Paper title: Automatically Generating Precise Oracles from Structured Natural Language Specifications
Paper number: 264
Purpose: To facilitate replicating experiments presented in our paper and to allow researchers to build on our work.
Badges claimed: We claim "Reusable" and "Available" badges. We are releasing a virtual machine of the artifact, and the source code as open source software (https://bitbucket.org/manishmotwani/swami) along with the detailed documentation.
Skills required: The reviewer will need to run a VirtualBox VM (running 64-bit Ubuntu 16) and execute shell and python scripts, following our instructions. If the reviewer wishes to verify our scripts, they will need knowledge of python and shell.

This artifact is a software system called Swami, a tool to automatically extract test oracles and generate executable tests from structured natural language specifications. Swami generates tests that encode exceptional or boundary behavior. The artifact evaluates Swami using ECMA-262, the official JavaScript specification and two JavaScript implementations: Mozilla Rhino and Node.js. Our results use ECMA-262 version 8, Rhino version 1.7.9, and Node.js version 10.7.0.

Note that reproducing all the results can take around 10 hours. The artifact contains four scripts, described below, together with the expected execution times.

The artifact infrastructure is set up on a VirtualBox virtual machine.

Steps to import the VM using VirtualBox software (copied from the INSTALL file)**:**

1. Download VirtualBox from https://www.virtualbox.org/
2. Download the swami.ova file from http://people.cs.umass.edu/~brun/ICSE2019SwamiVM
3. Please note that this is a large file (8GB) and may take a long time to download.
4. Open VirtualBox
5. Click on **File -> Import Appliance**
6. In the import appliance wizard, select the **swami.ova** file downloaded in step 2.
7. Click **continue**
8. Click **import**. This step may take 2-3 minutes to execute.

Start the VM. If you are asked for a password to log into the VM, that password is swami

The VM contains the following:

1. The ECMA-262 version 8 specification document stored in /home/swami/swami/data directory.
2. Mozilla Rhino version 1.7.9 compiler source code including developer-written test suite stored in /home/swami/rhino-Rhino1_7_9_Release directory.
3. Node.js version 10.8.0 JavaScript runtime source code stored in /home/swami/node-10.7.0 directory.
4. The source code and the driver scripts to run Swami, an automated test generation tool in /home/swami/swami directory. This source code is also available publicly at https://bitbucket.org/manishmotwani/swami

5. The BLUiR executable to preprocess natural language text and Java code before running IR based model stored in /home/swami/BLUiR-Package directory.
6. The Indri library to run OKAPI model stored in /home/swami/indri-5.3 directory.

The rest of this document describes the steps to replicate the results in the four research questions presented in our paper.

Research Question 1: How precise are Swami-generated tests?

Results reported in Section 4 for RQ1:
Result 1.1: Swami generates 83 compiling templates and 1000 tests for each of the templates resulting in 83,000 tests.
Result 1.2: Of the 83,000 tests, 32,379 (39%) are good tests, 535 (0.6%) are bad tests, and 50,086 (60.3%) are innocuous tests. Of the non-innocuous tests, 98.3% (32,379 out of 32,914) are good tests.

How to replicate:
Open a terminal in the VM and make sure the terminal window is maximized. (If the terminal is not maximized, the script will print **much** more output, but the final screen will look the same.) Execute the following commands sequentially.

```
cd /home/swami/swami/src
./reproduceRQ1.sh
```

The reproduceRQ1.sh script will take around 10 minutes to run and will print the results on the terminal as shown in Figure 1 below.

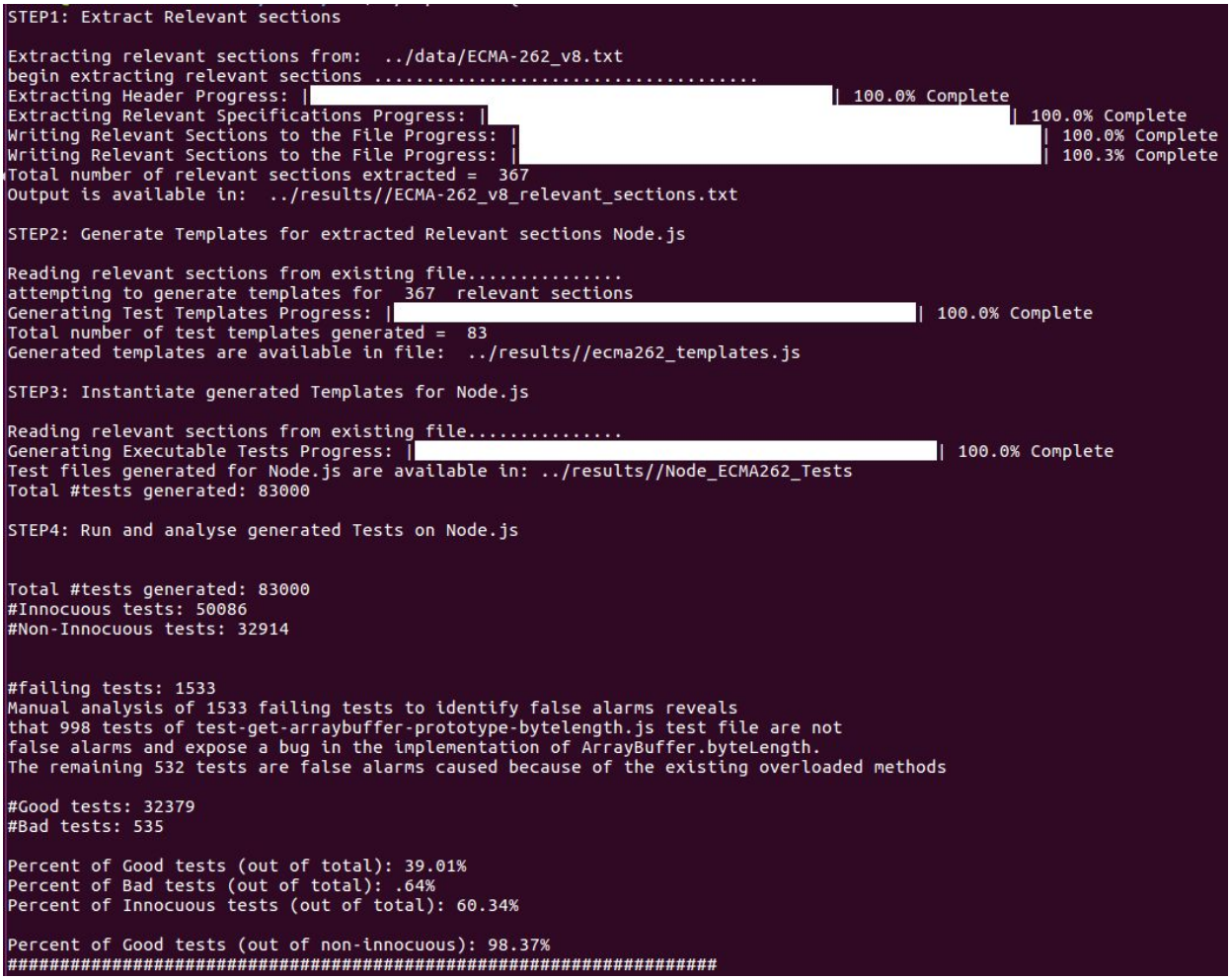


Figure 1: Output of executing the reproduceRQ1.sh script to replicate the RQ1 results

Script explanation:

STEP 1: The reproduceRQ1.sh script first invokes Swami's module that extracts the testable sections of the ECMA-262 document (Step 1 in Figure 1). Swami extracts 367 relevant sections from the document. The output is stored in /home/swami/swami/results/ECMA-262_v8_relevant_sections/txt.

STEP 2: Next, reproduceRQ1.sh invokes the module to generate test templates from the extracted sections (Step 2 in Figure 1). Swami generates 83 compilable templates. The output is stored in /home/swami/swami/results/ecma262_templates.js. This result is shown in the output line of Figure 1 that reads "Total number of test templates generated: 83".
This replicates the first half of Result 1.1.

STEP 3: Next, reproduceRQ1.sh invokes the module to instantiate these 83 templates using 1000 randomly generated test inputs for each template (Step 3 in Figure 1). The generated test files are stored in the directory /home/swami/swami/results/Node_ECMA262_Tests. This result is shown in the output line of Figure 1 that reads "Total #tests generated 83000".
This replicates the second half of Result 1.1.

STEP 4: Finally, reproduceRQ1.sh executes the 83,000 tests in Node.js to identify the good, bad, and innocuous tests. The script prints the fraction of good, bad, and innocuous tests. A failing test could be a good test if that test fails because of the bug in the implementation. If the test fails because of some other reason, such as malformed test inputs triggering overloaded methods, it is a bad test. We manually analyze the 1533 failing tests to identify these two types. 998 out of the 1533 failing tests were found to be good, while remaining 535 were false alarms or bad tests. This result is shown in the output lines of Figure 1 after the line that reads “STEP 4: Run and analyze generated tests on Node.js ”.
This replicates Result 1.2.

Research Question 2: Do Swami-generated tests cover behavior missed by developer-written tests?

Results reported in Section 4 for RQ2:

Result 2.1: Swami-generated tests identify 15 missing features in Rhino.

Result 2.2: Swami-generated tests identify 1 previously unknown defect in Node.js.

Result 2.3: Rhino's developer-written test suites have statement coverage of 71% and branch coverage of 66%

Result 2.4: For 12 Rhino methods belonging to the 8 Rhino classes, Swami-generated tests increase the statement coverage by 15.2% and branch coverage by 19.3% on average.

How to replicate:

Open a terminal in the VM and execute the following commands sequentially.

```
cd /home/swami/swami/src
./reproduceRQ2.sh
```

Script explanation:

The reproduceRQ2.sh script will take around 45 minutes to run. This script executes three test suites and computes their coverages:

(1) the Rhino's developer-written test suite,

(2) the Swami-generated test suite, and

(3) the Rhino's developer-written test suite augmented with Swami-generated test suite.

The reproduceRQ2.sh script generates HTML reports. Execute the following commands to view the reports.

- `firefox /home/swami/swami/results/buildSwami/reports/tests/test/index.html`
  This report shows the results of executing 83,000 swami-generated tests that expose 15 missing features in Rhino. Figure 2 shows the screenshot of the generated report. The 15 lines listed in this report each indicates one missing feature and clicking on these lines will open the details of the what caused the test to fail. For example, clicking on line "MozillaSuiteTest.runMozillaTest[34, js=testscr/tests/Rhino_ECMA262_Teressts/test_map_prototype_clear.js, opt=0]" will show the cause to be "Map is undefined" as shown in Figure 3.
  This replicates Result 2.1.

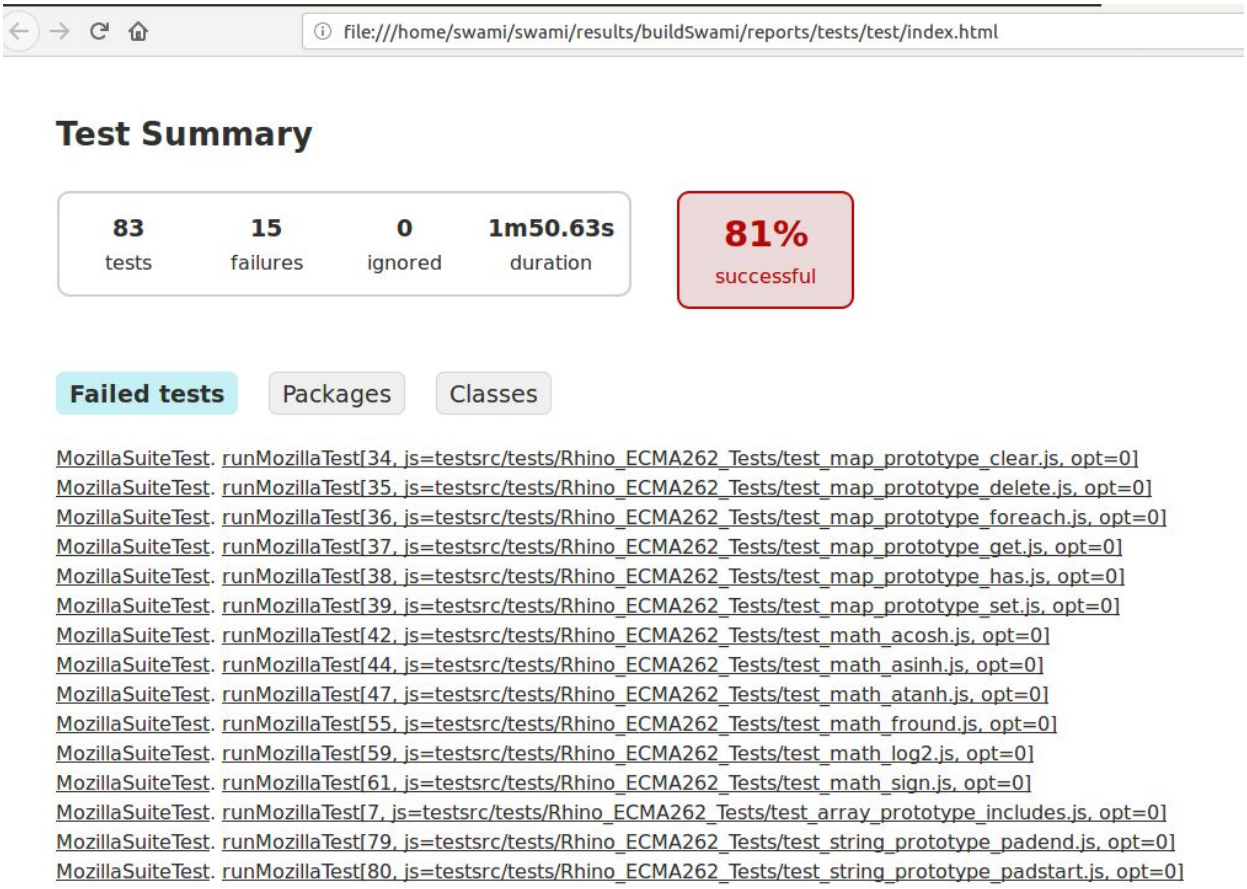


Figure 2: The test execution report of swami-generated tests exposing 15 missing features in Rhino

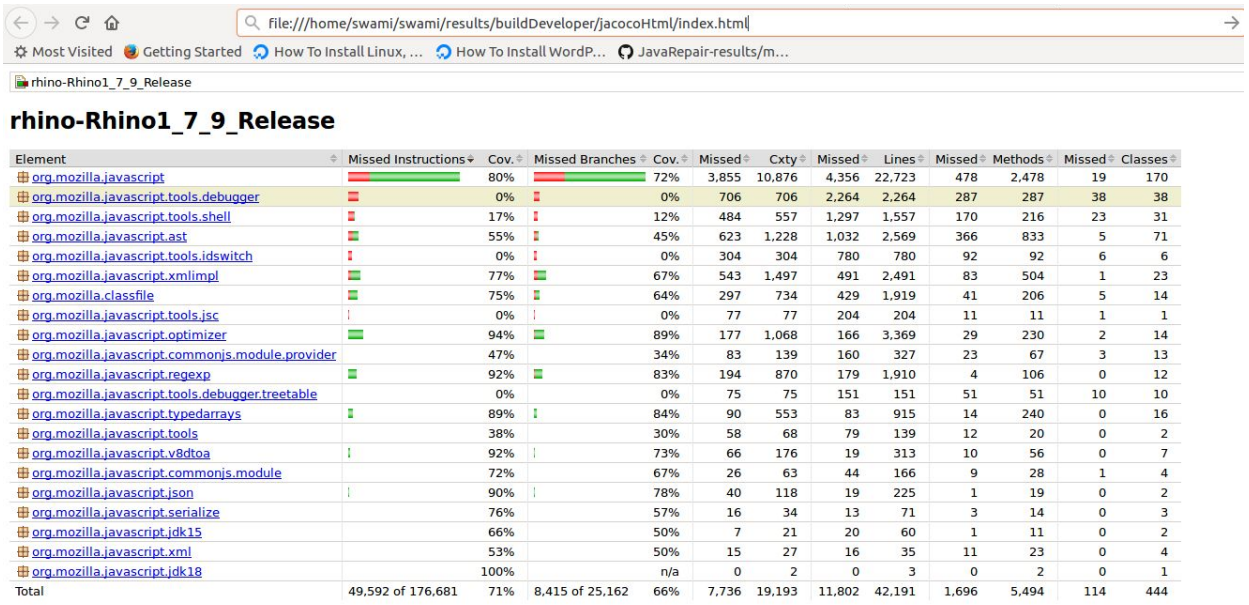
Figure 3: The test execution report of test_map_prototype_clear.js showing missing implementation of Map in Rhino which cause all the tests to fail.

- `~/node-10.7.0/node /home/swami/swami/results/Node_ECMA262_Tests/test-get-arraybuffer-prototype-bytelength.js | grep "Failed Test" | wc -l`

  This will print the number of tests that belong to test-get-arraybuffer-prototype-bytelength.js file and fail in Node.js. These are 998 tests and manual analysis of these 998 tests reveal that these are not false alarms.

  This replicates Result 2.2
- `firefox /home/swami/swami/results/buildDeveloper/jacocoHtml/index.html`

This report shows the statement and branch coverage of the Rhino's developer-written test suite as shown in Figure 4. See the 2nd and 3rd columns of the bottom row, labeled "Total", of the table in the HTML file.
This replicates Result 2.3.



rhino-Rhino1_7_9_Release

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.mozilla.javascript		80%		72%	3,855	10,876	4,356	22,723	478	2,478	19	170
org.mozilla.javascript.tools.debugger		0%		0%	706	706	2,264	2,264	287	287	38	38
org.mozilla.javascript.tools.shell		17%		12%	484	557	1,297	1,557	170	216	23	31
org.mozilla.javascript.ast		55%		45%	623	1,228	1,032	2,569	366	833	5	71
org.mozilla.javascript.tools.idswitch		0%		0%	304	304	780	780	92	92	6	6
org.mozilla.javascript.xmlimpl		77%		67%	543	1,497	491	2,491	83	504	1	23
org.mozilla.classfile		75%		64%	297	734	429	1,919	41	206	5	14
org.mozilla.javascript.tools.jsc		0%		0%	77	77	204	204	11	11	1	1
org.mozilla.javascript.optimizer		94%		89%	177	1,068	166	3,369	29	230	2	14
org.mozilla.javascript.commonjs.module.provider		47%		34%	83	139	160	327	23	67	3	13
org.mozilla.javascript.regexp		92%		83%	194	870	179	1,910	4	106	0	12
org.mozilla.javascript.tools.debugger.treetable		0%		0%	75	75	151	151	51	51	10	10
org.mozilla.javascript.typedarrays		89%		84%	90	553	83	915	14	240	0	16
org.mozilla.javascript.tools		38%		30%	58	68	79	139	12	20	0	2
org.mozilla.javascript.v8dtoa		92%		73%	66	176	19	313	10	56	0	7
org.mozilla.javascript.commonjs.module		72%		67%	26	63	44	166	9	28	1	4
org.mozilla.javascript.json		90%		78%	40	118	19	225	1	19	0	2
org.mozilla.javascript.serialize		76%		57%	16	34	13	71	3	14	0	3
org.mozilla.javascript.jdk15		66%		50%	7	21	20	60	1	11	0	2
org.mozilla.javascript.xml		53%		50%	15	27	16	35	11	23	0	4
org.mozilla.javascript.jdk18		100%		n/a	0	2	0	3	0	2	0	1
Total	49,592 of 176,681	71%	8,415 of 25,162	66%	7,736	19,193	11,802	42,191	1,696	5,494	114	444

Figure 4: The coverage report for the Rhino's developer-written test suite generated using Jacoco

- `firefox /home/swami/swami/results/coverage_analysis_developer.html`
  This report shows the difference between code coverage of developer-written test suite (Test Suite 1) and developer-written test suite augmented with Swami-generated tests (Test Suite 2) for all the Rhino's **classes. The report highlights the 8 Rhino classes in yellow for which Swami improves coverage.** Use the following command to view the report and scroll down to see the highlighted classes. Figure 5 shows the partial screenshot of the report.

  ```
  firefox /home/swami/swami/results/coverage_analysis_developer.html
  ```

ArrowFunction_ESTest_scaffolding	0 of 2010	0	0 of 2010	0	0 of 2010	0
ScriptRuntime	6407 of 7677	83	6448 of 7677	83	6448 of 7677	83
ObjToIntMap	566 of 762	74	566 of 762	74	566 of 762	74
HashSlotMap$1	35 of 40	87	35 of 40	87	35 of 40	87
NativeJSON_ESTest	0 of 290	0	0 of 290	0	0 of 290	0
NativeCall_ESTest	0 of 52	0	0 of 52	0	0 of 52	0
RhinoSecurityManager	0 of 37	0	0 of 37	0	0 of 37	0
NativeDate_ESTest_scaffolding	0 of 1822	0	0 of 1822	0	0 of 1822	0
BeanProperty	12 of 12	100	12 of 12	100	12 of 12	100
StackStyle	43 of 48	89	43 of 48	89	43 of 48	89
Token$CommentType	49 of 58	84	49 of 58	84	49 of 58	84
SecurityUtilities$1	10 of 10	100	10 of 10	100	10 of 10	100
ObjArray_ESTest	0 of 578	0	0 of 578	0	0 of 578	0
JavaMembers$MethodSignature	46 of 48	95	46 of 48	95	46 of 48	95
WrappedException_ESTest_scaffolding	0 of 702	0	0 of 702	0	0 of 702	0
NativeDate	3545 of 3951	89	3545 of 3951	89	3545 of 3951	89
NativeError	616 of 666	92	616 of 666	92	616 of 666	92
RhinoException$1	23 of 26	88	23 of 26	88	23 of 26	88
ContinuationPending_ESTest	0 of 76	0	0 of 76	0	0 of 76	0
ObjArray	304 of 794	38	304 of 794	38	304 of 794	38
NativeMath_ESTest_scaffolding	0 of 2290	0	0 of 2290	0	0 of 2290	0
NativeContinuation	78 of 120	65	78 of 120	65	78 of 120	65
NativeArrayIterator	54 of 58	93	54 of 58	93	54 of 58	93
Kit_ESTest	0 of 493	0	0 of 493	0	0 of 493	0
Token_ESTest	0 of 1484	0	0 of 1484	0	0 of 1484	0
NativeJavaArray	173 of 219	78	173 of 219	78	173 of 219	78
NativeObject$KeySet$1	53 of 65	81	53 of 65	81	53 of 65	81
CompilerEnvirons	187 of 264	70	187 of 264	70	187 of 264	70
NativeString	2023 of 2220	91	2024 of 2220	91	2024 of 2220	91
InterfaceAdapter_ESTest	0 of 25	0	0 of 25	0	0 of 25	0

Figure 5: The coverage comparison report highlighting the Rhino classes for which Swami-generated tests improve coverage over developer-written test suite

The reproduceRQ2.sh script parses the coverage reports (generated using Jacoco) of these 8 Rhino classes and generates a "::" separated csv file that shows the list of **12 Rhino methods** for which the Swami improves the coverage. The generated csv file can be viewed by executing the command

```
libreoffice home/swami/swami/results/coverage_analysis_developer.csv
```

This replicates Result 2.4.

Research Question 3: Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools?

Results reported in Section 4 for RQ3:

Result 3.1: 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%) fail.

Result 3.2: The EvoSuite-generated test suite achieved, on average, 77.7% statement coverage on the 251 Rhino classes.

Result 3.3: Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 Rhino classes by, on average, 19.5%.

How to replicate:

Open a terminal in the VM and execute the following commands sequentially.

```
cd /home/swami/swami/src
./reproduceRQ3.sh
```

Warning: The reproduceRQ3.sh script will take more than 5 hours to run! This script executes five EvoSuite-generated Rhino test suites, which are large. To save the reviewer's time, we provide the precomputed test execution results for each of the five EvoSuite-generated tests suites, as well as the merged EvoSuite test suites augmented with Swami tests.

Script explanation:

The precomputed test execution results for each of the five EvoSuite-generated test suites can be viewed by executing the following commands. Figures 6-10 show the screenshots of the test execution results of the five EvoSuite-generated test suites..

```
firefox /home/swami/swami/results/buildEvoSuite1/reports/tests/test/index.html
```

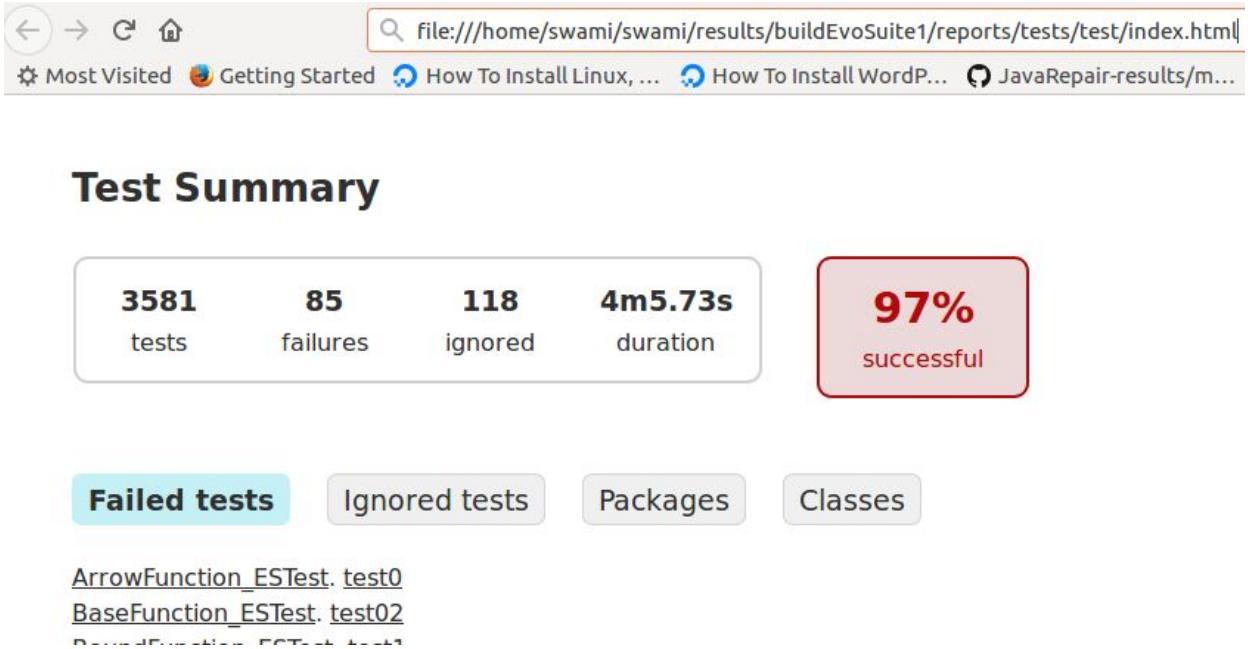


Figure 6: 85 tests fail out of 3581 tests generated using EvoSuite seed 1

```
firefox /home/swami/swami/results/buildEvoSuite2/reports/tests/test/index.html
```

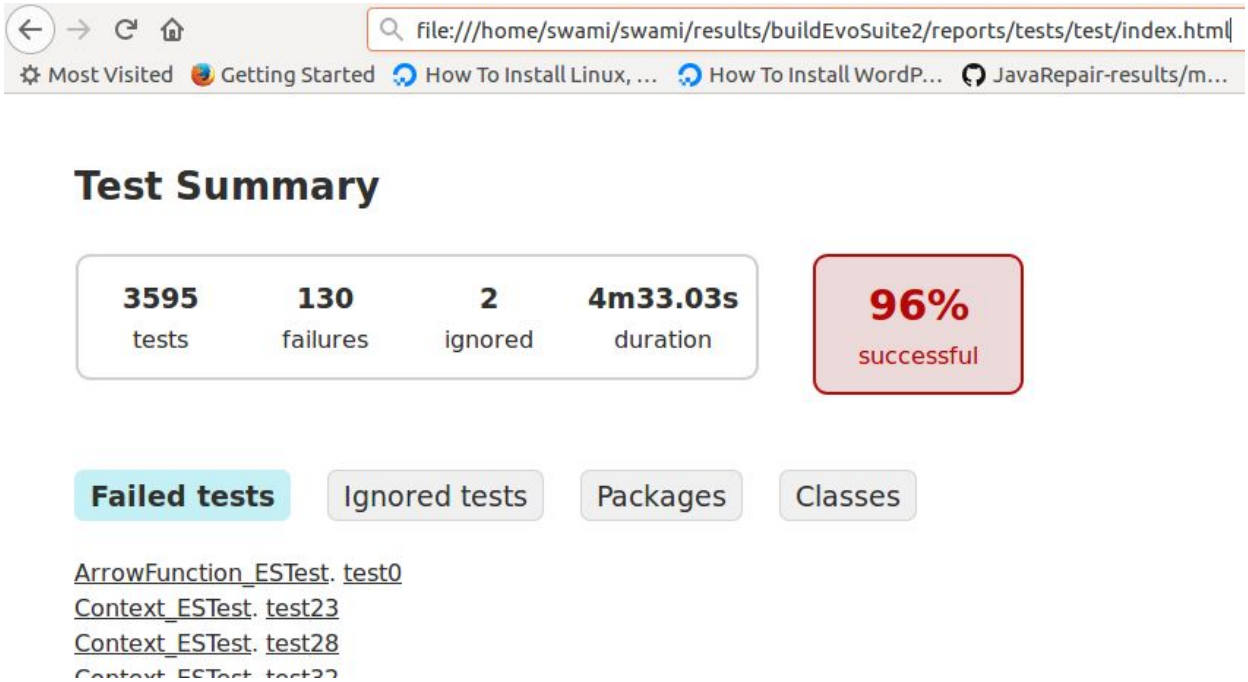


Figure 7: 130 tests fail out of 3595 tests generated using EvoSuite seed 2

```
firefox /home/swami/swami/results/buildEvoSuite3/reports/tests/test/index.html
```

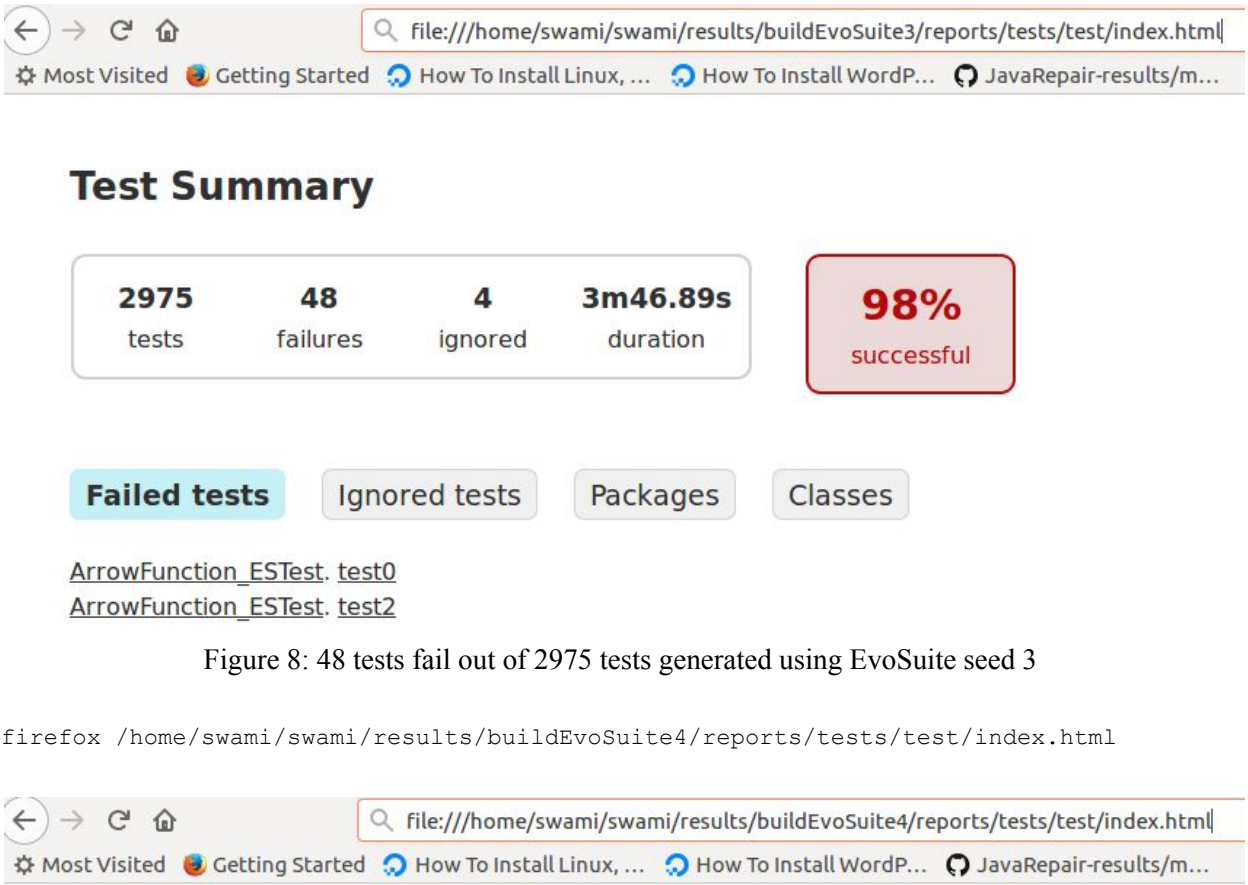


Figure 8: 48 tests fail out of 2975 tests generated using EvoSuite seed 3

```
firefox /home/swami/swami/results/buildEvoSuite4/reports/tests/test/index.html
```

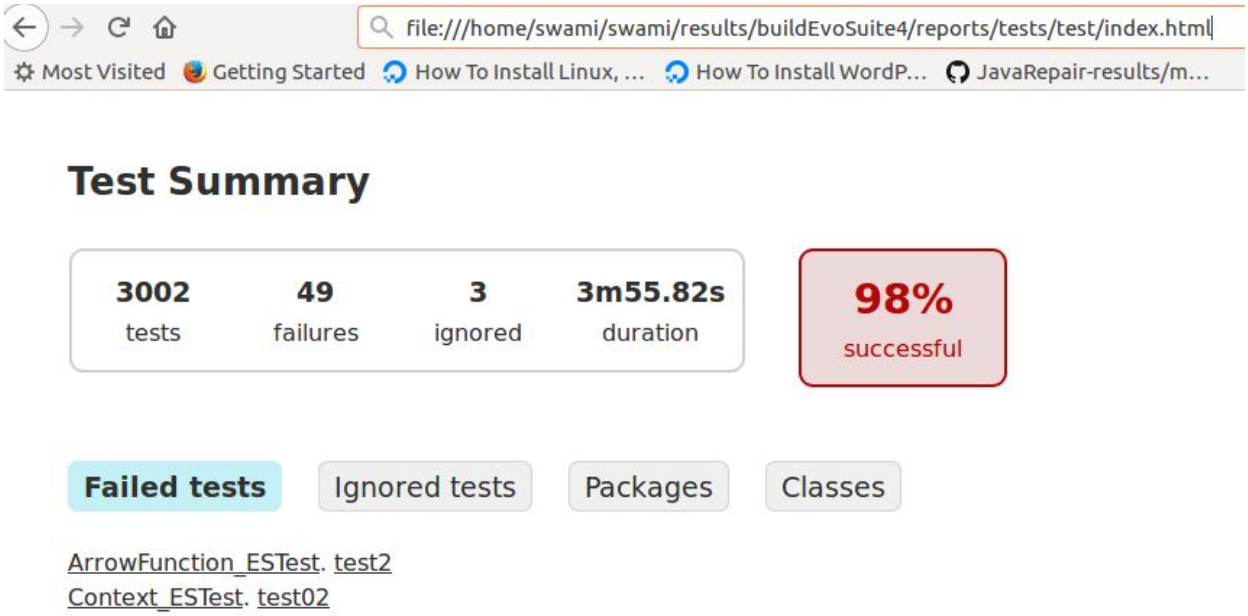


Figure 9: 49 tests fail out of 3002 tests generated using EvoSuite seed 4

```
firefox /home/swami/swami/results/buildEvoSuite5/reports/tests/test/index.html
```

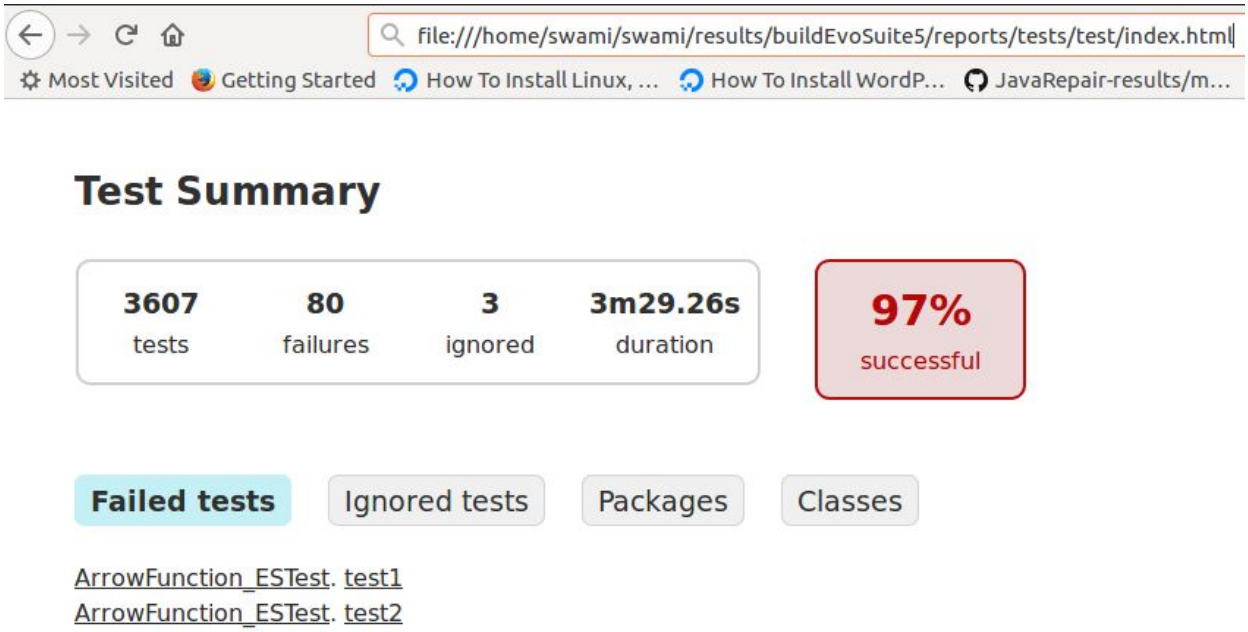


Figure 10: 80 tests fail out of 3607 tests generated using EvoSuite seed 5

Thus, a total of 16,760 (3581 + 3595 + 2975 + 3002 + 3607) tests are generated out of which 392 (85 + 130 + 48 + 49 + 80) fail.
This replicates Result 3.1.

EvoSuite generates a .csv file, for each of its test suites, that list that suite's statement coverage. The first column (TARGET CLASS) lists the 251 Rhino's classes and the third column (Coverage) shows the statement coverage obtained on a given Rhino class. These csv files can be viewed using following commands:

```
libreoffice /home/swami/swami/EvoSuiteTests/evosuite_1/evosuite-report/statistics.csv
libreoffice /home/swami/swami/EvoSuiteTests/evosuite_2/evosuite-report/statistics.csv
libreoffice /home/swami/swami/EvoSuiteTests/evosuite_3/evosuite-report/statistics.csv
libreoffice /home/swami/swami/EvoSuiteTests/evosuite_4/evosuite-report/statistics.csv
libreoffice /home/swami/swami/EvoSuiteTests/evosuite_5/evosuite-report/statistics.csv
```

The output obtained by running the script reproduceRQ3.sh shows the average statement coverage of all the five test suites. This can also be computed directly by running the following commands:

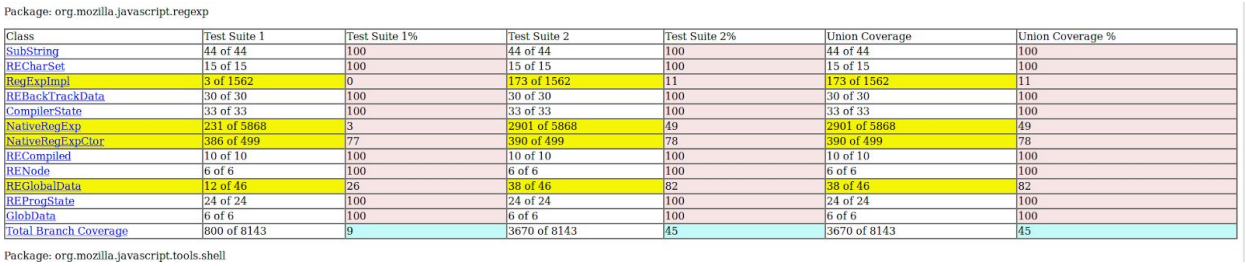
```
cd /home/swami/swami/src
./coverageAnalysisEvoSuite.sh
```

The output printed on the terminal reads "The average statement coverage of EvoSuite-generated tests on 251 Rhino classes is: 77.7086489606 %"
This replicates Result 3.2.

The report /home/swami/swami/results/coverage_analysis_evosuite.html shows the comparison between statement coverage of EvoSuite-generated test suite (Test Suite 1) and EvoSuite-generated test suite augmented with Swami-generated tests (Test Suite 2) on all the Rhino classes. **The report highlights the**

the 47 Rhino classes in yellow for which Swami improves coverage. Use the following command to view the report and scroll down to see the highlighted classes. Figure 11 shows the partial screenshot of the report.

```
firefox /home/swami/swami/results/coverage_analysis_evosuite.html
```



Package: org.mozilla.javascript.regexp

Class	Test Suite 1	Test Suite 1%	Test Suite 2	Test Suite 2%	Union Coverage	Union Coverage %
SubString	44 of 44	100	44 of 44	100	44 of 44	100
RECharSet	15 of 15	100	15 of 15	100	15 of 15	100
RegExpImpl	3 of 1562	0	173 of 1562	11	173 of 1562	11
REBackTrackData	30 of 30	100	30 of 30	100	30 of 30	100
CompilerState	33 of 33	100	33 of 33	100	33 of 33	100
NativeRegExp	231 of 5868	3	2901 of 5868	49	2901 of 5868	49
NativeRegExpCtor	386 of 499	77	390 of 499	78	390 of 499	78
RECompiled	10 of 10	100	10 of 10	100	10 of 10	100
RENode	6 of 6	100	6 of 6	100	6 of 6	100
REGlobalData	12 of 46	26	38 of 46	82	38 of 46	82
REProgState	24 of 24	100	24 of 24	100	24 of 24	100
GlobData	6 of 6	100	6 of 6	100	6 of 6	100
Total Branch Coverage	800 of 8143	9	3670 of 8143	45	3670 of 8143	45

Package: org.mozilla.javascript.tools.shell

Figure 11: The coverage comparison report highlighting the Rhino classes for which Swami-generated tests improve coverage over EvoSuite-generated test suite

For the ease of readability, we list these 47 Rhino classes along with their coverage details in comma separated /home/swami/results/coverage_analysis_evosuite.csv file. Use the following command to view the file:

```
libreoffice /home/swami/swami/results/coverage_analysis_evosuite.xlsx
```

The last row of the file shows the average improvement in the statement coverage for 47 Rhino methods. This replicates Result 3.3

Research Question 4: Does Swami's Okapi model precisely identify relevant specifications?

Results reported in Section 4 for RQ4:
Result 4.1: 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.

How to replicate:
Open a terminal in the VM and execute the following commands sequentially.

```
cd /home/swami/swami/src
./reproduceRQ4.sh
```

Script explanation:
The reproduceRQ4.sh script will take around 3 minutes to run. It executes another driver script swami/src/okapi/runOkapi.sh.
The runOkapi.sh script uses an existing bug location tool BLUiR (http://www.riponsaha.com/BLUiR.html) to preprocess and represent each section of ECMA-262 version 8 document as a query and each Java class of Rhino version 1.7.9 source code as a document.

Next, it executes the OKAPI model implemented in Indri library (http://www.lemurproject.org/indri.php) which is trained based on the ground truth data available at http://www.cs.columbia.edu/~eaddy/concerntagger/ (see Rhino case study). The output of this step are the mappings between the ECMA-262 section and the most highly ranked Rhino Java Class along with the similarly score. The retrieved mappings are further filtered out if the similarity score obtained is less than empirically determined threshold on the similarity score.
Finally, Precision and Recall values are computed using the ground truth data and the mappings retained after filtering based on threshold.
The identified mappings are stored in the file /home/swami/swami/results/spec_class_mappings_top1.txt which can be viewed using following command:

```
vim /home/swami/swami/results/spec_class_mappings_top1.txt
```

The relevant sections retrieved using OKAPI model are stored in file /home/swami/swami/results/retrieved_sections_ecma262_v8.txt which can be viewed using following command:

```
vim /home/swami/swami/results/retrieved_sections_ecma262_v8.txt
```

The output displayed on the terminal after the line that reads "Computing Precision and Recall" shows the replicated precision and recall results. Figure 12 shows the partial screenshot of the output displayed where
Precision = correctly retrieved / retrieved count * 100 and

Recall = correctly retrieved / relevant count * 100

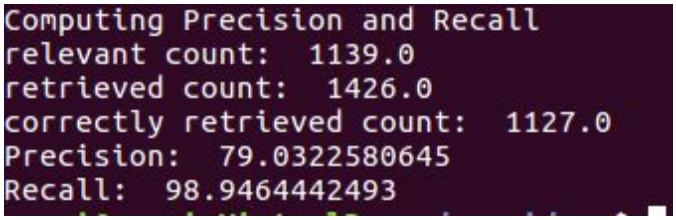


Figure 12: Precision and Recall of identifying relevant sections using OKAPI model

This replicates Result 4.1.