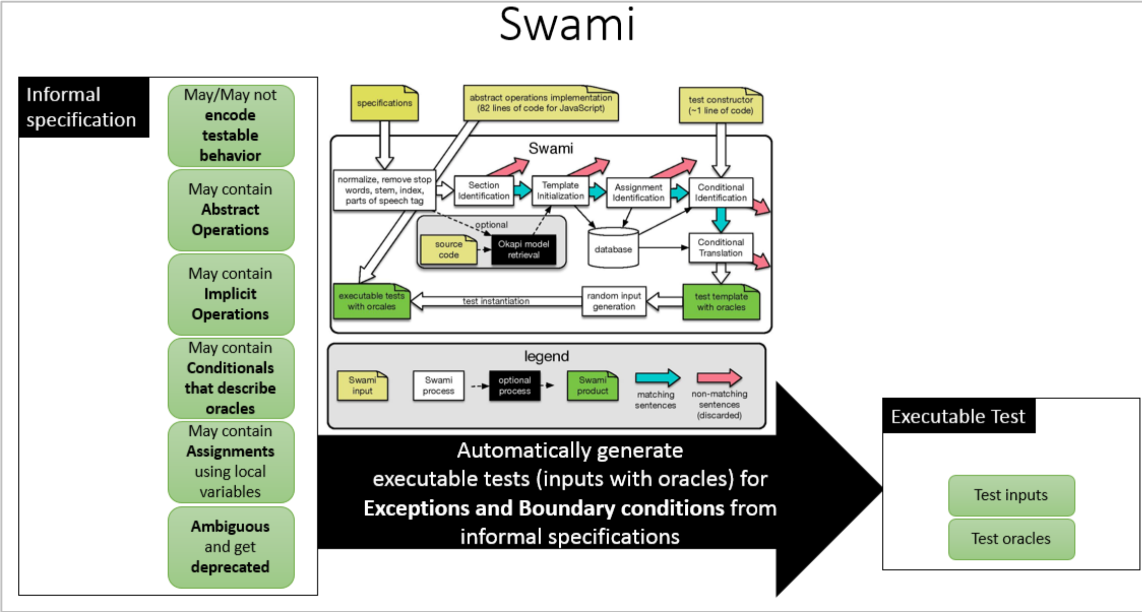




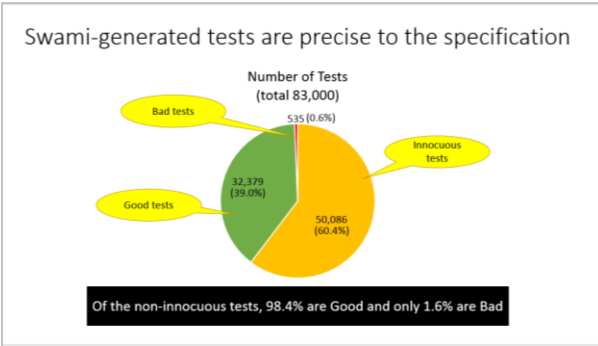


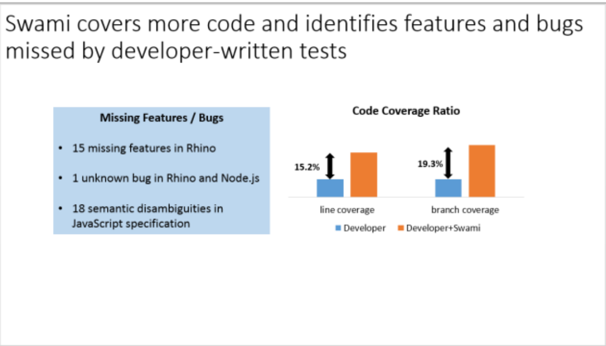


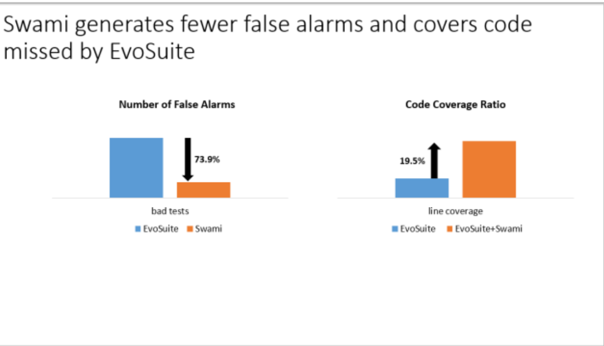
Contributions

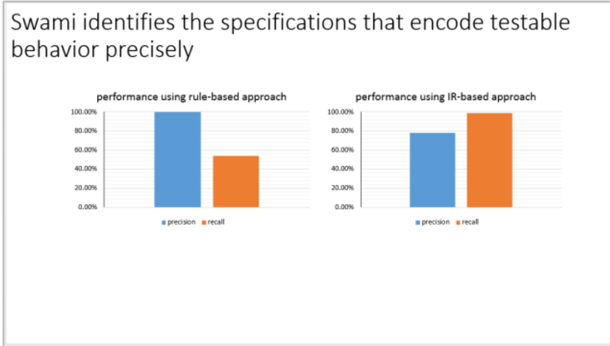


http://swami.cs.umass.edu









http://people.cs.umass.edu/~mmotwani