2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Automatically Generating Precise Oracles from Structured Natural Language Specifications

Manish Motwani and Yuriy Brun
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, Massachusetts 01003-9264, USA
{mmotwani, brun}@cs.umass.edu

***Abstract*—Software specifications often use natural language to describe the desired behavior, but such specifications are difficult to verify automatically. We present Swami, an automated technique that extracts test oracles and generates executable tests from structured natural language specifications. Swami focuses on exceptional behavior and boundary conditions that often cause field failures but that developers often fail to manually write tests for. Evaluated on the official JavaScript specification (ECMA-262), 98.4% of the tests Swami generated were precise to the specification. Using Swami to augment developer-written test suites improved coverage and identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification.**

I. Introduction

One of the central challenges of software engineering is verifying that software does what humans want it to do. Unfortunately, the most common way humans describe and specify software is natural language, which is difficult to formalize, and thus also difficult to use in an automated process as an oracle of what the software should do.

This paper tackles the problem of automatically generating tests from natural language specifications to verify that the software does what the specifications say it should. Tests consist of two parts, an input to trigger a behavior and an oracle that indicates the expected behavior. Oracles encode intent and are traditionally specified manually, which is time consuming and error prone. While formal, mathematical specifications that can be used automatically by computers are rare, developers do write natural language specifications, often structured, as part of software requirements specification documents. For example, Figure 1 shows a structured, natural language specification of a JavaScript `Array(len)` constructor (part of the ECMA-262 standard). This paper focuses on generating oracles from such structured natural language specifications. (Test inputs can often be effectively generated randomly [20], [50], and together with the oracles, produce executable tests.)

Of particular interest is generating tests for exceptional behavior and boundary conditions because, while developers spend significant time writing tests manually [2], [55], they often fail to write tests for such behavior. In a study of ten popular, well-tested, open-source projects, the coverage of exception handling statements lagged significantly behind overall statement coverage [24]. For example, Developers often focus on the common behavior when writing tests and forget to account for exceptional or boundary cases [2]. At the same time, exceptional behavior is an integral part of the software as important as the common behavior. An IBM study found that up to two thirds of production code may be devoted to exceptional behavior handling [13]. And exceptional behavior is often more complex (and thus more buggy) because anticipating all the ways things may go wrong, and recovering when things do go wrong, is inherently hard. Finally, exceptional behavior is often the cause of field failures [73], and thus warrants high-quality testing.

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**}.

https://www.ecma-international.org/ecma-262/5.1/#sec-15.4.2.2

Fig. 1. Section 15.4.2.2 of ECMA-262 (v5.1), specifying the JavaScript `Array(len)` constructor.

This paper presents Swami, a technique for automatically generating executable tests from natural language specifications. We scope our work by focusing on exceptional and boundary behavior, precisely the important-in-the-field behavior developers often undertest [24], [73].

Swami uses regular expressions to identify what sections of structured natural language specifications encode testable behavior. (While not required, if the source code is available, Swami can also use information retrieval techniques to identify such sections.) Swami then applies a series of four regular-expression-based rules to extract information about the syntax for the methods to be tested, the relevant variable assignments, and the conditionals that lead to visible oracle behavior, such as return statements or exception throwing statements. Swami then backtracks from the visible-behavior statements to recursively fill in the variable value assignments according to the specification, resulting in a test template encoding the

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00035

oracle, parameterized by test inputs. Swami then generates random, heuristic-driven test inputs to produce executable tests.

Using natural language specifications pose numerous challenges. Consider the ECMA-262 specification of a JavaScript `Array(len)` constructor in Figure 1. The specification:

- Uses natural language, such as "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)."
- Refers to abstract operations defined elsewhere in the specification, such as `ToUint32`, which is defined in section 9.6 of the specification (Figure 2).
- Refers to implicit operations not formally defined by the specification, such as `min`, `max`, `is not equal to`, `is set to`, `is an element of`, and `is greater than`.
- Describes complex control flow, such as conditionals, using the outputs of abstract and implicit operations in other downstream operations and conditionals.

We evaluate Swami using ECMA-262, the official specification of the JavaScript programming language [76], and two well known JavaScript implementations: Java Rhino and C++ Node.js built on Chrome's V8 JavaScript engine. We find that:

- Of the tests Swami generates, 60.3% are innocuous — they can never fail. Of the remaining tests, 98.4% are precise to the specification and only 1.6% are flawed and might raise false alarms.
- Swami generates tests that are complementary to developer-written tests. Our generated tests improved the coverage of the Rhino developer-written test suite and identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification.
- Swami also outperforms and complements state-of-the-art automated test generation techniques. Most tests generated by EvoSuite (which does not automatically extract oracles) that cover exceptional behavior are false alarms, whereas 98.4% of Swami-generated tests are correct tests that cannot result in false alarms. Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 Rhino classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.

While Swami's regular-expression-based approach is rather rigid, it performs remarkably well in practice for exceptional and boundary behavior. It forms both a useful tool for generating tests for such behavior, and a baseline for further research into improving automated oracle extraction from natural language by using more advanced information retrieval and natural language processing techniques.

Our research complements prior work on automatically generating test inputs for regression tests or manually-written oracles, such as EvoSuite [20] and Randoop [50], by automatically extracting oracles from natural language specifications. The closest work to ours is Toradacu [24] and Jdoctor [8], which focus on extracting oracles for exceptional behavior, and @tComment [65], which focuses on extracting preconditions related to nullness of parameters. These techniques are limited to using Javadoc comments, which are simpler than the specifications Swami tackles because Javadoc comments (1) provide specific annotations for pre- and post-conditions, including `@param`, `@throws`, and `@returns`, making them more formal [65]; (2) are collocated with the method implementations they specify, (3) use the variable names as they appear in the code, and (4) do not contain references to abstract operations specified elsewhere. Additionally, recent work showed that Javadoc comments are often out of date because developers forget to update them when requirements change [65]. Our work builds on @tComment, Toradacu, and Jdoctor, expanding the rule-based natural language processing techniques to apply to more complex and more natural language. Additionally, unlike those techniques, Swami can generate oracles for not only exceptional behavior but also boundary conditions. Finally, prior test generation work [8], [20], [24], [50] requires access to the source code to be tested, whereas Swami can generate black-box tests entirely from the specification document, without needing the source code.

Our paper's main contributions are:

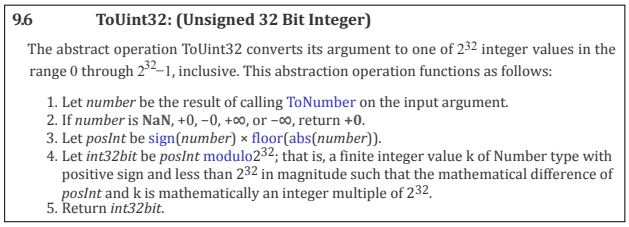
- Swami, an approach for generating tests from structured natural language specifications.
- An open-source prototype Swami implementation, including rules for specification documents written in ECMAscript style, and the implementations of common abstract operations.
- An evaluation of Swami on the ECMA-262 JavaScript language specification, comparing Swami-generated tests to those written by developers and those automatically generated by EvoSuite, demonstrating that Swami generates tests often missed by developers and other tools and that lead to discovering several unknown defects in Rhino and Node.js.
- A replication package of all the artifacts and experiments described in this paper.

The Swami implementation and a replication package are available at http://swami.cs.umass.edu/.

The rest of this paper is structured as follows. Section II illustrates Swami on an example specification in Figure 1. Section III details the Swami approach and Section IV evaluates it. Section V places our research in the context of related work, and Section VI summarizes our contributions.

II. Intuition Behind Swami

To explain our approach for generating test oracles, we will use ECMA-262, the official specification of the JavaScript programming language [76]. The ECMA-262 documentation consists of hundreds of sections, including an index, a description of the scope, definitions, notational conventions, language semantics, abstract operations, and, finally, methods supported by the JavaScript language.



9.6 ToUint32: (Unsigned 32 Bit Integer)

The abstract operation ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through $2^{32}-1$, inclusive. This abstraction operation functions as follows:

1. Let *number* be the result of calling ToNumber on the input argument.
2. If *number* is **NaN**, +0, −0, +∞, or −∞, return **+0**.
3. Let *posInt* be sign(*number*) × floor(abs(*number*)).
4. Let *int32bit* be *posInt* modulo 2^{32}; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such that the mathematical difference of *posInt* and k is mathematically an integer multiple of 2^{32}.
5. Return *int32bit*.

https://www.ecma-international.org/ecma-262/5.1/index.html#sec-9.6

Fig. 2. ECMA specifications include references to abstract operations, which are formally defined elsewhere in the specification document, but have no public interface. Section 9.6 of ECMA-262 (v5.1) specifies the abstract operation **`ToUnit32`**, referenced in the specification in Figure 1.

Our technique, Swami, consists of three steps: identifying parts of the documentation relevant to the implementation to be tested, extracting test templates from those parts of the documentation, and generating executable tests from those templates. We now illustrate each of these steps on the **`Array(len)`** constructor specification from ECMA-262 (Figure 1).

To use Swami to generate tests for a project, the developer needs to manually specify two things. First, a constructor for instantiating test cases. For example, for Rhino, the constructor is **`new TestCase(test name, test description, expected output, actual output)`**. This makes Swami project- and language-agnostic, for example, we were able to generate tests for Rhino and Node.js, two JavaScript engine implementations, one written in Java and one in C++, simply by specifying a different test constructor. Second, an implementation of abstract operations used in the specification. For example, the specification in Figure 1 uses the abstract operation **`ToUnit32(len)`**, specified in a different part of the specification document (Figure 2), and Swami needs an executable method that encodes that operation. For JavaScript, we found that implementing 10 abstract operations, totaling 82 lines of code, was sufficient for our purposes. Most of these abstract operation implementations can be reused for other specifications, so the library of abstract operation implementations we have built can be reused, reducing future workload.

A. Identifying parts of the documentation relevant to the implementation to be tested

The first step of generating test oracles is to identify which sections of the ECMA-262 documentation encode testable behavior. There are two ways Swami can do this. For documentation with clearly delineated specifications of methods that include clear labels of names of those methods, Swami uses a regular expression to match the relevant specifications and discards all documentation sections that do not match the regular expression. For example, Figure 1 shows the specification of the **`Array(len)`** constructor, clearly labeling the name of that method at the top. This is the case for all of ECMA-262, with all specifications of methods clearly identifiable and labeled. For specifications like this one, Swami uses a regular expression that matches the section number and the method name (between the section number and the open parenthesis). Section III-C will describe the full details of this step.

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

Fig. 3. Using the information-retrieval-based approach to identify the documentation sections relevant to the implementation, Swami finds these Rhino classes as most relevant to section 15.4.2.2 of the ECMA-262 (v5.1) documentation (the specification of the **`Array(len)`** constructor from Figure 1). `NativeArray.java` is the class that implements the **`Array(len)`** constructor, and the other four classes all depend on this implementation. Each of the classes listed appears in the `org.mozilla.javascript` package, e.g., `ScriptRuntime.java` appears in `org.mozilla.javascript.ScriptRuntime.java`.

When the documentation is not as clearly delineated, Swami can still identify which sections are relevant, but it requires access to the source code. However, to reiterate, this step, and the source code, are not necessary for ECMA-262, hundreds of other ECMA standards, and many other structured specification documents. Swami uses the Okapi information retrieval model [57] to map documentation sections to code elements. The Okapi model uses term frequencies (the number of times a term occurs in a document) and document frequency (the number of documents in which a term appears) to map queries to document collections. Swami uses the specification as the query and the source code as the document collection. Swami normalizes the documentation, removes stopwords, stems the text, and indexes the text by its term frequency and document frequency. (Section III-B describes this step in detail.) This allows determining which specifications are relevant to which code elements based on terms that appear in both, weighing more unique terms more strongly than more common terms. For example, when using the Rhino implementation, this approach maps the specification from Figure 1 as relevant to the five classes listed in Figure 3. The 5th-ranked class, `NativeArray.java`, is the class that implements the **`Array(len)`** constructor, and the other four classes all depend on this implementation. Swami uses a threshold similarity score to determine if a specification is relevant. We empirically found that a similarity score threshold of 0.07 works well in practice: If at least one class is relevant to a specification document above this threshold, then the specification document should be used to generate tests. Section IV-D will evaluate this approach, showing that with this threshold, Swami's precision is 79.0% and recall is 98.9%, but that Swami can use regular expressions to remove improperly identified specifications, boosting precision to 93.1%, and all tests generated from the remaining improperly identified specifications will fail to compile and can be discarded automatically.

B. Extracting test templates

For the specification sections found relevant, Swami next uses rule-based natural language processing to create test templates: source code parameterized by (not yet generated)

test inputs that encodes the test oracle. Swami generates two types of tests: boundary condition and exceptional condition tests. For both kinds, Swami identifies in the specification (1) the signature (the syntax and its arguments) of the method to be tested, the (2) expected output for boundary condition tests, and (3) the expected error for exceptional condition tests.

The `Array(len)` specification shows that the constructor has one argument: `len` (first line of Figure 1). The specification dictates that `Array(len)` should throw a `RangeError` exception if `len` is not equal to `ToUnit32(len)` (second paragraph of Figure 1), where `ToUnit32(len)` is an abstract operation defined elsewhere in the specification (Figure 2). We now demonstrate how Swami generates tests for this exceptional condition.

We have written four rules that Swami uses to extract this information. Each rule consists of a set of regular expressions that determine if a natural language sentence satisfies the conditions necessary for the rule to apply, and another set of regular expressions that parse relevant information from the sentence. The rule also specifies how the parsed information is combined into a test template. Swami applies the four rules (described in detail in Section III-C) in series.

The first rule, *Template Initialization*, parses the name of the method being tested and its arguments. The rule matches the heading of the specification with a regular expression to identify if the heading contains a valid function signature i.e., if there exists a function name and function arguments. In the example specification shown in Figure 1, the top line (heading) "15.4.2.2 new Array(len)" matches this regular expression. Swami then creates an empty *test template*:

```
function test_new_array(len){}
```

and stores the syntax for invoking the function (`var output = new Array(len)`) in its database for later use.

The second rule, *Assignment Identification*, identifies assignment descriptions in the specification that dictate how variables' values change, e.g., "Let posInt be sign(number) × floor(abs(number))" in Figure 2. For assignment sentences, Swami's regular expression matches the variable and the value assigned to the variable and stores this association in its database. Here, Swami would match the variable `posInt` and value `sign(number) × floor(abs(number))`. This populates the database with variable-value pairs as described by the specification. These pairs will be used by the third and fourth rules.

The third rule, *Conditional Identification*, uses a regular expression to extract the condition that leads to returning a value or throwing an error. The rule fills in the following template:

```
if (<condition>){
    try{
        <function call>
    }
    catch(e){
        <test constructor>(true, eval(e instanceof <
            expected error>))
    }
}
```

replacing "`<function call>`" with the method invoking syntax "`var output = new Array(len)`" from the database (recall it being stored there as a result of the first rule). The `<test constructor>` is a project-specific description of how test cases are written. This is written once per project, by a developer.

The sentence "If the argument len is a Number and ToUnit32(len) is not equal to len, a RangeError exception is thrown" (Figure 1) matches this expression and Swami extracts the condition "argument len is a Number and ToUnit32(len) is not equal to len" and the expected error "RangeError exception is thrown", populating the following template, resulting in:

```
if (argument len is a Number and ToUnit32(len) is not
     equal to len){
    try{
        var output = new Array(len);
    }
    catch(e){
        new TestCase("test_new_array_len", "
            test_new_array_len", true, eval(e
            instanceof a RangeError exception is thrown
            ));
        test();
    }
}
```

Finally, the forth rule, *Conditional Translation*, combines the results of the previous steps to recursively substitute the variables and implicit operations used in the conditionals from the previous rule with their assigned values, until all the variables left are either expressed in terms of the method's input arguments, or they are function calls to abstract operations. Then, Swami embeds the translated conditional into the initialized test template, creating a test template that encodes the oracle from the specification. In our example, this step translates the above code into:

```
function test_new_array(len){
    if (typeof(len)=="number" && (ToUint32(len)!=len)){
        try{
            var output = new Array ( len );
        }catch(e){
            new TestCase("array_len", "array_len", true,
                eval(e instanceof RangeError))
            test();
        }
    }
}
```

C. Generating executable tests

Swami uses the above test template with the oracle to instantiate executable tests. To do this, Swami generates random inputs for each argument in the template (`len`, in our example). ECMA-262 describes five primitive data types: Boolean, Null, Undefined, Number, and String (and two non-primitive data types, Symbol and Object). Swami generates random instances of the five primitive data types and several subtypes of Object (Array, Map, Math, DateTime, RegExp, etc.) using heuristics, as described in Section III-D. We have found empirically that generating such inputs for all arguments tests parts of the code uncovered by developer written tests, which is consistent with prior studies [24]. Combined with the generated oracle encoded in the template, each of these test inputs forms a complete test.

Swami generates 1,000 random test inputs for every test template. Many of these generated tests will not trigger the generated conditional, but enough do, as Section IV

```
/*ABSTRACT FUNCTIONS*/
function ToUint32(argument){
        var number = Number(argument)
        if (Object.is(number, NaN) || number == 0 || number
              == Infinity || number == -Infinity || number
             == +0 || number== -0){
            return 0
        }
        var i = Math.floor(Math.abs(number))
        var int32bit = i%(Math.pow(2,32))
    return int32bit
}
...
/*TEST TEMPLATE GENERATED AUTOMATICALLY*/
function test_new_array(len){
    if (typeof(len)=="number" && (ToUint32(len)!=len)){
        try{
            var output = new Array ( len );
        }catch(e){
            new TestCase("array_len", "array_len", true,
                 eval(e instanceof RangeError))
            test();
        }
    }
}
/*TESTS GENERATED AUTOMATICALLY*/
test_new_array(1.1825863363010669e+308);
test_new_array(null);
test_new_array(-747);
test_new_array(368);
test_new_array(false);
test_new_array(true);
test_new_array("V7KO08H");
test_new_array(Infinity);
test_new_array(undefined);
test_new_array(/[^.]+/);
test_new_array(+0);
test_new_array(NaN);
test_new_array(-0);
...
```

Fig. 4. The executable tests automatically generated for the `Array(len)` constructor from the specification in Figure 1.

evaluates. The bottom part of Figure 4 shows representative tests automatically generated for the `test_new_array` template.

Finally, Swami augments the generated test file with the manually implemented abstract operations. Figure 4 shows the final test file generated to test the `Array(len)` constructor.

III. The Swami Approach

This section details the Swami approach (Figure 5). Swami normalizes and parts-of-speech tags the specifications (Section III-A), identifies the relevant sections (Section III-B), uses regular-expression-based rules to create a test template encoding an oracle (Section III-C), and instantiates the tests via heuristic-based, random input generation (Section III-D).

A. Specification prepossessing

Swami uses standard natural language processing [27] to convert the specification into more parsable format. Swami normalizes the text by removing punctuation (remembering where sentences start and end), case-folding (converting all characters to lower case), and tokenizing terms (breaking sentences into words). Then Swami removes stopwords (commonly used words that are unlikely to capture semantics, such as "to", "the", "be"), and stems the text to conflate word variants (e.g., "ran", "running", and "run") to improve term matching. Swami uses the Indri toolkit [64] for removing stopwords and stemming. Swami then tags the parts of speech using the Standford coreNLP toolkit [39].

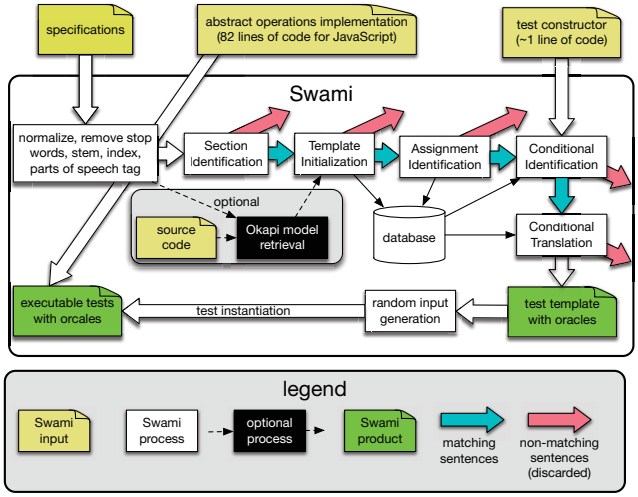


Fig. 5. Swami generates tests by applying a series of regular expressions to processed (e.g., tagged with parts of speech) natural language specifications, discarding non-matching sentences. Swami does not require access to the source code; however, Swami can optionally use the code to identify relevant specifications. Swami's output is executable tests with oracles and test templates that can be instantiated to generate more tests.

B. Identifying parts of the documentation relevant to the implementation to be tested

As Section II-A described, Swami has two ways of deciding which sections of the specification to generate tests from. For many structured specifications such as ECMA-262, the sections that describe methods are clearly labeled with the name of the method (e.g., see Figure 1). For such specifications, Swami uses a regular expression it calls *Section Identification* to identify the relevant sections.

Swami's *Section Identification* regular expression discards white space and square brackets (indicating optional arguments), and looks for a numerical section label (which is labeled as "CD" by the parts of speech tagger), followed optionally by the "new" keyword (labeled "JJ"), a method name (labeled "NN"), then by a left parenthesis (labeled "LRB"), then arguments (labeled "NN"), and then a right parenthesis (labeled "RRB"). If the first line of a section matches this regular expression, Swami will attempt to generate tests; otherwise, this section is discarded. For example, the heading of the specification in Figure 1, "15.4.2.2 new Array (len)" is pre-processed to

```
[('15.4.2.2', 'CD'), ('new', 'JJ'), ('Array', 'NN'), ('(
    ', '-LRB-'), ('len', 'NN'), (')', '-RRB-')]
```

matching the *Section Identification* regular expression.

If the specifications do not have a clear label, Swami can use information retrieval to identify which sections to generate tests for. (This step does not require the parts of speech tags.) The fundamental assumption underlying this approach is that some terms in a specification will be found in relevant source files. Swami considers the source code to be a collection of documents and the specification to be a query. Swami parses the abstract syntax trees of the source code files and identifies comments and class, method, and variable names. Swami then splits the extracted identifiers

into tokens using CamelCase splitting. Swami will index both the split and intact identifies, as the specifications can contain either kind. Swami then tokenizes the specification and uses the Indri toolkit [64] for stopword removal, stemming, and indexing by computing the term frequency, the number of times a term occurs in a document, and document frequency, the number of documents in which a term appears. Finally, Swami uses the Indri toolkit to apply the Okapi information retrieval model [57] to map specification sections to source code, weighing more unique terms more strongly than more common terms. We chose to use the Okapi model because recent studies have found it to outperform more complex models on both text and source code artifacts [53], [67]. The Okapi model computes the inverse document frequency, which captures both the frequency of the co-occurring terms and the uniqueness of those terms [80]. This model also weighs more heavily class and method names, which can otherwise get lost in the relatively large number of variable names and comment terms. Swami will attempt to generate tests from all specifications with a similarity score to at least one class of 0.07 or above (recall the example in Figure 3). We selected this threshold by computing the similarity scores of a small subset (196 sections) of ECMA-262, selecting the minimum similarity score of the relevant specifications (where relevant is defined by the ground-truth [17], described in Section IV-D). This prioritizes recall over precision. More advanced models [56] might improve model quality by improving term weights, but as Section IV-D will show, this model produces sufficiently high-quality results.

C. Extracting test templates

To generate test templates, Swami uses rule-based natural language processing on the specification sections identified as relevant in Section III-B. Swami applies four rules — *Template Initialization*, *Assignment Identification*, *Conditional Identification*, and *Conditional Translation* — in series. Each rule first uses a regular expression to decide if a sentence should be processed by the rule. If it should, the rule then applies other regular expressions to extract salient information, such as method names, argument names, assignments, conditional predicates, etc. When sentences do not match a rule, they are discarded from processing by further rules (recall Figure 5). Only the sentences that match all four rules will produce test templates.

Rule 1: ***Template Initialization*** locates and parses the name of the method being tested and the arguments used to run that method, and produces a test template with a valid method signature and an empty body. *Template Initialization* works similarly to the *Section Identification* regular expression; it matches the numerical section label, method name, and arguments by using their parts of speech tags and parentheses. Unlike *Section Identification*, *Template Initialization* also matches and extracts the method and argument names. Swami generates a test name by concatenating terms in the method under test's name with "_", and concatenating the extracted arguments with ", ", populating the following template:

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

Swami uses the `thisObj` variable as an argument to specify `this`, an object on which this method will be invoked.

For example, the specification section "21.1.3.20 String.prototype.startsWith (searchString [,position])" results in the empty test template

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

Swami identifies the type of the object on which the tested method is invoked and stores the method's syntax in a database. For example, the `String.startsWith` method is invoked on *String* objects, so Swami stores `var output = new String(thisObj).startsWith(searchString, position)` in the database.

Rule 2: ***Assignment Identification*** identifies assignment sentences and produces pairs of variables and their values, which it stores in Swami database. Specifications often use intermediate variables, described in the form of assignment sentences. For example, the `ToUint32` specification (Figure 2) contained the assignment sentence "Let `posInt` be `sign(number) × floor(abs(number))`" and `posInt` was used later in the specification. Assignment sentences are of the form "Let `<var>` be `<value>`", where `<var>` and `<value>` are phrases. *Assignment Identification* uses a regular expression to check if a sentence satisfies this form. If it does, regular expressions extract the variable and value names, using the parts of speech tags and keywords, such as "Let". Values can be abstract operations (e.g., `ToUint32`), implicit operations (e.g., `max`), constant literals, or other variables. Each type requires a separate regular expression. For the example assignment sentence from Figure 2, Swami extracts variable `posInt` and value `sign(number) × floor(abs(number))`. Swami inserts this variable-value pair into its database. This populates the database with assigned variable-value pairs as specified by the specification.

Rule 3: ***Conditional Identification*** identifies conditional sentences that result in a return statement or the throwing of an exception, and produces source code (an `if-then` statement) that encodes the oracle capturing that behavior. Our key observation is that exceptional and boundary behavior is often specified in conditional sentences and that this behavior can be extracted by regular expressions, capturing both the predicate that should lead to the behavior and the expected behavior. Conditional sentences are of the form "If `<condition> <action>`", where `<condition>` and `<action>` are phrases, often containing variables and method calls described in assignment statements. *Conditional Identification* uses three regular expressions to check if a sentence satisfies this form and if the `<action>` is a boundary condition or an exception. If `If * NN * is .* return .*` or `If * NN * the result is .*` match the sentence, Swami extracts the predicate `<condition>` as the text that occurs between the words `If` and either `return` or `the result is`. Swami extracts the `<action>` as the text that occurs after the words `return` or `the result is`. If `If * NN * is .* throw .* exception` matches the sentence, Swami extracts the predicate `<condition>` as the text that occurs between `If` and `throw` and the expected exception

as the noun (NN) term between `throw` and `exception`. For example, consider the following three sentences taken from section 21.1.3.20 of the ECMA-262 specification for the `String.prototype.startsWith` method. The sentence "If isRegExp is true, throw a TypeError exception." results in a `condition-action` pair `isRegExp is true - TypeError`. The sentence "If searchLength+start is greater than len, return false." results in `searchLength+start is greater than len - false`. And the sentence "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." results in `the sequence of elements of S starting at start of length searchLength is same as the full element sequence of searchStr - true`.

Finally, *Conditional Identification* generates source code by filling in the following templates for exceptional behavior:

```
if (<condition>){
    try{
        <function call>
        return
    }catch(e){
        <test constructor>(true, eval(e instanceof <
            action>))
        return;
    }
}
```

and for `return` statement behavior:

```
if (<condition>){
        <function call>
        <test constructor>(output, <action>))
        return;
    }
}
```

The `<function call>` placeholder is replaced with the code generated by *Template Initialization* (and stored in the database) to invoke the method under test, e.g., `var output = new String(thisObj).startsWith(searchString, position)`. The `<test constructor>` placeholder is a project-specific description of how test cases are written, and is an input to Swami (recall Figure 5 and Section II); for Rhino, it is `new TestCase(test name, test description, expected output, actual output)`. Figure 6 shows the code *Template Initialization* generates from the three conditional sentences in the `String.startsWith` specification.

Rule 4: *Conditional Translation* processes the oracle code generated by *Conditional Identification* by recursively filling in the variable value assignments according to the specification and injects the code into the template produced by *Template Initialization*, producing a complete test template.

Conditional Translation identifies the intermediate variables in the code generated by *Conditional Identification* and replaces them with their values from the database (placed in the database by *Assignment Identification*). If an intermediate variable does not have a value in the database (e.g., because the specification was incomplete or because the text describing this variable did not match *Assignment Identification*'s regular expressions), this test will fail to compile and Swami will remove the entire condition from the test. Next, Swami translates the implicit operations in the code. For example, `is greater than or equal to` and $\geq$ are translated to `>=`; `this value` is translated to `thisObj`; `is exactly`, `is equal to`, and `is` are translated to `===`; `<x> is one of <a>, <b>, ...` is translated to `x===a || x===b...`; `the number of elements in <S>` is translated to `<S>.length`; etc. Swami contains 54 such patterns, composed of keywords, special characters, wildcard characters, and parts of speech tags.

```
if (isRegExp is true){
    try{
        var output = new String(thisObj).startsWith(
            searchString, position)
        return
    }catch(e){
        new TestCase(<test name>, <test description>,
            true, eval(e instanceof TypeError))
        return;
    }
}
if (searchLength+start is greater than len){
        var output = new String(thisObj).startsWith(
            searchString, position)
        new TestCase(<test name>, <test description>,
            output, false))
        return;
    }
}
if (the sequence of elements of S starting at start of
     length searchLength is same as the full element
     sequence of searchStr){
        var output = new String(thisObj).startsWith(
            searchString, position)
        new TestCase(<test name>, <test description>,
            output, true))
        return;
    }
}
```

Fig. 6. The test code generated by the *Template Initialization* rule from the JavaScript `String.startsWith` specification.

An inherent limitation of the regular expressions is that they are rigid, and fail on some sentences. For example, Swami fails on the sentence "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." Swami correctly encodes most variable values and implicit operations, but fails to encode `the sequence of elements of S starting at the start of length searchLength`, resulting in a non-compiling `if` statement, which Swami removes as a final post-processing step.

Swami adds each of the translated conditionals that compiles to the test template initialized by *Template Initialization*. Figure 7 shows the final test template Swami generates for the `String.startsWith` JavaScript method.

D. Generating executable tests

Swami instantiates the test template via heuristic-driven random input generation. ECMA-262 describes five primitive data types: Boolean, Null, Undefined, Number, and String, and two non-primitive data types, Symbol and Object. Swami uses the five primitive data types and several subtypes of Object (Array, Map, Math, DateTime, RegExp, etc.). For Boolean, Swami generates `true` and `false`; for Null, `null`; for Undefined, `undefined`; for Number, random integers and floating point values. Swami also uses the following special values: `NaN`, `-0`, `+0`, `Infinity`, and `-Infinity`. For the Object subtypes, Swami follows several heuristics for generating Maps, Arrays, regular expressions, etc. For example, when generating an Array,

```
function test_string_prototype_startswith(thisObj,
    searchString, position){}
  if (isRegExp === true){
      try{
          var output = new String(thisObj).startsWith(
              searchString, position)
          return
      }catch(e){
          new TestCase(<test name>, <test description>,
              true, eval(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)
          new TestCase(<test name>, <test description>,
              output, false))
          return;
      }
  }
}
```

Fig. 7. The final test template Swami generates for the `String.startsWith` JavaScript method.

Swami ensures to generate only valid length arguments (an integer between 1 and 2^{32}). (Invalid lengths throw a `RangeError` at runtime, which would result in false alarm test failures.)

Finally, Swami augments the test suite with the developer written implementation of abstract operations used in the specification. Recall that for JavaScript, this consisted of 82 total lines of code, and that most of these abstract operation implementations can be reused for other specifications.

IV. Evaluation

We evaluate Swami on two well-known JavaScript implementations: Mozilla's Rhino (in Java) and Node.js (in C++, built on Chrome's V8 JavaScript engine). We answer four research questions:

- RQ1 **How precise are Swami-generated tests?** Of the tests Swami generates, 60.3% are innocuous — they can never fail. Of the remaining tests, 98.4% are precise to the specification and only 1.6% are flawed and might raise false alarms.
- RQ2 **Do Swami-generated tests cover behavior missed by developer-written tests?** Swami-generated tests identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. Further, Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods. The average statement coverage for these methods improved by 15.2% and the average branch coverage improved by 19.3%.
- RQ3 **Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools?** We compare Swami to EvoSuite and find that most of the EvoSuite-generated failing tests that cover exceptional behavior are false alarms, whereas 98.4% of the Swami-generated tests are precise to the specification and can only result in true alarms. Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.
- RQ4 **Does Swami's Okapi model precisely identify relevant specifications?** The Okapi model's precision is 79.0% and recall is 98.9%, but further regular expressions can remove improperly identified specifications, increasing precision to 93.1%; all tests generated from the remaining improperly identified specifications fail to compile and can be automatically discarded.

A. RQ1: How precise are Swami-generated tests?

Swami's goal is generating tests for exceptional behavior and boundary conditions, so we do not expect it to produce tests for most of the specifications. Instead, in the ideal, Swami produces highly-precise tests for a subset of the specifications.

We applied Swami to ECMA-262 (v8) to generate black-box tests for JavaScript implementations. Swami generated test templates for 98 methods, but 15 of these test templates failed to compile, resulting in 83 compiling templates. We manually examined the 83 test templates and compared them to the natural language specifications to ensure that they correctly capture specification's oracles. We then instantiated each template with 1,000 randomly generated test inputs, creating 83,000 tests. We instrumented test templates to help us classify the 83,000 tests into three categories: (1) good tests that correctly encode the specification and would catch some improper implementation, (2) bad tests that incorrectly encode the specification and could fail on a correct implementation, and (3) innocuous tests that pass on all implementations.

Of the 83,000 tests, 32,379 (39.0%) tests were good tests, 535 (0.6%) were bad tests, and 50,086 (60.3%) were innocuous.

It is unsurprising that innocuous tests are common. For example, if a test template checks that a RangeError exception is thrown (recall Figure 1), but the randomly-generated inputs are not outside the allowed range, the test can never fail. Innocuous tests cannot raise false alarms, since they can never fail, but could waste test-running resources. Existing test prioritization tools may be able to remove innocuous tests.

> Of the non-innocuous tests, 98.4% (32,379 out of 32,914) are good tests. We conclude that Swami produces tests that are precise to the specification.

We manually analyzed the 535 bad tests. (Of course, many of these tests could be grouped into a few equivalence classes based on the behavior they were triggering.) All bad tests came from 3 specifications. First, 176 of the tests tested the `ArrayBuffer.prototype.slice` method, and all invoked the method on objects of type `String` (instead of `ArrayBuffer`).

Because `slice` is a valid method for `String` objects, that other method was dynamically dispatched at runtime and no `TypeError` exception was thrown, violating the specification and causing the tests to fail.

Second, 26 of the tests tested the `Array.from(items,mapfn, thisArg)` method, expecting a `TypeError` whenever `mapfn` was not a function that can be called. The 26 tests invoke the `Array.from` with `mapfn` set to `undefined`. While `undefined` cannot be called, the specification describes different behavior for this value, but Swami fails to encode that behavior.

Third, 333 of the tests tested the `Array(len)` constructor with nonnumeric `len` values, such as Strings, booleans, etc. This dynamically dispatched a different `Array` constructor based on the type of the argument, and thus, no `RangeError` was thrown.

These imprecisions in Swami's tests illustrate the complexity of disambiguating natural language descriptions of code elements. The tests were generated to test specific methods, but different methods executed at runtime. Swami could be improved to avoid generating these bad tests by adding more heuristics to the input generation algorithm.

B. RQ2: Do Swami-generated tests cover behavior missed by developer-written tests?

Rhino and Node.js are two extremely well-known and mature JavaScript engines. Rhino, developed by Mozilla, and Node.js, developed by the Node.js Foundation on Chrome's V8 JavaScript engine, both follow rigorous development processes that include creating high-quality regression test suites. We compared the tests Swami generated to these developer-written test suites for Rhino v1.7.8 and Node.js v10.7.0 to measure if Swami effectively covered behavior undertested by developers.

Rhino's developer-written test suites have an overall statement coverage of 71% and branch coverage of 66%. Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods, increasing those methods' statement coverage, on average, by 15.2% and branch coverage by 19.3%. For Node.js, the developer-written test suites are already of high coverage, and Swami did not increase Node.js statement and branch coverage. However, coverage is an underestimate of test suite quality [69], as evidenced by the fact that Swami discovered defects in both projects.

Swami generated tests that identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. The Rhino issue tracker contains 2 open feature requests (but no tests) corresponding to 12 of 15 missing features in Rhino. We have submitted a bug report for the new defect[1] and a missing feature request for the 3 features not covered by existing requests[2].

The newly discovered defect dealt with accessing the `ArrayBuffer.prototype.byteLength` property of an `ArrayBuffer` using an object other than an `ArrayBuffer`. Neither Rhino nor Node.js threw a `TypeError`, failing to satisfy the JavaScript specification.

[1]https://github.com/mozilla/rhino/issues/522
[2]https://github.com/mozilla/rhino/issues/521

The missing features dealt with 6 `Map` methods[3], 6 `Math` methods[4], 2 `String` methods (`padStart` and `padEnd`), and 1 `Array` method (`includes`) not being implemented in Rhino, failing to satisfy the JavaScript specification.

The 18 semantic ambiguities are caused by JavaScipt's multiple ways of comparing the equality of values, e.g., `==`, `===`, and `Object.is`. In particular, `===` and `Object.is` differ only in their treatment of `-0`, `+0`, and `NaN`. The specification often says two values should be equal without specifying which equality operator should be used. This causes a semantic ambiguity in the specifications of methods that differentiate between these values. In fact, developer-written tests for Node.js and Rhino differ in which operators they use to compare these values, with Rhino's developers implementing their own comparator and explicitly stating that they consider differentiating between `+0` and `-0` unimportant[5], whereas the specification dictates otherwise. With this ambiguity in the specification, Swami has no obvious way to infer which equality operator should be used and can generate imprecise tests. When using `===`, Swami generated 18 false alarms revealing this ambiguity. Using `Object.is` removes these false alarms.

C. RQ3: Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools?

Random test generation tools, such as EvoSuite [20] and Randoop [50], can generate tests in two ways: using explicit assertions in the code (typically written manually), or as regression tests, ensuring the tested behavior doesn't change as the software evolves. As such, Swami is fundamentally different, extracting oracles from specifications to capture the intent encoded in those specifications. Still, we wanted to compare the tests generated by Swami and EvoSuite, as we anticipated they would cover complementary behavior.

We used EvoSuite to generate five independent test-suites for Rhino using 5 minute time budgets and line coverage as the testing criterion, resulting in 16,760 tests generated for 251 classes implemented in Rhino. Of these, 392 (2.3%) tests failed because the generated inputs resulted in exceptions EvoSuite had no oracles for (Rhino source code did not encode explicit assertions for this behavior). This finding is consistent with prior studies of automated test generation of exceptional behavior [8]. By comparison, only 1.6% of the non-innocuous Swami-generated tests and only 0.6% of all the tests were false alarms (recall RQ1). Swami significantly outperforms EvoSuite because it extracts oracles from the specifications.

Since EvoSuite, unlike Swami, requires project source code to generate tests, EvoSuite failed to generate tests for the 15 methods that exposed missing functionality defects in Rhino, which Swami detected by generating specification-based tests.

The EvoSuite-generated test suite achieved, on average, 77.7% statement coverage on the Rhino classes. Augmenting that test suite with Swami-generated tests increased the statement coverage of 47 classes by 19.5%, on average.

[3]https://github.com/mozilla/rhino/issues/159
[4]https://github.com/mozilla/rhino/issues/200
[5]https://github.com/mozilla/rhino/blob/22e2f5eb313b/testsrc/tests/shell.js

The tools most similar to Swami, Toradacu [24] and Jdoctor [8], are difficult to compare to directly because they work on Javadoc specifications and cannot generalize to the more complex natural language specifications Swami can handle. On nine classes from Google Guava, Toradocu reduced EvoSuite's false alarm rate by 11%: out of 290 tests, the false alarms went from 97 (33%) to 65 (22%) [24]. Meanwhile, on six Java libraries, Jdoctor eliminated only 3 of Randoop's false alarms out of 43,791 tests (but did generate 15 more tests and correct 20 tests' oracles) [8]. And again, without an implementation (or without the associated Javadoc comments), neither Toradacu nor Jdoctor could generate tests to identify the missing functionality Swami discovered. Overall, Swami showed more significant improvements in the generated tests.

D. RQ4: Does Swami precisely identify relevant specifications?

Swami's regular expression approach to *Section Identification* is precise: in our evaluation, 100% of the specification sections identified encoded testable behavior. But it requires specific specification structure. Without that structure, Swami relies on its Okapi-model approach. We now evaluate the Okapi model's precision (the fraction of the sections the model identifies as relevant that are actually relevant) and recall (the fraction of all the relevant sections that the model identifies as relevant).

Eaddy et al. [16] have constructed a ground-truth benchmark by manually mapping parts of the Rhino source code (v1.5R6) to the relevant concerns from ECMA-262 (v3). Research on information retrieval in software engineering uses this benchmark extensively [17], [25]. The benchmark consists of 480 specifications and 140 Rhino classes. On this benchmark, Swami's precision was 79.0% and recall was 98.9%, suggesting that Swami will attempt to generate tests from nearly all relevant specifications, and that 21.0% of the specifications Swami may consider generating tests from may not be relevant.

However, of the irrelevant specifications, 45.3% do not satisfy the *Template Initialization* rule, and 27.3% do not satisfy the *Conditional Identification* rule. All the test templates generated from the remaining 27.4% fail to compile, so Swami removes them, resulting in an effective precision of 100%.

E. Threats to validity

Regular expressions are brittle and may not generalize to other specifications or other software. We address this threat in three ways. First, our evaluation uses ECMA-based specifications which has been used to specify hundreds of systems [4], from software systems, to programming languages, to Windows APIs, to data communication protocols, to telecommunication networks, to data storage formats, to wireless proximity systems, and so on. This makes our work directly applicable to at least that large number of systems, and it can likely can be extended to other standards. Second, our evaluation focuses on the specification of JavaScript, a mature, popular language. The ECMA-262 standard is the official specification of JavaScript. Third, we evaluate our approach on two well-known, mature, well-developed software systems, Rhino, written in Java, and Node.js, in C++, demonstrating generalizability to test generation for different languages and systems.

The correctness of Swami-generated tests relies on the test constructor written by the developer. For example, the Rhino's test case constructor internally uses method `getTestCaseResult(expected, actual)`, which fails to distinguish `-0` from `+0` (recall Section IV-B). This inhibits the Swami-generated tests from correctly testing methods whose behavior depends on differentiating signed zeros.

Our evaluation of the Okapi-model-based approach and our choice of similarity score threshold rely on a manually-created ground truth benchmark [16]. Errors in the benchmark may impact our evaluation. We mitigate this threat by using a well-established benchmark from prior studies [17], [25], [26].

V. Related Work

Techniques that extract oracles from Javadoc specifications are the closest prior work to ours. Toradacu [24] and Jdoctor [8] do this for exceptional behavior, and @tComment [65] for null pointers. These tools interface with EvoSuite or Randoop to reduce their false alarms. JDoctor, the latest such technique, combines pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications for pre-conditions, and normal and exceptional post-conditions. Our approach builds on these ideas but applies to more general specifications than Javadoc, with more complex natural language. Unlike these tools, Swami does not require access to the source code and generates tests only from the specification, while also handling boundary conditions. When structured specifications, Javadoc, and source code are all available, these techniques are likely complementary. Meanwhile, instead of generating tests, runtime verification of Java API specifications can discover bugs, but with high false-alarm rates [32].

Requirements tracing maps specifications, bug reports, and other artifacts to code elements [15], [26], [77], which is related to Swami's *Section Identification* using the Okapi model [57], [64]. Static analysis techniques typically rely on similar information-retrieval-based approaches as Swami, e.g., BLUiR [51], for identifying code relevant to a bug report [58]. Swami's model is simpler, but works well in practice; recent studies have found it to outperform more complex models on both text and source code artifacts [53], [67].

Dynamic analysis can also aid tracing, e.g., in the way CERBERUS uses execution tracing and dependency pruning analysis [17]. Machine learning can aid tracing, e.g., via word embeddings to identify similarities between API documents, tutorials, and reference documents [79]. Unlike Swami, these approaches require large ground-truth training datasets. Future research will evaluate the impact of using more involved information retrieval models.

Automated test generation (e.g., EvoSuite [20] and Randoop [50]) and test fuzzing (e.g., afl [1]) generate test inputs. They require manually-specified oracles or oracles manually encoded in the code (e.g., assertions), or generate regression tests [20]. Swami's oracles can complement these techniques. Differential testing can also produce oracles by

comparing behavior of multiple implementations of the same specification [10], [12], [19], [59], [63], [78] (e.g., comparing the behavior of Node.js to that of Rhino), but requires multiple implementations, whereas Swami requires none.

Specification mining uses execution data to infer (typically) FSM-based specifications [5], [6], [7], [23], [29], [30], [31], [33], [34], [35], [36], [49], [54], [60]. TAUTOKO uses such specifications to generate tests, e.g., of sequences of method invokations on a data structure [14], then iteratively improving the inferred model [14], [68]. These dynamic approaches rely on manually-written or simple oracles (e.g., the program should not crash) and are complementary to Swami, which uses natural language specifications to infer oracles. Work on generating tests for non-functional properties, such as software fairness, relies on oracles inferred by observing system behavior, e.g., by asserting that the behavior on inputs differing in a controlled way should be sufficiently similar [22], [11], [3]. Meanwhile, assertions on system data can also act as oracles [46], [47], and inferred causal relationships in data management systems [21], [42], [43] can help explain query results and suggest oracles for systems that rely on data management systems [45]. Such inference can also help debug errors [70], [71], [72] by tracking and using data provenance [44].

Dynamic invariant mining, e.g., Daikon [18], can infer oracles from test executions by observing arguments' values method return values [48]. Such oracles are a kind of regression testing, ensuring only that behavior does not change during software evolution. Korat uses formal specifications of pre- and post-conditions (e.g., written by the developer or inferred by invariant mining) to generate oracles and tests [9]. By contrast, Swami infers oracles from the specification and neither requires source code nor an existing test suite.

Automated program repair, e.g., [74], [28], [37], [38], [40], [41], [61], [66], [75], relies on test suites to judge repair quality. Unfortunately, incomplete test suites lead to low repair quality [62], [52]. Swami's oracles may improve repair quality.

VI. Contributions

We have presented Swami, a regular-expression-based approach to automatically generate oracles and tests for boundary and exceptional behavior from structured natural language specifications. While regular expressions are rigid, Swami performs remarkably well in practice. Swami is the first approach to work on specifications as complex as the ECMA-262 JavaScript standard. Our evaluation demonstrates that Swami is effective at generating tests, complements tests written by developers and generated by EvoSuite, and finds previously unknown bugs in mature, well-developed software.

Acknowledgments

This work is supported in part by the National Science Foundation under grants no. CCF-1453474, CNS-1513055, and CCF-1763423.

References

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2018.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 1 edition, 2008.

[3] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Themis: Automatically testing software for discrimination. In *ESEC/FSE Demo*, pages 871–875, 2018.

[4] European Computer Manufacturer's Association. ECMA standards. https://ecma-international.org/publications/standards/Standard.htm, 2018.

[5] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE TSE*, 41(4):408–428, April 2015.

[6] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE*, pages 468–479, 2014.

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*, pages 267–277, 2011.

[8] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *ISSTA*, pages 242–253, 2018.

[9] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[10] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *S&P*, pages 114–129, 2014.

[11] Yuriy Brun and Alexandra Meliou. Software fairness. In *ESEC/FSE New Ideas and Emerging Results*, pages 754–759, 2018.

[12] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *ESEC/FSE*, pages 793–804, 2015.

[13] Flaviu Cristian. Exception handling. Technical Report RJ5724, IBM Research, 1987.

[14] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA*, pages 85–96, 2010.

[15] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[16] Marc Eaddy. Concern tagger case study data mapping the Rhino source code to the ECMA-262 specification). http://www.cs.columbia.edu/~eaddy/concerntagger/, 2007.

[17] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC*, pages 53–62, 2008.

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.

[19] Robert B. Evans and Alberto Savoia. Differential testing: A new approach to change detection. In *ESEC/FSE Poster*, pages 549–552, 2007.

[20] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE TSE*, 39(2):276–291, February 2013.

[21] Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. A characterization of the complexity of resilience and responsibility for self-join-free conjunctive queries. *Proceedings of the VLDB Endowment (PVLDB)*, 9(3):180–191, 2015.

[22] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: Testing software for discrimination. In *ESEC/FSE*, pages 498–510, 2017.

[23] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining behavior models from user-intensive web applications. In *ICSE*, pages 277–287, 2014.

[24] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA*, pages 213–224, 2016.

[25] Emily Hill. Developing natural language-based program analyses and tools to expedite software maintenance. In *ICSE Doctoral Symposium*, pages 1015–1018, 2008.

[26] Emily Hill, Shivani Rao, and Avinash Kak. On the use of stemming for concern location and bug localization in Java. In *SCAM*, pages 184–193, 2012.

[27] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Pearson Education, Inc., 2 edition, 2009.

[28] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *ASE*, pages 295–306, 2015.

[29] Ivo Krka, Yuriy Brun, George Edwards, and Nenad Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *ESEC/FSE*, pages 305–314, 2009.

[30] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions. In *ASE*, 2015.

[31] Tien-Duy B. Le and David Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *SANER*, 2015.

[32] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE*, pages 602–613, 2016.

[33] David Lo and Siau-Cheng Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.

[34] David Lo and Siau-Cheng Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE*, pages 265–275, 2006.

[35] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: Better together. In *ASE*, pages 387–396, 2010.

[36] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE*, pages 345–354, 2009.

[37] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, pages 166–178, 2015.

[38] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.

[39] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for simple program repairs. In *ICSE*, pages 448–458, 2015.

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016.

[42] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y. Halpern, Christoph Koch, Katherine F. Moore, and Dan Suciu. Causality in databases. *IEEE Data Engineering Bulletin*, 33(3):59–67, 2010.

[43] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment (PVLDB)*, 4(1):34–45, 2010.

[44] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Bringing provenance to its full potential using causal reasoning. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[45] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. Causality and explanations in databases. *Proceedings of the VLDB Endowment (PVLDB) tutorial*, 7(13):1715–1716, 2014.

[46] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *ESEC/FSE New Ideas*, pages 631–634, 2013.

[47] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Preventing data errors with continuous testing. In *ISSTA*, pages 373–384, 2015.

[48] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

[49] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *ASE*, pages 19–30, 2014.

[50] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *OOPSLA*, pages 815–816, 2007.

[51] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM TOSEM*, 21(4):23, 2012.

[52] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pages 24–36, 2015.

[53] Md Masudur Rahman, Saikat Chakraborty, Gail Kaiser, and Baishakhi Ray. A case study on the impact of similarity measure on information retrieval based software engineering tasks. *CoRR*, abs/1808.02911, 2018.

[54] Steven P. Reiss and Manos Renieris. Encoding program executions. In *ICSE*, pages 221–230, 2001.

[55] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, May 2002.

[56] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple BM25 extension to multiple weighted fields. In *CIKM*, pages 42–49, 2004.

[57] Stephen E. Robertson, Stephen Walker, and Micheline Beaulieu. Experimentation as a way of life: Okapi at TREC. *Information Processing and Management*, 36:95–108, January 2000.

[58] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[59] Vipin Samar and Sangeeta Patni. Differential testing for variational analyses: Experience from developing KConfigReader. *CoRR*, abs/1706.09357, 2017.

[60] Matthias Schur, Andreas Roth, and Andreas Zeller. Mining behavior models from enterprise web applications. In *ESEC/FSE*, pages 422–432, 2013.

[61] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI*, pages 43–54, 2015.

[62] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE*, pages 532–543, 2015.

[63] Varun Srivastava, Michael D. Bond, Kathryn S. McKinley, and Vitaly Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *PLDI*, pages 343–354, 2011.

[64] Trevor Strohman, Donald Metzler, HowardTurtle, and W. Bruce Croft. Indri: A language model-based search engine for complex queries. In *International Conference on Intelligence Analysis*, pages 2–6, 2005.

[65] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *ICST*, pages 260–269, 2012.

[66] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *ICSE*, pages 471–482, 2015.

[67] Chakkrit Tantithamthavorn, Surafel Abebe Lemma, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology*, 2018.

[68] Robert J. Walls, Yuriy Brun, Marc Liberatore, and Brian Neil Levine. Discovering specification violations in networked software systems. In *ISSRE*, pages 496–506, 2015.

[69] Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In *ICST*, pages 321–332, 2017.

[70] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. Data X-Ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245, 2015.

[71] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. QFix: Demonstrating error diagnosis in query histories. In *SIGMOD Demo*, pages 2177–2180, 2016.

[72] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. QFix: Diagnosing errors through query histories. In *SIGMOD*, pages 1369–1384, 2017.

[73] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA*, pages 419–431, 2004.

[74] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[75] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. In *ASE*, pages 625–636, 2017.

[76] Allen Wirfs-Brock and Brian Terlson. ECMA-262, ECMAScript 2017 language specification, 8th edition. https://www.ecma-international.org/ecma-262/8.0, 2017.

[77] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE TSE*, 42(8):707–740, 2016.

[78] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

[79] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *ICSE*, pages 404–415, 2016.

[80] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR*, pages 334–342, 2001.