A practical survey of automatic unit test generation for JavaScript applications

Niclas Ringbom, 40605
Master of Science (Technology), Master’s thesis in Computer Engineering
Faculty of Science and Engineering, Information Technologies
Åbo Akademi University
10/2023
Supervisor: Dragos Truscan

Abstract

Unit testing can and should be used during the software development lifecycle in order to continually verify that each component functions according to specifications. However, development hours are often more preferably used to create additional features, rather than ensure that the existing features are functioning properly. Automatic unit test generators could provide the test suites, which would spare the developer from exhausting development hours on writing exhaustive unit tests without jeopardizing the quality of the software. Several unit test generators exist for other programming languages such as Java, but few seem to support Javascript. The purpose of this thesis is to evaluate the availability and performance of automatic unit test generators for Javascript. Most of the tools found had not been updated for several years, and as such, were unusable. Out of the two seemingly capable tools, Ponicode was removed from the public market, and Examin generated test suites averaging a line coverage of 30.19%. As such, unit test generators for Javascript are not advanced enough to replace manual test writing

1. Introduction

Unit testing is an essential part of any good code base. It promotes reusable and reliable code, improves the overall quality of the code as well as simplifies integration and extension of the software. By using unit tests, we can validate that the software performs operations in the manner it is intended to.

Manually writing unit tests exhausts development hours that could be used to further improve or extend the software, and is one of the main reasons why unit tests are neglected [11]. This neglect opens up the software to faults, and could eventually result in a costly restructuring of the code base. Automatic test generators can be used to aid developers in ensuring that the software is adequately tested, even during time-sensitive development phases.

Early identification of faults in code is essential, as the problem or fault is fresh, which often leads to a quicker solution. This is due to the fact that the developer does not need to reacquaint themselves with the code base. Since testing can easily be ignored during the development process of a software project, frameworks that could automatically generate unit tests would help alleviate the issue of late refactoring due to software faults.

Automatic test generators allow the developer to forgo the creation and design phases included in testing cycles, and provide the test author with the resources needed to run the tests and validate the system under test. To produce tests, the generators may rely on source code or a variety of artifacts. These artifacts include resources such as class diagrams and design specifications. Software testing can be automated at various levels of the development life cycle.

The test cases written by the unit test generators are written separately for each function or component of code, and can be modified or expanded upon. This is essential for ensuring that the test suite created is able to test all parts of the code and offers an acceptable degree of fault detection.

Javascript is the most commonly used programming language, for the tenth year in a row, with a 65.36% usage during 2022 according to Stackoverflows' survey [21]. Javascript is mostly used in browser environments, with 97.3% of all websites using it as a client-side programming language [22]. In addition to the popularity Javascript has as a client-side language, it can also be utilized for non-browser usage [7]. These non-browser implementations are often server-side deployments with Node.js or similar frameworks.

The purpose of this thesis is to evaluate the state of unit test generators for Javascript, both through a survey of existing tools and a practical experiment.

1.1. Research questions

This thesis will answer the following research questions, each providing the reader with insights into the subject from four different perspectives:

- *RQ 1: What kind of tools and approaches exist for automatic unit test generation for Javascript?*
    - This RQ will be answered in chapter 3. State of the Art by conducting a survey on the subject "Javascript unit test generation".
- *RQ2: What is the availability and maturity level of existing tools for Javascript?*
    - Answered through an analysis of current tools, gathered through a search in relevant channels. This research question is answered in chapter 3. State of the Art.
- *RQ3: What is the performance of existing tools?*

    - This question will be answered through two subquestions, RQ 3.1 and RQ 3.2. These questions will measure the effectiveness in commonly used metrics such as code coverage and fault detection.
    - *RQ3.1 What is the level of* ***code coverage*** *achievable for automatically generated unit tests?*
        - Answered through a combination of two subjects; experiments conducted on viable generators in thesis, and reported results for generators not included in the experiment
    - *RQ3.2 What is the quality of the generated unit tests?*
        - This question is only applied to tests generated during the practical experiment. It is answered through a code review in this thesis' experiment.
- *RQ4 What are the limitations of the automated test generation frameworks?*
    - This final research question will be answered through a discussion of the subject where the results of the previous questions will be summarized and analyzed. This research question is answered in Chapter 6. Discussion.

The first two chapters, 1. Introduction and 2. Background, introduce concepts required to grasp the methodology of generating unit tests as well as understanding the challenges they face. These challenges are presented both from a general point of view and specifically for Javascript. The third chapter, 3. State of the Art, presents the methodology of the literature survey, and the results of it. Excluding the final chapters, 6. Discussion and 7. Conclusion, the remaining chapters describe the experiment and present the outcome of it.

2. Background

In this chapter of the thesis, subjects such as test oracles and unit test generation are presented and discussed. An overview of core Javascript functionality is also presented to give the reader a better understanding of the obstacles unit test generators face for this specific programming language. The purpose of this chapter is to further explain concepts strongly related to unit test generation in an effort to improve the readers' ability to interpret the discussion and analysis presented in the remaining chapters.

2.1. Javascript

Javascript as a programming language is prototype-based, single-threaded, and dynamic [7]. The language supports many different programming styles, due to its underlying object-oriented principles as well as imperative and declarative properties [7].

Javascript's prototype-based quality is one of the more appealing features of the language, as it allows the usage of classes and objects, without necessarily needing to explicitly define the properties of said object [8]. Compared to other popular object-oriented programming languages, such as C or Java, Javascript allows for a larger ambiguity as a default. As properties are added to empty objects during runtime in Javascript, the typing (among other things) of the objects does not need to be as clearly designed or planned for the code to run.

This behavior can also open up the code to faults, as objects and functions will allow almost any type of input. A simple summation of two variables can result in many different outcomes, depending on what type the variables represent. An example of this behavior is given in Figure 1, where two cases of integer addition are presented. In Figure 1-a , a string representation of the number 3 is added to 5, which results in the variable num1 (5) to be cast to the *string* type, to allow the concatenation of the two variables. The two variables (now both in the string type for this operation) are concatenated to result in the string representation of “53”. In the second case (Figure 1-b), an integer representation of the number 3 is added to the number 5, which gives us an expected result of 8.

```
const num1 = 5;
const num2_string = "3";

const sum = num1 + num2_string;
//53
```

a)

```
const num1 = 5;
const num2 = 3;

const sum = num1 + num2;
//8
```

b)

Figure 1. Summation of two values in Javascript

As made apparent in the example above, code execution in Javascript may not always give the expected outcome if the input or output is not controlled or checked. This uncertainty adds to the importance of testing Javascript code, to ensure that the methods give the expected results.

2.2. Javascript frameworks

Javascript implementations often use frameworks or libraries to expand upon the core functionality of pure Javascript (often called vanilla JS). These tools have their own intended use case, and provide additional functionality to support and effectivize development. Their syntax differs slightly from vanilla JS but they are able to interpret pure JS. The Javascripts communities' inclination to use frameworks and other extensions in order to expand upon the base toolkit offers a challenge to unit test generators as the generator is given an additional factor to consider when interpreting.

Two of the most popular frameworks for both front-end and back-end are introduced in this section. The statistics for the usage of frameworks are fetched from State of JS [12], which conducts an annual survey of the different technologies used by Javascript developers. Statistics for the usage of Javascript frameworks is presented below in Figure 8. [12] and Figure 9. [13].

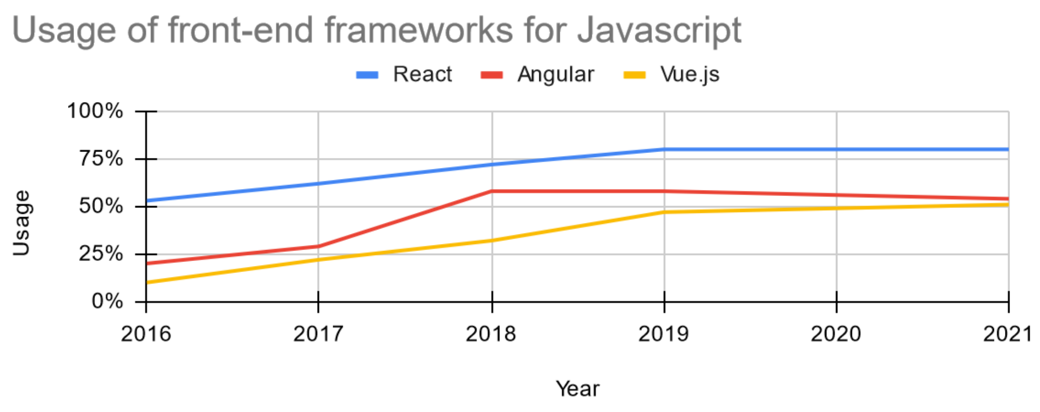


Figure 8. Usage of front-end frameworks for Javascript

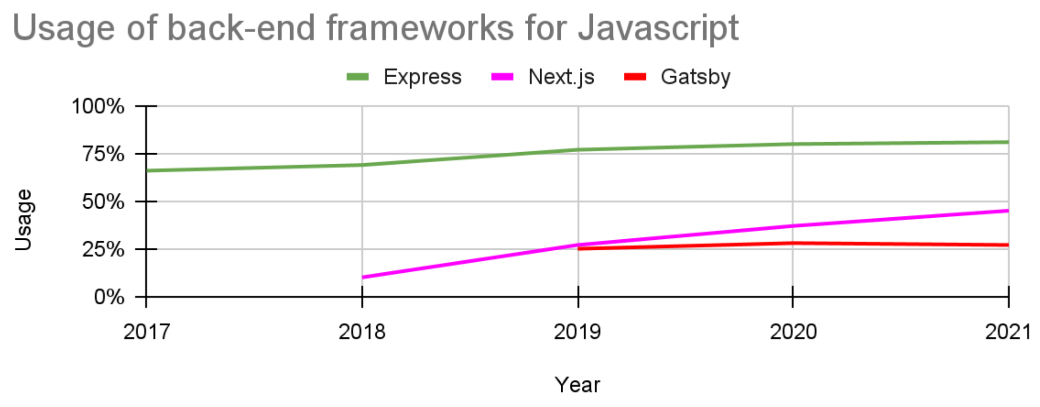


Figure 9. Usage of back-end frameworks for Javascript

2.2.1. React

React is a front-end library, even if it is often referred to as a framework. It is the most popular front-end tool for Javascript in 2021 according to State of JS [12] The library offers a simplified method for building a complete user interface by providing access to functionality such as individual components and state management. It also offers a virtual DOM, which allows the user interface to render and rerender only the essential changes. The component-based feature of the library incentivises the development of encapsulated units which are then combined into declarative views. Correctly utilized, this development style isolates components from each other which results in code that is easier to read and test.

2.2.2. Angular

Angular is the second most popular front-end tool [12]. Angular functions similarly to React, with components as the primary building blocks of the application. This separation of components offers a good starting point for quality control. Angular offers many features that impact the application development process, such as built-in Typescript, full dependency injection, and two way data binding. Lastly, being a full-featured framework, Angular provides the developer with all the tools required to create anything from small scale projects to enterprise solutions.

2.2.3. Express

Express is the most used back-end framework for JS [13]. It is used to build and manage servers and routers, with features such as APIs and middleware. The framework is minimalistic, and provides developers with little else than the most essential functionality to create a back-end.

2.2.4. Next.js

Next.js is a back-end framework often used in tandem with React, as it expands upon the functionality of React to allow websites to be rendered server-side rather than client-side. In fact, the React website recommends using Next.js for server-side rendering [16].

2.3. Software testing

Software testing refers to the process of verifying that the software functions as it is intended to. There are various different levels of software testing practices available to developers, which all have different use cases, and validate the systems' functionality in different ways. These are the four software testing levels [4]:

- Unit testing
    - The testing of individual units of source code. Depending on the coding practice, this unit may be a single function in a file or a complete segment of source code.
    - Used for continual testing, and often done during the early stage of development.
- Integration testing
    - The testing of the interaction between software components.
    - Done after the unit testing stage, to ensure that components with verified functionality interact with each other as designed.
- System testing
    - The testing of the whole system, where the system is compared to the original objectives [5, p. 130-132].
- Acceptance testing
    - The comparison of the complete product against the requirements and needs of the end users. This stage of the software testing process differs from the other types of testing

since this process is often done by the customer and not the developer. [5, p. 140-144]

This thesis will focus on the importance and effects of unit testing and briefly introduce the most commonly used testing levels. Since unit testing is not dependent on the other levels of testing, methods and techniques related to this level of testing can be analyzed in isolation.

The hierarchy of the software testing levels is presented in the figure below (Figure 2). As can be observed from the figure, the three broader testing levels (integration, system, and acceptance) are reliant on their narrower counterparts. For instance, when integration testing is performed on a combination of software components, an assurance of the correctness of respective components is needed. This assurance could be achieved through unit testing. This behavior is inherited to the broader levels, where system testing is aided by integration and acceptance testing by system testing.

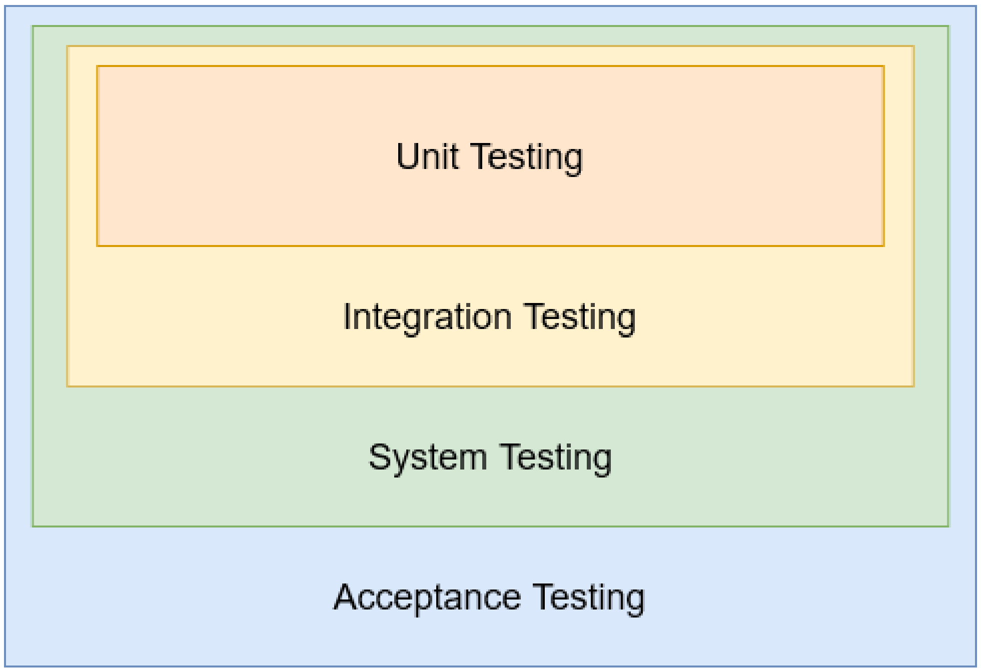


Figure 2. Software testing levels

2.4. Unit testing

Unit testing is one of the four software testing levels available, along with integration testing, system testing, and acceptance testing. Unit testing allows the testing of individual components of the source code. It is an integral part of a complete quality assurance system, as it ensures that each individual component of the code is working in accordance with design specifications.

In comparison to the other testing levels, unit testing has several advantages as well as disadvantages. One of the most prominent advantages is that it allows the developer to test small parts of the project, which helps with continuously verifying the quality of the project code. For larger, collaborative projects, unit tests provide a way for teams to test their own contribution prior to integrating with other branches of code. Since these tests are possible to execute on individual components, this testing process can be initiated as soon as the component is believed to function according to specifications. Identifying bugs and defects at an early development stage is essential, as eliminating these bugs becomes increasingly more difficult and time-consuming as the system is developed and software components are combined.

Unit testing also enables developers to refactor code at a later stage, while still verifying that the module or component is working as designed. There are many other advantages to unit testing, such as promoting code familiarity and allowing for more accurate design of each individual unit.

Disadvantages and limitations of unit testing are numerous, which is mostly due to the nature of writing explicit tests. The most obvious limitation of unit testing is the fact that it cannot be used to test interaction between components or modules. This results in the necessity to use other methods of testing in addition to unit testing, to ensure that the system as a whole functions as intended. Another disadvantage is that unit testing adds an additional layer of software that needs to be maintained. If the unit testing suites are poorly designed or their maintenance is neglected, they may provide the developer with faulty errors that are a result of inferior tests rather than poor code quality. Lastly, creating, executing, and maintaining unit testing suites are tasks that are quite time-consuming. This is often the reason why unit testing is neglected, since it often depletes resources that could be spent developing the system. This disadvantage, however, could be seen as an advantage as well, as writing tests for the system promotes familiarity with the code base, as was mentioned earlier.

The ultimate goal of unit testing is to ensure and verify that individual components of code function as they were designed. For more complex behavior, other testing levels prove superior to unit testing, as they are able to verify the interactions between components as well as the system as a whole. While there are several other software testing levels, unit testing is one of the more prominent and useful tools for a developer. Since unit testing requires a great deal of understanding what exactly the components should achieve in their functionality and how the code for them is written, unit testing is considered a white box approach. This implies that the writer (creator) of the tests has complete knowledge of the system under test. As unit tests are written for individual components of code, it is recommended to write tests for components during the development process.

In practice, unit tests use functionality of the code to test if certain inputs correspond to expected outputs or results. There are several syntactical ways to structure unit tests, of which the so-called “AAA method” and the table-driven method will be introduced in the following sections. Regardless of which structure is used, having a predetermined template of how unit tests should be written in a project promotes efficiency and readability.

2.4.1 Arrange, Act, Assert structure

The standard practice is to use the three A's of unit testing steps (Arrange, Act, Assert) [19], where the unit test is divided into three parts.
The three steps of the "Arrange, Act, Assert" structure promote readability by compartmentalizing each procedure.

The first step, "Arrange", should set up the test case by defining and assigning variables or objects. In addition to assigning needed variables, the arrange step should also initialize any features that provide the data or functions needed to test the unit. These features include things such as databases or a context, like logging in to a web app.

The behavior and business logic of the unit should be executed in the second step, "Act". This is the primary part of the unit test, and should mimic all of the different uses for the component. This includes all function calls or interactions with the logic of the unit or SUT.

After preparing the resources needed and executing the logic of the unit, the final step "Assert" is performed in order to verify that the outcome is as expected. This is usually done through an assertion check, which in the case of Jest, is the expect() function. Below is an example of how a unit test could be written in Jest [1], a popular Javascript unit testing framework, using the "AAA" structure. The source code to test is provided in the first code block and the test on the latter.

```
function sum(a, b) {
  return a + b;
}
```

```
describe('sum', () => {
    test('1 + 2 = 3', () => {
        // Arrange
        const a = 1;
        const b = 2;
        const expected = 3;

        // Act
        const result = sum(a, b);

        // Assert
        expect(result).toBe(expected);
    });
});
```

Figure 3. Jest unit testing using “AAA” structure

One disadvantage of the “AAA” structure is the need to write extra code, which for large test suites can result in a large amount of code. This results in costly refactoring of the tests. For example, the test above could be written in a single line instead of 5.

```
expect(sum(1, 2)).toBe(3);
```

Figure 4. Jest unit test single line

2.4.2 Table-driven structure

An alternative to the “AAA” structure is the table-driven structure, where the inputs and desired outcomes are sorted in a table, similarly to the “Arrange” step of Figure 3. The values from the table are then referenced in the assertion, which uses the different values when running multiple tests.

Applying table-driven testing to a unit test for the same sum() function could look like Figure 5.

```
describe('sum', () => {
    // Defines inputs 'a' & 'b', and expected outcomes
'expected'
    const tests = [
        {a: 1, b: 2, expected: 3},
        {a: 5, b: 5, expected: 10},
        {a: 4, b: 3, expected: 7},
    ]
    // Calls the function to be tested sum() inside an
assertion expect()
    tests.forEach(({a, b, expected}) => {
        expect(sum(a, b).toBe(expected);
    });
});
```

Figure 5. Table-driven unit test

The advantage of table-driven testing is that a single unit test is able to perform many different assertions. A single line for assertions also promotes maintainability, as the unit test is able to be reused for functions with similar logic, in this case basic arithmetic such as subtraction or multiplication.

After the test suite is completed, either through manual or automatic means, it needs to be run against the system under test to check for faults. This final stage of the unit testing process can either be done manually or automatically (often as a part of the deployment process). The automatic testing of the system ensures that new or modified components of code function as defined in specifications and requirements.

For the test suite to be accepted as adequate, it has to meet the defined system requirements. Once determined as adequate, the component is evaluated by running the unit test. If the test passes without any faults, the component can be verified to function as intended.

2.5. Unit testing metrics

Due to the strong connection between white box testing and unit testing, the most used metrics for unit testing rely on the fact that the source code is available. Two of these metrics are *code coverage* and *mutation score*.

Code coverage is defined as the degree to which a test suite (consisting of one or several tests) executes the source code [2]. The code coverage metric does not require the tests to be written in a complex manner, and could in some cases be considered to be an inaccurate metric for evaluating code quality. A study performed by Lucas Gren and Vard Antinyan found little to no correlation between code quality and code coverage [3]. Code coverage may be obtained through different levels, such as:

- Statement coverage: The ratio of all statements (lines) covered by the test suite in the source code.
- Branch coverage: The ratio of branches covered by the test suite in the source code. For example, if a test suite executes both branches of an if statement, the branch coverage for that code would be 100%.
- Function coverage: The ratio of functions in the source code called by the test suite.
- Condition coverage The ratio of boolean expressions that have been evaluated to true or false.

Mutation testing can be used to temporarily alter the source code, in order to simulate real faults. This is most often done automatically by tools such as Stryker [18]. Stryker supports JavaScript, TypeScript, C#, and Scala. The quality of a test suite can be evaluated via mutation testing. By using mutation testing to seed faults into the source code, one is able to test a system to a greater extent compared to using the original code. The higher the mutation score a test suite is able to achieve, the more thorough the test suite is. As a metric for unit test suites, mutation score is considerably more useful compared to code coverage [54]. An example of code mutation can be found from Figure 6 below.

```
function isUserOldEnough(user) {
  return user.age >= 18;
}
```

```
/* 1 */ return user.age > 18;
/* 2 */ return user.age < 18;
/* 3 */ return false;
/* 4 */ return true;
```

Figure 6. Stryker: Mutated return statements [17]

In the case presented in Figure 6, the return statement from the original source code on the left is mutated into the variants on the right. The modified program is then executed against the test suite to verify that the unit tests are written in a manner that tests every possible change (mutation) the statements or expressions could undergo. Summarily, mutation testing is able to evaluate the test suite's ability to find faults in the source code.

One of the ways to calculate mutation score is by calculating the percentage of generated mutants found in a test suite, but since this depends on how relevant and accurate the generation of mutants is, this might be an inaccurate measurement.

2.6. Mocking code for unit tests

Mocking of code is often performed in the unit testing phase, where the purpose is to unit test components that depend on other components which can not be verified as correct. These components are either not tested or not available for testing. This isolation is an essential part of testing Javascript code as units often depend on other components of software, whether it is another unit or an external component. The complex behavior is simulated via mocking so that a specific part or unit can be accurately tested.

In Javascript, this mocking can not only be performed on functions but on modules as well. This is an extremely important part of unit testing Javascript code, as there are often calls made to APIs (Application Programming Interface) which drastically reduce the reliability and speed of the tests. With mocking, this unreliability can be negated by simulating a response in the detriment of accuracy.

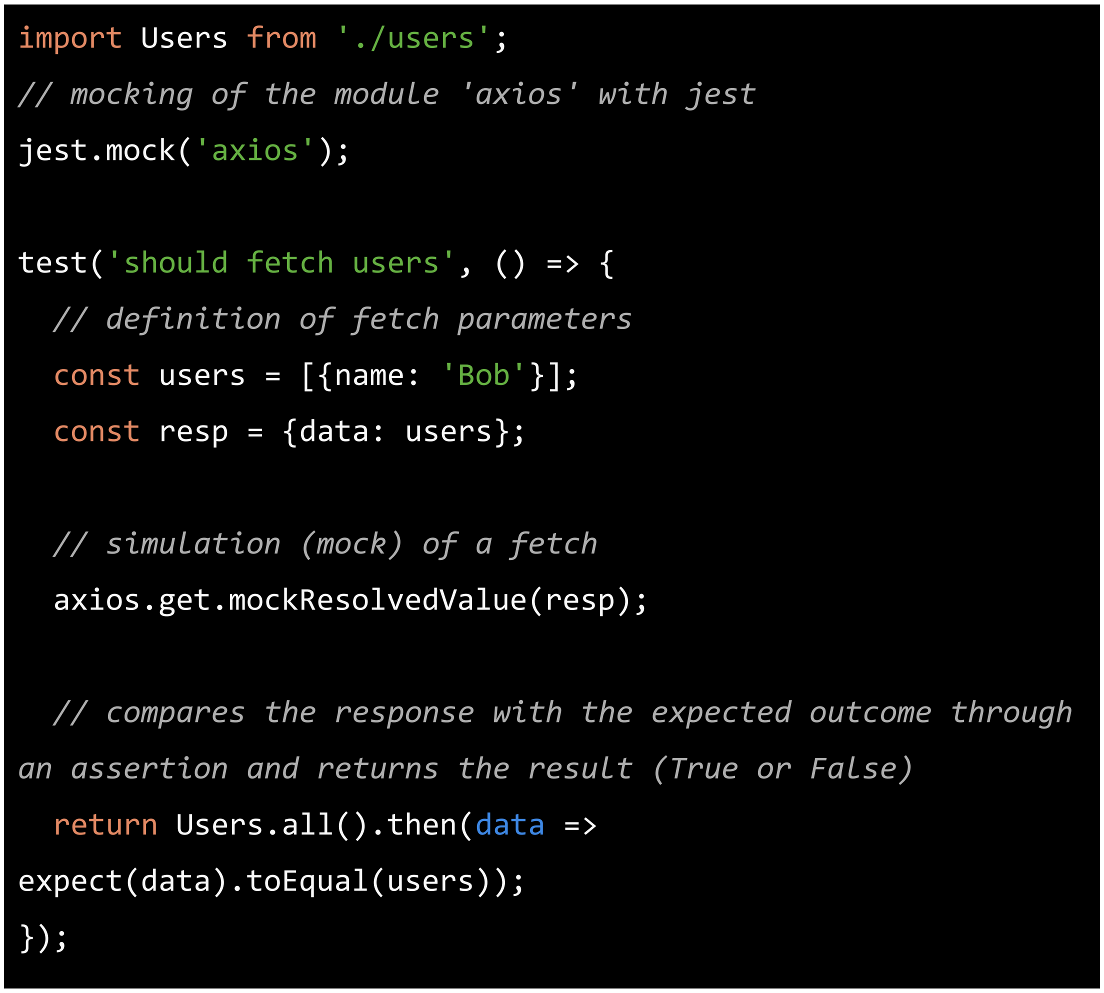
An example of module mocking in the testing framework Jest is presented below. Figure 7a is the system under test and Figure 7b is the test used to evaluate the code.

```
import axios from 'axios';

class Users {
  static all() {
    // axios fetches data from /users.json and assigns it to
resp.data
    return axios.get('/users.json').then(resp => resp.data);
  }
}
export default Users;
```

a)

```
import axios from 'axios';
```



```
import Users from './users';
// mocking of the module 'axios' with jest
jest.mock('axios');

test('should fetch users', () => {
  // definition of fetch parameters
  const users = [{name: 'Bob'}];
  const resp = {data: users};

  // simulation (mock) of a fetch
  axios.get.mockResolvedValue(resp);

  // compares the response with the expected outcome through
an assertion and returns the result (True or False)
  return Users.all().then(data =>
expect(data).toEqual(users));
});
```

b)

Figure 7. Module mocking in Jest

The framework uses the .mock() function to automatically mock the axios module, so that it is possible to simulate a response. In this particular case, the test simulates a fetch from a remote API, with the parameters of an array containing “name: ‘Bob’”. The variable definitions are done as one would without the mocking, but then the response is fetched through a mocked version of axios. Lastly, the simulated response is evaluated through an assertion using the expect(...) function. In essence, the mocking in this example allows the developer to test the unit without requiring a functional API.

2.7. Test oracles

When generating tests, a procedure that can differentiate between correct and incorrect behaviors is needed [9]. This procedure is referred to as a test oracle. The test oracle is aware of how the system under test should behave and what the desired outcome of the system and corresponding test is. These test oracles can then be used to design and create tests, as they provide knowledge on what is correct and incorrect. The different categories of test oracles are presented below [9].

- Implicit test oracles - Defined by implicit information on whether a system is working as expected, such as compilation faults or execution failures.
- Derived test oracles - Defined by information derived from documentation, system executions or previous versions of the SUT.
- Specified test oracles - Defined by formal specifications and software models. The only type of oracle to rely on mathematically based techniques.
- Human test oracles - In the absence of any form of information that could be used to define any of the other test oracles mentioned, human test oracles have to be utilized. In this case, the test oracle has to be based upon vague and abstract concepts such as “gut-feeling” or intuition since there is no tangible specification available.

Some of these types of oracles have their own disadvantages or inaccuracies depending on the information used to generate them. For implicit test oracles, the biggest challenge is the fact that not all faults are universal [9]. Behaviors in one system might be considered abnormal, while the same behavior in another system could be intended [9]. This results in the need to define new test oracles specifically depending on the system, as well as the context the oracle is used in. In the case of specified test oracles, the challenge is connected to the limitations of a formal specification. As formal specifications use abstraction to a great degree, the translation from abstract to definite may produce an inaccurate test oracle.

When the developers act as the test oracle, they are able to take formal and informal requirements into consideration when writing and examining tests. Naturally, a multitude of flaws exist when the developer acts as the oracle as this opens up the testing to a greater degree of human error and interpretation. This brings us to the test oracle problem which is a large hindrance for the effectiveness and correctness of automated testing as well as automatically generated testing.

The test oracle problem is defined as the challenge of differentiating between correct, intended behavior, and incorrect behavior [9]. This problem is particularly problematic for the effectiveness of automatically generating unit tests, as there is minimal supervision from the human aspect of the development process. If no useful information exists for creating test oracles, the generator is not able to write serviceable unit tests. If a test is written without assertions, the developer is required to review the incomplete test and complete the process [10], which renders the generation of the stubbed unit test unnecessary.

2.8. Unit test generation

Unit test generation is a complex process that differs greatly depending on the system under test. As each programming language has their own defining features, the same unit test generator might not be able to be used for a different language or even a different framework. For instance, when comparing plain Javascript with Java, many defining features of the languages differ. Javascript is an interpreted language, whereas Java is a compiled language. Another significant difference is the fact that Javascript is a weakly typed language, and Java is strongly typed.

There are many different methods that unit test generators can employ to analyze the code. As the generator has to have intricate knowledge of the system under test, this method of analysis is of great importance to the effectiveness of the unit tests. This knowledge may be derived either from the source code or from the specifications of the SUT. Without this knowledge, the generator would not be able to differentiate between a well constructed test and a simple test. Simple tests would lead to worse code coverage, and more importantly, to worse fault detection. This need for unit test generators to understand the system under test leads us to the oracle problem, explained in Section 2.1.

2.9 Unit testing frameworks in Javascript

Many unit testing frameworks are available for Javascript, from which a developer can choose a suitable framework depending on the use case. Frameworks such as Jest [1] or Jasmine [49] are used to test the functionality of the computational section of the code, whereas tools such as Puppeteer are used to determine if a site is rendered according to expectations.

Jest is one of the unit testing frameworks available for JS and was the most used testing framework in 2020 and 2021 [15]. It is compatible with most other frameworks and libraries used in JS projects, such as Node, React, Typescript, etc.

3. State of the art

This section of the thesis will cover the current state of unit test generation, both through a literature survey of academic publications and a survey of existing tools. The aim of this chapter is to present the past and current environment in which generators are made or prototyped. The primary analysis of the current state of unit test generation will be presented through the literature survey. This literature survey will consist of research from academic channels. Afterwards, a survey of existing tools, either found through the literature survey or other means will be introduced. The final subsection will summarize the results found from these surveys. This chapter will answer RQ1 and RQ2.

3.1. Literature survey

The first section of this chapter will present an academic literature survey of the topic “Unit test generators for Javascript” in order to collect resources and information required to answer the research questions in this thesis. These publications are collected from relevant academic channels such as Google Scholar [31] and IEEE Xplore [32]. The selection criteria for the literature survey is as follows:

- The tool is focused on unit test generation for JS
- The publication presents a fully functional tool
- The capabilities of the tool are presented through an experiment on open-source programs
- The generated tests should be fully functional unit tests

Materials referenced in collected resources will also be used in order to find similar materials. The resources in this literature survey were collected on 12/2022. The same queries were performed in both of the channels with the following search terms:

- “Unit test generation Javascript”
- “Automatic unit testing”

- “Test generation Javascript”

In total, approximately 30 publications were reviewed. As a result of these queries, and reviews of the resources referenced in them, the following publications were found to be the most relevant for the purpose of this survey:

Title	**Proposed tool**	**Year of publication**	**Found in Publication Database**
JSEFT: Automated Javascript Unit Test Generation [10]	JSEFT [25]	2015	IEEE
A framework for automated testing of javascript web applications [28]	Artemis [26]	2011	Google Scholar
A Symbolic Execution Framework for Javascript [27]	Kudzu (not public)	2010	Found through references in [28], available on IEEE

Table 1. List of resources for literature survey

As can be observed from the table above, the most relevant publications found in the literature survey are almost all a decade old, with the exception of JSEFT. This reflects the current state of academic publications within the field, as filtering results in both IEEE and Google Scholar from 2015 onwards returns no publication with a proposed tool. As such, it is not possible to assume these tools would be able to achieve the same degree of code coverage and fault detection if applied to code written today. This is due to the fact that Javascript has changed considerably since 2015, when ES6 (ECMAScript 6) was released. Changes to the language included the usage of “let” and “const” as variable keywords, promises, and arrow functions. The substantial changes to the syntax, as well as the industries' inclination to use frameworks such as React could severely impede the effectiveness of old tools. Some of the tools have received updates since their release to improve their capabilities. Of the tools listed in Table 1, Artemis is the only one to have received such updates. Regardless, the results presented in these publications offer valuable insights as to which degree unit test generators for Javascript could generate test suites.

3.2. Survey of existing tools

Similarly to the literature survey, relevant channels such as Google and Github were used to find existing tools using the following search terms:

- "Unit test generation Javascript"
- "Automatic unit testing"
- "Test generation Javascript"

The tools were filtered for those that were publicly available (community editions) or open source. The first search for tools was conducted on 05/2022 and the final search on 12/2022. No additional tools were found during the final search. Three additional tools were found as a result of the survey of existing solutions: Ponicode, Examin, and jest-test-gen. These tools were all found as a result of queries on Google.

3.3. Results

An introduction of the tools found as a result of the surveys in sections 3.1 and 3.2 are presented in Table 2 below, with a description of their test generation method, latest update, licensing, and a link to the tool.

Name	**Method**	**Last updated**	**Licensing**	**Link to tool**

Ponicode	AI, NLP processing of semantics and structure	2022	Commercial application, licensing not applicable	https://www.ponicode.com/developers
Examin	Feedback-directed	2021	MIT license	https://github.com/oslabs-beta/Examin
Artemis	Feedback-directed	2017	GPL v3.0	https://github.com/cs-au-dk/Artemis
JSEFT	Event-space exploration	2014	Undisclosed	https://github.com/saltlab/JSeft
jest-test-gen	Undetermined	2022	MIT license	https://github.com/egm0121/jest-test-gen

Table 2. Set of found unit test generators for Javascript

The advertised capabilities, availability, installation, as well as other relevant information on the tools found will be presented below.

Some of the tools found have been excluded from this set, as they are outside of the scope of the intended practical experiment or not publicly available. The excluded tools are listed below, along with a reason for their exclusion.

- Kudzu [27], not publicly available
- JS Test Gen [30], only generates test templates

In the following sections, possibly applicable unit test generation tools are reviewed.

3.3.1. Ponicode

Ponicode is an AI-powered unit test generator for JavaScript, TypeScript, Python, and Java. The tool analyzes the structure and semantics of the code using AI powered natural language processing [23], and uses the information obtained to generate unit tests.

For Java, the tool is available as an IntelliJ plugin and as a VSCode extension for the remainder of the supported technologies. A CLI (Command Line Interface) is also available, as an npm package. The CLI supports test generation for JavaScript, TypeScript, and Python. Ponicode generates unit tests for the following test frameworks, for each supported technology [33].

Language	**Test framework**
JavaScript	Jest
TypeScript	Jest
Python	Pytest

Table 3. Ponicode VSCode supported technologies.

The installation process is simple for both the CLI and the VSCode extension. NPM (Node Package Manager) is used to download and install the CLI. Before using the tool a small amount of configuration is required. In this configuration, properties of the system under test are defined. This configuration allows the tool to more accurately interpret the source code, which leads to higher quality tests. The VSCode extension can be installed through the VSCode marketplace and does not require any configuration.

Ponicode does not mention any results or studies done using the tool. As such, no details on the capabilities of the tool were found as a result of this survey. Further details on the capabilities of Ponicode are found in chapters 4 and 5.

3.3.2. Examin

Examin generates unit tests for React applications through a Google Chrome extension. The tool employs React Developer Tools to analyze the source code of the system under test, and generates unit tests for each individual component. Similarly to Ponicode, the tests are generated using Jest syntax.

Examin can be installed through the Chrome Web Store [35] or through manually building the extension using the GitHub repository [36]. No setup is required, with the exception of a few dependencies that are added to the system under test through NPM. In addition to the dependencies, Examin requires React Developer Tools [37] to function.

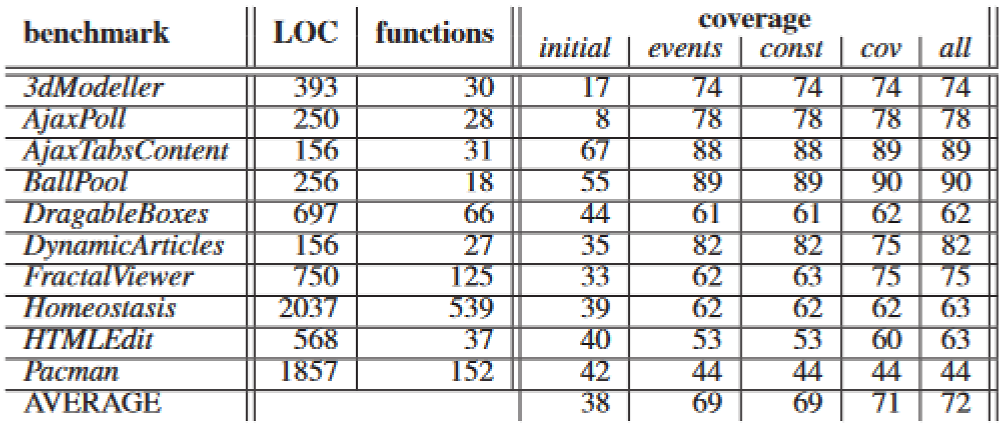
As was the case with Ponicode, Examin does not advertise any results or studies. As a result, the capabilities of this tool are not possible to determine without a practical experiment.

3.3.3. Artemis

Artemis is the first tool in the survey that is based on academic research. As such, intricate details about the tool and its capabilities are available in the research paper [28]. Based on the Rhino Javascript interpreter [38], it combines data from the interpreter, such as event handlers, constants, and read/write sets with feedback-directed random test generation algorithms [28].

The table below is a summary of the experimental results from the research paper [28] presenting Artemis. The columns are defined as follows:

- Benchmark, the system under test (SUT).
- LOC, total number of lines of code in the SUT.
- Coverage, line coverage (% of lines covered by generated tests) achieved by each test generation algorithm.



benchmark	LOC	functions	coverage				
			initial	*events*	*const*	*cov*	*all*
3dModeller	393	30	17	74	74	74	74
AjaxPoll	250	28	8	78	78	78	78
AjaxTabsContent	156	31	67	88	88	89	89
BallPool	256	18	55	89	89	90	90
DragableBoxes	697	66	44	61	61	62	62
DynamicArticles	156	27	35	82	82	75	82
FractalViewer	750	125	33	62	63	75	75
Homeostasis	2037	539	39	62	62	62	63
HTMLEdit	568	37	40	53	53	60	63
Pacman	1857	152	42	44	44	44	44
AVERAGE			38	69	69	71	72

Table 4. Artemis research publication experiment results [28].

A total of 100 tests were generated by each of the algorithms; *initial*, *events*, *const*, *cov*, and *all*. From the table we can observe that executing all of the test generation algorithms present in the tool against the systems under test resulted in an average of 72% code coverage. One negative aspect of the tool is that it requires a great deal of tests in order to achieve the code coverage presented. The results in Table 2 are all achieved after generating 100 tests, which for some of the programs is an unrealistic number of tests to maintain. All of the systems under test in the research were interactive and event-driven [28]. As such, the capabilities of the tool on other types of JavaScript programs is undetermined.

The installation of Artemis requires a great deal of effort in comparison to the other tools listed in this survey. A detailed installation file is included in the GitHub repository [26], which includes restrictions such as the usage of Linux-based systems.

3.3.4. JSEFT

The fourth tool included in this survey, JSEFT [10], is also a result of academic research. JSEFT uses dynamic exploration of the event-space through a function coverage maximization method [10] to interpret the code and generate unit tests. The tool was allowed to run for 10 minutes for each application, out of which 5 minutes were reserved for the dynamic exploration step [10]. There is no mention of how many tests were generated for each of the applications. In their research they claim to have achieved a higher code coverage than Artemis. In Figure 11 below, the code coverage percentage achieved in the experiment [10] of the tools JSEFT and Artemis are presented.

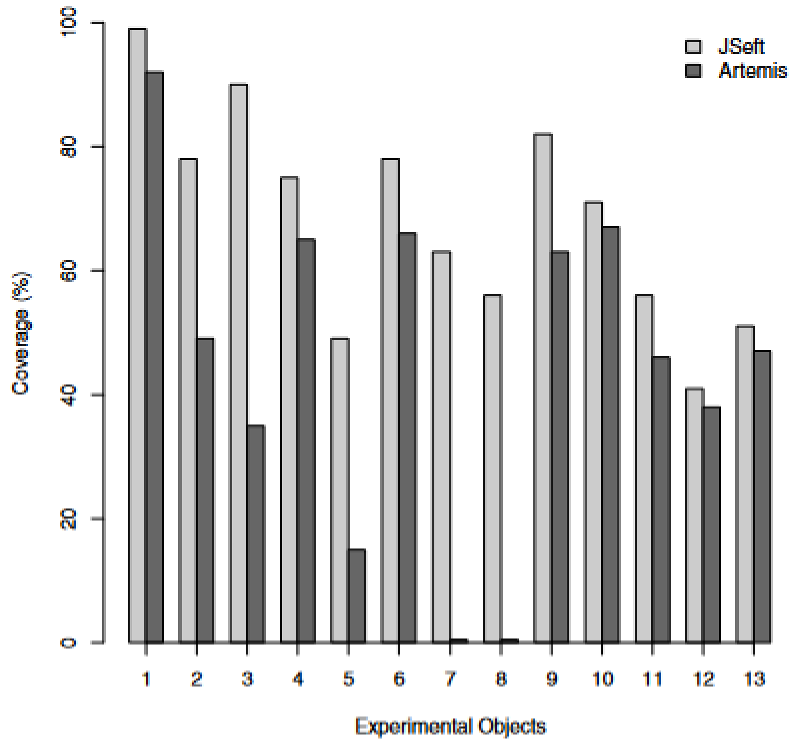


Figure 11. JSEFT & Artemis Code Coverage Comparison [10].

According to the research JSEFT is able to achieve 68.4% code coverage on average, while Artemis was only able to achieve 44.8% average code coverage on the same set of applications. In comparison to the benchmark programs used to evaluate Artemis [28], the experiment performed in the JSEFT paper uses many different application types. As a result, the experiment is more valid, as a considerable threat to validity in these experiments are the programs used to benchmark them. From the graph, we can observe that a more varied set of the systems under test result in a lower average code coverage for Artemis.

No installation guide is available on the Github repository [25], which severely hinders the availability of the tool.

3.3.5. Summary of survey

The capabilities of the tools Ponicode and Examin could not be determined in this survey, and would need to be included in the practical survey in order to evaluate their performance. One significant advantage of the tools appears to be their simple installation process, which lowers the initial cost for using (or evaluating) the tools. Another advantage of the tools is their usage of Jest syntax to structure the generated tests. This allows the tests to be easily integrated into existing test suites and expanded upon when required.

Both of the tools originating from academic research, JSEFT and Artemis, were possible to evaluate in the literature survey. Both of the research papers [10][28] included an experiment performed using the tools. Reviewing the experiments performed resulted in JSEFT appearing superior to Artemis. Even though these two tools seem promising after reviewing the research, the installation process for both of them appears cumbersome, which could negatively impact the usefulness of the tools. Another severe disadvantage of these tools is the fact that both papers (and the documentation of the tools) fail to mention in which syntax unit tests are generated. Compared to Examin and Ponicode, which use Jest, the tests generated by JSEFT and Artemis might not be able to comfortably integrate into testing frameworks.

Regarding RQ 2, many of the existing tools are not available as of today, either due to not being maintained or not possible to access. The remaining available tools do not seem to portray a high maturity level.

4. Evaluation

The purpose of this section is to present which unit test generation frameworks and how will be evaluated via a practical experiment. A brief explanation of how to generate tests with these frameworks will also be included along with a review of the readability of the generated tests. Lastly, a list of the metrics used to measure the effectiveness of the frameworks will be mentioned.

Out of the four tools found in the survey in chapter 3, two were chosen for the practical experiment. These tools were Ponicode and Examin. There were several reasons for only choosing these two tools out of the total 4 tools found, which will be listed below.

- Excluded from experiment
    - JSEFT
        - Unable to determine the syntax of the generated unit tests, likely to not be compatible with current test frameworks.
        - No installation guide included in the Github repository [25].
        - Installation of the tool proved unsuccessful, as a result of unresolved dependencies.
    - Artemis
        - Similarly to JSEFT, unable to determine the syntax of the generated tests.
        - Tedious installation process with heavy restrictions such as the usage of a Linux operating system.
- Included in experiment
    - Ponicode

        - Simple installation process with several options for usage platforms.
        - Promising early results.
        - Generates tests in Jest syntax, possible to integrate into existing test suites.
    - Examin
        - Browser based tool, effortless installation with no restrictions on development environment.
        - Generates tests in Jest or Enzyme syntax.

4.1. Test Generation Examples

A short running example of how to generate unit tests with Ponicode and Examin is presented below. Every step required to generate tests is briefly introduced, in order to provide the reader with insights into the process of generating unit tests. The following steps will be included in the examples:

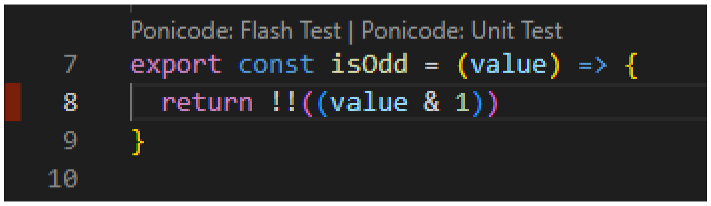
- Configuration
- Input required for test generation
- Test generation
- Generated test overview
- Test running and reporting

4.1.1. Ponicode

Ponicode offers two different methods for generating unit tests, a VSCode extension, and a CLI (Command Line Interface). Both methods require access to runnable source code. The CLI offers a simple and fast way to generate tests at the cost of customization, whereas the VSCode extension allows the developer to customize the generated unit tests to their own preference.

The VSCode extension does not require any configuration, but the tests themselves can be configured to the developers' needs. As mentioned earlier, the extension requires runnable source code as input, which in the case of Ponicode, is the only artifact used to generate test cases.

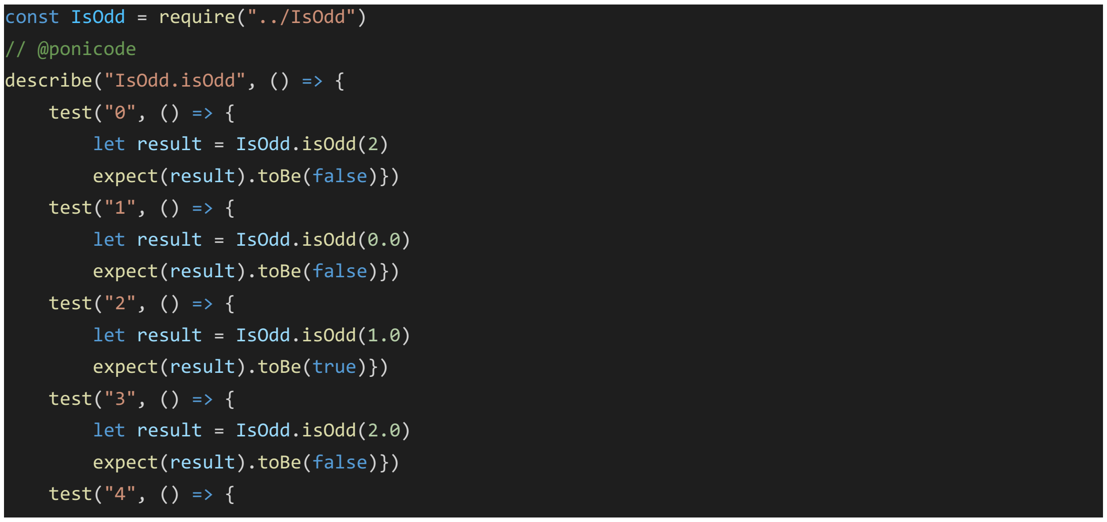
Tests can be generated in two ways, either using the “Flash Test" functionality or the “Unit Test” functionality. The “Flash Test” automatically generates 6 test cases, by default, for each function selected. The “Unit Test” functionality offers additional customization for the generation of the tests by opening a window from which the developer can select test cases from a pool of suggestions presented by the tool. The following function will be used as the unit under test for this running example.



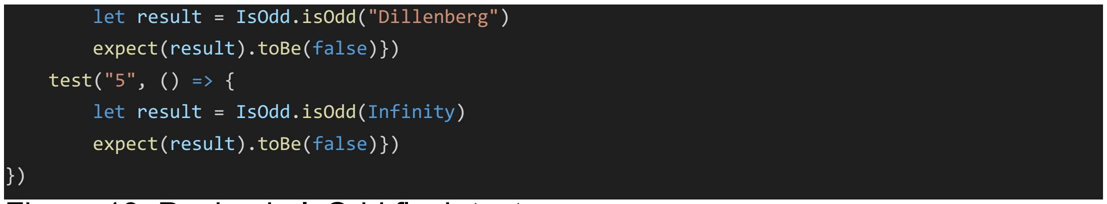
```
Ponicode: Flash Test | Ponicode: Unit Test
export const isOdd = (value) => {
  return !!((value & 1))
}
```

Figure 12. Ponicode example unit under test

In Figure 12 above, the two options for generating tests using Ponicode are visible. Upon selecting “Flash Test”, Ponicode generates the unit test in Figure 13 (formatted to take less space).



```
const IsOdd = require("../IsOdd")
// @ponicode
describe("IsOdd.isOdd", () => {
    test("0", () => {
        let result = IsOdd.isOdd(2)
        expect(result).toBe(false)})
    test("1", () => {
        let result = IsOdd.isOdd(0.0)
        expect(result).toBe(false)})
    test("2", () => {
        let result = IsOdd.isOdd(1.0)
        expect(result).toBe(true)})
    test("3", () => {
        let result = IsOdd.isOdd(2.0)
        expect(result).toBe(false)})
    test("4", () => {
```

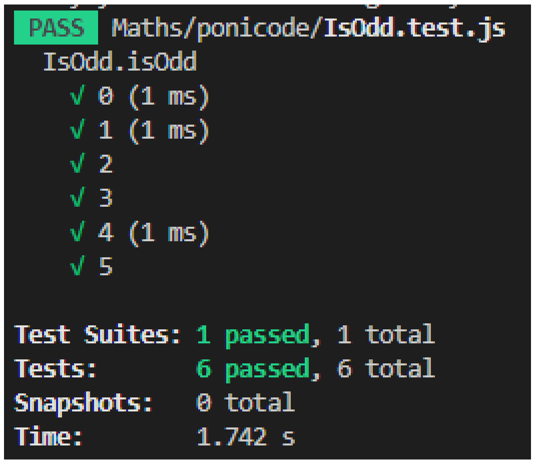


```
        let result = IsOdd.isOdd("Dillenberg")
        expect(result).toBe(false)})
    test("5", () => {
        let result = IsOdd.isOdd(Infinity)
        expect(result).toBe(false)})
})
```

Figure 13. Ponicode IsOdd flash test

The test suite contains 6 test cases in total, with 4 of them (tests 0-3) testing the logic of the function, and 2 of them (tests 4-5) testing edge cases for exception handling. This test suite is written into a separate subfolder, where the test can be added into existing test suites or run individually.

Tests generated by Ponicode are executed using Jest. These tests can be executed in multiple different ways, either through the ways recommended by Jest [14] or by opening the test suite and selecting “run”. The progress and result of the test execution is then reported in the terminal, along with any errors that might be present. The result of the test suite presented in Figure 14 produces the following result.

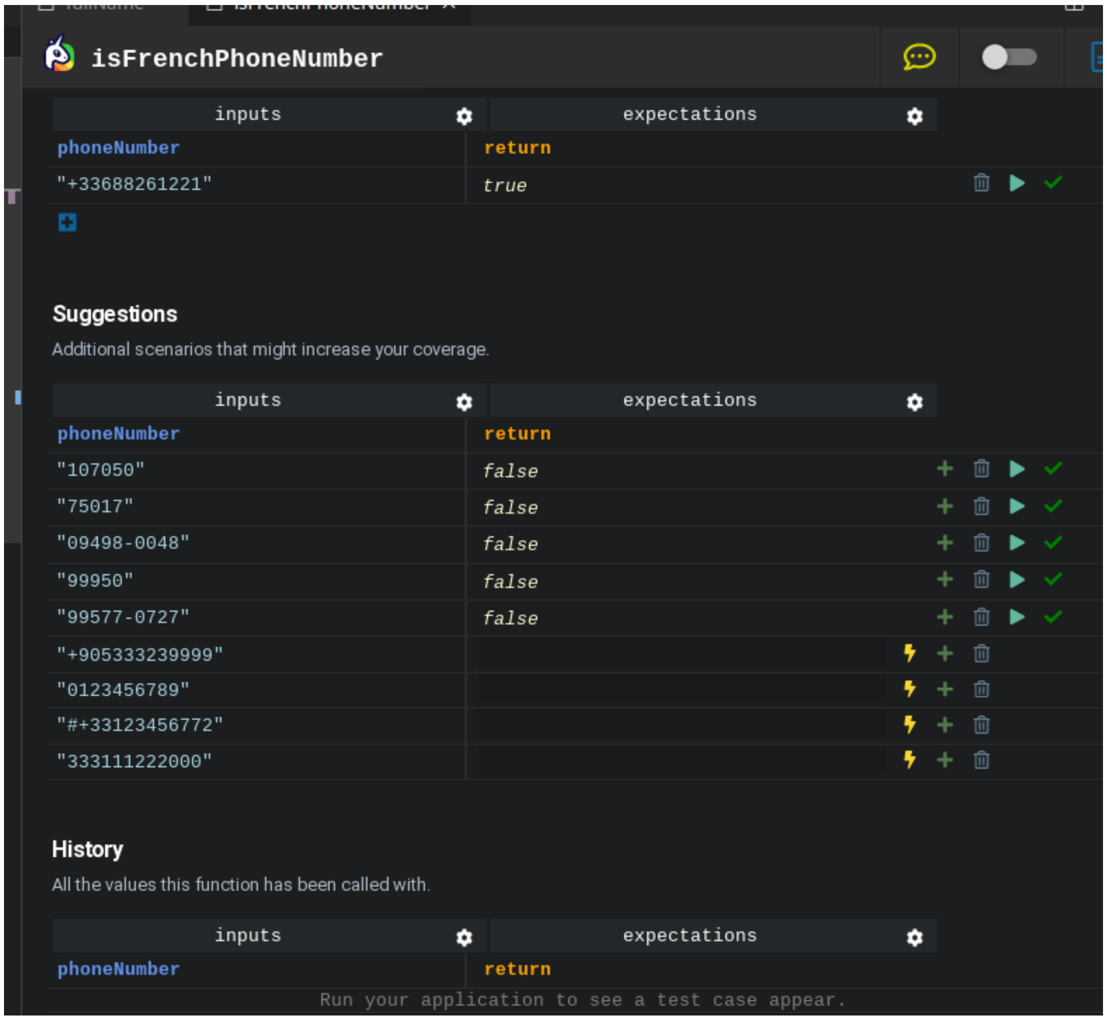


```
 PASS  Maths/ponicode/IsOdd.test.js
  IsOdd.isOdd
    √ 0 (1 ms)
    √ 1 (1 ms)
    √ 2
    √ 3
    √ 4 (1 ms)
    √ 5

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        1.742 s
```

Figure 14. IsOdd test execution result.

The “Unit Test” method functions similarly to the “Flash Test”, but instead of automatically generating a test suite, selecting the “Unit Test” opens a window in which the developer can add test cases from a list of suggestions. The developer can also add their own assertions to the unit test using the user interface. Figure 15 below presents the user interface of Ponicode's Unit Test feature.



isFrenchPhoneNumber

inputs	expectations
phoneNumber	return
"+33688261221"	true

Suggestions

Additional scenarios that might increase your coverage.

inputs	expectations
phoneNumber	return
"107050"	false
"75017"	false
"09498-0048"	false
"99950"	false
"99577-0727"	false
"+905333239999"	
"0123456789"	
"#+33123456772"	
"333111222000"	

History

All the values this function has been called with.

inputs	expectations
phoneNumber	return

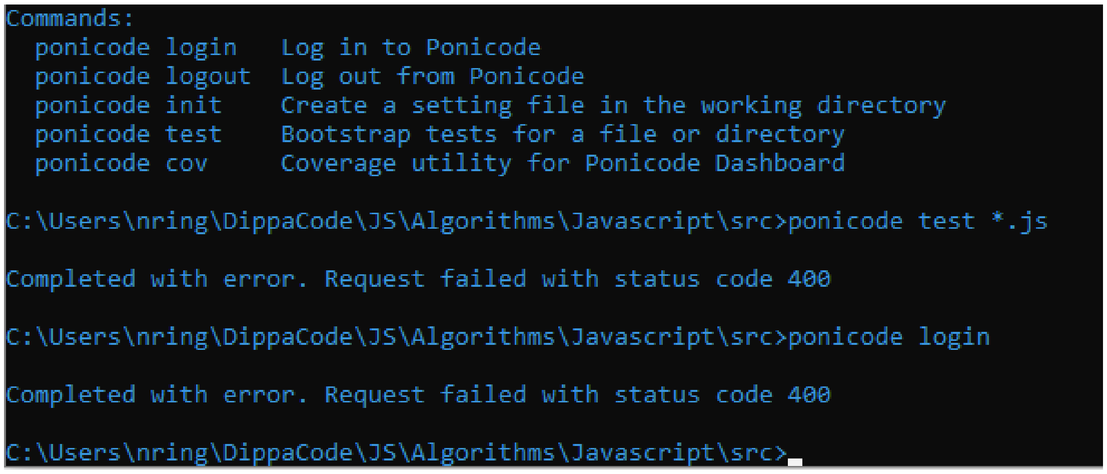
Run your application to see a test case appear.

Figure 15. Ponicode Unit Test User Interface [48]

The developers can also write their own test cases, which the extension then adds to the test suite.

As of 13.03.2023 all Ponicode solutions were discontinued and deprecated as a result of CircleCI acquiring the company Ponicode. All documentation related to the usage and features of the tool were removed from the public at the same date. CircleCI hopes to integrate the tool into their CI/CD product, and as such, the functionalities may become available again at a later date.

Attempting to use any of the solutions provided by Ponicode results in a failed request, as the test generation was hosted by the company itself. Further use of the tool is not possible at the time of the experiment, which results in the exclusion of Ponicode from the practical experiment. Below is a screenshot taken during an attempt to use the tool to perform the primary experiment.



```
Commands:
  ponicode login   Log in to Ponicode
  ponicode logout  Log out from Ponicode
  ponicode init    Create a setting file in the working directory
  ponicode test    Bootstrap tests for a file or directory
  ponicode cov     Coverage utility for Ponicode Dashboard

C:\Users\nring\DippaCode\JS\Algorithms\Javascript\src>ponicode test *.js

Completed with error. Request failed with status code 400

C:\Users\nring\DippaCode\JS\Algorithms\Javascript\src>ponicode login

Completed with error. Request failed with status code 400

C:\Users\nring\DippaCode\JS\Algorithms\Javascript\src>
```

Figure 16. Ponicode solution shut down.

Since the tool is no longer available for public use, the data found in the initial experiment to justify inclusion of the tool into the primary experiment will be used to review the capabilities of the tool. This review is presented in chapter 5.

4.1.2. Examin

Examin generates unit tests automatically from source code run on a local server. The tests are generated through a Google Chrome extension, which is opened on the browser, in the same window as the client connected to the local server. The unit tests are then automatically generated while using the application.

To install and configure Examin for a project, the following steps are required [36]:

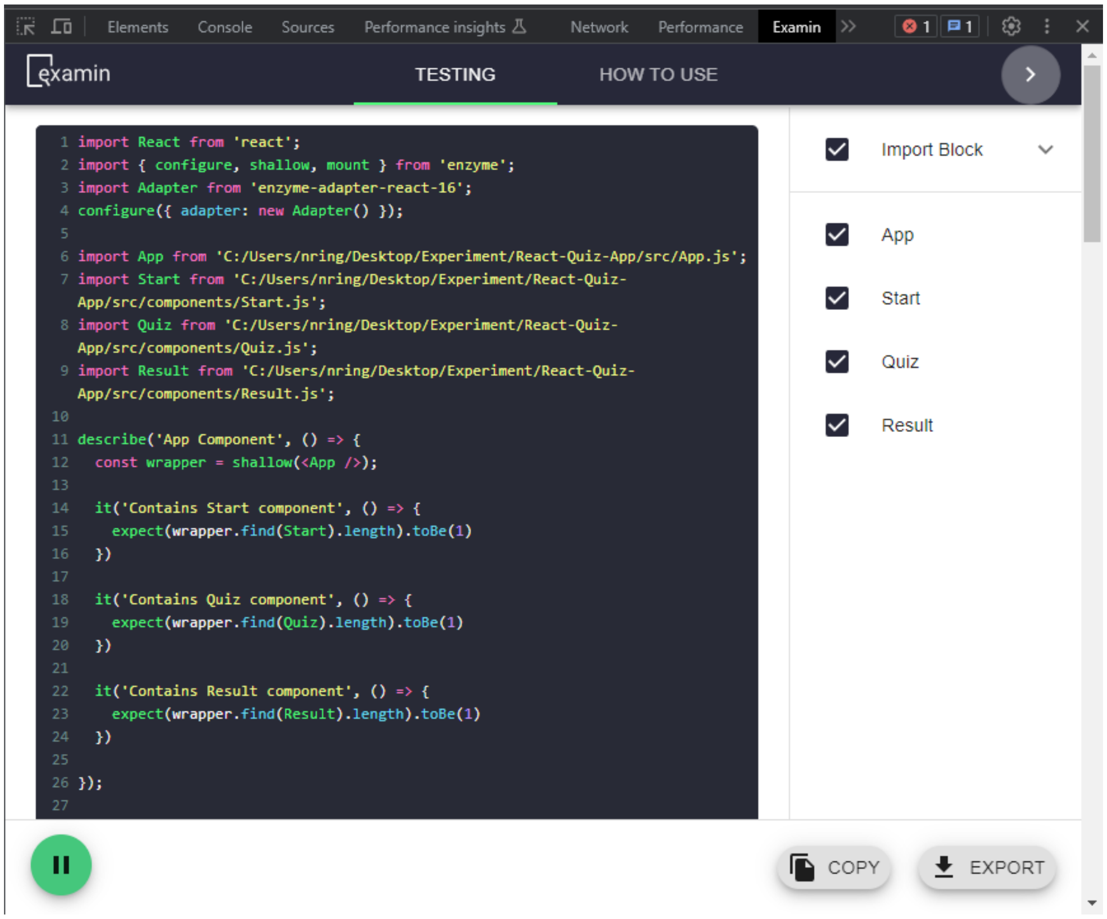
- Install the Examin extension for Google Chrome
- Install npm dependencies for Jest/Enzyme
- Navigate to the Examin panel in Chrome developer tools

After the installation, generate the unit tests by navigating through the application. The extension does not offer a great deal of customization, as the tool automatically generates test cases for each running component.

The simplistic interface of the extension offers a “How to use” tab, a “Start/Stop” function, as well as copy and export functions. The functions available in the extension are as follows:

- “How to use” tab, contains the same configuration information as on the GitHub page [36].
- Copy” function copies the contents of the generated test case to the clipboard.
- “Export” function exports the contents of the generated test case to a JavaScript file (.js).
- Detected components, allows the developer to show or hide test cases generated for each component.
- The “Start/Stop” function allows the developer to pause the unit test generation. This function is used to prevent generation of test cases for components that are rendered prior to reaching the unit to be tested (for instance pausing test generation during login to a webpage).

Figure 17 below is a screenshot of the tool during code execution.



```
import React from 'react';
import { configure, shallow, mount } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({ adapter: new Adapter() });

import App from 'C:/Users/nring/Desktop/Experiment/React-Quiz-App/src/App.js';
import Start from 'C:/Users/nring/Desktop/Experiment/React-Quiz-
App/src/components/Start.js';
import Quiz from 'C:/Users/nring/Desktop/Experiment/React-Quiz-
App/src/components/Quiz.js';
import Result from 'C:/Users/nring/Desktop/Experiment/React-Quiz-
App/src/components/Result.js';

describe('App Component', () => {
  const wrapper = shallow(<App />);

  it('Contains Start component', () => {
    expect(wrapper.find(Start).length).toBe(1)
  })

  it('Contains Quiz component', () => {
    expect(wrapper.find(Quiz).length).toBe(1)
  })

  it('Contains Result component', () => {
    expect(wrapper.find(Result).length).toBe(1)
  })

});
```

✓ Import Block

✓ App

✓ Start

✓ Quiz

✓ Result

Figure 17. Examin extension interface.

Figure 18 below consists of the test case generated for the “App” component of a simple tic-tac-toe game written in React [6] (Figure 24) used for the running example. This test case can then be added into a test suite and run with Jest.

```
//imports

describe('App Component', () => {
  const wrapper = shallow(<App />);

  it('Contains Header component', () => {
     expect(wrapper.find(Header).length).toBe(1)
  })

  it('Contains Board component', () => {
      expect(wrapper.find(Board).length).toBe(1)
  })

  it('Contains Footer component', () => {
      expect(wrapper.find(Footer).length).toBe(1)
  })

  it('App includes html elements', () => {
      expect(wrapper.find('[object
Object]').length).toEqual(1);
  });

});
```

Figure 18. Examin test example.

```
const x = 'clear'

const blank_boxes = Array(9).fill(null)

export const AppContext = createContext()

const App = () => {
    const [board, setBoard] = useState(blank_boxes)
    const [turn, setTurn] = useState(x)
    const { winner, winner_row } = check_winner(board)
    const running = game_over(board)
    return (
        <>
            <AppContext.Provider value={{ winner, turn, running }}>
                <Header />
                <Board
                    board={board}
                    setBoard={setBoard}
                    setTurn={setTurn}
                    winner_row={winner_row}
                />
            </AppContext.Provider>
            <Footer />
        </>
    )
}

export default App;
```

Figure 24. React tic-tac-toe App.js [6]

These generated test cases can then be used to supplement an existing Jest test suite, or be used as the basis for a new test suite. Examin offers no support for executing the generated tests or configuring a test environment. Test environment configuration and test execution should be performed as instructed in Jest Docs [14].

4.2. Metrics

In the practical experiment, line coverage is used to determine how well the frameworks are able to cover types of Javascript code. As line coverage alone is not a good metric for test quality, another metric should be used to confirm that the test not only tests the code, but confirms that the test the code adequately. Mutation testing was planned to use an additional metric, but due to the fact that Ponicode was no longer available, and dependency issues with Examin, a manual test review was performed on the generated tests.

4.3. Benchmark

This second section describing the practical experiment will present the importance of choosing different types of applications when evaluating the performance of unit test generators, as well as introduce the features of the applications chosen for this specific experiment.

4.3.1. Applications

Five different applications were chosen to evaluate the performance of Examin. All of these applications are written in React version 16.8 or higher. The defining features of the chosen applications are listed in Table 5 below.

Name	Description	Lines of code (JS)	React version	Github
react-calculator	Simple calculator	399	16.12	https://github.com/ManiruzzamanAkash/react-calculator
english-class-minigame	Word guessing game	502	18.0	https://github.com/bkfan1/english-class-minigame/blob/master/package.json
react-frontend-dev-portfolio	Portfolio website	769	16.13.1	https://github.com/Dorota1997/react-frontend-dev-portfolio
React-Quiz-App	Quiz app	201	18.0.0	https://github.com/md-kawsar-ali/React-Quiz-App
React-Weather-app	Simple weather app	318	18.2.0	https://github.com/codebucks27/React-Weather-app

Table 5. Properties of chosen applications.

4.4. Experiments

Details of how the experiment is conducted will be presented in this chapter. The two previous chapters, 4. Evaluation and 5. Benchmark explained which frameworks were included in this experiment and which applications they are being tested for. The contents of this chapter builds on those details, explaining how exactly the experiment is set up. These core details include how exactly the experiment is set up, in what environment, how data is collected, as well as the timeline for the experiment.

4.4.1. Experiment details

Only one tool is included in the practical experiment, as Ponicode, which was one of only two frameworks of notable performance found during the surveys in chapter 3. State of the art. As such, the experiment will be specifically designed around the needs of Examin, which is an automatic unit test generator for React applications with a version of 16.8 or higher.

The experiment in its entirety was performed by the author of this thesis. This includes everything from the execution of the tool against the selected applications to data collection. The experiments were performed on Windows 10 Version 10.0.19044, using VSCode as the IDE. NPM [46] was the package manager used to integrate Examin into the applications. Coverage data was fetched through Jests' built-in coverage reporter [47], which includes statement-, branch-, function-, and line coverage.

A thorough experiment was not possible to perform on Ponicode, as the tool was acquired by another company and terminated. The preliminary experiment was performed using a collection of algorithms written in pure JS [41], using the same environment and technologies mentioned above.

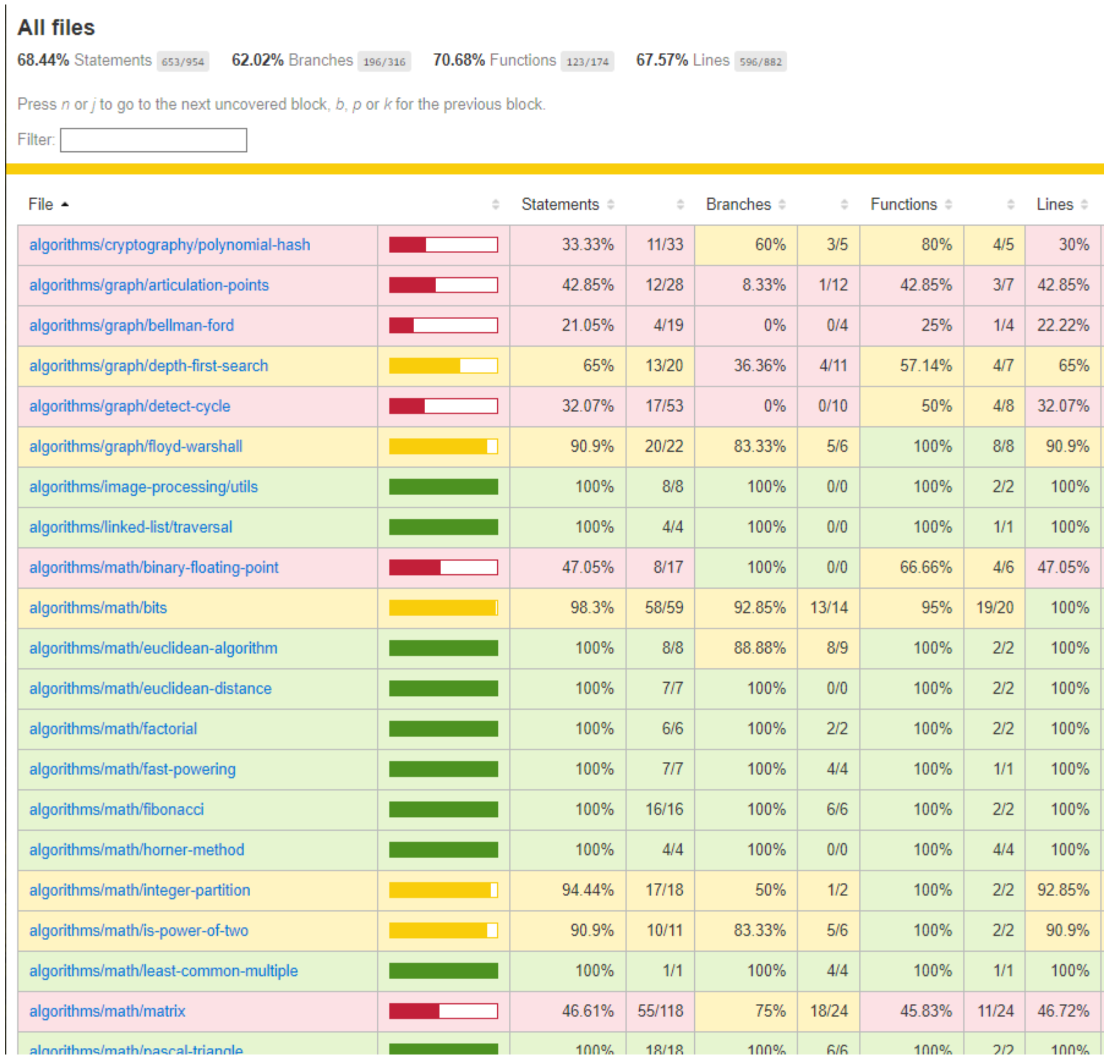
5. Results

This chapter will summarize the results of the experiment performed on the chosen frameworks. An analysis of the results will be done by answering RQ 3, 3.1, 3.2. In the case of Ponicode, the preliminary evaluation performed to justify inclusion into the practical experiment will be used to evaluate the framework. This chapter will answer RQ 3, RQ 3.1, and RQ 3.2.

5.1 Ponicode

During the preliminary evaluation of Ponicode on a set of algorithms written in Javascript [41], Ponicode managed to generate tests with an average 67.57% line coverage. Reviewing the coverage report in Figure 19 generated by these tests, it is apparent that the tool is only able to interpret certain types of code.

During the generation of the tests, several issues and limitations of the tool surfaced. Upon further testing the repository, Ponicode encountered 19 critical errors. The tool had difficulties instantiating classes, and did not understand some common-practice coding methodologies in JS, such as the use of methods.



All files

68.44% Statements 653/954 **62.02%** Branches 196/316 **70.68%** Functions 123/174 **67.57%** Lines 596/882

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File ▲		Statements		Branches		Functions		Lines
algorithms/cryptography/polynomial-hash		33.33%	11/33	60%	3/5	80%	4/5	30%
algorithms/graph/articulation-points		42.85%	12/28	8.33%	1/12	42.85%	3/7	42.85%
algorithms/graph/bellman-ford		21.05%	4/19	0%	0/4	25%	1/4	22.22%
algorithms/graph/depth-first-search		65%	13/20	36.36%	4/11	57.14%	4/7	65%
algorithms/graph/detect-cycle		32.07%	17/53	0%	0/10	50%	4/8	32.07%
algorithms/graph/floyd-warshall		90.9%	20/22	83.33%	5/6	100%	8/8	90.9%
algorithms/image-processing/utils		100%	8/8	100%	0/0	100%	2/2	100%
algorithms/linked-list/traversal		100%	4/4	100%	0/0	100%	1/1	100%
algorithms/math/binary-floating-point		47.05%	8/17	100%	0/0	66.66%	4/6	47.05%
algorithms/math/bits		98.3%	58/59	92.85%	13/14	95%	19/20	100%
algorithms/math/euclidean-algorithm		100%	8/8	88.88%	8/9	100%	2/2	100%
algorithms/math/euclidean-distance		100%	7/7	100%	0/0	100%	2/2	100%
algorithms/math/factorial		100%	6/6	100%	2/2	100%	2/2	100%
algorithms/math/fast-powering		100%	7/7	100%	4/4	100%	1/1	100%
algorithms/math/fibonacci		100%	16/16	100%	6/6	100%	2/2	100%
algorithms/math/horner-method		100%	4/4	100%	0/0	100%	4/4	100%
algorithms/math/integer-partition		94.44%	17/18	50%	1/2	100%	2/2	92.85%
algorithms/math/is-power-of-two		90.9%	10/11	83.33%	5/6	100%	2/2	90.9%
algorithms/math/least-common-multiple		100%	1/1	100%	4/4	100%	1/1	100%
algorithms/math/matrix		46.61%	55/118	75%	18/24	45.83%	11/24	46.72%
algorithms/math/pascal-triangle		100%	18/18	100%	6/6	100%	2/2	100%

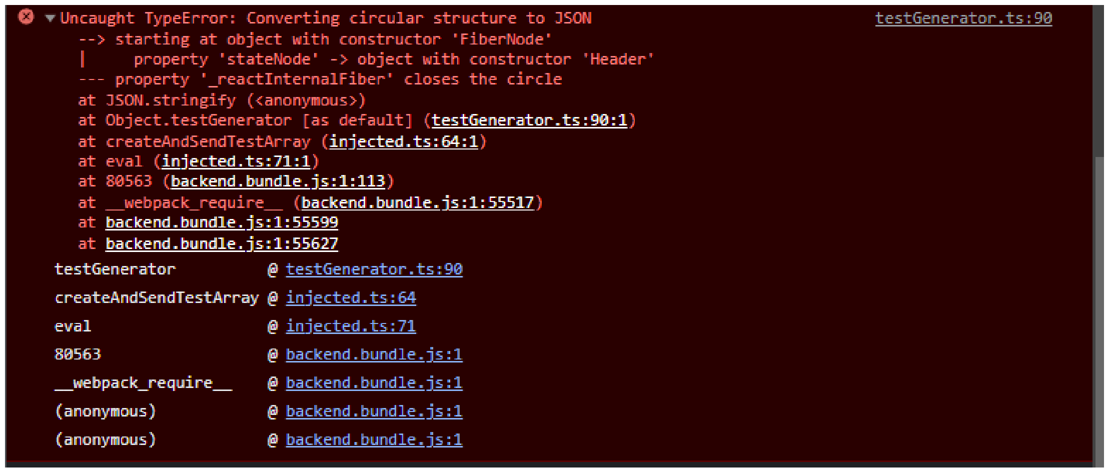
Figure 19. Ponicode initial results

5.2 Examin

Examin's test generation capabilities were tested on the applications listed in subsection 4.3.1. Applications. The tests were generated following Examin's instructions [24]. A detailed description of the test generation process is defined in subsection 4.1.2.

The configuration process for Examin was cumbersome, as all of the projects experienced dependency issues after installing the required dependencies for Examin.

Once configured, Examin generated the test suites almost instantly. However, the tool requires the developer to simulate every part of the application that is desired to generate unit tests for. The simulation entails navigating through all different branches of the program while it is running. Depending on the size and complexity of the application, this can slow down the generation process considerably. Examin failed to generate tests for two (english-class-minigame and react-frontend-dev-portfolio) out of the five chosen open-source applications. The tool encountered the same error in both cases, where it was unable to print the generated test case to the browser extension. A screenshot of the error encountered when generating tests for *English-class-minigame* is presented in Figure 20.



```
Uncaught TypeError: Converting circular structure to JSON                         testGenerator.ts:90
    --> starting at object with constructor 'FiberNode'
    |     property 'stateNode' -> object with constructor 'Header'
    --- property '_reactInternalFiber' closes the circle
    at JSON.stringify (<anonymous>)
    at Object.testGenerator [as default] (testGenerator.ts:90:1)
    at createAndSendTestArray (injected.ts:64:1)
    at eval (injected.ts:71:1)
    at 80563 (backend.bundle.js:1:113)
    at __webpack_require__ (backend.bundle.js:1:55517)
    at backend.bundle.js:1:55599
    at backend.bundle.js:1:55627
  testGenerator          @ testGenerator.ts:90
  createAndSendTestArray @ injected.ts:64
  eval                   @ injected.ts:71
  80563                  @ backend.bundle.js:1
  __webpack_require__    @ backend.bundle.js:1
  (anonymous)            @ backend.bundle.js:1
  (anonymous)            @ backend.bundle.js:1
```

Figure 20. Examin FiberNode error.

The error references an issue with the ‘FiberNode’ constructor, and as Examin uses React Fiber to generate the unit tests, this appears to be a core fault in the tool.

For the remaining three applications, Examin managed to achieve an average line coverage of 30.19%. The results of the generated test report are depicted in Figure 21. Each color in the graph represents an application under test, and the results are categorized by coverage type (statement-, branch-, function-, and line coverage).

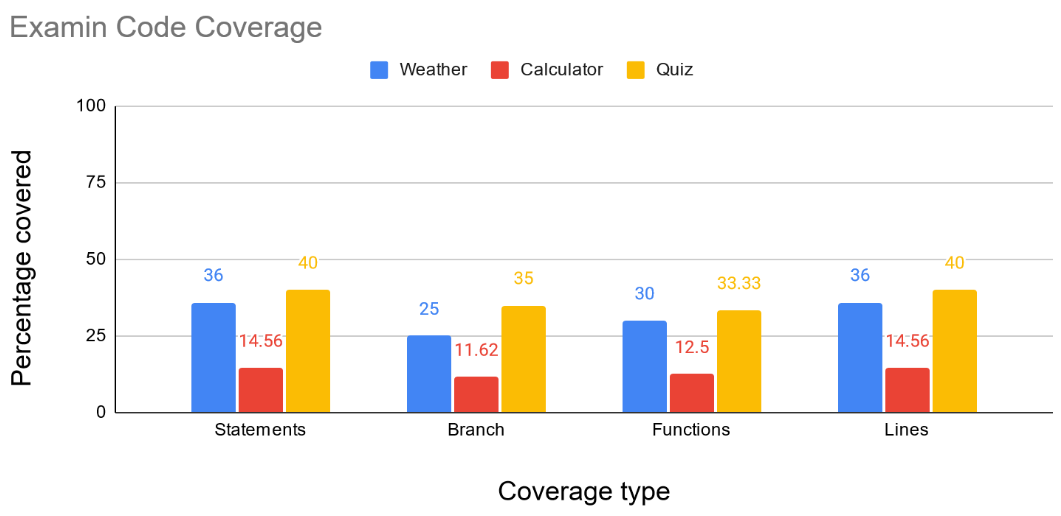
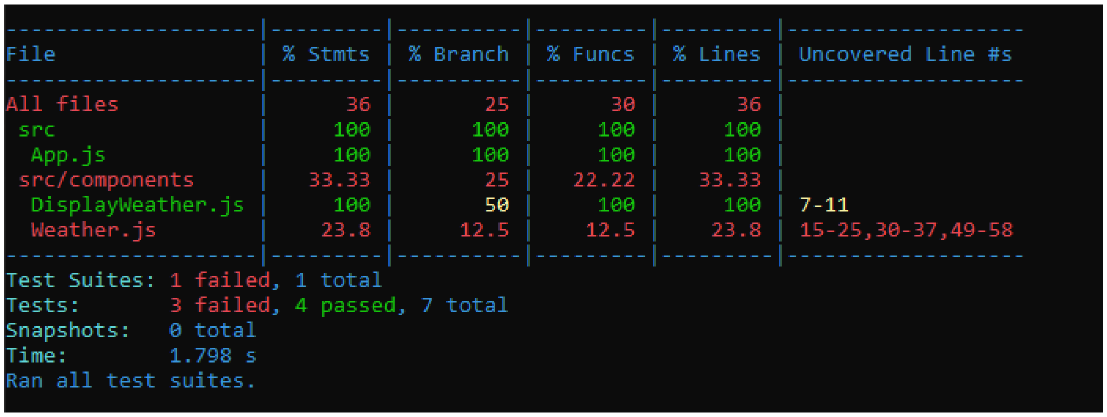


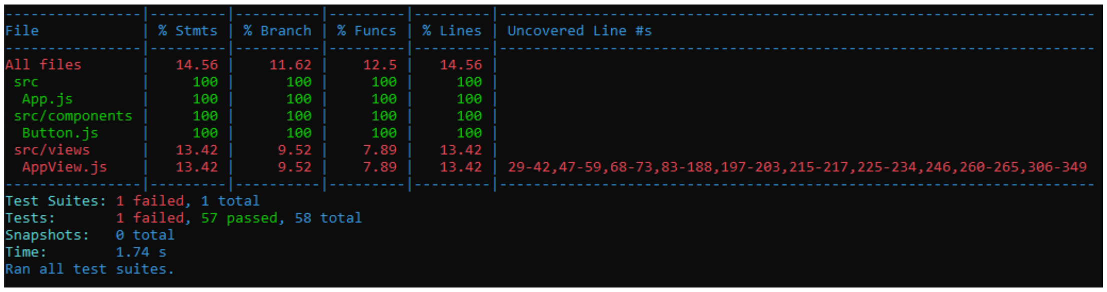
Figure 21. Examin Experiment Results Code Coverage.

As can be seen from the figure above, Examin did not manage to generate adequate tests for any of the systems under test. The worst results, after the two applications that Examin failed to generate any tests for, was the *calculator* application. For this application, Examin only managed to generate tests with a total line coverage of 14.56%. The best case was for the *quiz* application, with a total line coverage of 40%. It is also worth noting that the chosen applications had simple codebases that should be easier to interpret in comparison to real business applications. For further inspection, the generated reports are presented in Figures 22a), b), c).



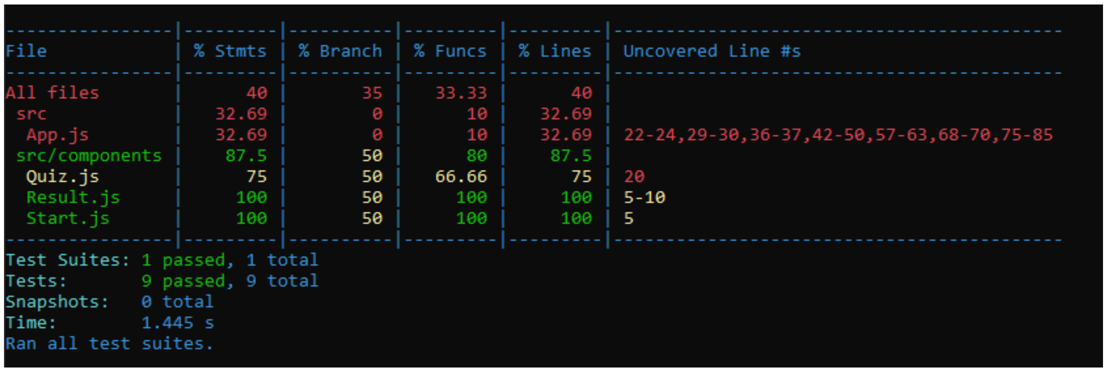
```
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |      36 |       25 |      30 |      36 |
 src                 |     100 |      100 |     100 |     100 |
  App.js             |     100 |      100 |     100 |     100 |
 src/components      |   33.33 |       25 |   22.22 |   33.33 |
  DisplayWeather.js  |     100 |       50 |     100 |     100 | 7-11
  Weather.js         |    23.8 |     12.5 |    12.5 |    23.8 | 15-25,30-37,49-58
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 failed, 1 total
Tests:       3 failed, 4 passed, 7 total
Snapshots:   0 total
Time:        1.798 s
Ran all test suites.
```

Figure 22a. Jest coverage report React-Weather-app



```
----------------|---------|----------|---------|---------|-------------------------------------------------------------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------------------------------------------------------------
All files       |   14.56 |    11.62 |    12.5 |   14.56 |
 src            |     100 |      100 |     100 |     100 |
  App.js        |     100 |      100 |     100 |     100 |
 src/components |     100 |      100 |     100 |     100 |
  Button.js     |     100 |      100 |     100 |     100 |
 src/views      |   13.42 |     9.52 |    7.89 |   13.42 |
  AppView.js    |   13.42 |     9.52 |    7.89 |   13.42 | 29-42,47-59,68-73,83-188,197-203,215-217,225-234,246,260-265,306-349
----------------|---------|----------|---------|---------|-------------------------------------------------------------------------
Test Suites: 1 failed, 1 total
Tests:       1 failed, 57 passed, 58 total
Snapshots:   0 total
Time:        1.74 s
Ran all test suites.
```

Figure 22b. Jest coverage report react-calculator



```
----------------|---------|----------|---------|---------|------------------------------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|------------------------------------------
All files       |      40 |       35 |   33.33 |      40 |
 src            |   32.69 |        0 |      10 |   32.69 |
  App.js        |   32.69 |        0 |      10 |   32.69 | 22-24,29-30,36-37,42-50,57-63,68-70,75-85
 src/components |    87.5 |       50 |      80 |    87.5 |
  Quiz.js       |      75 |       50 |   66.66 |      75 | 20
  Result.js     |     100 |       50 |     100 |     100 | 5-10
  Start.js      |     100 |       50 |     100 |     100 | 5
----------------|---------|----------|---------|---------|------------------------------------------
Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        1.445 s
Ran all test suites.
```

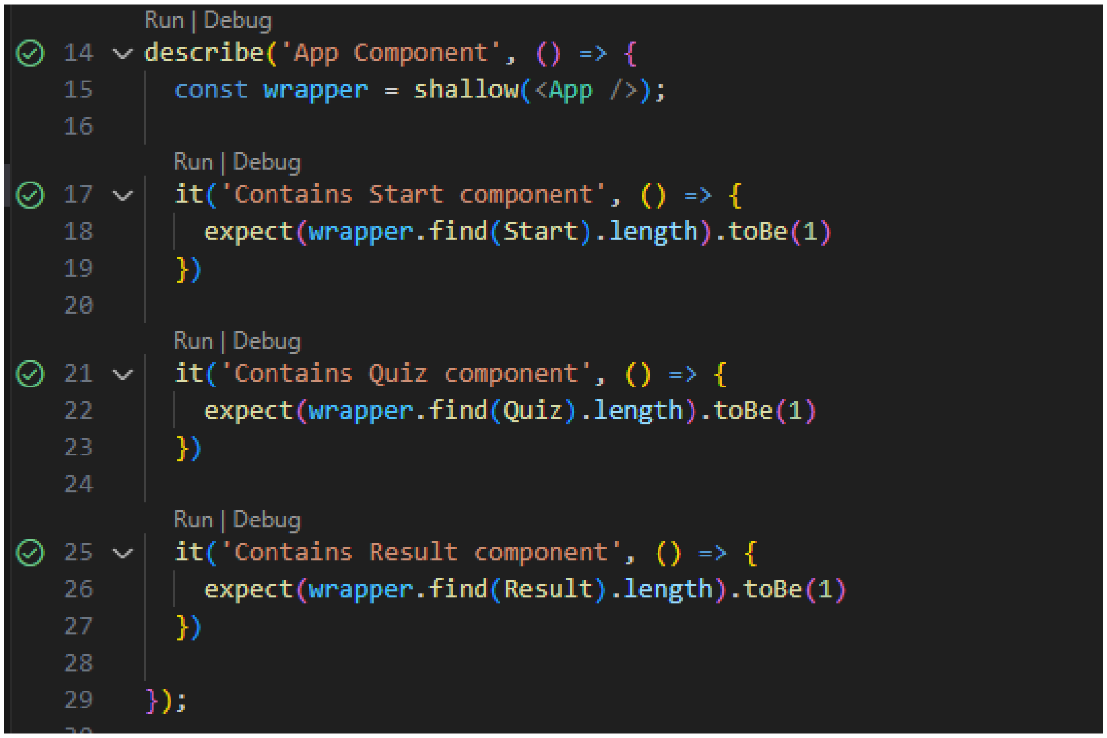
Figure 22c. Jest coverage report React-Quiz-App

Depending on the size and complexity of the application, this can slow down the generation process considerably. This also adds to the requirements of the test developer, as they might not have prior knowledge regarding the navigation of the application.

5.2.1 Test Quality

Considering the poor line coverage of the generated unit tests, it is unlikely they are of sufficient quality to be used as is in an actual test suite. To answer RQ 3.2, a manual review of the test quality is presented below.

For the *React-Quiz-App* the source code that determines the logic of the program is located in the *App.js* file. The rest of the relevant JS files, located in the *components* folder, affect the outlook of the program. All lines containing the essential logic of the program are not covered by the generated unit tests. Figure 23 contains the complete test generated by Examin for the *App.js* file.



```
Run | Debug
describe('App Component', () => {
  const wrapper = shallow(<App />);

  Run | Debug
  it('Contains Start component', () => {
    expect(wrapper.find(Start).length).toBe(1)
  })

  Run | Debug
  it('Contains Quiz component', () => {
    expect(wrapper.find(Quiz).length).toBe(1)
  })

  Run | Debug
  it('Contains Result component', () => {
    expect(wrapper.find(Result).length).toBe(1)
  })

});
```

Figure 23. Examin React-quiz-app generated unit test for App.js

As can be observed from the unit test, it clearly does not test any lines relevant to the logic of the program, only checks if the components are rendered into the application.

The generated tests for *react-calculator* and *React-Weather-app* suffer from the same discrepancies as for *react-quiz-app*. The generated tests contain almost no assertions for testing the logic of the program, and only check if components are rendered. Summarily, as the generated tests do not verify that the logic of the program functions as expected, the unit tests generated by Examin for these applications are of poor quality.

6. Discussion

This chapter will review the reliability of the experiments and surveys conducted in this thesis, as well as present an analysis of the validity of the aforementioned experiments. In addition to this, a discussion around the difficulties faced during the different practical parts of this thesis.

There were several difficulties faced while searching for tools and conducting the experiment. Finding an applicable unit test generator was significantly harder than expected. Out of the few applicable test generation tools, even fewer were usable for code written in newer versions of Javascript (including relevant frameworks and libraries such as React). Furthermore, installing the tools and conducting a preliminary evaluation often required solving several dependency issues. Although this hints at a poor availability of unit test generators for Javascript, it has negatively impacted the validity of the practical experiment. This is due to the fact that only one tool could be evaluated properly through the practical experiment. In addition to this, the tool (Examin) had a very specific use case as the program had to be executable with React v. 16.8 or higher.

7. Conclusion

Unit test generation offers an extremely helpful tool for reducing the development costs of a software project, but in the case of Javascript, the generators available display many limitations. The configuration process for many of them are extensive, and offer little to no help concerning the debugging process. If the tests generated by the tools are inadequate, they will require extensive review by the developers. Reviewing and expanding upon the tests will exhaust the same development hours the tools were meant to save. Although the unit test generators for Javascript suffer from limitations, they portray a promising start to be able to generate adequate test suites.

The conclusion of this thesis is that unit test generators for Javascript are not adequately advanced to be able to replace manual unit test development. As long as the test suites generated require extensive review and revision, manually writing the tests is the preferable option. For developers with a novice proficiency of unit testing, the process of reviewing and expanding upon a generated test suite that might contain odd syntax or irrelevant test cases could prove to be more difficult than personally writing the tests. Contrastingly, a developer with expert proficiency would likely write tests superior to the generated ones, without the need to exhaust development hours on configuring the tool for use.

Based on the results found both in the state of the art survey as well as the experiment, the conclusion of this thesis is that JavaScript unit test generators are not sufficiently advanced to replace manual test writing. They may be used to build templates for each unit test, but produce such unreliable logic that the developer cannot confirm whether the source code or the test itself is the issue. Regarding the future of JavaScript unit test generators, it may be possible for them to become sufficiently advanced to replace manual testing in some cases. This would require the test generators to be able to more accurately interpret the logic of the source code. Considering the dynamic and unpredictable nature of JavaScript, the slow improvement of JS unit test generators in the last 10 years, and the small size of the community, it is unlikely the test generators could achieve a sufficient level of success.

During the writing of this thesis, ChatGPT [53] has steadily gained popularity. ChatGPT-based unit test generators have As ChatGPT is a general AI, models that are specifically focused on programming should be assessed. These specialized models [50][51][52] could be a powerful tool for generating unit tests, and should be considered as a subject for future research into automatic unit test generation. For instance, CodiumAI [52] offers unit test generation support for Javascript and integrates directly into VSCode [44] and JetBrains IDEs [34].

8. Swedish summary

Enhetstestning är en viktig del av varje bra kodbas. Användningen av utförliga enhetstest främjar återanvändbarheten och pålitligheten av koden, så väl som förbättrar den övergripande kvaliteten på koden. Genom att använda enhetstester kan vi validera att programvaran utför operationer exakt så som den är avsedd att utföra dem. Att manuellt skriva enhetstester förbrukar värdefulla utvecklingstimmar som skulle kunna användas för att ytterligare förbättra eller utöka programvaran och är en av huvudorsakerna till att enhetstester förbises. Denna åsidosättning öppnar upp programvaran för bristfällig kod och kan i ett senare skede av utvecklingsfasen resultera i en kostsam omstrukturering av kodbasen.

Automatiska testgeneratorer tillåter utvecklaren att utelämna det manuella skapandet av enhetstest och direkt förse testförfattaren med de resurser som behövs för att köra testerna och validera systemet som testas. Generatorerna förlitar sig på källkod eller en mängd olika artefakter för att producera dessa enhetstest. Dessa artefakter kan bland annat vara resurser som klassdiagram, UML-modeller (Unified Model Language) och diverse designspecifikationer.

Testfallen som skrivs av enhetstestgeneratorerna skrivs separat för varje funktion eller kodkomponent. Dessa testfall kan vid behov förbättras manuellt av testförfattaren efter genereringen. Detta är viktigt för att säkerställa att den skapade testsviten kan testa alla delar av koden och erbjuder en acceptabel grad av feldetektering. Tidig identifiering av fel i kodbasen är väsentligt, eftersom problemet eller felet är aktuellt. Den tidiga identifieringen möjliggör att utvecklarna snabbare kan hitta en passlig lösning. Detta beror på att utvecklaren inte behöver återbekanta sig med kodbasen. Testning kan lätt åsidosättas under utvecklingsprocessen av ett programvaruprojekt. Ramverk som automatiskt kan generera enhetstester hjälper till att lindra kostnaden av mjukvarutestning.

Syftet med denna avhandling är att utvärdera prestandan av enhetstestgeneratorer för Javascript, både genom en litteraturundersökning av befintliga verktyg och ett praktiskt experiment.

Sökningen för verktygen (enhetstestgeneratorer för JavaScript) utfördes genom två olika kanaler. Dessa två kanaler var sökning av verktyg genom publicerad forskning och ett utförligt sök på nätet. Två olika kanaler användes för att inkludera verktyg av både akademiskt och icke-akademiskt ursprung i undersökningen. Följande verktyg hittades (Tabell 1):

Namn	**Metod**	**Senast uppdaterad**	**Licensering**	**Länk till verktyget**
Ponicode	AI, NLP processering av semantik och kodstruktur	2022	Kommersiell applikation	https://www.ponicode.com/developers
Examin	Responsriktad	2021	MIT licens	https://github.com/oslabs-beta/Examin
Artemis	Responsriktad	2017	GPL v3.0	https://github.com/cs-au-dk/Artemis
JSEFT	Evenemangsutrymmeutforskning	2014	Framgår ej	https://github.com/saltlab/JSeft
jest-test-gen	Obestämd	2022	MIT licens	https://github.com/egm0121/jest-test-gen

Tabell 1. Resultat av materialsökning för enhetstestgeneratorer.

Utav dessa verktyg, kunde verktygen som hittades genom akademiska kanaler (Artemis, JSEFT) utvärderas på basis av deras relaterade forskningspublikation. De andra verktygen (Ponicode, Examin, jest-test-gen) som hittades genom en webbsökning kunde inte utvärderas utan vidare undersökning. Verktygen med akademiskt ursprung verkade lovande, men installationen av dessa två verktyg visade sig vara ytterst krävande. När verktygen väl var installerade framgick det att de inte längre var funktionerliga. Dessa verktyg skulle kräva utförliga uppdateringar för att funktionera. jest-test-gen genererade endast basstrukturen för enhetstest och ansågs inte vara relevant för undersökningen.

Medan undersökningen av dessa verktyg pågick, köptes Ponicode av ett annat företag (CircleCI) som omedelbart förhindrade all åtkomst till verktyget. Som ett resultat av detta kunde endast en del av experimentet utföras på verktyget. Examin var således det enda återstående funktionerliga och tillgängliga verktyget att inkludera i experimentet.

Experimentet utfördes på diverse JavaScript applikationer med åtskiljande egenskaper. Verktygen användes för att generera enhetstest på basis av dessa applikationer, som sedan användes för att testa applikationerna. Målet med experimentet var att fastställa kvaliteten på enhetstesten genererade av verktygen. Detta mättes genom enhetstesternas förmåga att testa källkoden.

Ponicode lyckades generera lovande resultat på den enda applikationen som testades innan tillgängligheten till verktyget förnekades. Av 954 funktioner lyckades enhetstesten genererade av Ponicode täcka 653 stycken, 68.4%. Under experimentet uppkom tydliga begränsningar i Examins möjlighet att integreras i existerande programvaruprojekt. Integreringen i funktionella projekt lyckades endast för 1 av 5 fall. Utöver detta är verktyget begränsat till kod med en React version av 16.8 eller högre, som grovt förminskar verktygets genomsnittliga användbarhet.

Generering av enhetstest erbjuder ett användbart verktyg för att minska utvecklingskostnaderna i ett programvaruprojekt, men angående Javascript har de tillgängliga generatorerna många begränsningar. Konfigurationsprocessen för många är krävande och erbjuder obetydlig information när det gäller felsökningsprocessen. Om testerna som genereras av verktygen är otillräckliga kommer de att kräva omfattande granskning av utvecklarna. Även om enhetstestgeneratorerna för Javascript lider av begränsningar visar de en lovande startpunkt för att kunna generera utförliga testsviter. Slutsatsen av denna avhandling är att enhetstestgeneratorer för Javascript inte är tillräckligt avancerade för att kunna ersätta manuell enhetstestutveckling på en allmän nivå. Så länge som testsviterna som genereras kräver omfattande granskning och omstrukturering, är manuell skrivning av testerna det bättre alternativet.

9. References

1. *Jest*, Jest, accessed 29 October 2023,https://jestjs.io/
2. Ivanković, Marko, et al. "Code coverage at Google." *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019.
3. Gren, L. and Antinyan, V., 2017, August. On the relation between unit testing and code quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 52-56). IEEE.
4. Sneha, K. and Malle, G.M., 2017, August. Research on software testing techniques and software automation testing tools. In *2017 international conference on energy, communication, data analytics and soft computing (ICECDS)* (pp. 77-81). IEEE.
5. Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C., 2004. *The art of software testing* (Vol. 2). Chichester: John Wiley & Sons.
6. *Tic-tac-react*, Github, accessed 29 October 2023, https://github.com/rfce/tic-tac-react
7. *Javascript web docs*, Mozilla, accessed 29 October 2023, https://developer.mozilla.org/en-US/docs/Web/JavaScript
8. *Prototype based programming MDN web docs*, Mozilla, accessed 29 October 2023, https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming
9. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S., 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, *41*(5), pp.507-525.
10. S. Mirshokraie, A. Mesbah and K. Pattabiraman, "JSEFT: Automated Javascript Unit Test Generation," *2015 IEEE 8th International*

Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1-10, doi: 10.1109/ICST.2015.7102595.

11. Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," 2012 7th International Workshop on Automation of Software Test (AST), 2012, pp. 36-42, doi: 10.1109/IWAST.2012.6228988.
12. *State of JS 2021, front-end frameworks*, State of JS, accessed 29 October 2023, https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/
13. *State of JS 2021, back-end frameworks*, State of JS, accessed 29 October 2023, https://2021.stateofjs.com/en-US/libraries/back-end-frameworks/
14. *Jest Getting started*, Jest, accessed 29 October 2023, https://jestjs.io/docs/getting-started
15. *State of JS 2021, Testing libraries*, State of JS, accessed 29 October 2023, https://2021.stateofjs.com/en-US/libraries/testing
16. *React, Create a new app*, React, accessed 29 October 2023, https://reactjs.org/docs/create-a-new-react-app.html
17. *Stryker mutator docs*, Stryker, accessed 29 October 2023, https://stryker-mutator.io/docs/
18. *Stryker*, accessed 29 October 2023, https://stryker-mutator.io/
19. *Ponicode, AAA or Table-driven*, Internet Archive, accessed 29 October 2023, https://web.archive.org/web/20221207080828/https://www.ponicode.com/shift-left/arrange-act-assert-or-table-driven-testing
20. "IEEE/ISO/IEC International Standard - Software and systems engineering--Software testing--Part 4: Test techniques - Redline," in ISO/IEC/IEEE 29119-4:2021(E) - Redline , vol., no., pp.1-286, 28 Oct. 2021.

21. *Stackoverflow Survey 2022*, Stackoverflow, accessed 29 October 2023, https://survey.stackoverflow.co/2022/#most-popular-technologies-language
22. *W3techs client-side usage statistics*, W3Techs, accessed 29 October 2023, https://w3techs.com/technologies/details/cp-javascript
23. *Ponicode*, CircleCI, accessed 29 October 2023, https://circleci.com/blog/ponicode-and-circleci/
24. *Examin*, Examin, accessed 29 October 2023, https://www.examin.dev/
25. *JSEFT tool*, Github, accessed 29 October 2023, https://github.com/saltlab/JSeft
26. *Artemis tool*, Github, accessed 29 October 2023, https://github.com/cs-au-dk/Artemis
27. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song, "A Symbolic Execution Framework for JavaScript," 2010 IEEE Symposium on Security and Privacy, 2010, pp. 513-528, doi: 10.1109/SP.2010.38.
28. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of javascript web applications. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). Association for Computing Machinery, New York, NY, USA, 571–580. https://doi.org/10.1145/1985793.1985871
29. Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. ACM Comput. Surv. 50, 5, Article 66 (September 2018), 36 pages. https://doi.org/10.1145/3106739
30. *JS Test Gen*, Github, accessed 29 October 2023, https://js-test-gen.github.io/

31. *Google Scholar*, Google, accessed 29 October 2023, https://scholar.google.com/
32. *IEEE Xplore*, IEEE, accessed 29 October 2023, https://ieeexplore.ieee.org/Xplore/home.jsp
33. *Ponicode VSCode extension supported technologies*, Internet Archive, accessed 29 October 2023, https://web.archive.org/web/20230201223057/https://docs.ponicode.com/docs/vscode_extension/supported_technologies/index/
34. *Jetbrains*, Jetbrains, accessed 29 October 2023, https://www.jetbrains.com/
35. *Examin Chrome extension*, Google, accessed 29 October 2023, https://chrome.google.com/webstore/detail/examin/ihhopbmcfgkpjklemfdbhgingabdkcpe
36. *Examin Github Repository*, Github, accessed 29 October 2023, https://github.com/oslabs-beta/Examin
37. *React Developer Tools*, Google, accessed 29 October 2023, https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
38. *Rhino Interpreter*, Github, accessed 29 October 2023, https://github.com/mozilla/rhino
39. *React Magic*, Github, accessed 29 October 2023, https://github.com/Sylvenas/react-magic
40. *React Snakke*, Github, accessed 29 October 2023, https://github.com/diogomoretti/react-snakke
41. *JavaScript Algorithms*, Github, accessed 29 October 2023, https://github.com/trekhleb/javascript-algorithms
42. *JavaScript Snake*, Github, accessed 29 October 2023, https://github.com/patorjk/JavaScript-Snake
43. *Angular Start Application*, Github, accessed 29 October 2023, https://github.com/DeborahK/Angular-GettingStarted

44. *Visual Studio Code*, Visual Studio Code, accessed 29 October 2023, https://code.visualstudio.com/
45. *Algorithms Github*, Github, accessed 29 October 2023, https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms
46. *Npm*, npm, accessed 29 October 2023, https://www.npmjs.com/
47. *Jest CLI*, Jest, accessed 29 October 2023, https://jestjs.io/docs/cli
48. *PoniCode : My feedback and a mixed overall feeling about the tool*, Sylvian Leroy, accessed 29 October 2023, https://sylvainleroy.com/2020/07/23/ponicode-my-feedback-and-a-mixed-overall-feeling-about-the-tool/
49. *Jasmine Documentation*, Jasmine, accessed 29 October 2023, https://jasmine.github.io/
50. Yuan, Zhiqiang, et al. "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation." *arXiv preprint arXiv:2305.04207* (2023).
51. Xie, Zhuokui, et al. "ChatUniTest: a ChatGPT-based automated unit test generation tool." *arXiv preprint arXiv:2305.04764* (2023).
52. *Codium.ai*, accessed 29 October 2023, https://www.codium.ai/
53. *ChatGPT*, accessed 29 October 2023, https://openai.com/blog/chatgpt
54. G. Petrović, M. Ivanković, G. Fraser and R. Just, "Does Mutation Testing Improve Testing Practices?," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, ES, 2021, pp. 910-921, doi: 10.1109/ICSE43902.2021.00087.