Selective Regression Testing on Node.js Applications

by

Charlie Chen

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Science

in

Honour in Computer Science, Software Engineering

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2020

The following individuals certify that they have read, and recommend to the Faculty of Science for acceptance, the thesis entitled:

Selective Regression Testing on Node.js Applications

Submitted by **Charlie Chen** in partial fulfillment of the requirements for the degree of **Bachelor of Science**

in **Honour in Computer Science combined Software Engineering**

Examining Committee:

Reid Holmes, Computer Science
Supervisor

Abstract

Node.js is one of the most popular frameworks for building web applications today. As software system becomes mature, the cost of performing the retest-all regression test becomes significant. A technique of reducing running time is Selective Regression Testing (SRT). By rerunning a subset of tests based on code change, selective regression testing can detect software failures more efficiently. However, previous studies mainly focused on standard desktop applications. The Node.js applications are considered hard to perform test reduction because its asynchronous, event-driven programming model and JavaScript is a loosely typed, dynamic programming language.

In this thesis, we present NodeSRT, a Selective Regression Testing framework for Node.js applications. By performing static and dynamic analysis, NodeSRT gets relationship between changed method and their relationships with each test, then reduce the whole regression test suite to only tests that are affected by change, which would improve the execution time of the regression test suite. To evaluate our selection technique, we applied NodeSRT to two open-source projects: Uppy and Simorgh, then compared our approach with retest-all strategy and current industry used SRT technique: Jest OnlyChange option. The results demonstrate that NodeSRT correctly selects affected tests based on changes and is more precise than the Jest OnlyChange option.

Lay Summary

Do you have to wait for a long time before your regression test suite finished? Do you have a hard time finding which change caused test fail? This thesis presents a technique to selects a subset of tests based on changes to reduce the execution time of running regression test suite. We also did an empirical study on two open-source projects and compared our approach with retest all strategy and Jest OnlyChange option to analyze the difficulties and benefits of Selective Regression Testing on Node.js applications.

Preface

The work presented in this thesis was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey campus. All projects and associated methods were approved by the University of British Columbias Research Ethics Board [certificate #H18-02971].

Reid Holmes is the supervisor of this project and was involved throughout the project in concept formation and manuscript composition.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor Reid Holmes and my honour advisor Mark Greenstreet. This work would not be possible without their guidance and support.

I would like to thank all my professors and friends at the University of British Columbia for my past four years at university.

Finally, special thanks to my parents for backing me up, encouraging me, and supporting whatever choices I make.

Chapter 1

Introduction

When working on software projects, regression testing is a necessary maintenance activity to show that the code has not been adversely affected by changes [1]. However, running the whole regression test suite is expensive. Some test suite takes even days to finish. Besides, when submitting a pull request, if the CI strategy is to rerun all regression tests, developers have to wait for a long time before getting feedbacks. A way to reduce the running time is Selective Regression Testing (SRT). Selective Regression Testing rerun a subset of tests based on code changes. It can run the regression test more efficiently and ensure the inclusiveness of failure.

With the continuous growth of web applications, Node.js has become one of the most popular frameworks for web application development [2]. Since JavaScript is a loosely typed, dynamic language, test selection on JavaScript projects is hard. Besides, modern web applications are usually composed of different kinds of components; running unit tests only does not judge the overall behaviour of the web application [3].

In this thesis, we first discovered a regression test selection technique for Node.js application then evaluate the selection technique by performing an empirical study on a Node.js project. During the empirical study, we tried to address the following research questions:

> **RQ1** Comparing to retest all strategy, how much running time can selective regression test improve?

RQ2 How effective is the selective strategy in terms of inclusiveness, precision, efficiency?

RQ3 For method-level granularity and file-level granularity, which suit Node.js application better in regression test selection?

There are two phases required to run selective regression testing. The first step is to select tests based on change and test dependency graph generated by static and dynamic analysis. The second phase is running the selected tests. For some project, if the running time of selecting tests plus the running time of executing selected tests is greater than the running time of executing the whole test suite, applying the selection technique is meaningless. RQ1 is trying to measure the benefit of using the selection technique.

Let *P* be a program, let *P'* be a modified version of *P*, and let *S* and *S'* be the specifications for *P* and *P'* respectively. Let *T* be a test suite created to test *P*. If a test *t* passed in *P* but failed in *P'*, we say t is ***fault-revealing*** for *P'*. A test *t* is ***modification-revealing*** for *P* and *P'* if and only if it causes the outputs of *P* and *P'* to differ. Rothermel et al. defined four categories when evaluating the quality of the selected regression tests [1].

- **Inclusiveness**: inclusiveness measures the extent to which regression test selection technique(*M*) chooses modification-revealing tests from *T* for inclusion in *T'*. To calculate inclusiveness, suppose *T* contains *n* tests that are modification revealing for *P* and *P'*, and suppose *M* selects *m* of these tests. The inclusiveness of *M* relative to *P*, *P'* and *T* is the percentage given by the expression *(100(m/n))*
- **Precision**. Precision measures the extent to which *M* omits tests that are non-modification-revealing. Suppose *T* contains *n* tests that are non-modification-

revealing for *P* and *P'* and suppose *M* omits *m* of these tests. The precision of *M* relative to *P*, *P'* and *T* is the percentage given by the expression *(100(m/n))*

- **Efficiency**: efficiency measures the time and space requirements for the regression test selection technique.
- **Generality**: the generality of a test selection technique is its ability to function in a comprehensive and practical range of situations.

We say a selection technique(M) is safe if it achieves 100% inclusiveness. Our empirical study measures the inclusiveness, precision, and efficiency to address RQ2. Generality is out of scope of this thesis.

There are four different levels of granularity of test selection technique: statement, method, file, module. The common two are method-level and file-level. File-level granularity analysis selects files that include the changes to the program, where method-level analysis selects methods. Since file-level test reduction selects more tests than needed, file-level analysis is less precise. The study done by Gligoric et al. [4] shown that file-level granularity analysis runs 32% faster than the method-level granularity analysis in large scale Java programs. RQ3 tries to find if the same statement holds for our experiment.

In this thesis, we are also going to show the challenges faced when building a regression test selection framework for Node.js applications. Since JavaScript is a loosely typed, dynamic language, ES6 introduced some new features to JavaScript, selection techniques for traditional Object-oriented programming languages do not work for JavaScript. This thesis will describe how we handle corner cases when developing NodeSRT.

Chapter 2

Related Work

2.1 Selection algorithms overview

Regression test selection technique has been reviewed by several researchers. Lin et al. [5] have proved that the Regression test suite minimization (TSM) is NP-complete since the minimum set-covering problem can be reduced to the TSM problem in polynomial time. Rothermel and Harrold [1] proposed several metrics and reviewed twelve techniques. E. Engström et al. [6] performed a survey that analyzed selection techniques before 2006. Biswas et al. [7] surveyed SRT techniques proposed after that. Zarrad [8] reviewed 22 papers on Regression Testing for Web-based applications. And several empirical studies have been done on Selective Regression Testing. In general, no selection technique can be said outperformed to others since they are designed for different frameworks and different deployment environments. In this section, we will select some representative techniques for each language and framework to show the merits and demerits of them.

2.2 Selection Techniques for Procedural Programs

Procedural programs were the first group of programs that have been studied by SRT. These techniques make use of control flow analysis, data or control dependency analysis or lexical analysis to select relevant test cases based on change.

Harrold and Souffa [9], Taha et al. [10] proposed an SRT technique based on Dataflow analysis. It works by detecting the def-use information for variables that are affected by program modifications, then select test cases that exercise the paths from the definition of modified variables to their uses. However, this technique is not safe because it omitted modification-revealing tests when code deletion happened. Also, the dataflow technique might not select tests to execute a new or modified output statement that contains no variable uses [1].

In 1997, Rothermel and Harrold [11] proposed a safe, efficient SRT technique based on control flow analysis. Their approach compares the CFGs of original program P and modified program P' respectively to identify dangerous edges. Then selects tests that exercise dangerous edges. This technique is considered safe because it selects all the modification-traversing tests. An empirical study [12] shows that the tool implemented their approach "DejaVu" saves 41% of the time on seven C programs with an extensive test pool. But 16% worse when it comes to realistic code-coverage-based test suites.

TestTube [13] presented by Yih-Farn Chen et al. is a differencing-based SRT technique for C programs. It divides programs into different entities such as functions, types, variables, and macros then collect coverage information on which entity is covered by which tests. When the system under test is modified, TestTube identifies which entities were changed. Using the coverage and change information, TestTube reduced 50% of unit test cases in their empirical study.

Another empirical study was done by Beszedes et al. [14] on procedural programs, where they implement a basic test selection technique based on code coverage information. Then select tests on C++ functions and method level. In their experiment, they applied their selection method to an open-source web browser engine WebKit to investigate technical difficulties and benefits. WebKit system consists of about 2.2 million lines of code, nearly 24,000 regression test cases. The result shows the total

execution time of the selected test is 21.4% of execution time of full test suite. However, manual verification indicates that the overall inclusiveness is 75.38%.

2.3 Selection Techniques for Object-Oriented Programs

Comparing to procedure programs, Object-Oriented Programs include some program concepts such as inheritance, polymorphism, encapsulation, abstraction, etc. These concepts added complexity to program analysis.

The first safe SRT technique for Java programs was proposed by Harrold et al. [15]. Their approach is based on an extended control flow graph, which also captures information for classes. Similar to control flow based analysis, their technique traverses the graph to identify dangerous edges, then based on coverage matrix obtained through instrumentation, it selects test cases that exercised dangerous edges.

Chianti [16] is another regression test selection tool for Java. It uses dynamic and static analysis to classify each atomic change to different categories (added classes, deleted classes, added methods, deleted methods, changed methods, added fields, deleted fields, and lookup changes). Comparing to Harrold's technique, Chianti does not rely on traversal of two representations of the program to find semantic differences. Instead, Chianti derives a set of atomic changes to form nodes and edges in the call graphs for the tests in the original version of the program. Chianti also handles Java Dynamic dispatch and anonymous functions well according to their experiment.

Other Regression test selection techniques such as Firewall-based techniques, Design model-based techniques, and Specification-based techniques are considered unsafe and less precise than Program model-based techniques. These techniques are beyond the discussion of this section.

2.4 Regression Testing on Node.js applications

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside web browsers. Node.js lets developers use JavaScript to write command-line tools and for server-side scripting – running scripts server-side to produce dynamic web page content before the pages are sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web application development around a single programming language, rather than different languages for server- and client-side scripts [17]. Regression Testing on Node.js applications usually consists of two parts: unit test and end-to-end test. Frameworks used for unit testing are mocha, Jest, Jasmine, etc. Frameworks used for end-to-end testing are Selenium, Cypress, etc.

Several studies have been done on JavaScript. JSART [18] is an automated JavaScript regression testing technique based on dynamic analysis to infer invariant assertions. By injecting back obtained assertions to JavaScript code, JSART uncovers regression faults in subsequent revisions of the web application under test. Mutandis [19] is a generic mutation testing approach that guides the mutation generation process. It works by leveraging static and dynamic program analysis to guide the mutation generation process a-priori towards parts of the code that are error-prone or likely to influence the program's output. Although Mutandis build the relationship between tests and changes, it only detects functions affected by the changes, no static changes included. This is sufficient for mutation testing, but it is not safe for regression test selection.

Some studies have been done on Node.js applications. NodeMOP [2] is a runtime verification tool for Node.js applications that uses a similar approach with JSART. It dynamically injects assertions to ensure the correctness of the application. NodeMOP also supports error recovery by allowing developers to define custom handlers in case of property violations. Tochal [20] is another change impact analysis tool for JavaScript.

Through dynamic code injection and static analysis, this approach incorporates a ranking algorithm for indicating the importance of each entity in the impact set.

Currently, the test selection technique used by industry is based on static analysis to get file dependency graph. Jest [21] has a onlyChanged option that identifies which tests to run based on which files have changed in the current repository. In the section below, we will compare our proposed technique with Jest onlyChanged option to show the merits and demerits of our approach.

Chapter 3

Methodology

In this section, we explain the techniques and how the tool works to select regression tests

3.1 Challenges

Many special language features make JavaScript hard to test and perform analysis. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions. As a multi-paradigm language, JavaScript supports event-driven, functional, and imperative programming styles. It has APIs for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM). I will describe each of the special features below and show how traditional regression test selection technique does not work for JavaScript.

3.1.1 first-class functions

```
1. function square(number) {
2.   return number * number;
3. }
4.
5. const square2 = function(number) { return number * number }
6.
7. const factorial = function fac(n) {
8.   return n < 2 ? 1 : n * fac(n - 1)
9. }
```

Figure 1 Function Declarations

```
1. function map(f, a) {
2.   let result = []; // Create a new Array
3.   let i; // Declare variable
4.   for (i = 0; i != a.length; i++)
5.     result[i] = f(a[i]);
6.   return result;
7. }
8. const f = function(x) {
9.    return x * x * x;
10.  }
11.  let numbers = [0, 1, 2, 5, 10];
12.  let cube = map(f,numbers);
13.  console.log(cube);
```

Figure 2 Function as argument

```
1. // returns the factorial of 10.
2. alert((function(n) {
3.   return !(n > 1)
4.     ? 1
5.     : arguments.callee(n - 1) * n;
6. })(10));
7.
8. // anonymous arrow functions
9. alert(((n) => {
10.    return !(n > 1)
11.      ? 1
12.      : arguments.callee(n - 1) * n;
13.  })(20));
```

Figure 3 Anonymous functions

Figure 1 shows three ways of declaring a function in JavaScript. Since functions are first-class, functions can be assigned to variables (Figure 1, line 5 and line 7), or be passed as argument (Figure2). Figure 3 shows anonymous functions in JavaScript. Since JavaScript uses just-in-time (JIT) compilation strategy, anonymous functions are not translated to bytecode. Therefore, these first-class functions are hard to track in traditional approaches because traditional approaches try to identify affected methods with their name.

3.1.2 Inheritance and the prototype chain

JavaScript is a prototype-based language, as it is dynamic and does not provide a classic class implementation. When it comes to inheritance, JavaScript only has one construct: Object. Each object has a private property which holds a link to another object: prototype. That prototype object has a prototype of its own until an object reaches `null`. By definition, `null` has no prototype, and it acts as the final link of the prototype chain [22].

```
1.   var o = {
2.     a: 2,
3.     m: function() {
4.       return this.a + 1;
5.     }
6.   };
7.
8.   console.log(o.m()); // 3
9.   // When calling o.m in this case, 'this' refers to o
10.  var p = Object.create(o);
11.  // p is an object that inherits from o
12.
13.  p.a = 4; // creates a property 'a' on p
14.  console.log(p.m()); // 5
```

Figure 4 Inheritance functions

```
1. function MyFunc() {
2.     this.x = 100;
3. }
4.
5. var obj1 = new MyFunc();
```

Figure 5 Create object with new keyword

```
1.  function Car() {}
2. car1 = new Car();
3. car2 = new Car();
4.
5. console.log(car1.color);     // undefined
6.
7. Car.prototype.color = 'original color';
8. console.log(car1.color);     // 'original color'
9.
```

```
10.  car1.color = 'black';
11.  console.log(car1.color);    // 'black'
12.
13.  console.log(Object.getPrototypeOf(car1).color); // 'original
   color'
14.  console.log(Object.getPrototypeOf(car2).color); // 'original
   color'
15.  console.log(car1.color);   // 'black'
16.  console.log(car2.color);   // 'original color'
```

Figure 6 Prototype example

```
1. // initialize object
2. var myCar = new Object();
3. myCar.make = 'Ford';
4. myCar.model = 'Mustang';
5. myCar.year = 1969;
6.
7. var myCar2 = {
8.     make: 'Ford',
9.     model: 'Mustang',
10.      year: 1969
11.  };
12.
13.  // access object field
14.  myCar.color; // undefined
15.  myCar['make'] = 'Ford';
16.  var propertyName = 'make';
17.  myCar[propertyName] = 'Ford';
18.  myCar['mo'+'del'] = (() => 'explorer')();
```

Figure 7 Working with object

As shown in Figure 4, functions can be assigned to be a property of an object (line 3). Also, when creating a property 'a' on `p`, the output of `p.m()` changed. Figure 5 shows another way of creating inheritance using the `new` keyword. Figure 6 shows the object field can be added or deleted at anytime instead of pre-defined like object-oriented languages. Figure 7 shows different ways of initializing an object and different ways to access object field. Line 18 gives an extreme example of dynamic typing. Table 1 summarizes the differences between class-based (Java) and prototype-based (JavaScript) object systems. All these examples shows that JavaScript's dynamic, prototype-based

language features make it hard for a traditional object-oriented approach to keep track of object fields.

Category	Class-based (Java)	Prototype-based (JavaScript)
Class vs. Instance	Class and instance are distinct entities	All objects can inherit from another object
Definition	Define a class with a class definition. Instantiate a class with constructor methods	Define and create a set of objects with constructor functions.
Creation of new object	Create a single object with the `new` operator.	Same
Construction of object hierarchy	Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inheritance model	Inherit properties following the class chain	Inherit properties following prototype chain
Extension of properties	Class definition specifies all properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an initial set of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

Table 1 Comparison of Class-based (Java) and prototype-based (JavaScript) object system

3.1.3 Asynchronous and Error handling

A Promise is a proxy for a value not necessarily known when the promise is created. The adaption of Promise in JavaScript lets asynchronous methods return values like synchronous methods – instead of returning final value immediately. The asynchronous method returns a promise to supply the value at some point in the future [22].

```
1.  const myPromise =
2.    (new Promise(myExecutorFunc))
3.    .then(handleFulfilledA)
4.    .then((res) => {
5.       return handleFulfillB(res)
6.    })
7.    .then(handleFulfilledC)
8.    .catch(handleRejectedAny);
```

Figure 8 Promise chain

```
1. // call our promise
2. var askMom = function () {
3.     console.log('before asking Mom'); // log before
4.     willIGetNewPhone
5.         .then(showOff)
6.         .then(function (fulfilled) {
7.             console.log(fulfilled);
8.         })
9.         .catch(function (error) {
10.             console.log(error.message);
11.          });
12.     console.log('after asking mom'); // log after
13. }
```

Figure 9 Asynchronous Promises

Figure 8 shows a Promise chain. Any changes happen in the upper part of the chain will affect the result of the lower part. The error handling is also different in JavaScript. Any error thrown in any parts of Promise chain will be handled by the final catch block. Besides, as shown in line 4, the handler functions in Promises are usually anonymous, which makes it challenge to identify. Figure 9 shows the asynchronous

nature of Promises. Line 3 and line 12 are executed before code blocks in promise. This is because asynchronous tasks are pushed to the event queue then execute when the call stack is empty. All in all, asynchronous functions and its error handling is a special feature in JavaScript that requires to be handled by the test selection technique.

3.2 Modification-based Regression Test Selection Approach

The Modification based works by analyzing modified code entities then select tests based on modifications. This strategy is relatively simple and proved safe by [13] [7] but highly imprecise. However, empirical study [4] [12] shows the complicated SRT technique has a high overhead, which results in higher total running time when the SRT technique is applied. This is the reason why the current industry used test selection technique is in file-level granularity, the only static analysis involved. Our SRT approach tries to improve current selection precision but does not add too much overhead to it. Therefore, we used a combination of static and dynamic analysis, then perform a modification-based test selection algorithm in method level. While tackling technical difficulties in Node.js, this approach is safe and four times more precise on average than the industry used SRT technique in our empirical study. We have implemented this technique as a command-line tool NodeSRT. This tool can also be used in CI/CD environment. The tool consists of five parts: dynamic analysis, static analysis, change analysis, test selection, and selected test runner. In the next four sections below, we will explain each of the modules in detail.

3.3 Dynamic analysis module

The dynamic analysis module works by injecting analysis code into the original codebase, then running the full test suite will generate a call graph. The dynamic analysis module has three parts: logging server, code injector, test runner.

3.3.1 Logging server

Unlike desktop applications, modern web applications have the characteristics that client-side, server-side separated. This means the code in different modules may be running in a different environment. E.g. The server-side code is running in Node.js environment. Client-side code is running in the browser environment. To collect runtime information of both modules, NodeSRT uses HTTP requests. The injected code will send logging messages to the server. The server will collect all the logging messages and generate a call graph because the network connection is sometimes unstable (this rarely happened when setting up correctly). The logging server has an auto-recovery feature to generate a call graph in correct JSON format. As noted by [13] [14], when the code base becomes large, code analysis result should be store in database to ensure performance does not get too slow.

3.3.2 Code injector

Code injector works by injecting analytic code to Abstract Syntax Tree (AST), then generate code from injected AST would give us injected version of original codebase. More specifically, the code injector would inject code that sends logging message to the logging server at the start of every function when the injected file is a regular js file. Inject in the beginning and end of each test when the injected file is a test file. The logging message contains function name, injected filename. If the injected file is a test file, the logging message will also include test name and suite name, test start indicator and test end indicator.

When the functions are anonymous, the code injector will assign a unique name to it. Like Chianti [16], the code injector collects its parent id, the class name it belongs to, and the number of parameters then generates a name based on these. If two anonymous functions in a file are assigned to the same name, the code injector will assign them a unique number at the end of the assigned name to differentiate them. For example, in Figure 10, an anonymous function is assigned to a variable stdArrayModelBlock. The code injector identifies the variable is going to be assigned to and record it with a logging message. The SRTlib here is an injected library to buffer messages to be sent and handle communications with the logging server. Figure 11 shows an anonymous function is used as a function argument. In this case, the code injector will trace all the AST ancestors of this anonymous function node, then extract the id of each ancestor to form a unique name. Figure 12 shows a promise chain. There are three anonymous functions are used. For each anonymous function, the code injector assigned a unique name to it based on its position in the promise chain. The first anonymous function is named "stopApp.then", the second one is named "stopApp.then.then", and so on.

When the function is inside a class, the code injector will also record the class name and its parent class name if it has one.

```
1. // before injection
2. export const stdArrayModelBlock = (blockType, modelArray) =>
3.   blockBase(blockType, blockArrayModel(modelArray));
```

```
1. // after injection
2. export const stdArrayModelBlock = (blockType, modelArray) => {
3.
   SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":false,"function"
   :"stdArrayModelBlock","fileName":"/src/app/models/blocks/index.js
   ","paramsNumber":2},`);
4.
   SRTlib.send('{"type":"FUNCTIONEND","function":"stdArrayModelBlock
   "},');
5.   return blockBase(blockType, blockArrayModel(modelArray));
```

```
6.
   SRTlib.send('{"type":"FUNCTIONEND","function":"stdArrayModelBlock
   "},');
7. };
```

Figure 10 Anonymous function

```
1. // before injection
2. const bitratesFromData = jsonData.content.blocks[0].playlist.map(
3.               item => (item.bitrate / 1000).toString(),
4.             );
```

```
1. // after injection
2.  const bitratesFromData =
   jsonData.content.blocks[0].playlist.map(item => {
3.
   SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":
   "bitratesFromData.jsonData.content.blocks.playlist.map","filename
   ":"/AdHocCypress/cypress/integration/topcat2MediaPlayback/index.j
   s","paramsNumber":1},`);
4.
   SRTlib.send('{"type":"FUNCTIONEND","function":"bitratesFromData.j
   sonData.content.blocks.playlist.map"},');
5.               return (item.bitrate / 1000).toString();
6.
   SRTlib.send('{"type":"FUNCTIONEND","function":"bitratesFromData.j
   sonData.content.blocks.playlist.map"},');
7.             });
```

Figure 11 Using anonymous function as argument

```
1. // before injection
2. stopApp()
3.     .then(() => {
4.       if (isDev) return Promise.resolve();
5.
6.       spinner.text = 'Building app...';
7.       return buildApp();
8.     })
9.     .then(() => {
10.        spinner.text = isDev
11.          ? 'Starting app in developer mode...'
12.          : 'Starting app...';
13.        return startApp();
14.      })
15.      .then(
16.        () =>
17.          new Promise(resolve => {
18.            spinner.text = 'Running tests...';
```

```
            setTimeout(() => {
              spinner.stop();
              resolve();
            }, 2000);
          }),
      )
      .then(runTests)
      .then(stopApp);
```

```
// After injection
stopApp().then(() => {
SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":"stopApp.then","fileName":"/src/integration/utils/runTests/index.js","paramsNumber":0},`);
    if (isDev) {
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then"},');
      return Promise.resolve();
    }
      spinner.text = 'Building app...';
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then"},');
      return buildApp();
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then"},');
    }).then(() => {
SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":"stopApp.then.then","fileName":"/src/integration/utils/runTests/index.js","paramsNumber":0},`);
      spinner.text = isDev ? 'Starting app in developer mode...' : 'Starting app...';
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then.then"},');
      return startApp();
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then.then"},');
    }).then(() => {
SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":"stopApp.then.then.then","fileName":"/src/integration/utils/runTests/index.js","paramsNumber":0},`);
SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then.then.then"},');
```

```
23.      return new Promise(resolve => {
24.
   SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":
   "ReturnStatement.NewExpression###1","fileName":"/src/integration/
   utils/runTests/index.js","paramsNumber":1},`);
25.        spinner.text = 'Running tests...';
26.        setTimeout(() => {
27.
   SRTlib.send(`{"type":"FUNCTIONSTART","anonymous":true,"function":
   "setTimeout","fileName":"/src/integration/utils/runTests/index.js
   ","paramsNumber":0},`);
28.          spinner.stop();
29.          resolve();
30.
   SRTlib.send('{"type":"FUNCTIONEND","function":"setTimeout"},');
31.        }, 2000);
32.
   SRTlib.send('{"type":"FUNCTIONEND","function":"ReturnStatement.Ne
   wExpression###1"},');
33.      });
34.
   SRTlib.send('{"type":"FUNCTIONEND","function":"stopApp.then.then.
   then"},');
35.    }).then(runTests).then(stopApp);
```

Figure 12 Injection to promise chain

3.3.3 Test runner

The test runner takes a preconfigured test running command to run tests in the injected codebase. The tests running in injected codebase should have the same result as original codebase because the injected code should not change the return value of each function. After running all tests in the injected codebase, the logging server should have generated a call graph. This call graph should have information about the dependencies of each test. Since injected codebase uses HTTP requests to send logging messages to the logging server, the total running time of running all tests in the injected codebase is slower than running all tests in the original codebase. This should be a minor issue because according to DART (Daily Automated Regression Tester) module [23], the full suite of regression tests are running at night. Developers can still get selected test suite running after each commit by using the call graph generated nightly. If we use an incremental strategy like

Harrold and Marry [9], an additional module is required. Plus, this will increase the overhead of running selected test suite. So, we think this is unnecessary for now.

3.4 Static analysis module

The static analysis module scan all files in the codebase with extensions of `.js`, `.jsx`, and `.json` to get file dependencies. How it works is by traversing the AST and looking for keywords such as `require` and `import`, then resolve each of the require statement statically.

```
1. import {
2.   runCoreCanonicalTests,
3.   runCanonicalAnalyticsTests,
4. } from '../../common';
5.
6. const getClientEnvVars = require('./src/clientEnvVars');
```

Figure 13 File dependency static analysis

Figure 13 show two most common ways to import another module in Node.js. To increase accuracy, the static analysis module maps resolved file dependencies to assigned IDs in assignment statement of test files, e.g. in line 6 of Figure 13, static analysis module first resolve `'./src/clientEnvVars'` and find all file dependencies of that resolved file recursively. Let's call an array of all dependent files `F`. Then, static analysis module maps `F` to id `getClientEnvVars`. Finally, find all tests `ts` in this file used id `getClientEnvVars`, we can say each test in `ts` depends on files `F`. One caveat is that since this is pure static analysis, string computations in the argument of require calls will not be resolved.

3.5 Change analysis module

The change analysis module compares the ASTs of files appear in the diff patch, then generates a JSON representation of changes. Several steps involved in this analysis: First, change analysis module parse the diff patch to get a list of all changed files, then use AST parser to get original AST and changed AST on changed files. Second, change analysis module compares two ASTs and finds the different AST nodes. Third, find ancestors of each different AST nodes. Fourth, since NodeSRT uses function-level granularity, change analysis module finds the closest function name of each different AST node based on their ancestors. If the function is anonymous, the change analysis will generate a unique name using the method mentioned in 3.3.2 code injector. If the modification is inside a class, the change analysis module will also collect its Class properties. Finally, change analysis module collects all information produced in previous steps to generate a JSON representation of changes. The JSON object should include the following properties: filename, ancestors, class properties, unique function name, and number of parameters. Below, I further explain change analysis module through a simple example.

```
1. // in file A.js
2. // before change
3. function getSum (arr) {
4.   let sum = 0;
5.   arr.forEach(ele => {
6.      sum = sum + ele;
7.   })
8.   return sum;
9. }
```

```
1.  // in file A.js
2.  // after change
3. function getSum (arr) {
4.   let sum = 0;
5.   arr.forEach(ele => {
6.     let temp = ele * 2; // add a new line
7.     sum = sum + temp;  // changed ele to temp
8.   })
9. return sum;
10.  }
```

Figure 14 A simple `getSum` function

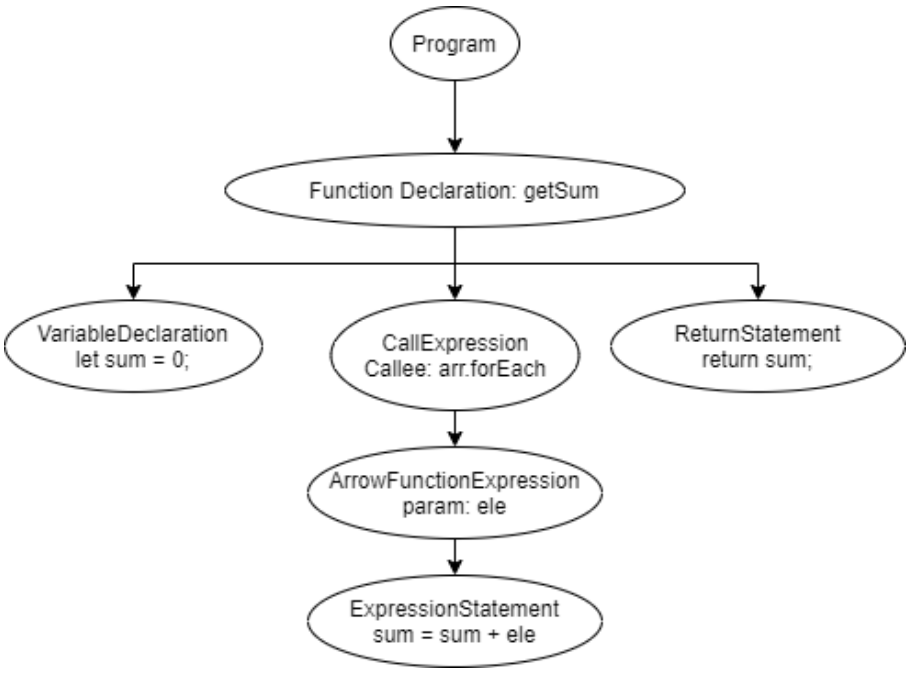

Figure 15 Sample AST of `getSum` function before change

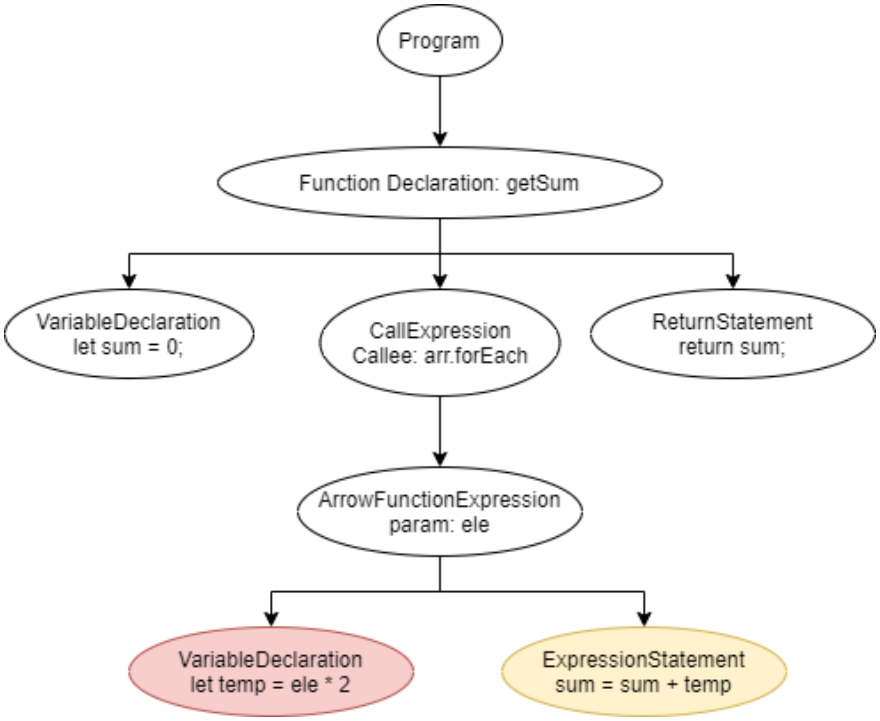

Figure 16 Sample AST of `getSum` function after change

Figure 14 shows a simple function `getSum` in file `A.js`. The lower part of Figure 14 shows the function `getSum` after change. We can see the change that happened in an anonymous function in `forEach` Call Expression. When doing change analysis on this function, the first step of the module will generate an AST for both before and after the change of `getSum` function as shown in Figure 15 and Figure 16. The red bubble in Figure 16 means an addition of an expression. The yellow bubble means a modification to an expression. The second step is to traverse both AST to find nodes that make the difference. In this case, step 2 should identify the red bubble and yellow bubble in Figure 16. The third step is to get the ancestors of AST nodes that make the differences. In this case it is `[Program, FunctionDeclaration, CallExpression, ArrowFunctionExpression]`. The fourth step is to get the nearest parent function name of different AST nodes. In this case, the nearest function is

an anonymous function. So we use the method mentioned in the code injector to generate a unique name: `arr.forEach`. Finally, collect all information produced in previous steps and generate a JSON representation of the changes. Figure 17 shows the result of the generated JSON object.

```
1. {
2.   filename: "path to A.js",
3.   ancestors: [Program, FunctionDeclaration, CallExpression,
   ArrowFunctionExpression],
4.   classProperty: null,
5.   uniqueFunctionName: "arr.forEach",
6.   paramNumber: 1
7. }
```

Figure 17 JSON object representation of change

3.6 Test selector module

The test selector module selects affected tests based on changes. The input of the test selector is a list of changes produces by the Change analysis module, a call graph produced by dynamic analysis module, and a file dependency graph produced by static analysis module. For each change, test selector finds tests affected in call graph by looking for matched `unqiueFunctionName, filename, paramNumber` and `classProperty`. If the changed function is not found in the call graph or there is no function name in the change, this means the change happened outside a function, in other words, the change occurred in a static field. The test selector module will select tests depend on the changed file based on file dependency graph to guarantee safety. When the `uniqueFunctionName` matches to a function name with a counter in the end, for example, ReturnStatement.NewExpression###1, the test selector will match all functions with the same function name regardless of the ending counter. If the changed function is inside a class declaration, then the test selector will match all functions with the same

name in its parent class and chile class. The output of the test selector module should be a list of selected tests. This should be a JSON array with each element consists of test name, test suite name, test filename.

3.7 Selected test runner module

The selected test runner module takes a list of selected tests produced by the test selector module. It works by lexically replace `it` to `it.only` of the selected tests in selected test file. Then runs test start command. The JavaScript test module `Jest` or `mocha` will only run tests with `it.only`.

3.8 End-to-end test selection

The end-to-end tests are usually tests running in the browser with end-to-end test framework selenium or cypress. These tests are difficult to perform SRT because these tests integrate frontend and backend, plus the frontend JavaScript code is usually interpreted into different versions of ECMAScript code to ensure compatibility of different browser, or JavaScript bundling to optimize performance. However, Since NodeSRT uses HTTP requests to record logging messages and build a dynamic call graph, code running in both browser environment and Node.js environment can be captured. The logging server of dynamic analysis module provides an additional endpoint to handle logging messages for end-to-end tests. After all, messages recorded, the logging server will generate a call graph for end-to-end tests, then the test selector will select tests based on this graph. Because each Node.js application has their end-to-end test suite set up differently, the end-to-end test selection module needs to be customized for each Node.js application. It should be noted that since we could not perform static analysis on

end-to-end tests, no file dependency graph can be generated. Therefore, the test selection for end-to-end tests is considered unsafe.

3.9 Limitations

The NodeSRT uses the babel-parser[1] module to get JavaScript AST, babel-walk[2] module to traverse the AST, babel-generator[3] module to generate injected code. Some comments on the code will be misplaced after code injection such as `ts-ignore`, `eslint-disable global-require`. This will cause build failure when the comments are meaningful and necessary for the build process. The cause of the issue is babel-parser attached comments to AST lexically, i.e., it only records the line number of the comments. A better parser is required to solve this issue. For now, the NodeSRT code injector will ignore problematic files. Besides, NodeSRT could not handle JavaScript eval function.

[1] babel-parser module. https://babeljs.io/docs/en/babel-parser
[2] babel-walk module. https://github.com/pugjs/babel-walk
[3] babel-generator module. https://babeljs.io/docs/en/babel-generator

Chapter 4

Empirical Study

To better evaluate our test selection technique, we performed two empirical studies to address research questions raised in Chapter 1.

4.1 Experimental Subject

There are a lot of open-source Node.js projects out there but not a lot with tests and well maintained. Using the method mentioned by Labuschagne et al. [24], we were able to find 23 open source projects with tests and have a reasonably number of commits to perform our study. Among those, Uppy[1] and Simorgh[2] caught our eyes. Uppy is a sleek, modular JavaScript file uploader that integrates seamlessly with any application. It uses asynchronous Promises heavily, so it is a good testbed for our SRT technique. Uppy has 112k lines of code, 33k lines of JavaScript code. This project has 216 unit tests and nine end-to-end tests, achieves 20% of code coverage. Simorgh is the BBC website rebuild using React.js. It is a standard RESTful API project with well-maintained tests. This project has 698k lines of code, 85k lines of JavaScript code. It also includes 2801 unit tests, achieves 97% of code coverage.

[1] Uppy, https://github.com/transloadit/uppy
[2] Simorgh, https://github.com/bbc/simorgh

4.2 Design and Procedure

To analyze the effectiveness of our SRT technique, we performed test selection on a total of 588 commits of two subjects. For each commit, we generate a diff patch from its previous commit as input to NodeSRT using `git diff` command. The purpose of performing SRT is to reduce the number of tests needed to validate newly committed code and to reduce the total running time. Therefore, the number of selected tests and their execution time are two essential metrics of our experiment.

Suppose we are going to run an experiment on commit `C1`, and the previous commit is `C0`. The diff patch `D` is generated between `C0` and `C1`. The experiment procedure runs as followed. First, revert subject repo to `C0` and run all tests on it. After tests finished, we collect the total number of tests and all tests running time. Then use NodeSRT to perform test selection technique with `D` and collect the running time of executing selection steps. After that, we collect the number of selected tests and run all selected tests. Collect the running time of executing selected tests. Lastly, we run the Jest OnlyChange option as a comparison and collect the number of Jest OnlyChange selected tests and its running time.

The experiment is running on a 4 core x86_64 CPU with 16 GB of RAM, AWS cloud Linux server. To automate the process, we used Docker containers. We usually run four containers at a time in one Linux instance. Due to the fact that the internet speed is not unchanged, plus the computing speed of shared instance is relatively unstable, we assume that in one experiment, these factors are unchanged. Therefore, we use the percentage of tests selected and the percentage of SRT full process running time to represent the result. The percentage of test selected is calculated by the formula:

$$\frac{number\ of\ selected\ tests}{total\ number\ of\ tests} * 100\%$$

The percentage of SRT full process running time is calculated by the formula:

$$\frac{selection\ step\ running\ time + selected\ tests\ running\ time}{all\ tests\ running\ time} * 100\%$$

4.3 Experiment result

In this section, we are going the describe the result of our experiment.

4.3.1 Uppy

As an initial experiment, we run experiment on commits created from Jan 6, 2020, to July 31, 2020, which includes 480 commits. Since the tests of Uppy only has around 20% of the code coverage, most of the commits are not covered by tests. As a result, only 63 commits have test selected.

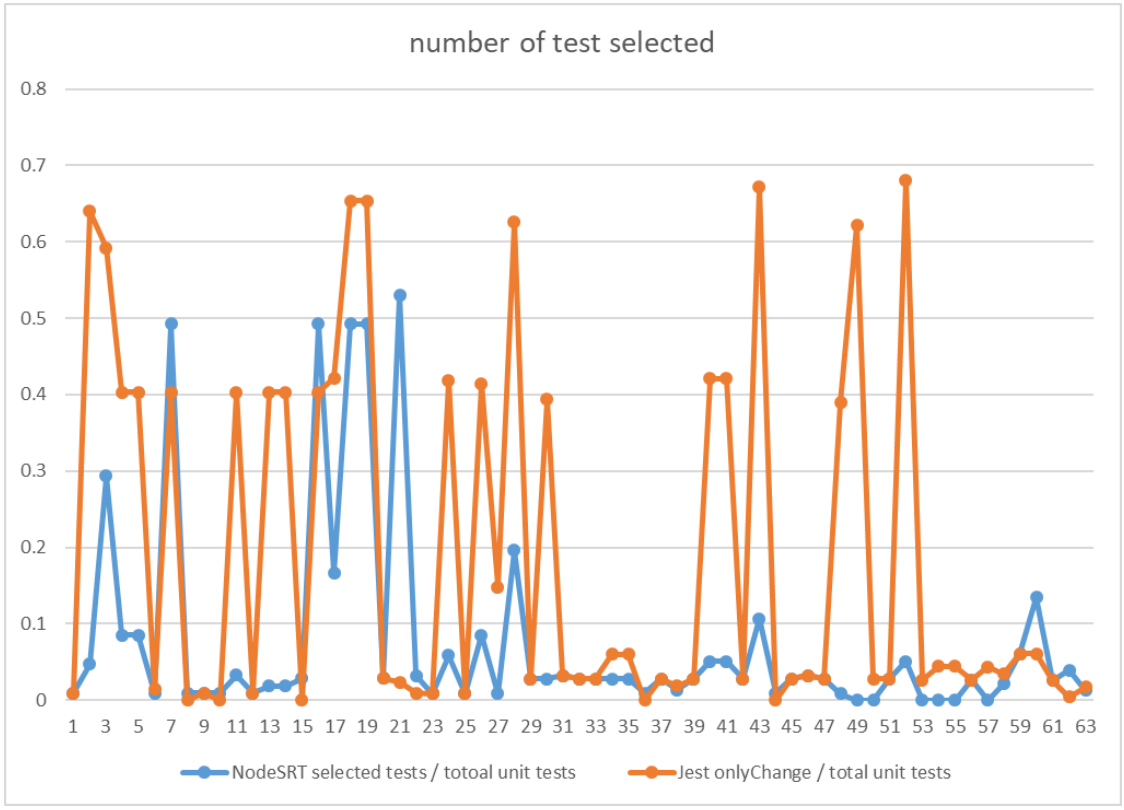

Figure 18 Uppy comparing NodeSRT and Jest OnlyChanged on number of tests selected

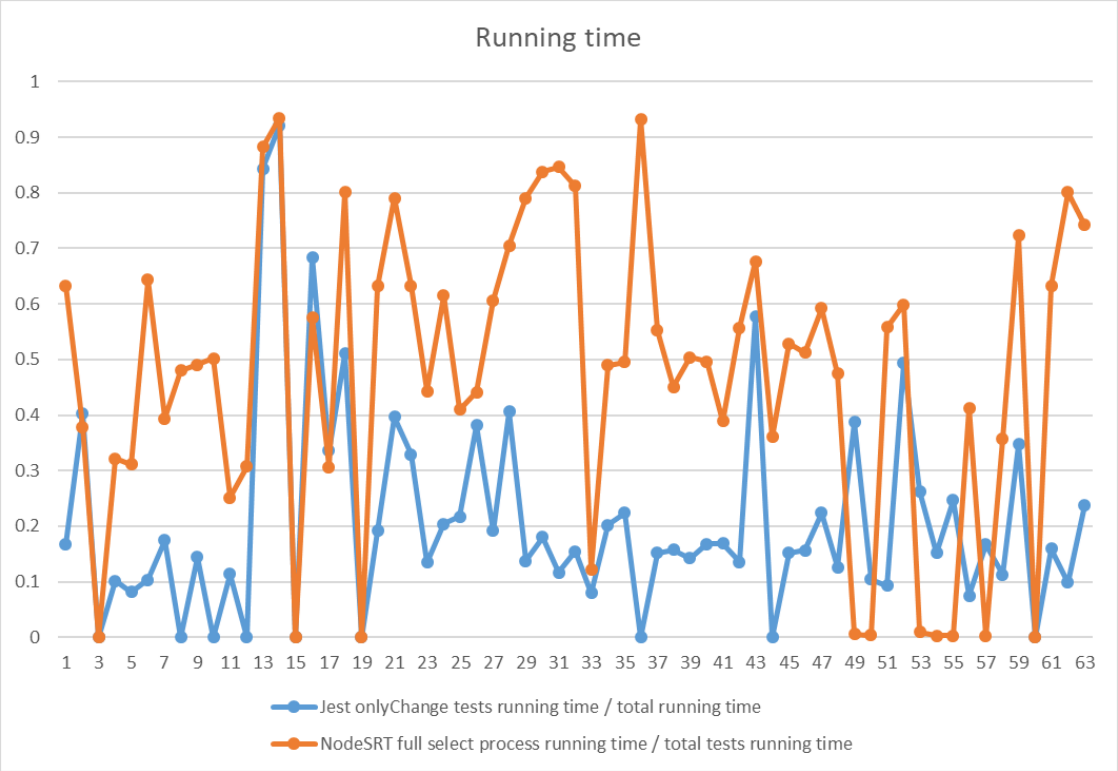

Figure 19 Uppy comparing NodeSRT and Jest OnlyChanged on selected test running time

The result shows, for these 63 commits, the test selection time is very low, on average of 490.1 milliseconds, with the highest of 1785.5 milliseconds and lowest of 94.9 milliseconds. As for the number of tests selected, NodeSRT selects 8.397% of total tests on average. Jest OnlyChange option selects 20.66% of total tests on average. In Figure 18, we can see that Jest OnlyChange option selects more tests than NodeSRT most of the time. There were six times when Jest OnlyChange option selects tests whereas NodeSRT did not select any tests. This is because Jest also does static file dependency analysis on files other than JavaScript using the jest-haste-map[1] module. Therefore, changes in non-

[1] jest-haste-map, https://www.npmjs.com/package/jest-haste-map

JavaScript files such as package.json, index.html can be captured by Jest OnlyChange Option. We can conclude that for Uppy, Jest OnlyChange option is better in terms of inclusiveness. NodeSRT is better in terms of precision. In fact, NodeSRT selects almost 1.5 times fewer tests than the Jest OnlyChange option.

What surprised us most is the total running time. In Figure 19, we can see that although Jest OnlyChange selects more tests than NodeSRT, Jest OnlyChange selected tests runs faster than NodeSRT selected tests. Jest OnlyChange took on average of 23.63% of all tests running time to run selected tests. NodeSRT took on average of 50.46% of all tests running time to run the full SRT process (This includes time to run selection steps and run selected tests). After investigation, the reason why NodeSRT took longer time is not because the NodeSRT selection steps took too long. Since Jest OnlyChange is an option of running jest cli, it makes use of their own jest-haste-map module and jest customized file system module watchman[1] to select and execute tests. In comparison, NodeSRT lexically replaces `it` to `it.only` for selected tests in affected test file makes Jest have to scan over all test files to get a list of files to run, then start running these tests.

[1] watchman, https://facebook.github.io/watchman/docs/install.html

4.3.2 Simorgh

For Simorgh, we ran 108 commits with NodeSRT, 77 of those have tests selected. This project has a relatively high code coverage rate (about 97%).

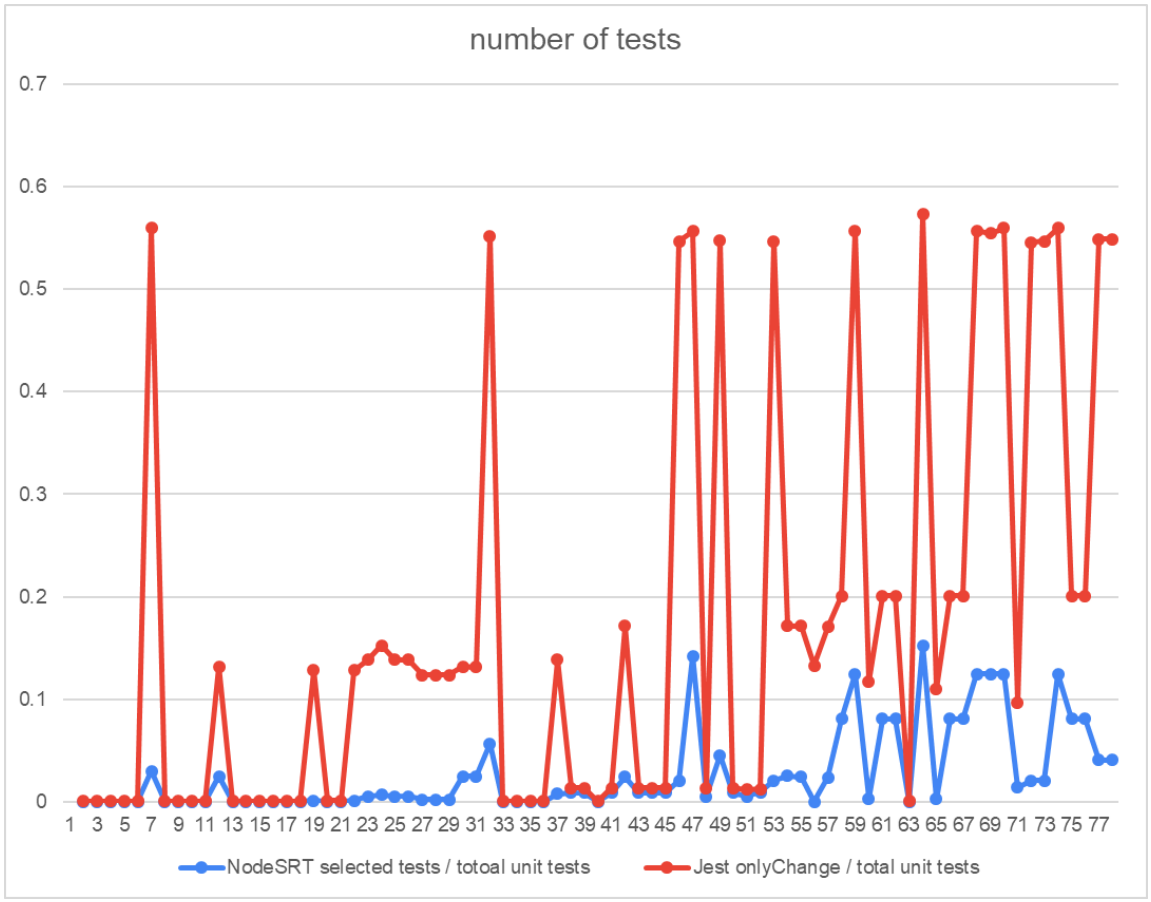

Figure 20 Simorgh comparing NodeSRT and Jest OnlyChanged on number of tests selected

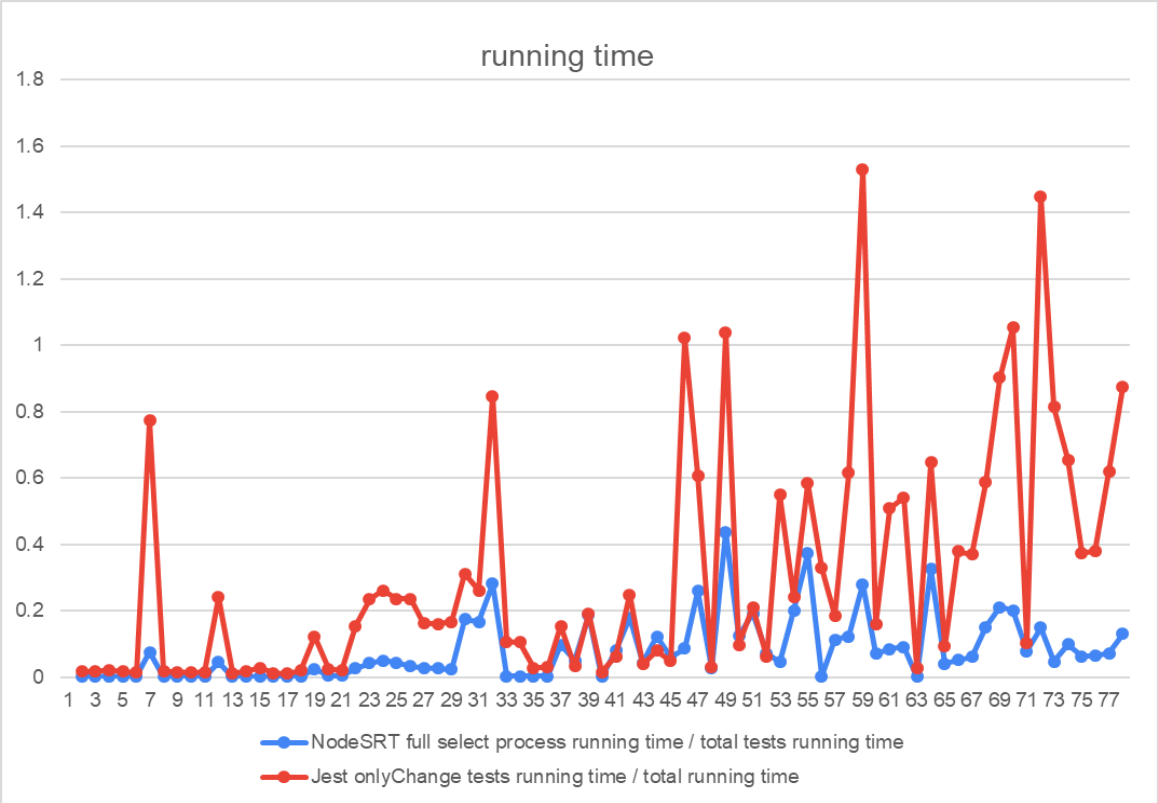

Figure 21 Simorgh comparing NodeSRT and Jest OnlyChanged on selected test running time

For these 77 commits, NodeSRT selected 2.73% of total tests on average. Jest OnlyChange selected 17.26% of total tests on average. The time needed for selection steps is still very low, with 2256.3 milliseconds on average, highest of 16239.6 milliseconds, and lowest of 290.7 milliseconds. Since Simorgh has a bigger codebase, we can roughly conclude that the selection step running time and the size of the codebase being analyzed are positively correlated. The experiment on Simorgh also indicates that our technique is more precise than Jest OnlyChange and less inclusive than Jest OnlyChange.

In terms of running time, Figure 21 shows that NodeSRT is better than Jest OnlyChange in general. In fact, NodeSRT took an average of 8.09% of all test running time to run the full SRT process. Jest OnlyChange took on average of 30.18% of all tests running time to run selected tests. Even though Jest OnlyChange can make use of their own file system module, the number of tests selected slow down the process significantly (Jest OnlyChange selects 5.32 times more tests than NodeSRT on average).

4.4 Summary

In this chapter, we discussed the empirical studies performed on NodeSRT. By running the selection process on two different Node.js projects with a total of 588 round of experiments, we validated our SRT technique has high inclusiveness for JavaScript file changes. The running time for selection step of our technique is also low. It only took less than 5% of full SRT process running time for both projects. Comparing to Jest Onlychange, NodeSRT selects fewer tests in general, which means NodeSRT is more precise. For projects with high code coverage, NodeSRT is five times more precise than Jest OnlyChange. However, due to the fact that Jest OnlyChange is highly integrated with Jest and uses its own file system module, it outperformed NodeSRT in terms of inclusiveness and selected tests running time. Future improvements can be worked on in these areas.

Chapter 5

Discussion

In this section, we discuss threats to validity and future works.

5.1 Internal Validity

Since all Empirical study experiments are done on the AWS cloud server, the network speed and computing speed might vary from time to time. Thus, it might cause interference to the measuring of test running time. To mitigate this threat, we assume the network speed and computing speed is constant during one experiment process and used the ratio of full reduction process running time to whole test suite running time to represent the running time of our SRT process.

Besides, test failing is another thread that affect the number of tests selected. Failed tests can cause incomplete process of dependency graph collection, thus cause test selection failed. Therefore, we discard samples with test fail exists before or after changes applied to the codebase.

5.2 External Validity

The primary threat to external validity is regard to the representativeness of our sample population. We tried to mitigate this thread by choosing two Node.js projects with different size of codebase and different test coverage. These two projects used the most

common frameworks in Node.js such as React and Webpack. Another concern is regarding the representativeness of our selected changes. We applied 588 commits in total for our empirical study to mitigate this thread. These commits were from the mature period of software development, where regression testing becomes essential.

5.3 Future works

Plugins Improvement. Since NodeSRT is independent of Jest, it cannot access Jest's own file system module and test running module. As a result, as shown in the empirical study on Uppy, even though NodeSRT selects fewer tests than Jest OnlyChange, Jest OnlyChange can still run reduced tests faster than NodeSRT. Future work on making NodeSRT a plugin of Jest can further improve performance. Besides, by accessing Jest's own file dependency graph can reduce the redundant works and improve inclusiveness.

Generalizability. Although NodeSRT provides a command-line interface, it can be further improved in terms of generalizability to make it work on more projects. A lot of other Node.js frameworks have not been tested for NodeSRT. Besides, a more user-friendly interface for end-to-end test selection is required.

Chapter 6

Conclusion

As software projects become mature, the cost of regression testing becomes significant. This thesis presents an approach to perform selective regression testing on Node.js applications. By making use of a change-based selection technique, obtaining a function call relationship with dynamic analysis, collecting file dependency information with static analysis, our technique reduces regression test in relatively short running time and high inclusiveness and precision.

We conducted an empirical study on two Node.js projects with different codebase size and code coverage to evaluate our technique. We further compared our approach with the current industry used test selection technique regarding metrics of inclusiveness, precision, and performance. Results show that our technique achieves the same level of inclusiveness and performance but with much higher precision. For Uppy, our technique is 1.46 times more precise than Jest OnlyChange. For Simorgh, our technique is 5.32 times more precise than Jest OnlyChange.

Future work can be done in integrating our technique with the Jest testing framework to improve its performance and inclusiveness. Also, more empirical evaluations are needed to enhance generalizability.

Bibliography

[1] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering,* vol. 22, p. 529–551, 1996. doi: 10.1109/32.536955.

[2] F. Schiavio, H. Sun, D. Bonetta, A. Rosà and W. Binder, "NodeMOP: runtime verification for Node.js applications," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019. doi: 10.1145/3297280.3297456.

[3] M. Hirzel, "Selective regression testing for web applications created with google web toolkit," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform Virtual machines, Languages, and Tools - PPPJ' 4*, 2014. doi: 10.1145/2647508.2647527.

[4] M. Gligoric, L. Eloussi and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, 2015. doi: 10.1145/2771783.2771784.

[5] J.-W. Lin, C.-Y. Huang and C.-T. Lin, "Test suite reduction analysis with enhanced tie-breaking techniques," in *2008 4th IEEE International Conference on Management of Innovation and Technology*, 2008. doi: 10.1109/icmit.2008.4654545.

[6] E. Engström, M. Skoglund and P. Runeson, "Empirical evaluations of regression test selection techniques," in *Proceedings of the Second ACM-IEEE international*

symposium on Empirical software engineering and measurement - ESEM '08, 2008. doi: 10.1145/1414004.1414011.

[7] S. Biswas, R. Mall, M. Satpathy and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica,* vol. 35, 2011.

[8] A. Zarrad, "A Systematic Review on Regression Testing for Web-Based Applications," *Journal of Software,* vol. 10, p. 971–990, 8 2015. doi: 10.17706//jsw.10.8.971-990.

[9] M. J. Harrold and M. L. Souffa, "An incremental approach to unit testing during maintenance," in *Proceedings. Conference on Software Maintenance, 1988.*, 1988. doi: 10.1109/icsm.1988.10188.

[10] A.-B. Taha, S. M. Thebaut and S.-S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," in *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. doi: 10.1109/cmpsac.1989.65142.

[11] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, p. 173–210, Apr. 1997. doi: 10.1145/248233.248262.

[12] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering,* vol. 24, p. 401–419, Jun. 1998. doi: 10.1109/32.689399.

[13] Y.-F. Chen, D. S. Rosenblum and K.-P. Vo, "TestTube: A system for selective regression testing," in *Proceedings of 16th International Conference on Software Engineering*, 1994. doi: 10.1109/ICSE.1994.296780.

[14] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in WebKit," in *2012 28th IEEE international conference on software maintenance (ICSM)*, 2012. pp. 46–55.

[15] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon and A. Gujarathi, "Regression test selection for Java software," *ACM SIGPLAN Notices*, vol. 36, p. 312–326, Nov. 2001. doi: 10.1145/504311.504305.

[16] X. Ren, B. G. Ryder, M. Stoerzer and F. Tip, "Chianti: a change impact analysis tool for Java programs," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*. doi: 10.1109/icse.2005.1553643.

[17] *"Node.js," Wikipedia. Wikimedia Foundation, Aug. 2020, [Online]. Available: https://en.wikipedia.org/wiki/Node.js.* [Accessed: 29-Aug-2020].

[18] S. Mirshokraie and A. Mesbah, "JSART: JavaScript Assertion-Based Regression Testing," in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2012, p. 238–252.

[19] S. Mirshokraie, A. Mesbah and K. Pattabiraman, "Efficient JavaScript Mutation Testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013. doi: 10.1109/icst.2013.23.

[20] S. Alimadadi, A. Mesbah and K. Pattabiraman, "Hybrid DOM-Sensitive Change Impact Analysis for JavaScript," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Dagstuhl, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.321.

[21] *"Jest CLI Options · Jest," Jest. [Online]. Available: https://jestjs.io/docs/en/cli#–onlychanged.* [Accessed: 29-Aug-2020].

[22] *“JavaScript,” MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.* [Accessed: 29-Aug-2020].

[23] A. Memon, I. Banerjee, N. Hashmi and A. Nagarajan, "DART: a framework for regression testing "nightly/daily builds" of GUI applications," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* doi: 10.1109/ICSM.2003.1235451.

[24] A. Labuschagne, L. Inozemtseva and R. Holmes, "Measuring the cost of regression testing in practice: a study of Java projects using continuous integration," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017, doi: 10.1145/3106237.3106288.

Appendix

project name	total round of experiment	valid round of experiment	average full unit test running time (s)	total unit test number	average selected unit test number	average selection step running time (ms)	average NodeSRT selected unit running time (s)	average jest selected tests	average jest selected test running time	average percentage of NodeSRT selected test number
Uppy	480	63	69.3	217	16.23	490.1074	65.74	40.95238	55.70355	8.39661%
Simorgh	108	77	450.7072	2803	111.1698	2256.295	31.88217	483.7792	115.277	2.72996%

Project name	average percentage of Jest OnlyChange selected test number	average percentage of NodeSRT full process running time	average percentage of Jest OnlyChange running time
Uppy	20.66006%	50.23649%	23.62704%
Simorgh	17.25890%	8.09741%	30.18125%

Table 2 Empirical study result summary