Automatically Generating Precise Oracles from Structured Natural Language Specifications

Manish Motwani

http://swami.cs.umass.edu

Yuriy Brun
The Test Oracle Problem

Input → Software Under Test → Actual Output
The Test Oracle Problem

Input → Software Under Test → Actual Output

Test Oracle (Expected Output)

Correct  Incorrect
The Test Oracle Problem

Input → Software Under Test → Actual Output

Test Oracle (Expected Output)

Correct

Incorrect

Easy to generate
The Test Oracle Problem

- Input
- Software Under Test
- Actual Output
- Test Oracle (Expected Output)

Correct

Incorrect

Easy to generate

Hard to generate
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]], [[Enumerable]], true, [[Configurable]]: true.

Executable Test

```javascript
/*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);
...
```

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

Number of Tests
(total 83,000)

50,086
(60.4%)

Innocuous tests
Swami-generated tests are precise to the specification

Number of Tests
(total 83,000)

- Innocuous tests: 50,086 (60.4%)
- Good tests: 32,379 (39.0%)
Swami-generated tests are precise to the specification

Number of Tests (total 83,000)

- Innocuous tests
  - 50,086 (60.4%)
- Bad tests
  - 535 (0.6%)
- Good tests
  - 32,379 (39.0%)
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

<table>
<thead>
<tr>
<th>Missing Features / Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 15 missing features in Rhino</td>
</tr>
<tr>
<td>• 1 unknown bug in Rhino and Node.js</td>
</tr>
<tr>
<td>• 18 semantic disambiguities in JavaScript specification</td>
</tr>
</tbody>
</table>
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

**Code Coverage Ratio**

- Line coverage: Developer 15.2%, Developer+Swami 19.3%
- Branch coverage: Developer 15.2%, Developer+Swami 19.3%
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?
Why is it hard to derive oracles from informal specifications?

Encode testable behavior
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}. 

Implicit Operations

Encode testable behavior

Abstract Operations
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?

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?

- **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?

- **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., \(@\text{params}\), \(@\text{throws}\) 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

Concrete oracles for uncommon 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.
What kind of oracles exist in informal specifications?

Informal specifications typically contain oracles for Exceptions and Boundary conditions.

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 infinities of opposite sign, is not defined.
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%

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

Goal of this work

Automatically generate executable tests (inputs with oracles) for Exceptions and Boundary conditions from structured informal specifications.
Automatically generate executable tests (inputs with oracles) for Exceptions and Boundary conditions from structured informal specifications.
Step 1: Identify specifications which encode testable behavior

Rules are regular expressions composed of POS tags, keywords, and wild card characters
Step 1: Identify specifications which encode testable behavior

Rule-based approach

Heading RE: [CD new* NN LRB NN.* RRB]
Body RE: [If .* return .*] [if .* throw .* exception]
Step 1: Identify specifications which encode testable behavior

Rule-based approach

when the format of specification document is unknown

Source code

Information Retrieval-based approach

<table>
<thead>
<tr>
<th>section ID</th>
<th>matched Java class</th>
<th>similarity score</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.4.2.2</td>
<td>ScriptRuntime.java</td>
<td>0.37</td>
</tr>
<tr>
<td>15.4.2.2</td>
<td>Interpreter.java</td>
<td>0.31</td>
</tr>
<tr>
<td>15.4.2.2</td>
<td>BaseFunction.java</td>
<td>0.25</td>
</tr>
<tr>
<td>15.4.2.2</td>
<td>ScriptableObject.java</td>
<td>0.24</td>
</tr>
<tr>
<td>15.4.2.2</td>
<td>NativeArray.java</td>
<td>0.21</td>
</tr>
</tbody>
</table>

OKAPI model
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
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).
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.
Step 2: Extract **method signature** from specification heading and initialize Test Template

```javascript
function test_string_prototype_startswith(thisObj, searchString, position) {} // Initialized Test Template
```

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

The following steps are taken:

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

```javascript
function test_< method name > (thisObj,<[ method args ]>) {} // Initialized Test Template
```

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`.

```javascript
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).
2. Let \( isReexp \) be ? Isregexp(searchString).
3. Let \( isReexp \) be ? Isregexp(searchString).
4. If isRegExp is true, throw a TypeError exception.
5. Let \( searchString \) 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. If \( pos \) is negative, let \( pos \) be \( len + pos \).
9. Let \( pos \) be ? ToInteger(pos).
10. Let \( suffix \) be \( S \) from \( \text{pos} \) to \( \text{len} \).
11. If \( suffix \) is undefined, return true.
12. Otherwise, return false.

**Initialized Test Template**

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

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

**Method invocation code**
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 \( searchString \) 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 \( searchString \).
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 \( searchString \), 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).
4. If isRegExp is true, throw a TypeError exception.
5. Let searchString 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 searchString.
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 searchString, 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($O$).
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.
Step 4: Identify and parse **Conditionals** to populate the conditional templates

```javascript
if (<condition>) {
    try {
        var output = <method invocation>;
        return;
    } catch (e) {
        <test constructor>(true, (<code>e</code> instanceof <expected error>));
        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`.  

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

Exception

```javascript
if (<condition>) {
    var output = <method invocation>;
    return;
} catch(e) {
    <test constructor>(true, (e instanceof <expected error>));
    return;
}
```
Step 4: Identify and parse **Conditionals** to populate the conditional templates

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

### Exception oracle

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

### Boundary condition oracle

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 `ToIntInteger(position)`. (If `position` is undefined, this step produces `null`)
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`. 
Step 4: Identify and parse **Conditionals** to populate the conditional templates

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

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

```javascript
if (isRegExp is true) {
    try {
        var output = new String(thisObj).startsWith(searchString, position);
        return;
    } catch(e) {
        assert.StrictEqual(true, (e instanceof TypeError));
        return;
    }
}
```
Step 4: Identify and parse **Conditionals** to populate the conditional templates

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

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

```javascript
if (isRegExp is true) {
    try {
        var output = new String(thisObj).startsWith(searchString, position);
        return;
    } catch (e) {
        assert.StrictEqual(true, (e instanceof TypeError));
        return;
    }
}
```
Step 4: Identify and parse **Conditionals** to populate the conditional templates.

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

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

Exception oracle

From step 2
Step 4: Identify and parse **Conditionals** to populate the conditional templates

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

```javascript
if (isRegExp is true) {
  try {
    var output = new String(thisObj).startsWith(searchString, position);
    return;
  } catch (e) {
    assertStrictEqual(true, (e instanceof TypeError));
    return;
  }
}
```
Step 4: 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(searchStr).`
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` length `searchLength` is the same as the full element sequence of `searchStr`, return `true`.
12. Otherwise, return `false`.

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

**Boundary Condition**

Boundary condition oracle
Step 5: Recursively substitute **local variables** and **implicit operations**

```javascript
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;
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>RequireObjectCoercible(this value)</td>
</tr>
<tr>
<td>S</td>
<td>ToString(this)</td>
</tr>
<tr>
<td>isRegExp</td>
<td>IsRegExp(searchString)</td>
</tr>
<tr>
<td>searchString</td>
<td>ToString(searchString)</td>
</tr>
<tr>
<td>pos</td>
<td>ToInteger(position)</td>
</tr>
<tr>
<td>len</td>
<td>length of S</td>
</tr>
<tr>
<td>start</td>
<td>min(max(pos,0),len)</td>
</tr>
<tr>
<td>searchLength</td>
<td>length of searchString</td>
</tr>
</tbody>
</table>

**Method Arguments:**

- `thisObj`
- `searchString`
- `position`
Step 5: Recursively substitute **local variables** and **implicit operations**

```javascript
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;
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>RequireObjectCoercible(this value)</td>
</tr>
<tr>
<td>S</td>
<td>ToString(O)</td>
</tr>
<tr>
<td>isRegExp</td>
<td>IsRegExp(searchString)</td>
</tr>
<tr>
<td>searchString</td>
<td>ToString(searchString)</td>
</tr>
<tr>
<td>pos</td>
<td>ToInteger(position)</td>
</tr>
<tr>
<td>len</td>
<td>length of S</td>
</tr>
<tr>
<td>start</td>
<td>min(max(pos, 0), len)</td>
</tr>
<tr>
<td>searchLength</td>
<td>length of searchString</td>
</tr>
</tbody>
</table>

**Method Arguments:**
- thisObj
- searchString
- position
Step6: Add conditionals to the initialized test template and check if it compiles

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

function test_stringPrototype_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) {
      assertStrictEqual(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);
    assertStrictEqual(output, false);
    return;
  }
}
Step 7: Instantiating test template by generating test inputs using random input generation

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

```javascript
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;
        }
    }else {
        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;
        }
    }
}
```
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($O$).
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).
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.
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).
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 $searchStr$, return true.
12. Otherwise, return false.

```javascript
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;
  }
}
```

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

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

Swami

http://swami.cs.umass.edu
Contributions

Swami-generated tests are precise to the specification

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

Swami generates fewer false alarms and covers code missed by EvoSuite

Swami identifies the specifications that encode testable behavior precisely

http://swami.cs.umass.edu
Contributions

Swami-generated tests are precise to the specification

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

Swami generates fewer false alarms and covers code missed by EvoSuite

Swami identifies the specifications that encode testable behavior precisely

http://swami.cs.umass.edu

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