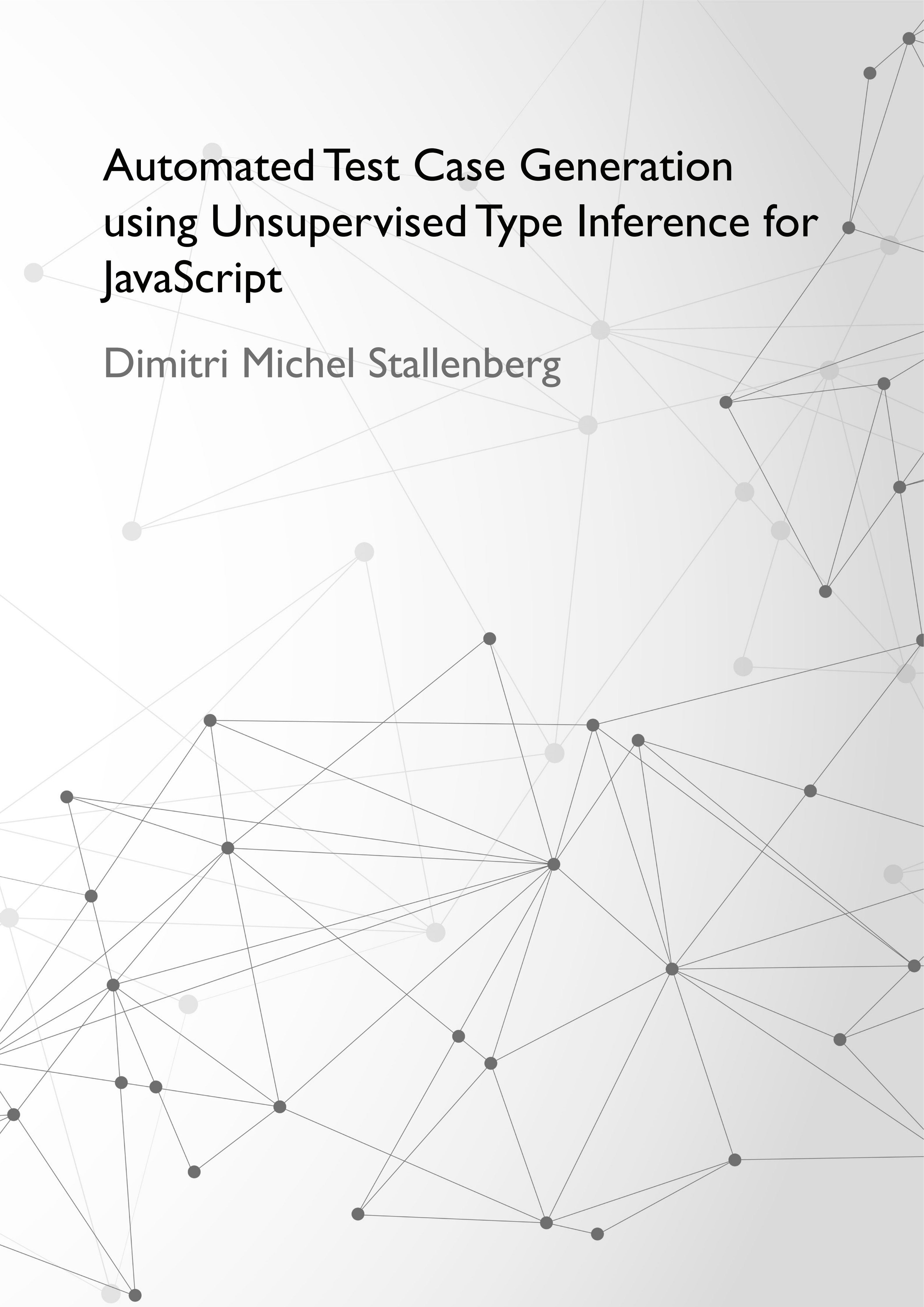
Automated Test Case Generation using Unsupervised Type Inference for JavaScript

Dimitri Michel Stallenberg



Automated Test Case Generation using Unsupervised Type Inference for JavaScript

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dimitri Michel Stallenberg
born in Purmerend, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Automated Test Case Generation using Unsupervised Type Inference for JavaScript

Abstract

Traditional software testing is a labor-intensive and expensive manual process. To mitigate the high cost of manual test case generation, researchers have developed various techniques for automated test case generation over the last few decades. These techniques make use of static type information to determine which data types should be used in new test cases. Dynamically typed languages like JavaScript do not provide type information. The lack of type information poses a new challenge for test case generation techniques.

In this thesis, we propose a novel unsupervised probabilistic type inference approach to infer data types in a test case generation context. The approach uses both static and dynamic type inference techniques. We implemented the approach in a novel tool called SYNTEST-JAVASCRIPT which is an extension of the SYNTEST-FRAMEWORK. We evaluate the performance of the approach compared to random type sampling with respect to branch coverage. The evaluation is done using a custom benchmark of 97 units under test.

Our results show that using statically inferred type achieves a statistically significant increase in 54% of the benchmark files compared to the baseline. The combination of using both statically and dynamically inferred types improves the approach slightly with a significant increase in 56% of the benchmark files compared to the baseline. Finally, the results show that the time consumed by the static and dynamic type inference is insignificant compared to the total time budget and is worthwhile given the performance boost type inference provides.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. A. Panichella, Faculty EEMCS, TU Delft
University Supervisor:	Ir. M. J. G. Olsthoorn, Faculty EEMCS, TU Delft
Committee Member:	Dr. L. Y. Chen, Faculty EEMCS, TU Delft

Preface

With this thesis, I conclude my five-year study at the Technical University of Delft. During this time I have had the privilege to work with the Software Engineering Research Group on multiple occasions.

Several people helped me get where I am, and I would like to thank them. First of all, I am incredibly grateful to my supervisors, Annibale Panichella and Mitchell Olsthoorn, who guided me and were always trying to help me elevate the quality of my work. Their knowledge and experience helped me overcome some of the challenges I faced during this nine-month journey. I want to thank Arie van Deursen and Lydia Chen for reading my work and for being part of my thesis committee. I also wish to thank the people close to me. First of all my parents and sister, who have always inspired me to do the best I can and never settle for less. Next, I want to thank my girlfriend Lisette, who supported me throughout the long nights of studying. Your confidence in my ability to write this thesis provided me with the motivation I required. Finally, I would like to thank my friends, who were always there for me whenever I needed a distraction.

Dimitri Michel Stallenberg
Delft, the Netherlands
October 3, 2022

Contents

Acronyms

AST Abstract Syntax Tree. 18, 19, 30–32, 35

AUC Area Under the Curve. 47, 48, 51

CC Cyclomatic Complexity. 42

CFG Control Flow Graph. 31, 37, 67

CLI Command Line Interface. 37, 38

DA Dynamic Analysis. xi, 2, 3, 8, 9, 15, 17, 27, 28, 46, 50, 52, 55–63, 66, 67, *Glossary:* Dynamic Analysis

DTL Dynamically Typed Language. 1–3, 7, 11, 12, 15, 17, 19, 29, 42, *Glossary:* Dynamically Typed Language

IEI Incorporating Execution Information. *Glossary:* Execution Information

IQR Inter-Quartile-Range. 47, 48, 50, 52, 55, 63

NLP Natural Language Processing. 8, 14, 15, 17

SA Static Analysis. xi, 2, 3, 8, 15, 17, 28, 46, 49–53, 55–63, 65–67, *Glossary:* Static Analysis

SBST Search-Based Software Testing. 6, 7

STL Statically Typed Language. 1, 7, *Glossary:* Statically Typed Language

SUT Software Under Test. 13, 18, 19, 23, 24, 26, 27, 39, 42, 47

UuT Unit under Test. 32, 34, 39, 42–48, 61, 62, 66, 67

List of Figures

List of Tables

Chapter 1

Introduction

Traditional software testing is a labor-intensive and expensive manual process [8, 46]. At the same time, testing is an integral part of the software engineering discipline. It provides the means to assure software quality [42, 47]. To mitigate the high cost of manual test case generation, researchers have developed various techniques for automated test case generation over the last few decades [8, 17, 43, 43]. Earlier studies have shown that automated testing effectively finds real-world faults while allowing developers to spend less time manually writing test cases [56, 57]. Search-based approaches in particular have been shown to effectively achieve higher code coverage [35] and have fewer smells compared to manually-written test cases [52], and detect unknown bugs [4, 6, 25]. Most research, however, is focused on Statically Typed Languages (STLs). Since Dynamically Typed Languages (DTLs) have gained significant popularity over the last decade [59], it is crucial to investigate whether the proposed techniques translate to DTLs.

In this chapter, we will go over the problem description, the research aim, the research significance, and the outline of the rest of the study.

1.1 Problem description

DTLs are prevalent among software developers [44, 59]. They do not require explicit type declarations, saving developers time that would otherwise be spent on developing a type system [29]. Examples of DTLs are Python and JavaScript, which are among the most popular languages used by developers around the world [2, 3]. However, most research on automated test generation revolves around STLs. These approaches rely on static type information to generate correctly typed inputs for functions and constructor calls within the test cases. In DTLs, developers do not have to provide such type information, allowing developers to quickly prototype functionalities without the need for a complex type system [29].

DTLs pose a new challenge for automated test case generation approaches. Since the type information is unavailable, the automatic testing approach has to guess which types to use for the inputs. Guessing the correct type in addition to generating a value that helps increasing structural coverage, dramatically increases the search space. Lukasczyk et al. have recently published their tool Pynguin for test case generation for Python. Their work shows that including type information is crucial for test generation. Their tool uses type annotations which are optional in Python. In JavaScript, such type annotations do not exist.

To address this problem, we have to infer the variable types, i.e., make accurate type predictions. There are several ways to infer types. For example, we can evaluate the usage of a variable to make assumptions about its type, i.e., investigate the variable's context.

Type inference has been studied previously to convert untyped code to typed code [32, 41, 44, 53, 63]. Listing 1.1 shows an example of such a conversion. In the example, the function *max* has two parameters and a return value. In Listing 1.1a, there is no information present regarding the types of the parameters or return value. In Listing 1.1b, there are type annotations directly next to the parameters and the function indicating that the *max* function requires two numbers as input and returns a number as well.

```
function max(a, b) {
    if (a < b) {
  return a
    } else {
  return b
    }
}
```

(a) Untyped

```
function max(a: number, b: number
    ): number {
    if (a < b) {
  return a
    } else {
  return b
    }
}
```

(b) Typed

Figure 1.1: Converting untyped code to typed code

Gao et al. showed that the lack of static types within JavaScript leads to bugs that could have been easily identified with a static type system [29]. Converting untyped code to typed code is thus a topic of interest because large-scale projects are more maintainable when written in typed languages [15, 29].

Earlier research often relies on SA and machine learning to infer types [32, 41, 44, 53, 63]. Training machine learning models often is expensive and requires retraining for unseen types. This is undesirable in search-based test case generation since the goal of test generation is to discover as many faults in the shortest amount of time. Retraining the machine learning model whenever new code is presented is thus far from ideal.

Additionally, SA means the code is only analyzed and not executed. We argue that adding DA techniques can provide additional information regarding the variables' types. We can then use this information to make more accurate type inferences.

This study proposes a novel approach to deal with DTLs during automated test case generation. The approach uses an unsupervised type inference strategy that combines SA with a novel DA technique. The DA technique consists of using Execution Information from the generated test cases. During the search process, we execute the generated test cases to evaluate their achieved coverage; thus, the Execution Information is available by default. This makes the DA technique uniquely appropriate for type inference in a test case generation context. Essentially, the Execution Information allows the evaluation of the statically inferred type's accuracy and adjust the inferred type accordingly. This information allows the approach to adjust the likelihood of the type options for a specific variable.

This study evaluates the impact of statically and dynamically inferred types on the test-case generating capabilities of the *SynTest* [1] project. The *SynTest* project is a framework for automated test case generation. To evaluate the novel approach, we realized the approach within a JavaScript-specific version of the *SynTest* tool. The tool is one of the contributions of this study, and we call it SYNTEST-JAVASCRIPT.

1.2 Research aim

This study aims to:

- Investigate the performance impact of type inference on test case generation for JavaScript.
- Measure the performance impact of using execution information to improve type inference accuracy for JavaScript.
- Evaluate whether the usage of type inference can be a time inexpensive process.

The performance impact can be measured by looking at the achieved structural coverage and the fault-detecting capabilities of the generated test cases. We evaluate whether the process can be time inexpensive by comparing the time the type inference process used to the total time used by the test case generation process. The general hypothesis is that the execution information of the generated tests allows for more accurate type inference; in addition, improving the inferred types improves the test-case generation capabilities. This reciprocity can create a feedback loop that, in the end, improves the test-case generation capabilities even further.

The specific research questions that follow from these research aims are listed in Chapter 6.

1.3 Research significance

The research aim is to improve search-based test case generation for DTLs by evaluating the importance of the availability of variable types. Below we itemize the main contributions of this study.

- **Research:** By answering the research questions posed in Chapter 6, it becomes clear what the impact of using (SA and/or DA) type inference during test case generation is. At the time of writing this study, the state-of-the-art test case generation research does not utilize DA type inference techniques. This study thus contributes by improving the test case generation capabilities for DTLs.

[1] https://www.syntest.org/

- **Tool:** To evaluate the proposed approach, the approach is implemented by extending the SYNTEST-FRAMEWORK with a JavaScript-specific version called SYNTEST-JAVASCRIPT. We published the tool as an open-source project on GitHub [2]. It allows developers to use the tool for their work and personal projects. Additionally, it serves as a basis for other researchers to build upon.

- **Benchmark:** In addition to the tool, we created a benchmark consisting of various JavaScript projects. We use the benchmark to evaluate the approach. The benchmark has been made publicly available on GitHub [3]. Future work can use this benchmark to compare the different test case generators.

- **Replication package:** We published a replication package on GitHub[4] allowing anyone to replicate the results from this study. The package ensures the validity and reliability of the research.

1.4 Outline

The remainder of this study is organized as follows. First, in Chapter 2 we provide essential background information. Next, in Chapter 3, we discuss related work and identify the research gap. Chapter 4 describes the novel unsupervised type inference for test case generation approach. After that, Chapter 5 presents the implementation details of the conceived tool that realizes the approach. Chapter 6 discusses the experimental setup used to evaluate the approach. Chapter 7 summarizes the obtained results from the empirical evaluation. Finally, Chapter 8 concludes the results and gives suggestions regarding future work.

[2]https://github.com/syntest-framework/syntest-javascript
[3]https://github.com/syntest-framework/syntest-javascript-benchmark
[4]https://github.com/dstallenberg/syntest-javascript-thesis-replication-package

Chapter 2

Background

In Chapter 1, we went over the problem description and research aim of this study. This chapter explains the core background concepts related to the subject. We will discuss software testing, automated software testing, dynamically typed languages, and finally, type inference.

2.1 Software Testing

Software Testing is a vital aspect of software engineering [8]. It allows developers to ensure the software performs its intended purpose [42, 47]. Testing comes in many forms [60]. The most basic type of testing is unit-level testing. Unit-level testing aims to verify the lowest functionality level of the software by isolating the functionalities. Integration-level testing essentially aims to test several functionalities in combination to assess whether they work together correctly. System-level testing aims to verify the software as a whole. To ensure that the software keeps working in the future, developers create test suites, which can be run whenever necessary to check the code base. Continuously running such test suites after modifying the code is called regression testing.

2.1.1 Code Coverage

Determining whether a test suite is sufficient enough to detect faults/bugs can be challenging. As guidance, software testers often use code coverage metrics [31]. Code coverage allows developers to see what parts of the code are reached when a test is run. In a previous study, Kochhar et al. found a moderate to strong correlation between the effectiveness of the test suite and code coverage [37]. Here the effectiveness of the test suite is determined by its ability to find real-world bugs.

There are several types of code coverage metrics. Each focuses on different aspects of the code. For example, statement coverage determines what statements are covered during certain software execution. However, code is not simply a piece of text which is executed linearly statement after statement. Instead, code can be viewed as a graph, where each conditional creates a `true` and a `false` ”branch”. Branch coverage gives insight into which branches within the software graph are covered. There is a subsumption relation between these two, that is, 100% branch coverage automatically means 100% statement coverage.

For this reason, it is wise to use the coverage metric highest in the subsumption hierarchy. However, the highest metric is called path coverage. Path coverage measures how many of the possible paths through the ”code graph” are covered. The number of possible code paths increases exponentially with the size of the program [65]. This makes it extremely expensive to cover all possible code paths and questionable if it is worthwhile. For this reason, branch coverage is often used by software developers.

2.2 Automated Test Case Generation

Current industrial practices rely on manually written tests, which requires a large amount of time and effort from software developers [8, 17, 43]. Hence, researchers have proposed automated test case generation solutions since the 1970s [19]. This has resulted in various techniques and tools that allow tests to be automatically generated using software. These techniques include symbolic execution [14, 36], concolic execution [30], random testing [18], and search-based software testing [43]. These techniques will be explained in more detail in the next three subsections.

Random Testing & Fuzzing

Random testing is the most straightforward technique. Tools that use this technique randomly generate inputs for functions/programs. These function calls are then executed and checked for crashes/exceptions. Random testing often does not involve creating an actual test case, instead, the inputs which induce crashes/exceptions are stored to be later shown to the end user. Since random testing is purely random, it does not have any guidance in achieving high coverage or high-quality test cases.

Symbolic Execution & Concolic Testing

Symbolic execution in a software testing context aims to discover the requirements for reaching a certain point in the software. To be more specific, what input combination leads to the execution of certain parts of the software. The symbolic execution is done by using symbolic values for inputs. This way, we can create constraints for reaching each conditional branch in terms of those symbols. Concolic testing is very similar to symbolic execution. However, it uses concrete values instead of symbols to generate concrete inputs to maximize code coverage.

Search-based Software Testing

Search-Based Software Testing (SBST) techniques often have a random component similar to random testing techniques. However, SBST techniques also incorporate objectives. Often these objectives are branch objectives. Branch objectives are essentially uncovered branches in the code. The goal of the algorithm is to cover these branch objectives. We can calculate the approach level and branch distance to guide the algorithm. The approach level equals the number of branches between the closest covered branch and the branch objective. In Chapter 5 we show an example of this. The branch distance is equal to the variable values evaluated at the conditional expression of the objective branch.

SBST techniques have been successfully used in literature to automatically generate test cases at the different testing levels [43], such as unit [26], integration [22], and system-level testing [11].

Various search algorithms have been proposed for the purpose of generating test cases. A few examples include: whole suite [27], MIO [10], MOSA [50], and DynaMOSA [51]. Recent studies have shown that DynaMOSA is more effective and efficient than other state-of-the-art genetic algorithms for unit-level test generation of Java [16] and Python [40] programs.

2.3 Dynamically Typed Languages

In STLs, type checking occurs at compile time. Conversely, type checking occurs during run time for DTLs. This means that the correctness of types is not verified upfront. Instead, the types are checked only when the program is executing. Common examples of DTLs include JavaScript, Python, Ruby, PHP, Lua, and Perl. Python and JavaScript have been among the most popular programming languages for a number of years [2, 3].

Figure 1.1 in Chapter 1 shows the difference between a DTL and a STL. The STL requires a developer to explicitly state the data type of each variable when the variable is declared. Function signatures are also required to state their return type explicitly. In DTLs this is not required.

The popularity of DTLs can be attributed to the flexibility DTLs offer developers [59]. This type-flexibility allows developers to quickly write pieces of code without needing to maintain a complex type system.

One of the downsides of DTLs is that the type errors are only detected during execution, making DTLs error-prone [29].

JavaScript

As mentioned, JavaScript and Python are examples of DTLs. However, in contrast to Python, JavaScript is a weakly-typed language. In a weakly-typed language, variables are not bound to a specific type. In other words, weakly-typed languages make conversions between unrelated types implicitly. In a weakly-typed language, it is allowed to, for example, add variables of different types together. In a strongly-typed language, this would not be allowed. For example, the expression $c = "abc" + 1$ is not allowed in Python as we are mixing two different types in one expression. In JavaScript, this is perfectly fine since it is weakly-typed.

2.4 Type Inference

Type inference is the ability to deduce the type of an expression automatically. There are many techniques to perform type inference. These techniques can be categorized as Static Type Inference and Dynamic Type Inference.

2.4.1 Static Type Inference

Static Type Inference makes use of SA to extract information from source code. In literature, three techniques are mainly used. These techniques are Logical Constraint Processing, Contextual Hint Processing, and Natural Language Processing (NLP).

Logical constraints

In essence, logical constraints are expressions within the source code that constrain the types of the involved variables. An example of such a logical constraint is an assignment. If, for example, a variable is assigned a boolean literal, we know that the variable is a boolean at that point in the code.

Contextual hints

Among developers, it is common to contain the variable's type in the variable's name, i.e., if a variable of the type "tree" contains a family tree, the variable might be named "familyTree". In contrast to NLP, the association between variable types and variable names is not learned. Instead, they are based on the occurrence of the name of the type in the name of the variable. This is a contextual hint.

Another contextual hint is the usage of the variable. The variables might be involved in several operations and expressions. These operations and expressions can provide an insight into what the type of the variable might be.

Natural Language Processing

NLP often encompasses learning to associate the entire context around a variable with a variable's type. For NLP, the variable's type does not have to be specified in the variable's context. Instead, the type is inferred based on what the NLP model learned to associate with the variable's context. For example, the word "count" is often used in variable names which are of the numeric type. The NLP model will learn to associate "count" with the numeric type. Another example is processing the comments left by developers to learn about a variable's type. For this study, the NLP method has not been considered because of reasons described in Chapter 4.

2.4.2 Dynamic Type Inference

Besides Static Type Inference, there also exists Dynamic Type Inference. Dynamic Type Inference uses DA of the execution of a program to extract information on the variable's types.

DA can be used to, for example, infer static types using information gathered from dynamic runs [7]. In another study DA is used for type specialization [28].

Chapter 3

Related Work

In this study, we focus on type inference for test case generation. Chapter 2 provided background on automated test case generation, DTLs, and type inference. This chapter reviews previous work on test case generation for DTLs and type inference to establish the research gap on which this study will focus.

3.1 Test Case Generation for Dynamically Typed Languages

There exist several automated test case generation tools that deal with DTLs. This section highlights some of these tools to discuss their aim, results, and shortcomings.

For JavaScript, the test case generation approaches can be classified by the input space on which they operate. There are two such spaces: the *event space* and the *value space* [54]. The event space concerns the order of events in JavaScript. The event space mostly revolved around user interfaces, for example, the order in which certain buttons are clicked. On the other hand, the value space is about the values that are used to execute certain functionalities. Both these classifications focus on client-side JavaScript. Even more so, most of the literature on JavaScript test case generation is focused on client-side applications. However, unit testing for server-side JavaScript is similar to value-space-oriented approaches.

Artemis & SymJS

Artemis is one of the first test case generation tools for JavaScript [13]. It uses a feedback-directed random testing algorithm to test JavaScript web applications. *Artemis* operates on both the event space and the value space. It does not use any form of type inference.

SymJS is automatic symbolic testing framework for client-side JavaScript web applications [39]. It is based on *Artemis* [13] and aims to improve the value-space exploration using concolic execution. *SymJS* only accounts for two types: *numbers* and *strings*. Tanida et al. [58] proposed an improvement by creating symbolic inputs based on manual type annotations. In other words, they use the type annotations developers can define in the documentation of the JavaScript function.

Although using documentation to extract types is a solid solution to the problem, it requires that the documentation is written in a specific format that includes the type of the required variables. It also requires the type documentation to be very specific. For example, if a function requires a specific type of object, it is not very helpful to list the type of the argument as “Object”. Instead, it should specify the exact properties of the object.

JSeft

JSeft is a JavaScript test generation tool focused at creating event-based tests and function-level unit tests [45]. In their paper, Mirshokraie et al. focus on web application testing and oracle generation. *JSeft* starts by creating a state-flow graph (SFG). This SFG is then used to generate event-based test cases. Based on the elements of the web application, the JavaScript function states are extracted to generate function-level unit tests. Since *JSeft* works on the Document Object Model (DOM) level from which it directly extracts function calls with arguments, it does not have to bother with inferring types. It can simply extract the argument types from the function calls it finds. Although this makes *JSeft* less relevant to this study, it does provide an interesting idea that can be used during the type inference. We can extract hints on the types of arguments to use by looking at the calls made to a specific function.

Jalangi

Jalangi was first introduced by Sen et al. [55] as a framework for the dynamic analysis of JavaScript. Although it is not a test case generation tool, it performs concolic testing. Initially, the types of input values were chosen based on their immediate use in the branch conditions used by concolic testing. In other words, the broader context in which the variables are used was not considered. Dhok et al. [23] proposed type-awareness to improve the approach. In this context, type-awareness meant that the concolic testing algorithm handled the type constraints separately from the branch constraints. The addition of type awareness significantly reduced the number of redundant inputs, i.e., inputs that do not achieve new code coverage while using other argument types.

Both search-based testing and concolic testing suffer from scaling issues when the number of possible input combinations is large. The results from Dhok et al. indicate that being aware of type constraints reduces the number of possible input combinations. Access to the arguments’ type information could also prove beneficial for search-based testing techniques.

Pynquin

Pynguin is an automated unit test generation framework for Python introduced in a paper by Lukasczyk et al. [40]. The paper aims to verify that test generation is also effective for DTLs. Additionally, the goal is to empirically evaluate the impact of type information on *Pynguin*’s test generation capabilities. *Pynguin* uses a search-based technique and takes a whole-suite test-generation approach. Just like *SymJS*, it uses type annotations added by developers. The paper results show that search-based test generation is effective for most of the benchmarks when working with DTLs. The authors also conclude that incorporating type information allows *Pynguin* to cover larger parts of the code. However, it is said that

SUTs which use more complex types benefit more than SUTs which only utilize simple types.

Summary

In summary, most automated testing tools for JavaScript are focused on client-side applications instead of Node.js [1] server-side applications [13, 39, 45]. The event-space-oriented approaches are not applicable to server-side applications. However, value-space-oriented techniques can be applied to automated testing tools for server-side applications. Tools that do focus on JavaScript server-side applications often only create system-level test cases by operating on s [21, 33, 61]. The tools and approaches discussed either use only a few type hints or ignore argument types completely. The used typed hints mostly consist of type annotations provided by the developer.

3.2 Type inference

Type inference has been a research topic for quite some time now [9, 34]. This section will briefly cover the field's history and discuss the state-of-the-art techniques for type inference.

JSNice

One of the earliest studies on inferring types for JavaScript was published by Raychev et al. in 2015 [53]. The study introduces a scalable prediction engine called *JSNice*. The engine aims to predict the names of identifiers and the type annotations of variables. *JSNice* operates by parsing the input program and converting it to a dependency network relating unknown with known properties. Based on training data, this dependency network is then used to infer types and names of the unknown properties.

The study is focused on the general problem of inferring program properties and thus serves as the foundation for further type inference research.

One of the shortcomings of *JSNice* is that it can only predict basic JavaScript types. This means it cannot assert that a particular variable is of a user-defined type. According to Mir et al. another problem with *JSNice* is that it is unable to consider a wide context within a program or function, i.e., it is unable to differentiate variables in different contexts with the same name.

JSNice validates the generated code by running the Google Closure Compiler, a tool that type checks JavaScript with optional type annotations. However, this is not foolproof. The Closure Compiler can only check if the given type is possible, not if it is the type that the developer intended.

Deeptyper

DeepTyper is a deep learning model created to provide type suggestions for JavaScript [32]. It consists of GRUs (Gated Recurrent Units), a type of neural network. *DeepTyper* is trained

[1] https://nodejs.org/

using a large set of TypeScript projects from GitHub. These projects are used to learn variable types in certain contexts.

In their paper Hellendoorn et al. states that *DeepTyper* performs similar to *JSNice*. It is also noted that combining the two tools provides significantly better results. In this hybrid mode, *JSNice* is consulted first on a certain type. If *JSNice* does not have an answer *DeepTyper* is used. The author attributes the results of the hybrid mode to the two approaches being remarkably complementary. *JSNice* is almost always correct about a predicted type, but it often is uncertain and does not provide a type. *DeepTyper* on the other hand, does make more predictions, but the predictions are incorrect more often. This notion shows that when the methods used by *JSNice* provide an uncertain result, some additional prediction mechanisms can help the type inference.

DeepTyper focusses on the 11000 most commonly used types. This restricts the tool from inferring user-defined types, i.e., types defined within the source code for which we are trying to infer types. To enable *DeepTyper* to learn the user-defined types, *DeepTyper* would have to be retrained using the source code where those types are defined.

NL2Type

NL2Type is like *DeepTyper* a deep learning model [41]. *Nl2Type* consists of a LSTM (Long Short Term Memory) network, a type of neural network. In contrast to *DeepTyper*, *NL2Type* uses comments in combination with code as input to predict types. The authors Malik et al. have shown that it significantly outperformed both *JSNice* and *DeepTyper*. Similar to the non-hybrid *DeepTyper* approach, *NL2Type* is a NLP approach.

Just as *DeepTyper*, *NL2Type* is only able to predict a defined set of types. *NL2Type* works optimally for 1000 types.

LambdaNet

LambdaNet uses a combination of logical constraints and context hints [63]. Similar to *JSNice* it uses the hints and logical constraints to build a dependency graph. However, in contrast to *JSNice*, it uses a GNN (Graph Neural Network) to infer types. The authors of *LambdaNet* note that their approach outperforms *DeepTyper* significantly. *LambdaNet* can predict 100 different types.

Type4Py

At the time of writing this study, *Type4Py* is among the state-of-the-art regarding type inference. *Type4Py* was introduced by Mir et al.. It focuses on *Python* and solves the limited type vocabulary problem by employing a deep similarity learning strategy. *Type4Py* outperforms other state-of-the-art approaches such as *Typilus* which also uses deep similarity learning [5]. *Type4Py* considers contextual and natural type hints by feeding identifiers, code context, and visible type hints as features from which it learns type associations.

Although *Type4Py* can handle an unlimited amount of types, i.e., an infinitely large type vocabulary, it cannot make predictions for types that lie beyond its pre-defined type clusters.

Summary

Most related work on type inference uses machine learning to perform NLP. The approaches are unable to infer types that are defined in the source code by the developer. In other words, their type vocabulary is limited to the provided training data. Additionally, the described techniques focus on SA only.

3.3 Research Gap

Given the literature, we found that test case generation for JavaScript is mainly focused on client-side applications. To the best of our knowledge, we can conclude that the current automated test case generation approaches for DTLs do not use full-fledged type inference techniques.

Regarding the type inference techniques, most suffer from a fixed-size type vocabulary. Without retraining the NLP models, the techniques cannot predict user-defined types. Retraining is expensive and thus unwanted during automated test case generation.

In this study, we propose a novel unsupervised approach to dealing with DTLs during automated test case generation. This approach will use several of the discussed SA type inference techniques in combination with a novel DA technique unique to test case generation. The novel approach is unsupervised, does not require any training, and has an infinite-size type vocabulary.

Chapter 4

Unsupervised Type Inference for Test Case Generation

Chapter 2 explained essential background information. Chapter 3 discussed related work. This chapter will explain how different SA and DA techniques are integrated into the test case generation for DTLs approach.

This study evaluates the performance impact of using Static Type Inference for test case generation for JavaScript. In addition, the impact of using both Static Type Inference and Dynamic Type Inference is measured. The general idea behind this is that if we can improve the accuracy of the inferred types by incorporating Execution Information from the search process, i.e., perform DA, the quality of the generated test cases will also improve. That is, the approach will achieve higher structural coverage because it has more information about the types of arguments. To evaluate this hypothesis, we propose a novel approach that integrates unsupervised probabilistic type inference into the search-based test case generation process to infer required type information.

4.1 Unsupervised Probabilistic Type Inference

The approach builds upon previous type inference studies. It relies on SA techniques to perform the initial type inference, similar to the studies discussed in Chapter 3. The techniques used are logic-based inference combined with context-based inference. Ideally, test case generation approaches dedicate most of their time budget to the search for high-coverage test cases. Using machine learning techniques is thus not favorable since training such models is expensive. Since NLP techniques generally require some form of machine learning, we have not considered using such techniques.

In contrast to most related work, the novel approach also uses DA type inference techniques. It considers information about the correctness of the inferred types by analyzing Execution Information of generated test cases. We use this information is then used to improve the probabilistic type models created by the SA type inference techniques.

Unsupervised probabilistic type inference involves four phases, as shown in Figure 4.1.

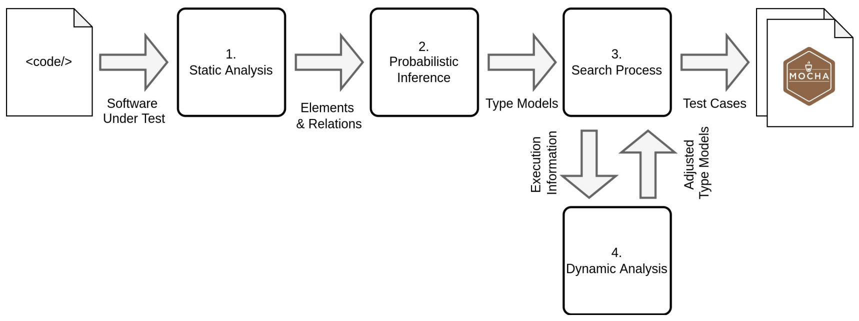


Figure 4.1: Unsupervised Probabilistic Type Inference for Test Case Generation Flowchart

The first phase consists of static analysis, with regards to type inference, of the Software Under Test (SUT) and results in a set of extracted elements and relations. The second phase uses the extracted elements and relations to create probabilistic type models for each element. The third phase consists of the search process. During the search process, the type models are used to sample inputs that result in test cases that cover branch objectives. The generated test cases are executed during the search process. From the execution, we extract information about the correctness of the types used in the test case. The fourth phase uses the execution information to adjust the probabilistic type models such that they become more accurate than before. The third and fourth phase keep interacting with each other for as long as the budget allows or all branch objectives are covered. Since the goal is to find a set of test cases that covers most branches and exposes bugs, the final set of test cases is the output of the process.

In the following sections, we will discuss these phases in more detail. The first section will go over the static analysis phase. In the second section, the probabilistic type inference phase is discussed. The third section describes the relevant parts of the search process. The fourth section covers the last phase of the approach. To answer the research questions stated in Chapter 6 we created five approach variants. The final section describes these different variants.

4.2 Static Analysis

The first phase consists of inspecting the SUT and its dependencies to gather information that can be used to infer types. The gathered information consists of:

- Elements: identifiers and literals.
- Relations: expressions and operations involving one or more elements.
- User-defined types: Type descriptions based on classes, interfaces, prototyped functions, or object initializers.

To extract the required information, all code is converted to Abstract Syntax Tree (AST)s. We traverse these ASTs to find the relations and their involved elements. Additionally, all user-defined objects are extracted from the AST to create object type descriptions.

4.2.1 Elements

Elements represent the parts of the SUT for which we want to infer the types. An element can be a variable identifier or literal. Identifiers are named references to variables, functions, and properties. Literals are raw values that can be assigned to variables or constants. Since literals directly represent a type, we do not require inference. However, as described in Chapter 2 in DTLs, identifiers do not have explicit types. To find out more about the type of an identifier, we can use the context of the identifier, i.e., the relations.

4.2.2 Relations

Relations are expressions and operations involving one or more elements. They describe how the elements are used and how they relate to other elements. Relations can tell us more about the elements' types. As an example relation, assume variable L is assigned R ($L = R$). If R is a boolean literal, we can infer that L must also be boolean at that point in the code.

The relations are extracted from the AST and converted to a format that allows easy identification. Figure 4.2a shows a *smaller than* relation between variable a and literal 6 on line 2. The relation is converted and recorded as $[L < R, a, 6]$ as shown in Figure 4.2b. A full list of the possible relations is shown in Table 4.2. The table specifies the operator category, the operator name, and the relation format for all operators available in JavaScript. Most of these operations are taken from the MDN web documentation by Mozilla [1].

```
1  function example (a) {
2      if (a < 6) {
3          return 0
4      }
5      return a
6  }
7
8  example(5)
```

(a) Code

1. $[L\_R, example, a]$
2. $[L < R, a, 6]$
3. $[L \rightarrow R, example, 0]$
4. $[L \rightarrow R, example, a]$
5. $[L(R), example, 5]$

(b) Extracted Relations

Figure 4.2: Example of relation extraction from code

Besides the obvious relations like a binary operation or assignment expression, Table 4.2 also contains relations like function arguments, function returns, and function calls. Recording such additional relations gives extra insight into the possible types of the elements. For example, Figure 4.2a shows that the example function has a parameter a. Assume we want to infer the type a. Further in the code snippet, line 8 shows that the example function is called using 5 as an argument, which is a numeric literal. This indicates that parameter a might be required to be numeric. However, since JavaScript is a DTL, we cannot say this with certainty, i.e., the example function might allow a to be both numeric and string, or any other type.

Often code is not as straightforward as the example given in Figure 4.2a. Code regularly contains nested relations. Nested relations are relations where the involved elements are

Operator Category	**Operator Name**	**Relation**
Primary Expressions	This	this.L
	Define Function	function L
	Define Class	class L
	Define Generator Function	function* L
	Pause and Resume Generator Function	yield L
	Delegate to another Generator Function	yield* L
	Define Async Function	async function L
	Wait for Promise Resolution/Rejection	await
	Array initializer	[L]
	Object initializer	{L:R}
	Regular Expression	/L/
	Grouping Operator	(L, R)
Left-hand-side Expressions	Property Accessor	L.R
	New	new L()
	Spread	...L
Increment/Decrement	PostFix Increment	L++
	PostFix Decrement	L- -
	PreFix Increment	++L
	PreFix Increment	- -L
Unary	Delete	delete L
	Void	void L
	Type Of	typeof L
	Unary Plus	+L
	Unary Negation	-L
	Bitwise NOT	∼L
	Logical NOT	!L
Arithmetic	Addition	L+R
	Subtraction	L-R
	Division	L/R
	Multiplication	L*R
	Remainder	L%R
	Exponentiation	L**R
Relational	In	L in R
	Instance Of	L instanceof R
	Less than	L<R
	Greater than	L>R
	Less or Equal	L<=R
	Greater or Equal	L>=R

Table 4.1: JavaScript Relations

Operator Category	**Operator Name**	**Relation**
Equality	Equality	L==R
	Inequality	L!=R
	Strict Equality	L===R
	Strict Inequality	L!==R
Bitwise Shift	Left Shift	L<<R
	Right Shift	L>>R
	Unsigned Right Shift	L>>>R
Binary Bitwise	Bitwise AND	L&R
	Bitwise OR	L|R
	Bitwise XOR	LˆR
	Logical AND	L&&R
Binary Logical	Logical OR	L||*R*
	Nullish Coalescing	L??R
Ternary	Conditional	C?L:R
Optional Chaining	Optional Chaining	L?.R
Assignment	Assignment	L=R
	Multiplication Assignment	L*=R
	Exponentiation Assignment	L**=R
	Division Assignment	L/=R
	Remainder Assignment	L%=R
	Addition Assignment	L+=R
	Subtraction Assignment	L-=R
	Left Shift Assignment	L<<=R
	Right Shift Assignment	L>>=R
	Unsigned Right Shift Assignment	L>>>=R
	Bitwise AND Assignment	L&=R
	Bitwise XOR Assignment	Lˆ=R
	Bitwise OR Assignment	L|=R
	Logical AND Assignment	L&&=R
	Logical OR Assignment	L|| = *R*
	Logical Nullish Assignment	L??=R
	Destructuring Assignment	[a,b] = [1,2]
	Destructuring Assignment	{a, b} = {a:1, b:2}
Comma	Comma	L,R
Function	Parameter	L_R
	Return	L->R
	Call	L(R)

Table 4.2: JavaScript Relations (continued)

```
const x = (a == b ? 6 : 10)
```

(a) Nested Relation

1. $[L = R, x, y*]$
2. $y* = [C?L : R, z*, 6, 10]$
3. $z* = [L == R, a, b]$

(b) Extracted Relations

Figure 4.3: Example of nested relation extraction from code

relations themselves. In Figure 4.3a we show an example of such nested relations. Constant x is assigned a value, so the formatted relation is equal to $[L = R, x, y*]$. The formatted relation does not contain the whole right part of the assignment; instead, $y*$ is defined. $y*$ is an artificial element that points to the relation that is the right part of the assignment. The right part of the assignment turns out to be a ternary statement. $y*$ points to $[C?L : R, z*, 6, 10]$. Once again, an artificial element $z*$ is created that points to the equality relation in the conditional part of the ternary statement. Hence $z*$ points to the final relation $[L == R, a, b]$. Although the code in Figure 4.3a seems rather simple, there are three relations involved that are contained in each other. These relations are shown in Figure 4.3b.

These nested relations provide additional insight into the types of the elements, as depicted in Figure 4.4. There are two relations present. The first is the assignment relation between x and $y*$, that is, $[L = R, x, y*]$. Here $y*$ points to the comparison, $[L < R, a, b]$. The comparison relation is nested within the assignment relation. Since the relation $L < R$ always results in a boolean value, we can conclude that x must be boolean at this point in the code.

```
const x = a < b
```

(a) Nested Relation

1. $[L = R, x, y*]$
2. $y* = [L < R, a, b]$

(b) Extracted Relations

Figure 4.4: Example of type insights from nested relations

4.2.3 Scopes

As shown in the previous subsections, information about the elements' types can be extracted from the relations in which those elements are involved. However, one important aspect of the elements we have not yet discussed is scoping. The scope of an identifier determines its accessibility. Figure 4.5 shows an example of what scoping does. First, the constant x is assigned the value *5*. The constant x is defined in the so-called global scope. Next, a function is defined, which creates a new scope. This scope has access to references of the global scope but can also have its own references, which are only available to itself and its sub-scopes. Within the function, we observe that another constant x is defined. Note that from this point on, every reference to x in the scope of the function refers to the newly defined constant rather than the one from the global scope. This phenomenon is called variable shadowing. Briefly put, when an identifier is redeclared in a narrower scope, the declaration in the narrower scope shadows the declaration in the broader scope. In the context of the extraction phase, this shadowing principle is fundamental because variable x in the global scope is not the same variable as the one from the function scope. They might have different types. In this case, the x from the global scope is numerical, while the x from the function scope is a string.

To keep track of the scope of the elements, we save the elements and relations with their respective scope identifier. The scope identifier allows the inference techniques to couple different types to elements with equal names.

```
const x = 5

function example(a) {
    const x = "Hello "
    return x + a
}
```

Figure 4.5: Example of variable shadowing with a function scope

4.2.4 Complex Types

In JavaScript, objects are an essential building block of the language. Objects in JavaScript are stores of key-value pairs. Besides primitive types such as booleans or numbers, almost everything can be represented as an object. An array can, for example, be viewed as a special object where the keys are numbers.

In modern JavaScript versions, developers can define classes and interfaces that induce a more object-oriented approach to JavaScript.

Since objects play such a prominent role in JavaScript, it is crucial that object types can also be inferred. To infer what type an object is, it is required first to extract all object type descriptions available to the SUT. These include class and interface definitions and standard objects like an array or a function.

4.3 Probabilistic Inference

After extracting all the elements, relations, and possible types from the SUT, the second phase starts. In this phase, we build a probabilistic type model for each element. This is straightforward for elements that represent literals as the type can be directly inferred from the literal type. However, not every element in the code is a literal. Based on the extracted relations, we make assumptions. For example, when the relation $[L = R, x, 5]$ is processed, it can be inferred that at this particular point in the code, x must be of a numerical type since it is assigned the literal value 5. It is, however, not certain that x is numerical before or after this particular relation. Variable x might be re-assigned to another type. Hence, a scoring system is used instead of fixing x to be numerical. The type models consist of a map linking types to scores. If an element is possibly a certain type, then that type has a non-zero score in the element's type model. We derive these scores from the extracted relation information. Using the previous example relation: $[L = R, x, 5]$ x is assigned a point for the numeric type. If x is later assigned a string value, it also receives a point for the string type. The example is straightforward since the right part of the assignment is a literal value. However, as mentioned before, JavaScript is weakly typed. The following relation showcases this: $[L + R, x, 5]$. In this relation, 5 is added to x. Although the right part is once again a literal value, we cannot say anything about x. We can make the assumption that x is probably numerical, but it is just as likely a string. For this particular relation, it is appropriate to give both an equal score for the string and numerical type.

4.3.1 Complex Type Resolving

In order to identify which elements of the SUT are objects, we check the elements for the *Property Accessor* relation. We compare the accessed properties to the available object type descriptions if an element is involved in one or more *Property Accessor* relations. If there is an overlap between the element's properties and the properties of an object type description, we assign the type description as a possible type of the element. In Figure **??**, we show a Venn diagram of an example from one of the benchmarks. In the benchmark, we find a complex type called *Command*. This *Command* object has several properties such as a name, a set of arguments, a parent *Command* etc. In the Venn diagram, the *Command* object is shown as the largest circle. Now assume that we are trying to resolve the type of an element x. In the (hypothetical) code context of x, we observe that the name, arguments, options, and aliases properties of x are used. We show this as the smaller circle in the Venn diagram. There clearly is an overlap between the properties of x and the properties of the *Command* type. Because of this overlap, we assign a score to the *Command* type for element x.

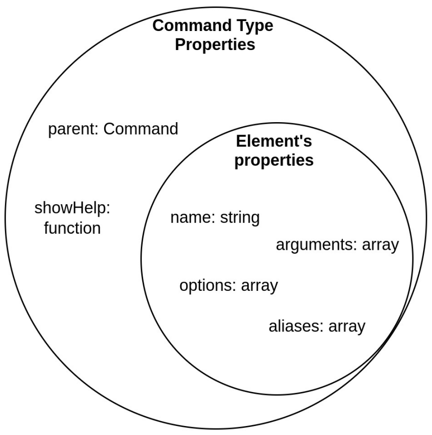


Figure 4.6: Example Venn diagram of a complex type description and an elements' accessed properties

In addition to the matching object descriptions, an anonymous object type is created and assigned as a possible type. This anonymous object type exactly matches the properties of the element. The anonymous object type is useful when there is no matching type description in the source code. This ensures that we can still use a matching type during the search process.

4.3.2 Probabilities

After assigning the scores to the possible types, we use the scores to calculate the probabilities of the types. The higher the score of a particular type, the higher the probability that the element is of that type. The probability p_i of type i is calculated by dividing score s_i by the total score $\sum_i s_i$, as shown in equation 4.1. The sum of probabilities adds up to 1.

$$p_i = \frac{s_i}{\sum_i s_i} \tag{4.1}$$

The probabilities are used during the search phase to sample arguments of the correct type. In order to sample an argument, one of the types in the type model has to be picked. For this study, we created two modes. We call the first *Rank-Based Sampling* mode. In this mode, the type with the highest probability is always selected. We call the second *Proportional Sampling* mode. In this mode, the type is randomly picked based on its probability, i.e., if a type has a 50% probability of being the correct type, it has a 50% chance of being selected.

4.3.3 Type Dependencies

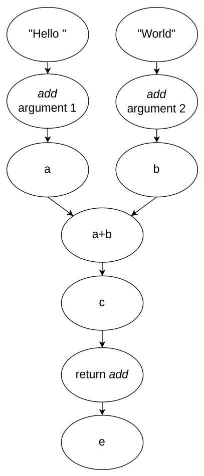
Next to the element types, relations themselves also have a result type. Consider for example the relation: $[L > R, a, b]$. In this relation the output type is boolean no matter what the types of a and b are. Without further context, this is not very useful. However, we often encounter nested relations within code. For example, $[L = R, c, d*]$ where $d*$ equals $[L > R, a, b]$. Since we know that the outcome of $d*$ is a boolean value, we can infer that c must also be a boolean. However, instead of assigning a score for boolean to c we make the type of c depend on the type of $d*$. We do this because there are relations where the outcome of the relation depends on the involved elements. By making the types of elements dependent on each other, we are essentially creating a type probability network.

Figure 4.7 shows an example of such a type probability network. Figure 4.7a shows the source code. We consider the *add* function with two arguments a and b. On line 2 variable c is defined to be equal to $a+b$. The type result of relation $a+b$ is dependent on the types of a and b. Variable c is directly dependent on the result of the relation. We show the dependency relations as a directed graph in Figure 4.7b. In the graph, arrows point from the dependent to the dependee. On line 3 we find a return statement returning c. In the graph, this is shown as the return type of the *add* function being dependent on the type of c. In the second function on line 7, we see that the *add* function is called with two arguments, namely, “Hello ” and “World”. Now we can say that the type of the first argument of *add* depends on the type of the literal “Hello ”. This literal is a string, which indicates that a is probably also that type. This propagates through the entire network. The type probability network ensures that whenever new information about the type of a certain element becomes available, it is propagated to all linked type models. Because of the type

```
function add(a, b) {
  const c = a + b
  return c
}

function main() {
    const e = add("Hello ", "World")
    print(e)
}
```

(a) Source code



(b) Type model network

Figure 4.7: Example type model network

dependencies between elements, it is crucial that for each element, a type model is created, and not just for the elements of interest.

4.4 Search process

The third phase of the approach is the search process. In this phase, we use a heuristic to generate test cases with the goal of uncovering faults in the SUT. In this study, we use an evolutionary approach. Specifically, the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [51] is used. As mentioned in Chapter 2, this algorithm outperforms other algorithms when it comes to automated test case generation.

The evolutionary approach randomly generates a set of test cases. These test cases are then evaluated by looking at how close they are to covering certain branch objectives. The most promising test cases are selected to be part of the next generation, the others are 'killed'. The remaining population of test cases is then used to create a new population, the offspring. This offspring population is created by using variational operators such as mutation and crossover on the members of the parent population. The offspring population is then added to the total population, and the process repeats. Both during the random generation and mutation of test cases, the type models are used to sample arguments for function or constructor calls.

The evaluation of the test cases requires that we run the test cases against the SUT. While executing the test cases, exceptions might occur in the SUT. These exceptions can sometimes provide information on the correctness of the types of the used arguments. This information is stored and used in the fourth phase.

4.5 Dynamic Analysis

In the third phase, the probabilistic type models are used to create test cases with possibly correct types. However, since the models are probabilistic, there is no guarantee that the correct type is used. While these test cases are executed, some test cases may trigger exceptions in the SUT. Some might be actual bugs or intentional exceptions in the SUT. Others are type errors indicating that there is a type mismatch somewhere in the source code. A wrongly inferred argument type possibly causes this mismatch. During the DA phase, the thrown exceptions are intercepted and processed.

Besides the type scores that indicate an element is of a certain type, each element also has execution scores. We use these execution scores to update the adjust the type models using the execution results of the test cases. If a TypeError exception is thrown during the execution of a test case, we proceed to scan the error message or stack trace for variable names that are used in the generated test case. If we find a matching variable, we assign a negative execution score to the type that was used for that variable. In Listing 4.8 we show an example stack trace. From the first line of this stack trace, we can conclude that the description property of the cmd object should have been a function. In this example, the description property was sampled as a string, so the type model of the description property will be assigned a negative point for the type string. The type model's execution scores are used to calculate the type's likelihood. The idea here is that for each TypeError that is thrown, we adjust the type models to become more accurate.

```
TypeError: cmd.description is not a function
    at Help.subcommandDescription (.syntest/instrumented/benchmark/
       top10npm/commanderjs/lib/help.js:3945:16)
    at _callee$ (.syntest/tests/tempTest.spec.js:5:22)
    at tryCatch (node_modules/regenerator-runtime/runtime.js:63:40)
    at Generator.invoke [as _invoke] (node_modules/regenerator-
       runtime/runtime.js:294:22)
    at Generator.next (node_modules/regenerator-runtime/runtime.js
       :119:21)
    at asyncGeneratorStep (.syntest/tests/tempTest.spec.js:7:103)
    at _next (.syntest/tests/tempTest.spec.js:9:194)
    at .syntest/tests/tempTest.spec.js:9:364
    at new Promise (<anonymous>)\n at Context.<anonymous> (.syntest/
       tests/tempTest.spec.js:9:97)
```

Figure 4.8: Example stack trace of a useful type exception

To calculate the final probability p_i of type i we use equation 4.2. The first fraction in the equation equals equation 4.1. The second fraction represents the execution score probability. Since the execution score, es_i is either zero or negative (when a used type gives TypeErrors), we add the absolute value of the most negative score $min_i(es_i)$ to the execution scores of all types. This ensures that the (original) most negative score now has a score of zero, and all the other scores are positive. These modified scores are identified by es'_i. After doing this, we do the same as with the regular scores, i.e., divide the es'_i score by the sum of all es'_i scores.

$$p_i = \frac{s_i}{\sum_i s_i} \cdot \frac{es_i + |min_i(es_i)|}{\sum_i (es_i + |min_i(es_i)|)} = \frac{s_i}{\sum_i s_i} \cdot \frac{es_i'}{\sum_i es_i'} \tag{4.2}$$

Now it must be noted that after doing this multiplication, the sum of probabilities is not equal to 1 anymore. To fix this, we divide all probabilities by the sum of probabilities at the end of the process.

4.6 Approach Variants

Given the phases described in this chapter, we have created five different approach variants to answer the research questions. These variants are listed in Table 4.3. The first variant acts as the baseline to which we will compare the four other variants. The baseline does not use any Static Type Inference or Dynamic Type Inference. The types used for the baseline are completely random, i.e., all types have an equal probability of being used. Because no static analysis is conducted, the baseline is only able to sample primitive types (number, boolean, etc) and the standard JavaScript complex types, i.e. *Object*, *Array*, *String*. The second and third variants only use Static Type Inference for their type models. In other words, they do not use phase 4 in Figure 4.1. The second and third variants use *Rank-Based Sampling* and *Proportional Sampling* respectively, as described in Section 4.3. The fourth and fifth variants use both Static Type Inference and Dynamic Type Inference. Again one uses the *Rank-Based Sampling* mode and the other uses the *Proportional Sampling* mode.

Variant	Sampling Mode	Analysis Mode
Baseline	-	-
SA-only Rank-Based Sampling	Rank-Based	Static Analysis
SA-only Proportional Sampling	Proportional	Static Analysis
SA+DA Rank-Based Sampling	Rank-Based	Static + Dynamic Analysis
SA+DA Proportional Sampling	Proportional	Static + Dynamic Analysis

Table 4.3: Overview of the approach variants

Chapter 5

Syntest JavaScript

We described the approach in the previous Chapter 4. To evaluate the approach, we implemented a tool. The tool will be referred to as SYNTEST-JAVASCRIPT. This chapter will describe the implementation details of the conceived tool.

First, we will detail the foundation on which we build SYNTEST-JAVASCRIPT. Second, we discuss the architecture of SYNTEST-JAVASCRIPT. Third, the process of running SYNTEST-JAVASCRIPT is laid out. Finally, the options SYNTEST-JAVASCRIPT provides are explained.

5.1 Foundation

SYNTEST-JAVASCRIPT belongs to the SynTest project [1]. SYNTEST is a project dedicated to generating synthetic tests for JavaScript-based languages. The foundation of the SYNTEST project is the SYNTEST-FRAMEWORK[2]. SYNTEST-FRAMEWORK consists of a library written in TypeScript that provides generic interfaces to build search-based software testing tools. The first such SYNTEST tool is called SYNTEST-SOLIDITY and as the name implies is a tool for automated software testing of *Solidity* smart contracts[3] [48].

SYNTEST-FRAMEWORK acts as a common core for all SYNTEST projects. It consists of a set of generic search algorithms. These search algorithms are configurable such that they can be optimally used for any type of encoding. Next to the search algorithms, SYNTEST-FRAMEWORK contains a few interfaces to be used in the different language-specific SYNTEST Tools. Finally, it contains several utility classes.

SYNTEST-JAVASCRIPT is the second SYNTEST tool that builds upon SYNTEST-FRAMEWORK. During the development of SYNTEST-JAVASCRIPT we have proposed several improvements for the SYNTEST-FRAMEWORK project in order to make the project more generic and more capable of handling any JavaScript-based language. Besides the language difference between SYNTEST-JAVASCRIPT and SYNTEST-SOLIDITY it is important to note that SYNTEST-JAVASCRIPT is the first SYNTEST tool that focusses on DTL. For this reason,

[1]https://www.syntest.org/
[2]https://github.com/syntest-framework
[3]https://docs.soliditylang.org/

SYNTEST-JAVASCRIPT has a special module dedicated to type inference. It must also be noted that compared to *Solidity*, JavaScript is a more complex and, simultaneously, flexible language. This added complexity and flexibility required a more rigorous static analysis and encoding structure.

Since SYNTEST-FRAMEWORK is written in TypeScript it is possible to create a language-specific tool in both JavaScript and TypeScript. However, due to the advantages that TypeScript offers during the development of large-scale applications like this, it was deemed the more logical choice to build SYNTEST-JAVASCRIPT upon [24]. SYNTEST-JAVASCRIPT is created using *Node.js* v16 which at the time of writing is the active LTS version of *Node.js*.

SYNTEST-JAVASCRIPT supports both CommonJS and ECMAScript *Node.js*[4] modules. Currently, the resulting test cases adhere to the ECMAScript module style.

We created SYNTEST-JAVASCRIPT using Object Oriented Programming principles and various design patterns. It is thus set up to be highly extensible. This setup allows future developers of the tool to seamlessly replace the inner working of the various components of the tool.

5.2 Architecture

This section gives the details of the architecture of SYNTEST-JAVASCRIPT. Figure 5.1 represents the high-level architecture of the SYNTEST-JAVASCRIPT tool. The figure consists of a component diagram that showcases how the different components of the system interact. The figure shows three dashed boxes containing one or more components with a common purpose. This section will detail most of the components in Figure 5.1. First, we will discuss the components within the "Static Analysis" box. Second, the contents of the "Type Inference" box. Finally, we will describe the leftover components. We will not discuss the component in the "Framework" box since it is not specific to SYNTEST-JAVASCRIPT; instead, it is part of the SYNTEST-FRAMEWORK.

5.2.1 Static Analysis

The first component group of the tool revolves around the static analysis. In Chapter 4 the general idea of the static analysis is laid out. In this subsection, we will explain some implementation details. Figure 5.1 shows that the static analysis consists of an *Abstract Syntax Tree Generator*, a *Control Flow Graph Generator*, a *Target Visitor*, *A Type Visitor*, and an *Element & Relation Visitor*.

Abstract Syntax Tree Generator

The static analysis starts by converting the source code to AST. We used the *Babel* library[5] to do this. Other components of the static analysis also use this library to traverse the AST using a visitor pattern.

[4]https://nodejs.org/
[5]https://babeljs.io/

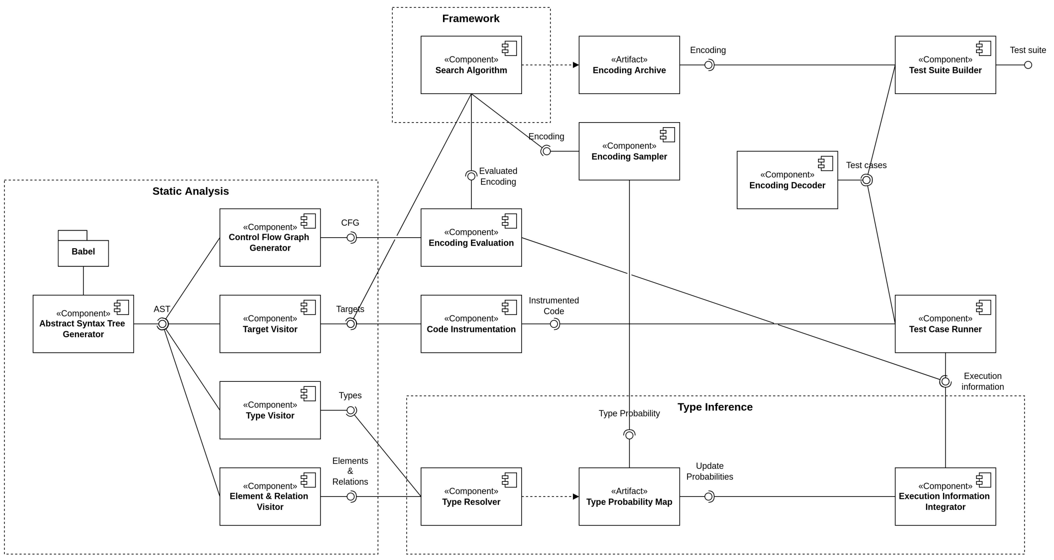

Framework
«Component» Search Algorithm
«Artifact» Encoding Archive
Encoding
«Component» Test Suite Builder
Test suite
Encoding
«Component» Encoding Sampler
«Component» Encoding Decoder
Test cases
Evaluated Encoding
Static Analysis
Babel
«Component» Control Flow Graph Generator
CFG
«Component» Encoding Evaluation
«Component» Abstract Syntax Tree Generator
AST
«Component» Target Visitor
Targets
«Component» Code Instrumentation
Instrumented Code
«Component» Test Case Runner
Execution information
«Component» Type Visitor
Types
Type Probability
Type Inference
Elements & Relations
Update Probabilities
«Component» Element & Relation Visitor
«Component» Type Resolver
«Artifact» Type Probability Map
«Component» Execution Information Integrator

Figure 5.1: Component diagram of SYNTEST-JAVASCRIPT

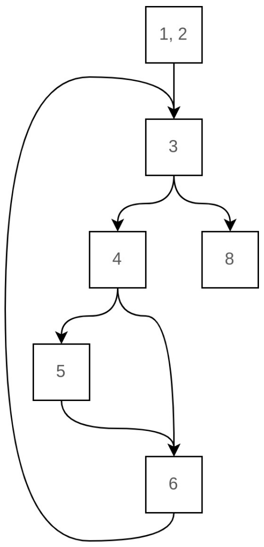
Control Flow Graph Generator

The next component of interest is the *Control Flow Graph Generator*. As the name indicates, this component uses the AST to create a Control Flow Graph (CFG). A CFG is a graph representation of all code paths that might be traversed during program execution. Nodes in this graph represent pieces of code without any jumps. The directed edges between the nodes represent the jumps in the control flow. In Figure 5.2 an example of such a CFG is given. In Figure 5.2a the source code with numbered lines is shown. In Figure 5.2b the resulting CFG, with the corresponding line numbers for each node, is shown. On line 3 we encounter the first branching point, the for loop. In the CFG two branches are exiting node 3. The left-most branch is the case where $i < arr.length$ and thus we enter the for loop and end up on line 4. The rightmost branch is essentially the 'false' branch where we do not enter the for loop and thus directly go to the return statement on line 8.

The generated CFG is used by the search process to determine how far a certain test case is from covering a certain branch. We do this by looking at what branches the test case covers. For example, say we execute test case X on the *countZeros* function in Listing 5.2a. The coverage results show that lines 1, 2, 3, and 8 are covered, i.e. the for loop has not been entered. Now we want to know how far we are from covering line 5. We do this by looking at the CFG. We calculate the shortest paths from each of the covered nodes/lines to node 5. It is important to note that these paths are directed, i.e., in the example, it is impossible to go from 8 to 3. Next, of all the shortest paths we again take the shortest; in this case, the shortest path from a covered node to 5 is the path 3, 4, 5 which has length 2. This metric is called the approach level.

```
1 function countZeros(arr) {
2   let count = 0
3   for (let i = 0; i < arr.length; i++) {
4     if (arr[i] === 0) {
5         count += 1
6     }
7   }
8   return count
9 }
```

(a) Source code



(b) Control flow graph

Figure 5.2: Example of code to control flow graph translation

Target Visitor

By extracting all classes, methods, and functions, the *Target Visitor* gathers possible targets from the ASTs. We only create tests for targets that are exported in the source code. The resulting targets contain information regarding the scope of the class/method/function, the required parameters, and the return parameter.

Type Visitor

The *Type Visitor* is responsible for finding the user-defined types. As described in Chapter 4 these user-defined types are used to infer which elements should have such types. The *Type Visitor* extracts classes, user-defined objects, prototyped functions, and interfaces. The extraction results in a set of "complex objects". It is important to note that the *Type Visitor* does not exclusively look into the source code of the Unit under Test (UuT). Instead, it analysis the entire code base in which the UuT is situated.

Element & Relation Visitor

In addition to the *Type Visitor* we also have the *Element & Relation Visitor*. This visitor gathers all elements and the relations they are involved in from the AST. Thus, at the end of this visitor process, we have a complete set of all interactions between variables and constants.

5.2.2 Type Inference

The next component group is the Type Inference group. Although the type inference is strongly connected to the static analysis, it requires its own component group. This group consists of the *Type Resolver* and the *Execution Information Integrator* as shown in Figure 5.1. The most important artifact in this component group is the *Type Probability Map* which is created by the *Type Resolver* and later modified by the *Execution Information Integrator*. In essence, the *Type Resolver* performs the initial Static Type Inference while the *Execution Information Integrator* performs the Dynamic Type Inference.

Type resolver

The *Type Resolver* uses the results of the type-related static analysis; the *Type Visitor* and the *Element & Relation Visitor*. We use these results to infer the elements' types.

The first step of the *Type Resolver* is to resolve the types of each of the primitive elements. This step is relatively simple as it is evident that, for example, a string primitive is of the type string. However, it is a crucial step to make the inference network complete.

Next, we resolve the relations. We do this in two steps. First, we try to infer the involved elements of the relation. This is done by making assumptions about certain relations. For example, a comparison relation likely involves numeric elements. However, this is not always a given as JavaScript is very flexible regarding what types of elements are usable in any relation. Another example might be that we have an equality relation. In this case, we expect the two elements to be of the same type. We thus make the type probabilities of the two elements loosely dependent on each other. The second step of the relation resolving is resolving the actual relationship itself. Sometimes it is possible to infer the resulting type of a relation without knowing the types of the involved elements. For example, the *typeof* relation always returns a string. On the other hand, a return relation's type is equal to that of the returned element.

Finally, we resolve the “complex” elements. These consist of all elements which are the object in a *Property Accessor* relation. Now, by comparing the properties that are being accessed in the scope of the element to the properties of the ”complex objects” found by the *Type Visitor*, we can infer which “complex object” type belongs to which “complex” element. Additionally, we always create one anonymous complex object type that matches the properties being accessed. This is done such that there is always a possible type present in the type probability map, which can be used for the test cases.

As explained in Chapter 4, the entire probabilistic inference uses a scoring system where on every hint of an element being of a certain type, we assign a point to that type within the type probability map of the element. Once the static analysis is finished, we use the scores to calculate the probability of each type. We described the equation for this calculation in Chapter 4.

The final result of the inference is a *Type Probability Map* per element. This probability map consists of a probability per possible type. More likely types have a higher probability.

Execution Information Integrator

The *Execution Information Integrator* is the second component used for type inference. It is responsible for adjusting the type probability map of each element once new execution information becomes available. This execution information is provided by executing test cases during the search process.

Most types in an element's type probability map are not actually applicable. This is because the elements often only have one valid type, while the probability maps can contain multiple. For example, an element X might have a probability map with a probability 0.3 for string and 0.7 for number. In JavaScript, it is possible to have code that allows X to be a string or a number. However, more often than not, the creator of that code intended X to be only one of the two. Although JavaScript is very flexible, using more complex types can result in unexpected behavior when the wrong type is used. The code might, for example, throw a TypeError. During the search process, these TypeErrors are caught and used to improve the accuracy of the probabilities in the type probability map. The current version of the tool does this by scanning the error messages for the names of parameters used by the UuT. If there is a match, we assume that the type used for that parameter is likely wrong. We thus give the type in question a negative point for the execution score. Note that this is a different score than the type score. The calculation of the new type probabilities is described in Chapter 4.

5.2.3 Instrumentation

As mentioned in Chapter 4 the search process of test-case generation using search-based techniques relies on branch coverage as guidance to create a sufficient test suite. We need to know which branches are evaluated during a certain execution to track the branch coverage. We can do this by modifying the original source code of which we want to track the coverage. In Listing 5.3, you can view the modification being made. In Listing 5.3a, you can see the source code, which consists of a simple function that returns the maximum value of the two given arguments a and b. In this example, the if statement creates exactly one branch. In Listing 5.3b you can observe the resulting instrumented code. On line 2, the first call to our track function is made. It tracks that function '0' has been called. Similarly, we see on line 5 that branch '0' is tracked. Additionally, we track that this branch's 'true' part has been executed. On line 8 we again see branch '0' being tracked, but this time the 'false' part of the branch is executed.

In addition to the branch coverage, we also record the branch conditions and the values of the involved elements as they are evaluated. On line 3, we show an example of this. Here we record the condition and the values of the elements in the condition of branch '0'. This extra information allows for the calculation of the branch distance in addition to the approach level.

For the instrumentation, we used an older version of *istanbul.js* [6] as inspiration. However, since *istanbul.js* is not meant for automated test case generation tools, some modifications had to be made to create a version that meets the requirements of SYNTEST-

[6] https://istanbul.js.org/

```
function max(a, b) {
  if (a < b) {
    return a
  } else {
    return b
  }
}
```

(a) Source code

```
function max(a, b) {
  track('function', 0)
  record(0, 'a < b', [a, b])
  if (a < b) {
    track('branch', 0, true)
    return a
  } else {
    track('branch', 0, false)
    return b
  }
}
```

(b) Instrumented code

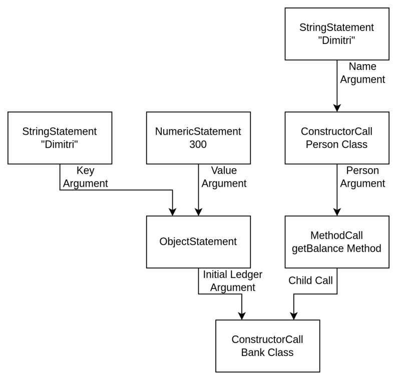
Figure 5.3: Example of source code to instrumented code translation

JAVASCRIPT. For example, the original does not treat loops as branches. This creates the problem that the approach level is not entirely correct, giving the search algorithm less guidance. Another modification is the addition of the aforementioned recording of branch conditions and element values.

5.2.4 Encoding

The goal of SYNTEST-JAVASCRIPT is to generate test cases, which means we are trying to synthesize code using a genetic algorithm; this is called Genetic Programming [38]. However, genetic programming is more than simply mutating a string until it forms a test case. Instead, we use an encoding that describes how the test case should be formatted. Since code can be represented as a tree (for example, an AST), it makes sense to also represent the encoding of the test cases as a tree. This encoding is a tree of statements. Figure 5.4a gives an example of such a tree.

The encoding is split up into four classes. The first class is the primitive class, consisting of the Bool, Null, Numeric, String, and Undefined statements. This class contains the most basic building blocks of the encoding. The second class is the complex class, consisting of the Array, Object, and Arrow Function statements. These statements are more complex as they can have child statements. An array statement, for example, can be empty but can also be filled with, e.g., numeric statements. The object statement always has child statements in pairs; one is the key, and the other is the value. The key statement must always be a string. The arrow function statement only has a return statement in the current implementation. Note that these complex statements can, next to primitive statements, also have complex children. The third class is the root class; it consists of the function call statement and constructor call statement. We call this class the root class because the root of the encoding tree is always a constructor or a function call. Both these statements can hold argument statements. Like child statements of complex statements, the argument statements are used to provide the arguments required for the constructor or function call. The constructor call differs from the function call as it can have children next to the required arguments. These additional children consist of method calls from the fourth and final class. This final class is the action class and consists of only the method call statement. These method calls are children of the constructor call statement. They are essentially calls to the methods of the instantiated class.


StringStatement "Dimitri"
Name Argument
StringStatement "Dimitri"
NumericStatement 300
ConstructorCall Person Class
Key Argument
Value Argument
Person Argument
ObjectStatement
MethodCall getBalance Method
Initial Ledger Argument
Child Call
ConstructorCall Bank Class

(a) Encoding tree

```
it("getBalance method", () => {
    const initialBalance = 300
    const ledger = {
        "Dimitri": 300
    }
    const bank = new Bank(ledger)

    const name = "Dimitri"
    const person = new Person(
        name)

    const balance = bank.
        getBalance(person)
})
```

(b) Decoded test case

Figure 5.4: Example of decoding an encoding to a test case

In Figure 5.4a an example of an encoding tree is given. At the bottom of the tree, we find the constructor call as the root. Here the *Bank* class is instantiated using a single argument. This argument is an object statement containing a single key-value pair. The constructor call also contains one method call statement named *getBalance*. This method call requires one argument of the type *Person*, which also is a class and thus can be instantiated by another construct call. The *Person* class requires a single argument, the name of the *Person*, which is a string statement. The decoded test case of the example encoding tree is shown in Listing 5.4b.

5.2.5 Sampler

We sample these encoding trees through the *Encoding Sampler* component. The Search Algorithm uses this component constantly. At first, by sampling the initial population, and later when new encodings need to be sampled to keep a high diversity. The tree structure of the encoding allows the sampler to resample the entire encoding tree or sub-trees. The *Encoding Sampler* uses the targets provided by the *Target Visitor* in combination with the *Type Probability Maps* to sample encoding trees with the corresponding types. Although the goal is to create test cases with types that match the source code, it is interesting to use the wrong types every now and then. For example, using the wrong type of argument for a certain function call can lead to unexpected executions which should not have been possible, resulting in the discovery of a bug. Finding these kinds of bugs can be very valuable. For this reason, the sampler samples a random type with a small probability defined by the *random_type_probability* option.

5.2.6 Evaluation

In order to evolve the population of encodings to a set of useful test cases, i.e., a set of test cases with high branch coverage, the encodings need to be evaluated such that the most promising encodings can proceed to the next generation. To do this evaluation, we send the encodings to the *Test Case Runner*, which uses the *Encoding Decoder* to convert the encoding tree to a textual representation. The *Test Case Runner* then proceeds to run the test case against the instrumented source code. Once the test case run is done, the execution information is returned to the *Encoding Evaluation* component. As mentioned before, this information is then combined with the CFG to calculate the approach level and the branch distance of the encoding.

5.2.7 Suite Builder

Once the budget of the search algorithm has run out, it creates the encoding archive. This archive consists of all encodings which are key to covering certain branches or have induced unique crashes. We then pass the archive to the *Test Suite Builder*, which uses the archive and the *Encoding Decoder* to create the final test suite. In addition to simply putting all the tests together, the *Test Suite Builder* also creates all the assertions that a proper test case should have. These assertions consist of checks on the runtime return values of function and method calls. Finally, if exceptions occur while executing a test case, the failing statement is asserted to throw an exception.

5.3 Running the tool

Developers can use the tool through the easy-to-use Command Line Interface (CLI) that is publicly available on the Node Package Manager repository [7]. Another way is through cloning the public git repository [8] and installing the tool by following the instructions in the `README.md`. This document also includes instructions for creating a docker image specific to running the tool.

The CLI is highly configurable and offers a range of options. These options will be detailed in Chapter 6. The options can be given directly as arguments to the CLI or by creating a *.syntest.js* file in the root of their repository. Within this file, each option can be configured.

If a developer is interested in running the tool on their code, they can run the following command after installing the package:

```
syntest-javascript --target_root_directory="<PATH_TO_ROOT_DIRECTORY>"
--include="<PATH_TO_TARGETS>" --search-time=120
```

This command will start a test case generation process with a search time of 120 seconds per target. It will perform the static analysis using all the source code available in the given

[7]https://www.npmjs.com/package/@syntest/javascript
[8]https://github.com/syntest-framework/syntest-javascript

```
it('test for detectUndirectedCycle', async () => {
    const _isDirected_boolean_WdyT = true;
    const _graph_Graph_lk5v = new Graph(_isDirected_boolean_WdyT)
    const _vertex_undefined_5ax = undefined;

    expect(JSON.parse(JSON.stringify(_graph_Graph_lk5v))).to.deep.equal({
        "vertices":{},"edges":{},"isDirected":true})

    try {
        const _returnValue_any_wkP1 = await _graph_Graph_lk5v.
            getNeighbors(_vertex_undefined_5ax)
    } catch (e) {
        expect(e).to.be.an('error')
    }
});
```

Figure 5.5: Example generated test case

`<PATH_TO_ROOT_DIRECTORY>`. The `<PATH_TO_TARGETS>` can be a single file or a pattern that selects a set of files.

Running the tool will give a bunch of information regarding the process. First, some general information, such as the parameter settings and the included files, is given. Next, a loading bar indicating how far we progressed with the coverage is shown. Next to this bar, the remaining time budget is given. Finally, a small overview of the covered branches, statements, and functions is given once the search process is done.

Once the tool is done, the results are located within the `syntest` folder. This folder includes the process logs, the run statistics, and the generated test cases. An example of such a test case is shown in Listing 5.5. These test cases adhere to the Mocha[9] format. The variable names are of the format `_<argument name>_<type>_<random string>`. As shown, the results of the function calls are checked using assertions. When an exception is expected, we wrap the function call in a try-catch block.

The statistics in the `syntest` folder are created in two different formats. The first contains statistics that are available at the end of the experiment. This includes different types of structural coverage, the timing of the initialization and search, and values of certain parameters, such as the random seed that was used. The second type contains several types of structural coverage for each generation together with a timestamp. The second type of statistic allows for the investigation of the performance over time.

5.4 Options

As mentioned in the previous section, SYNTEST-JAVASCRIPT is highly configurable, either through the configuration file *.syntest.js* or directly by giving arguments to the CLI.

Most of the configurable options of SYNTEST-JAVASCRIPT come directly from SYNTEST-FRAMEWORK. These are already detailed in the documentation of SYNTEST-FRAMEWORK

[9] https://mochajs.org/

and will thus not be discussed in this thesis. However, some of the most important options are worth mentioning. The first four rows in table 5.1 show these options. The last three rows in the tablet show the configurable options unique to SYNTEST-JAVASCRIPT.

Option	Argument	Description
target_root_directory	string	The path to the root directory
include	array of strings	The paths/patterns to include
exclude	array of strings	The paths/patterns to exclude
search_time	number	the budget in seconds per UuT
type_inference_mode	string	The type inference mode
incorporate_execution_information	boolean	Incorporate Execution Information
random_type_probability	number	The random type probability

Table 5.1: Overview of relevant configurable options of SYNTEST-JAVASCRIPT

The first option is the path to the root directory of the SUT. Everything in the root directory will be used during the static analysis to, for example, find types. The next option is the include argument. This option should receive an array of string paths or patterns. These paths/patterns will be used to find possible UuTs. The exclude option does the exact opposite; it excludes any UuTs found using the given paths/patterns. It is important to note that the exclude option overrules the include option. The last option of SYNTEST-FRAMEWORK is the search time option. It dictates the budget in seconds per UuT. The first option of SYNTEST-JAVASCRIPT is the type inference mode. This option can have one of three possible values, namely: `none`, `proportional`, and `ranked`. These modes directly correspond to the baseline, the *Proportional Sampling*, and the *Rank-Based Sampling* approach variants. Next, we have the incorporate execution information option. This option decides whether to use the execution information received from the test case execution of the search process. Finally, the random type probability option should be a decimal number dictating the probability that a random type is used instead of the inferred type.

Chapter 6

Empirical Evaluation

In the previous chapter, we detailed the ins and outs of the implemented approach. In this chapter, the experimental setup we used to evaluate the approach will be laid out. First, we discuss the benchmark used for the evaluation. Here the choices for the benchmark will be explained. Second, we explain the research questions that aid in achieving the research aim. Thirdly, relevant parameters are enumerated together with the values chosen for those parameters. In the same light, the parameters we varied during the experiments are discussed. Next, we discuss the experimental protocol used to compare the approaches. Finally, we describe the threats to this study's validity and reproducibility and what we did to minimize them.

6.1 Benchmark

We created a benchmark to evaluate the effectiveness of the conceived approach. This benchmark consists of 5 projects, namely: *Express*[1] a web framework for *Node.js*. *Commander.js*[2] a command-line framework for *Node.js*. *Moment.js*[3] a JavaScript library for parsing, validating, manipulating, and formatting dates. *JavaScript Algorithms*[4] a library containing popular algorithms and data structures. *Lodash*[5] a JavaScript library which provides utility functions for common programming tasks. These projects were picked based on the number of stars on *GitHub* or weekly downloads from the Node Package Manager [6] in the JavaScript community. Additionally, we have chosen the projects such that together, they represent a diverse set of syntax and code styles.

We have hand-picked a subset of files from the benchmark projects to be used for this evaluation. The files were chosen based on several factors. First, the file must contain something testable, i.e., an exported function or class. Secondly, the file needs to have a

[1] https://expressjs.com/
[2] https://tj.github.io/commander.js/
[3] https://momentjs.com/
[4] https://github.com/trekhleb/javascript-algorithms
[5] https://lodash.com/
[6] https://www.npmjs.com

Cyclomatic Complexity (CC) of at least 2. Since a lower CC means fewer linearly independent paths through the source code, we will likely require fewer tests to achieve high structural coverage [64]. Although, for this study, the case could be made that achieving high coverage in files with lower CC is still non-trivial since that would sometimes require correctly inferred types. For this reason, for large projects with lots of files, we chose a subset of files with a range of CC values. In Tables 6.1 and 6.2 the CC per file is given indicated by the **CC** column.

To calculate the CC, we used *Plato* [7], which is a tool for static analysis of JavaScript source code. In addition to the CC, we give the number of branches indicated by the **B** column. Finally, we provide the number of Source Lines Of Code indicated by the **SLOC** column, also known as the physical lines of code.

For some projects, files had to be excluded or modified. For example, in the *Commander.js* project, two files contain statements that exit the entire program; this is the intended behavior for the project. However, the entire tool exits during test case generation when these statements are reached. For this reason, such files should be excluded. However, it is often the case that files within a project depend on each other. So when these dependencies are not properly mocked and the tool tests a related file it actually also tests parts of the excluded files. In some cases, the exiting statements are again reached. For this reason, we modified the problematic files such that they do not exit the program. Of course, this is not an optimal solution. For future work, a long-term solution would be to properly mock the dependencies of the SUT.

As previously mentioned, only files with exported functions or classes were considered for the benchmark. These functions and classes are the UuTs that the tool is testing. The question might arise, why not test individual class methods. Well, functions generally do not have or modify states. On the other hand, class methods often modify the parent class's state. This does not apply to all code, but the consensus in the computer science community is that class methods should interact with the class state. Otherwise, they might just as well be static functions. So for these reasons, the approach tests either a function with just one call to that function, or a class by calling one or more of its methods. The UuTs per file are given in the Tables 6.1 and 6.2 indicated by the **UuT** column. Note that there might be more non-exported functions and classes present in the files, which increase the CC, the number of Branches, and the number of Source Lines Of Code.

6.2 Research Questions

This study aims to answer three questions revolving around using type inference during test case generation. Generating meaningful tests while types are unknown is difficult. Type inference has been researched but, to the best of our knowledge, not in the context of test case generation for DTLs. This leads to the first research question:

1. What is the performance impact of using inferred types versus random types on the test coverage generated by automated test case generation

[7]https://github.com/es-analysis/plato

Benchmark	**File**	**UuT**	**CC**	**SLOC**	**B**
Commander.js	help.js	1	50	406	66
	option.js	2	20	324	18
	suggestSimilar.js	1	21	100	32
Express	query.js	1	5	47	6
	layer.js	1	17	181	22
	route.js	1	23	225	30
	application.js	1	42	661	52
	request.js	1	35	525	44
	response.js	1	133	1169	174
	utils.js	7	28	304	34
	view.js	1	14	182	16
Moment.js	valid.js	2	21	51	8
	date-from-array.js	2	7	35	8
	from-string.js	3	31	258	50
	from-array.js	1	39	187	46
	from-string-and-format.js	1	24	135	32
	from-string-and-array.js	1	12	67	16
	from-object.js	1	4	20	4
	from-anything.js	2	26	117	34
	constructor.js	3	19	80	32
	get-set.js	5	16	73	22
	add-subtract.js	1	10	61	14
	calendar.js	2	16	53	22
	compare.js	6	28	72	28
	diff.js	1	15	79	12
	format.js	4	16	78	26
	locale.js	2	4	34	6
	min-max.js	2	13	62	16
	start-end-of.js	2	35	164	20

Table 6.1: Benchmark statistics breakdown

Benchmark	**File**	**UuT**	**CC**	**SLOC**	**B**
JavaScript Algorithms	breathFirstSearch.js	1	7	75	8
	graphBridges.js	1	5	95	8
	detectDirectedCycle.js	1	5	93	8
	detectUndirectedCycle.js	1	5	59	8
	dijkstra.js	1	6	80	10
	eulerianPath.js	1	9	101	14
	floydWarshall.js	1	4	72	6
	hamiltonianCycle.js	1	6	134	10
	kruskal.js	1	6	62	10
	prim/prim.js	1	8	73	12
	stronglyConnectedComponents.js	1	5	133	8
	bfTravellingSalesman.js	1	7	104	14
	Knapsack.js	1	24	195	40
	CountingSort.js	1	9	78	14
	Matrix.js	12	26	309	38
	RedBlackTree.js	1	22	323	34
Lodash	equalArrays.js	1	19	84	24
	hasPath.js	1	11	53	8
	random.js	1	11	73	14
	result.js	1	6	53	10
	slice.js	1	11	47	20
	split.js	1	9	42	8
	toNumber.js	1	12	65	20
	transform.js	1	10	59	12
	truncate.js	1	19	113	34
	unzip.js	1	5	43	6

Table 6.2: Benchmark statistics breakdown (continued)

tools?

Furthermore, because this study involves test case generation and execution, the approach has unique access to execution information that can be dynamically analyzed. This could lead to more accurate type prediction as it allows verification of the types. Since the use of execution information may have an impact on the accuracy of inferred types, it could have an impact on the performance of the test generation tool, hence the second research question:

2. How does the incorporation of execution information impact the performance of automated test generation tools when using inferred types?

Finally, since test case generation tools are often evaluated on their ability to generate as much coverage as possible in a short period, it is important to investigate the time it takes to use the type inference techniques. Thus the third and final question follows:

3. How significant is the amount of time used for the type inference?

These three research questions will be answered in Chapter 8 by analyzing the results of the experiments, which can be found in Chapter 7.

6.3 Parameters

Complex approaches and tools like the one conceived during this study often have numerous configuration options in the form of parameters. For the empirical evaluation of the approach, choices about the values of the parameters had to be made. In this section, we discuss the values of the parameters which are most relevant to the research aim. Some of the values for these parameters are derived from previous work or picked for specific reasons and thus not varied. The varied parameters are the ones that help answer the research questions and have the highest impact on the performance of the approach.

6.3.1 Constant parameters

Starting with the constant parameters, the first that comes to mind is the choice of search algorithm. Based on the findings of the related work, we concluded that DynaMOSA[51] is the most appropriate choice. DynaMOSA is a many-objective genetic algorithm specialized for test-case generation. This enables DynaMOSA to deal with the scalability issues that arise when dealing with hundreds of branch objectives.

Since DynaMOSA is adapted to deal with the scalability issues other many-objective genetic algorithms have, it does not require an enormous population size to maintain genetic diversity. A common default value for the population size is 50 [51]. Now since the number of branches objectives per benchmark file is not extremely large, the default population size of 50 was deemed sufficient.

The maximum allowed search time per UuT amounted to 120 seconds, which is common in literature [29, 49]. From the results, we concluded that 120 seconds is sufficient time to showcase the capabilities of the approach. The benchmark consists of a total of 97

UuTs, meaning that running a single experiment took 3 hours and 14 minutes. Additionally, due to the stochastic nature of the approach, each experiment needs to run for at least 50 times to measure the average performance. This brings the run-time per experiment to 161 hours and 40 minutes. Of course, this can be partly parallelized, reducing the run-time.

The other configurable parameters the SYNTEST-FRAMEWORK offers were kept at their default values.

6.3.2 Varied parameters

To answer the research questions, some parameters must be varied over the experiments. This allows us to investigate the impact of certain parameters on the performance of the approach.

The first parameter of interest is the “type inference mode”. As the name indicates, this parameter decides which type inference mode is used to decide on the types of the elements. At the time of writing, there are three modes. The first is None, which is the baseline mode as it omits the type inference altogether. In the baseline mode, the type models are ignored, and thus each type has an equal probability of being sampled. The second mode is `ranked`, the *Rank-Based Sampling* mode. In the *Rank-Based Sampling* mode, we always select the type with the highest likelihood in the type model. That is, for any element, we always sample the most likely type. The third and final mode is `proportional`, the *Proportional Sampling* mode. In this mode, the algorithm samples a random type from the type model based on the likelihoods. In other words, if a certain type has a high likelihood, it has a high probability of being sampled. In contrast to *Rank-Based Sampling* mode in *Proportional Sampling* mode unlikely types are not excluded from being sampled.

The second varied parameter is the ”incorporate execution information” parameter. This parameter requires a boolean value and determines whether the approach incorporates execution information to improve the accuracy of inferred types, i.e., Dynamic Type Inference. If we consider Figure 4.1, this parameter decides whether phase 4 is active or not. If this parameter is set to `false` the type models are purely based on Static Type Inference. When this parameter is `false` we refer to the approach variant as *SA-only*, if it is `true` we refer to the variant as *SA+DA*.

6.3.3 Other parameters

Due to the nature of the approach, there are a bunch of additional parameters which can be tuned. These parameters can be found in the scoring system. As described in Chapter 4, for a selection of relations, type scores are assigned to the involved elements. However, there is no guarantee that these scores are representative of the real world. As a suggestion for future work, these parameters can be tuned through some machine learning model, or by mining massive data sets of JavaScript code to learn the actual type scores. However, there is some beauty in the simplicity of the current model, as it is based on basic JavaScript rules and general syntax. The use of more advanced models might result in over-fitting for certain syntax types and in-explainable choices for type scores.

Part	**Name**	**Information**
CPU	2x AMD EPYC 7H12 64-Core Processor	2x128 Threads 1.5-2.6 GHz
RAM	-	256 GB
Drive	-	10 Gbs networked SSD network share
OS	Ubuntu	20.04 LTS

Table 6.3: System setup specification

6.4 Experimental Protocol

As mentioned in the previous section, we ran each experiment 50 times to calculate an average performance. This is necessary due to the stochastic nature of the approach. During each trial each of the 97 UuTs is considered separately for 120 seconds. Meanwhile, during those 120 seconds, every generation the number of covered objectives is recorded with the current timestamp. After the 120 seconds are over, the final branch coverage is recorded separately with other statistics about the run.

Each experiment was run on the same system to ensure that the hardware did not influence the results of the experiments. The relevant specifications of the system setup can be found in Table 6.3. Each experiment was allowed to use a maximum of 8 GB of ram. This maximum allowed a maximum of 50 experiments/trials to be run in parallel.

To compare the performance difference between the various experiment settings, the median and the Inter-Quartile-Range (IQR) per experiment are calculated from the final branch coverage. These statistics can be found in Chapter 7. For these experiments, the median was chosen over the mean to prevent anomalies from twisting the results. We also report the IQR to give a sense of the spread of the results.

In addition to the median and IQR it is crucial to determine whether the results of the different experiments are significantly different from one another. To this end, we used the unpaired Wilcoxon signed-rank test [20] with a threshold of 0.05. This non-parametric statistical test determines if two data distributions are significantly different. In other words, if the data distributions are not significantly different, they might be sampled from the same distribution. The Wilcoxon signed-rank test is combined with the Vargha-Delaney $\hat{A}_{12}$ statistic [62] to describe the effect size of the result, which determines the magnitude of the difference between the two data distributions.

Since, for some benchmarks, the results in Chapter 7 are inconclusive, i.e., there is no significant difference between certain techniques, I decided to also compare the Area Under the Curve (AUC) for each of the benchmark files. If there is not a clearly superior technique given the final coverage percentage, the AUC values can give more insight. AUC values are especially insight-full when working with metrics with a clear bound such as branch coverage, i.e., there are only so many branches within the SUT. For example, if multiple techniques achieve 100% coverage, there is no clear winner unless we look at the AUC and find that one technique achieves 100% coverage much earlier in the process. In other words, AUC indicates how fast a technique achieves a certain level. In this case, the AUC can be calculated by taking the branch coverage at each second and summing those values.

For the AUC values the median and IQR are also calculated together with the pair-wise comparisons of the AUC values between the techniques.

6.5 Threats to Validity & Reproducability

Threats to *construct validity*. We rely on well-established metrics in software testing to compare the different approach variants, namely branch coverage, fault detection capability, and running time. As a stopping condition for the search, we measured the search budget in terms of running time (i.e., 120 seconds) rather than considering the number of executed tests. Given that the different approach variants in the comparison use different types of inference with different overheads, execution time provides a fairer measure of time allocation.

Threats to *internal validity*. Our prototype is an extension of the open-source SYNTEST-FRAMEWORK. Although we thoroughly tested the extension, there is no guarantee that the code is completely bug-free. Any bugs introduced in the extension or in the integration with SYNTEST-FRAMEWORK could potentially influence the results of the experiment and therefore impact the validity. To minimize this risk, we published the entire code base in an open-source code repository for other people to inspect and replicate the experiment.

Threats to *external validity*. An important threat regards the number of projects in the benchmark. We selected five projects based on their popularity in the JavaScript community. The projects are diverse in terms of size, application domain, purpose, syntax, and code style. Further experiments on a larger set of projects would increase the confidence in the generalizability of the study. The benchmark has a limited number of projects because of the high cost of running experiments with so many UuTs and the difficulty of supporting all types of syntax and code styles. There is no guarantee that the results will generalize to other projects.

Threats to *conclusion validity* are related to the randomized nature of the test case generation algorithms. To minimize this risk, each experiment has been executed 50 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [12] and analyzed the possible impact of different random seeds on the results. We used the unpaired Wilcoxon signed-rank test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

Chapter 7

Results & Discussion

This thesis aims to evaluate the performance impact of using type inference for test-case generation. Additionally, we want to investigate the performance impact of using dynamic execution information to improve the type inference. Finally, we evaluate the significance of the amount of time used by the type inference.

In Chapter 6, we discussed the experimental protocol used to evaluate SYNTEST-JAVASCRIPT. SYNTEST-JAVASCRIPT is the implementation of the novel approach proposed in Chapter 4. In this chapter, we go over the results of the experiments and discuss them to answer the three research questions proposed in Chapter 6.

To measure the performance differences, four variants plus a baseline variant of the approach described in Chapter 4 have been implemented as described in Chapter 5. These four variants are compared to the baseline approach as well as to each other.

The following three sections will go over the three research questions in order. Each section is split into a results subsection and a discussion subsection. In the results subsection, the relevant statistics of the experiments are given. In the discussion subsection, the results are analyzed and explained.

7.1 Inferred Types versus Random Types

The first research question, as stated in Chapter 6 is:

> **1. What is the performance impact of using inferred types versus random types on the test coverage generated by automated test case generation tools?**

To answer the first research question, we use the achieved structural coverage to measure and compare the performance of the *SA-only Rank-Based Sampling* and *SA-only Proportional Sampling* variants against the baseline. To be more specific, we will look at the achieved branch coverage.

7.1.1 Results

First, we look at the total branch coverage, i.e., the branch coverage achieved at the end of the search budget. In Table 7.1, we report the achieved median branch coverage per benchmark file for each variant and the baseline. Additionally, the IQR is given. The benchmark name and filename can be found in the table's two leftmost columns. Next to these two columns from left to right the median and IQR of: the baseline, SA-*only* *Rank-Based Sampling*, SA+DA *Rank-Based Sampling*, SA-*only* *Proportional Sampling*, and finally SA+DA *Proportional Sampling* are provided. Superior median values are marked gray. If there is a tie, both variants are marked gray.

As mentioned in Chapter 6 it is important to check whether the performance differences between the variants are significant. In Table 7.2 the baseline is compared to the SA-*only* *Rank-Based Sampling* variant and the SA-*only* *Proportional Sampling* variant. Additionally, the SA-*only* *Rank-Based Sampling* variant is compared to the SA-*only* *Proportional Sampling* variant. The two leftmost columns of the table report the benchmark and the benchmark file. After the first two columns, the table reports the comparisons. For each comparison, we give the Wilcoxon signed-rank *p*-value to indicate the significance of the performance difference. Additionally, we provide the Vargha-Delaney $\hat{A}_{12}$ statistic to give an indication of the magnitude of the performance difference. Significant *p*-values, i.e. $p \leq 0.05$ are marked gray.

To answer the first research question, we only need to look at the results of the baseline, static analysis (SA) *Rank-Based Sampling* variant, and static analysis (SA) *Proportional Sampling* variant.

Commander.js: Table 7.1 together with Table 7.2 show that on the *Commander.js* benchmark the *Rank-Based Sampling* variant significantly outperforms the baseline only on the *help.js* file. On the other 2 files, the baseline outperforms the *Rank-Based Sampling* variant. The *Proportional Sampling* variant outperforms both the baseline and the *Rank-Based Sampling* on all 3 files.
Express: For the *Express* benchmark, we find that the *Rank-Based Sampling* variant significantly outperforms the baseline on the *utils.js* file. The baseline outperforms the *Rank-Based Sampling* variant on 2 files. The difference is insignificant on the other 3 files. The *Proportional Sampling* variant significantly outperforms the baseline on 2 files. The difference is insignificant on the other 4 files. The same results can be observed when comparing the *Proportional Sampling* variant against the *Rank-Based Sampling* variant.
JavaScript Algorithms: Although the *JavaScript Algorithms* benchmark consists of several sub-benchmarks, we will discuss them as one. The tables show that the *Rank-Based Sampling* variant significantly outperforms the baseline on 9 files. It is outperformed 2 times significantly. It performs similarly on the other 8 files. The *Proportional Sampling* variant outperforms the baseline on 13 files and is outperformed 0 times. The *Proportional Sampling* outperforms the *Rank-Based Sampling* variant 8 times significantly.

Lodash: On the *Lodash* benchmark we observe that the *Rank-Based Sampling* variant significantly outperforms the baseline on 3 files. The baseline outperforms *Rank-Based Sampling* significantly on 1 file. There is no significant difference on the other 6 files. The *Proportional Sampling* variant performed similarly to the *Rank-Based Sampling* variant, however, the baseline never outperformed the *Rank-Based Sampling* variant. The *Rank-Based Sampling* variant outperforms the *Proportional Sampling* variant significantly on 2 files.
Moment: On the *Moment* benchmark, we find that the *Rank-Based Sampling* variant significantly outperforms the baseline on 10 files. There is no significant difference on the other 9 files. The same results hold for the *Proportional Sampling* variant. The *Proportional Sampling* variant outperforms the *Rank-Based Sampling* variant significantly on 1 file and is outperformed significantly on another file.

Area Under the Curve

For some of the benchmark files, the results are similar for all variants. However, as noted in Chapter 6 one of the variants may reach the achieved structural coverage earlier in the process. To investigate this AUC is often used. However, after careful investigation of the AUC results, we found no significant difference with the results found in Table 7.1. For this reason, this study has not included the AUC results.

Summary

To summarise the findings of Table 7.1 and Table 7.2:

- The SA-*only* *Rank-Based Sampling* variant outperforms the baseline 24 times of which 24 are significant. It is outperformed 7 times of which 6 are significant. It has a tie 26 times.

- The SA-*only* *Proportional Sampling* variant outperforms the baseline 31 times of which 31 were significant. It is outperformed 0 times. It has a tie 26 times.

- The SA-*only* *Proportional Sampling* variant outperforms the SA-*only* *Rank-Based Sampling* variant 16 times of which 16 were significant. It is outperformed 1 of which 1 significant. It has a tie 40 times.

7.1.2 Discussion

Generally, given the results, we can say that the usage of type inference during test case generation improves the test case generation capabilities of SYNTEST-JAVASCRIPT. It is also clear that it does not provide a benefit in some situations. We can also say that *Proportional Sampling* clearly outperforms *Rank-Based Sampling* when only static analysis is used to infer types. The results indicate that using the *Proportional Sampling* variant rarely gives a disadvantage over both the baseline and *Rank-Based Sampling* variant, given the current benchmark.

Benchmark	File	Baseline		*Rank-Based Sampling*				*Proportional Sampling*			
				SA-only		*SA+DA*		*SA-only*		*SA+DA*	
		Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
commanderjs	help.js	0.20	0.015	0.40	0.045	0.44	0.076	0.53	0.030	0.52	0.045
commanderjs	option.js	0.44	0.111	0.33	0.056	0.50	0.000	0.50	0.111	0.50	0.000
commanderjs	suggestSimilar.js	0.66	0.219	0.55	0.156	0.55	0.188	0.75	0.031	0.75	0.062
express	application.js	0.63	0.019	0.62	0.019	0.63	0.019	0.65	0.019	0.65	0.019
express	query.js	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000
express	request.js	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000	0.30	0.000
express	response.js	0.15	0.011	0.14	0.016	0.14	0.017	0.15	0.011	0.16	0.010
express	utils.js	0.56	0.029	0.62	0.000	0.62	0.000	0.62	0.029	0.60	0.029
express	view.js	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000	0.06	0.000
js algorithms graph	articulationPoints.js	0.00	0.000	0.00	0.000	0.08	0.000	0.08	0.000	0.08	0.000
js algorithms graph	bellmanFord.js	0.00	0.000	0.17	0.000	0.33	0.125	0.33	0.167	0.33	0.000
js algorithms graph	bfTravellingSalesman.js	0.00	0.000	0.08	0.000	0.08	0.167	0.08	0.000	0.12	0.167
js algorithms graph	breadthFirstSearch.js	0.25	0.000	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000
js algorithms graph	depthFirstSearch.js	0.17	0.167	0.17	0.167	0.17	0.000	0.17	0.000	0.17	0.167
js algorithms graph	detectDirectedCycle.js	0.00	0.000	0.12	0.000	0.38	0.000	0.38	0.000	0.38	0.000
js algorithms graph	detectUndirectedCycle.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
js algorithms graph	dijkstra.js	0.00	0.000	0.10	0.000	0.20	0.000	0.10	0.100	0.20	0.000
js algorithms graph	eulerianPath.js	0.00	0.000	0.00	0.000	0.21	0.000	0.21	0.000	0.21	0.000
js algorithms graph	floydWarshall.js	0.00	0.000	0.67	0.000	0.67	0.000	0.67	0.000	0.67	0.000
js algorithms graph	graphBridges.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
js algorithms graph	hamiltonianCycle.js	0.00	0.000	0.00	0.000	0.00	0.200	0.00	0.000	0.20	0.200
js algorithms graph	kruskal.js	0.10	0.000	0.30	0.100	0.30	0.000	0.40	0.100	0.40	0.100
js algorithms graph	prim.js	0.08	0.000	0.17	0.000	0.17	0.083	0.17	0.000	0.17	0.000
js algorithms graph	stronglyConnectedComponents.js	0.00	0.000	0.00	0.000	0.38	0.500	0.25	0.000	0.38	0.500
js algorithms knapsack	Knapsack.js	0.57	0.000	0.50	0.150	0.50	0.150	0.57	0.050	0.57	0.069
js algorithms matrix	Matrix.js	0.74	0.046	0.71	0.026	0.74	0.053	0.79	0.079	0.76	0.046
js algorithms sort	CountingSort.js	0.92	0.083	0.92	0.000	0.92	0.000	0.92	0.000	0.92	0.000
js algorithms tree	RedBlackTree.js	0.21	0.000	0.26	0.000	0.26	0.000	0.26	0.029	0.26	0.029
lodash	equalArrays.js	0.08	0.000	0.71	0.042	0.67	0.083	0.75	0.042	0.75	0.042
lodash	hasPath.js	0.75	0.000	0.75	0.000	0.75	0.000	0.75	0.000	0.75	0.000
lodash	random.js	1.00	0.000	1.00	0.000	1.00	0.054	1.00	0.000	1.00	0.000
lodash	result.js	0.90	0.000	0.80	0.000	0.80	0.000	0.90	0.000	0.90	0.000
lodash	slice.js	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000
lodash	split.js	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000
lodash	toNumber.js	0.60	0.000	0.65	0.000	0.65	0.000	0.65	0.050	0.65	0.050
lodash	transform.js	0.83	0.000	0.83	0.167	0.83	0.250	0.83	0.000	0.83	0.062
lodash	truncate.js	0.38	0.000	0.59	0.029	0.59	0.000	0.59	0.000	0.59	0.000
lodash	unzip.js	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000	1.00	0.000
moment	add-subtract.js	0.00	0.000	0.71	0.071	0.71	0.054	0.71	0.071	0.71	0.071
moment	calendar.js	0.05	0.000	0.45	0.091	0.45	0.000	0.45	0.091	0.45	0.091
moment	check-overflow.js	0.05	0.000	0.60	0.000	0.60	0.050	0.60	0.000	0.60	0.000
moment	compare.js	0.14	0.000	0.14	0.000	0.14	0.000	0.14	0.000	0.14	0.000
moment	constructor.js	0.38	0.000	0.56	0.000	0.56	0.000	0.56	0.031	0.56	0.062
moment	date-from-array.js	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000	0.88	0.000
moment	diff.js	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000	0.00	0.000
moment	format.js	0.08	0.000	0.08	0.000	0.08	0.000	0.08	0.000	0.08	0.000
moment	from-anything.js	0.74	0.029	0.76	0.029	0.76	0.029	0.76	0.029	0.76	0.029
moment	from-array.js	0.02	0.000	0.04	0.000	0.04	0.000	0.04	0.000	0.04	0.000
moment	from-object.js	0.50	0.000	0.50	0.000	0.50	0.000	0.50	0.000	0.50	0.000
moment	from-string-and-array.js	0.00	0.000	0.31	0.000	0.31	0.000	0.31	0.000	0.31	0.000
moment	from-string-and-format.js	0.06	0.000	0.59	0.031	0.50	0.180	0.53	0.180	0.52	0.219
moment	from-string.js	0.06	0.000	0.16	0.000	0.16	0.000	0.16	0.000	0.16	0.000
moment	get-set.js	0.14	0.000	0.23	0.034	0.41	0.045	0.45	0.045	0.45	0.045
moment	locale.js	0.33	0.167	0.33	0.000	0.33	0.000	0.33	0.000	0.33	0.000
moment	min-max.js	0.12	0.000	0.12	0.000	0.12	0.000	0.12	0.000	0.12	0.000
moment	start-end-of.js	0.10	0.000	0.10	0.000	0.10	0.000	0.10	0.000	0.10	0.000
moment	valid.js	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000	0.38	0.000

Table 7.1: Median branch coverage together with the Inter-Quartile-Range per bench- mark file for each variant and the baseline. Superior values are marked gray

Benchmark	File	Baseline vs *SA-only Rank-Based*		Baseline vs *SA-only Proportional*		*SA-only Rank-Based* vs *SA-only Proportional*	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
commanderjs	help.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.995 (large)
commanderjs	option.js	0.00	0.206 (large)	0.00	0.646 (small)	0.00	0.922 (large)
commanderjs	suggestSimilar.js	0.51	0.519 (negligible)	0.00	0.980 (large)	0.00	0.916 (large)
express	application.js	0.00	0.232 (large)	0.00	0.588 (small)	0.00	0.830 (large)
express	query.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
express	request.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
express	response.js	0.00	0.203 (large)	0.29	0.510 (negligible)	0.00	0.801 (large)
express	utils.js	0.00	0.847 (large)	0.00	0.820 (large)	0.00	0.350 (small)
express	view.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
js algorithms graph	articulationPoints.js	1.00	0.500 (negligible)	0.00	1.000 (large)	0.00	1.000 (large)
js algorithms graph	bellmanFord.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.840 (large)
js algorithms graph	bfTravellingSalesman.js	0.00	1.000 (large)	0.00	1.000 (large)	0.08	0.530 (negligible)
js algorithms graph	breadthFirstSearch.js	0.00	0.992 (large)	0.00	0.992 (large)	1.00	0.500 (negligible)
js algorithms graph	depthFirstSearch.js	0.32	0.510 (negligible)	0.05	0.540 (negligible)	0.08	0.530 (negligible)
js algorithms graph	detectDirectedCycle.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.970 (large)
js algorithms graph	detectUndirectedCycle.js	1.00	0.500 (negligible)	0.03	0.550 (negligible)	0.03	0.550 (negligible)
js algorithms graph	dijkstra.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.690 (medium)
js algorithms graph	eulerianPath.js	0.00	0.610 (small)	0.00	1.000 (large)	0.00	0.890 (large)
js algorithms graph	floydWarshall.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
js algorithms graph	graphBridges.js	1.00	0.500 (negligible)	0.32	0.510 (negligible)	0.32	0.510 (negligible)
js algorithms graph	hamiltonianCycle.js	1.00	0.500 (negligible)	0.01	0.570 (negligible)	0.01	0.570 (negligible)
js algorithms graph	kruskal.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.630 (small)
js algorithms graph	prim.js	0.00	1.000 (large)	0.00	1.000 (large)	0.16	0.520 (negligible)
js algorithms graph	stronglyConnectedComponents.js	1.00	0.500 (negligible)	0.00	1.000 (large)	0.00	1.000 (large)
js algorithms knapsack	Knapsack.js	0.00	0.030 (large)	0.00	0.360 (small)	0.00	0.913 (large)
js algorithms matrix	Matrix.js	0.00	0.230 (large)	0.00	0.668 (medium)	0.00	0.863 (large)
js algorithms sort	CountingSort.js	0.03	0.542 (negligible)	0.01	0.572 (negligible)	0.08	0.530 (negligible)
js algorithms tree	RedBlackTree.js	0.00	0.954 (large)	0.00	0.962 (large)	0.04	0.630 (small)
lodash	equalArrays.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.740 (large)
lodash	hasPath.js	0.44	0.504 (negligible)	0.01	0.585 (small)	0.00	0.597 (small)
lodash	random.js	0.00	0.410 (small)	1.00	0.500 (negligible)	0.00	0.590 (small)
lodash	result.js	0.00	0.045 (large)	0.03	0.559 (negligible)	0.00	0.953 (large)
lodash	slice.js	0.01	0.570 (negligible)	1.00	0.500 (negligible)	0.01	0.430 (negligible)
lodash	split.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
lodash	toNumber.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.640 (small)
lodash	transform.js	0.00	0.446 (negligible)	0.00	0.688 (medium)	0.00	0.694 (medium)
lodash	truncate.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.610 (small)
lodash	unzip.js	0.03	0.550 (negligible)	1.00	0.500 (negligible)	0.03	0.450 (negligible)
moment	add-subtract.js	0.00	1.000 (large)	0.00	1.000 (large)	0.16	0.480 (negligible)
moment	calendar.js	0.00	1.000 (large)	0.00	1.000 (large)	0.32	0.497 (negligible)
moment	check-overflow.js	0.00	1.000 (large)	0.00	1.000 (large)	0.26	0.586 (small)
moment	compare.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	constructor.js	0.00	1.000 (large)	0.00	1.000 (large)	0.10	0.556 (negligible)
moment	date-from-array.js	0.32	0.510 (negligible)	0.32	0.510 (negligible)	1.00	0.500 (negligible)
moment	diff.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	format.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	from-anything.js	0.00	0.724 (medium)	0.00	0.814 (large)	0.00	0.634 (small)
moment	from-array.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
moment	from-object.js	1.00	0.500 (negligible)	0.00	0.580 (small)	0.00	0.580 (small)
moment	from-string-and-array.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
moment	from-string-and-format.js	0.00	1.000 (large)	0.00	0.950 (large)	0.00	0.194 (large)
moment	from-string.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
moment	get-set.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	1.000 (large)
moment	locale.js	0.00	0.340 (small)	0.00	0.360 (small)	0.16	0.520 (negligible)
moment	min-max.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	start-end-of.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	valid.js	1.00	0.500 (negligible)	0.08	0.530 (negligible)	0.08	0.530 (negligible)

Table 7.2: Pairwise comparison of the baseline, *Rank-Based Sampling*, and *Proportional Sampling* variant. The Wilcoxon signed-rank p-value and the Vargha-Delaney $\hat{A}_{12}$ statistic are reported. Significant p-values are marked gray

Given the five benchmark projects, it is interesting that some benchmarks benefit more from the inferred types than others. A deeper analysis of the benchmarks was required to find out why this difference exists.

The *Commander.js* benchmark consists of classes with methods that often have one of the user-defined objects as an argument. This explains why using type inference gives such a performance boost for this benchmark, as using random type arguments on such methods would often result in crashes.

Although the *Express* benchmark consists of functions with both primitive and complex type arguments, the functions often only pass the arguments to other functions, i.e., there are not many interactions with the complex type arguments. This makes it hard for the current setup to infer the types, which could explain the relatively low benefit of using type inference on this benchmark.

The *JavaScript Algorithms* benchmark contains numerous data structures and algorithms. The data structures each have their own type. The algorithms, in turn, make use of these data structures. Using type inference should thus be advantageous in achieving high structural coverage during test case generation. However, as previously shown, the type inference is beneficial only in 13 out of 19 files. After investigating the generated test cases, it turns out that the type inference variants did get the type right in most cases. However, due to the complexity of the functions at hand, the approach variants could still not cover more than the baseline. This is especially true for the 5 files where the median coverage is 0. To be more specific, the files in question make use of callback functions which are passed to an external function. These callback functions contain most of the branches. At its current state, SYNTEST-JAVASCRIPT cannot get a correct Control Flow Graph of such functions, and thus the guidance towards the branch objectives is not functioning. In other words, once the tool is improved to deal with this kind of complexity, we might find that the type inference actually does provide an advantage for the benchmark files in question.

The *Lodash* benchmark turned out to consist of functions with mostly native JavaScript type arguments. This explains why the type inference did not affect the performance of most of the files. The files that were most impacted by the type inference, i.e., *equalArrays.js* and *truncate.js*, are the two files with the most complex argument types. For example, *truncate.js* was the only function that required an options object with specific attributes.

Finally, the *Moment* benchmark. For this benchmark, there were 9 files for which the type inference did not give an advantage. Since the benchmark uses complex types, type inference is expected to be beneficial. After analysis of the benchmark, it seems that *Moment* uses a syntax where the exported functions are not stand alone as they make use of the 'this' keyword. Instead of defining these functions as methods in a class, *Moment* imports all the functions and assigns them to a moment object in the main file. This setup makes it impossible to run some of the functions in isolation. Because of this, whenever a 'this' keyword is encountered during testing, an exception is thrown, preventing SYNTEST-JAVASCRIPT from achieving any further coverage.

7.2 Incorporating Execution Information

The second research question, as stated in Chapter 6 is:

> **2. How does the incorporation of execution information impact the performance of automated test generation tools when using inferred types?**

To answer the second research question, we will again use the achieved branch coverage to measure and compare the performance of the *Rank-Based Sampling* and *Proportional Sampling* variants which use Dynamic Type Inference against the baseline. In addition, we will compare these variants to their *SA-only* counterparts. In order to make this comparison, we again make use of Table 7.1, where we can find the final branch coverage per benchmark file. We also use Table 7.3 which contains the significance statistics of the comparison between the baseline and the *SA+DA* variants. In addition, we use Table 7.4a and 7.4b which contains the significance statistics of the comparison between the *SA-only* variant and the *SA+DA* variant of *Rank-Based Sampling* and *Proportional Sampling* respectively.

7.2.1 Results

As mentioned in the previous section, Table 7.1 reports the median branch coverage per benchmark file for each variant. Additionally, the IQR is given. For the second research question, we will mainly look at the results of the 5th and 7th columns, i.e., the *SA+DA Rank-Based Sampling* variant and the *SA+DA Proportional Sampling* variant.

Table 7.3 show the Wilcoxon signed-rank p-value and the Vargha-Delaney $\hat{A}_{12}$ statistic to indicate the significance of the performance differences between the *SA+DA* variants and the baseline. Table 7.4a and 7.4b indicate the significance of the performance differences between the *SA+DA* variants and their *SA-only* counterparts.

To answer the second research question, we will review the results per benchmark and then summarize the findings.

Commander.js: Table 7.1 together with Table 7.3 show that on the *Commander.js* benchmark the *SA+DA Rank-Based Sampling* variant significantly outperforms the baseline on 2 files, one more than the *SA-only* variant. We also observe from Table 7.4a that the *SA+DA Rank-Based Sampling* variant outperforms its *SA-only* counterpart on 2 files. The baseline outperforms both variants on 1 file.

Both *Proportional Sampling* variants outperforms the baseline on all 3 files. From Table 7.4b we learn that the *SA+DA* variant is outperformed once against its *SA-only* counterpart.

The *SA+DA Proportional Sampling* variant outperforms the *SA+DA Rank-Based Sampling* significantly 2 times.

Express: For the *Express* benchmark, we find that the *SA+DA Rank-Based Sampling* variant significantly outperforms the baseline on 1 file. The baseline outperforms the *SA+DA Rank-Based Sampling* variant on 1 files. The difference is insignificant on the other 4 files.

The SA+DA *Rank-Based Sampling* variant significantly outperforms the SA-*only* variant on 2 files.

The SA+DA *Proportional Sampling* variant significantly outperforms the baseline on 3 files. The difference is insignificant on the other 3 files. The SA+DA *Proportional Sampling* variant outperforms the SA-*only* variant significantly on 1 file.

The SA+DA *Proportional Sampling* variant outperforms the SA+DA *Rank-Based Sampling* significantly 2 times. It is outperformed 1 time.

JavaScript Algorithms: From the tables, we observe that the SA+DA *Rank-Based Sampling* variant significantly outperforms the baseline on 12 files. It is outperformed 1 times significantly. It performs similarly on the other 6 files. The SA+DA *Rank-Based Sampling* variant significantly outperforms the SA-*only* variant on 7 files. They perform similarly on the other 12 files.

The SA+DA *Proportional Sampling* variant significantly outperforms the baseline on 13 files and is outperformed 0 times. The SA+DA *Proportional Sampling* variant outperforms the SA-*only* variant 4 times significantly. It is outperformed 1 time.

The SA+DA *Proportional Sampling* variant outperforms the SA+DA *Rank-Based Sampling* significantly 5 times.

Lodash: On the *Lodash* benchmark we observe that the SA+DA *Rank-Based Sampling* variant significantly outperforms the baseline on 3 files. The baseline outperforms SA+DA *Rank-Based Sampling* variant significantly on 1 file. There is no significant difference on the other 6 files. The SA+DA *Rank-Based Sampling* variant outperforms its SA-*only* counterpart 0 times and is outperformed 1 time.

The SA+DA *Proportional Sampling* variant significantly outperforms the baseline on 3 files and lost 0. The SA+DA *Proportional Sampling* performed equal to its SA-*only* counterpart.

The SA+DA *Proportional Sampling* variant outperforms the SA-*only* *Rank-Based Sampling* significantly 2 times.

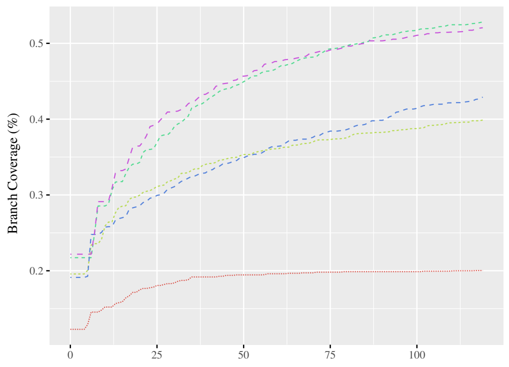
Moment: On the *Moment* benchmark, we find that the SA+DA *Rank-Based Sampling* variant significantly outperforms the baseline on 10 files. There is no significant difference on the other 9 files. The SA+DA *Rank-Based Sampling* variant outperforms the SA-*only* variant 1 time. It is outperformed 1 time.

The SA+DA *Proportional Sampling* variant also outperforms the baseline 10 times. The SA+DA *Proportional Sampling* variant outperforms its SA-*only* counterpart on 0 files and is outperformed significantly on 1 file.

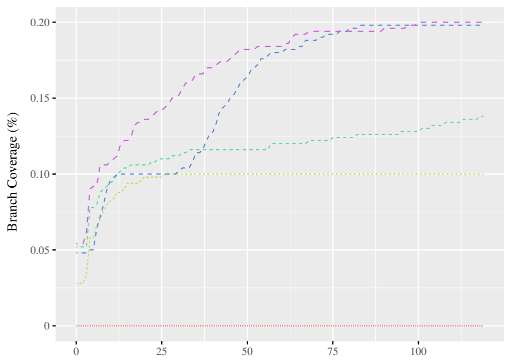
The SA+DA *Proportional Sampling* variant outperforms the SA-*only* *Rank-Based Sampling* significantly 2 times.

Coverage over Time

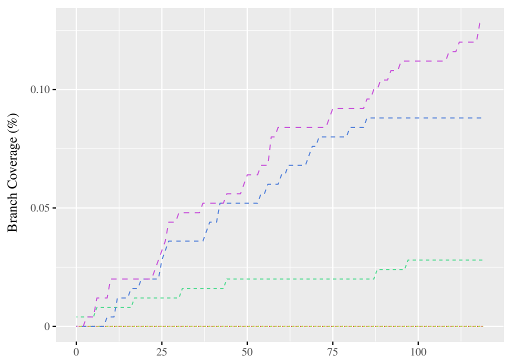
To give an impression of the performance differences between the different variants and the baseline, we can look at Figure 7.1. Here the coverage over time for some of the interesting benchmark files is shown. In Figure 7.1d the legend is given. Figure 7.1b gives a clear example of a possible limit being reached by the SA+DA variants. Figure 7.1c shows that for the *hamiltoniancycle.js* the coverage limit is not yet reached as the structural coverage seems to still be climbing.



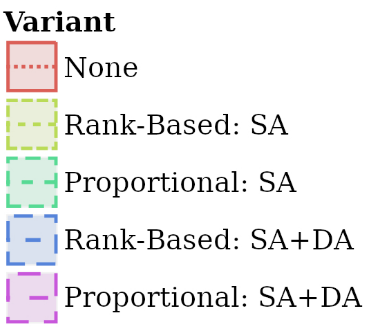
(a) Commander.js: help.js



(b) JS algorithms graph: dijkstra.js



(c) JS algorithms graph: hamiltoniancycle.js



(d) Legend

Figure 7.1: Coverage over time comparison of the approach variants on several benchmark files

Summary

To summarise the findings of Table 7.1, 7.3, 7.4a, and 7.4b:

- The *SA*+*DA* *Rank-Based Sampling* variant outperforms the baseline 28 times of which 28 are significant. It is outperformed 4 times, of which 4 are significant. It has a tie 25 times.
- The *SA*+*DA* *Proportional Sampling* variant outperforms the baseline 33 times of which 32 were significant. It is outperformed 0 times. It has a tie 24 times.
- The *SA*+*DA* *Rank-Based Sampling* variant outperforms the *SA-only* *Rank-Based Sampling* variant 12 times of which 12 are significant. It is outperformed 2 times, of which 2 are significant. It has a tie 43 times.

- The *SA*+*DA* *Proportional Sampling* variant outperforms the *SA*-*only* *Proportional Sampling* variant 5 times of which 5 were significant. It is outperformed 4 times, of which 3 are significant. It has a tie 48 times.
- The *SA*+*DA* *Proportional Sampling* variant outperforms the *SA*+*DA* *Rank-Based Sampling* variant 13 times of which 13 were significant. It is outperformed 1 time, of which 1 significant. It has a tie 43 times.

7.2.2 Discussion

Given the results, we can say that for both the *Rank-Based Sampling* and *Proportional Sampling*, the incorporation of execution information (i.e., Dynamic Type Inference), does improve their performance over the baseline. To be specific, the *SA*+*DA* *Rank-Based Sampling* variant outperformed the baseline 28 times while the *SA*-*only* *Rank-Based Sampling* variant only outperformed the baseline 24 times. The *SA*+*DA* variant also lost 3 times less against the baseline. The *SA*+*DA* *Proportional Sampling* variant outperformed the baseline 1 time more compared to the *SA*-*only* *Proportional Sampling* variant.

Furthermore, the results show that incorporating execution information does not always give an advantage over the *SA*-*only* variants. The *SA*+*DA* *Proportional Sampling* variant does outperform its *SA*-*only* counterpart 5 times, but it is outperformed significantly 3 times. On the other hand, the *SA*+*DA* *Rank-Based Sampling* variant outperforms its *SA*-*only* counterpart 13 times and is outperformed 1 time. This indicates that incorporating execution information is especially beneficial to the *Rank-Based Sampling* approach. The nature of the two approaches can explain this. The *Rank-Based Sampling* approach always uses the type with the highest likelihood. If the type with the second highest likelihood is the actual correct type, then the *Rank-Based Sampling* approach will never use the correct type. However, incorporating execution information can change the likelihoods such that the correct type gets the highest likelihood. On the other hand, the *Proportional Sampling* approach does not suffer from this problem. This is the case because the *Proportional Sampling* approach picks a random type proportional to the types' likelihood. The second type might have a lower likelihood, but it can still be picked. The *Proportional Sampling* approach thus benefits less from incorporating execution information. However, having the highest likelihood for the right type can still be beneficial in preventing repeated usage of the wrong type during the search process.

In summary, the *Rank-Based Sampling* approach can get stuck with an incorrect type if the likelihoods are not changed, which is the case if we only use static analysis. The *Proportional Sampling* approach does not suffer from this and thus benefits less from the incorporation of execution information (i.e., Dynamic Type Inference).

Benchmark	File	*Baseline* vs *SA*+*DA* *Rank-Based*		*Baseline* vs *SA*+*DA* *Proportional*		*SA*+*DA* *Rank-Based* vs *SA*+*DA* *Proportional*	
		p-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$
commanderjs	help.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.891 (large)
commanderjs	option.js	0.00	0.846 (large)	0.00	0.811 (large)	0.00	0.424 (small)
commanderjs	suggestSimilar.js	0.00	0.636 (small)	0.00	0.973 (large)	0.00	0.825 (large)
express	application.js	0.00	0.249 (large)	0.00	0.581 (small)	0.00	0.801 (large)
express	query.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
express	request.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
express	response.js	0.00	0.393 (small)	0.00	0.596 (small)	0.00	0.695 (medium)
express	utils.js	0.00	0.842 (large)	0.00	0.820 (large)	0.02	0.366 (small)
express	view.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
js algorithms graph	articulationPoints.js	0.00	1.000 (large)	0.00	1.000 (large)	0.03	0.550 (negligible)
js algorithms graph	bellmanFord.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.423 (small)
js algorithms graph	bfTravellingSalesman.js	0.00	1.000 (large)	0.00	1.000 (large)	0.02	0.552 (negligible)
js algorithms graph	breadthFirstSearch.js	0.00	0.992 (large)	0.00	0.975 (large)	0.16	0.480 (negligible)
js algorithms graph	depthFirstSearch.js	0.03	0.550 (negligible)	0.32	0.510 (negligible)	0.05	0.460 (negligible)
js algorithms graph	detectDirectedCycle.js	0.00	1.000 (large)	0.00	1.000 (large)	0.56	0.481 (negligible)
js algorithms graph	detectUndirectedCycle.js	1.00	0.500 (negligible)	0.03	0.550 (negligible)	0.03	0.550 (negligible)
js algorithms graph	dijkstra.js	0.00	1.000 (large)	0.00	1.000 (large)	0.32	0.510 (negligible)
js algorithms graph	eulerianPath.js	0.00	1.000 (large)	0.00	1.000 (large)	0.08	0.470 (negligible)
js algorithms graph	floydWarshall.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
js algorithms graph	graphBridges.js	0.16	0.520 (negligible)	0.16	0.520 (negligible)	1.00	0.500 (negligible)
js algorithms graph	hamiltonianCycle.js	0.00	0.720 (medium)	0.00	0.820 (large)	0.00	0.600 (small)
js algorithms graph	kruskal.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.810 (large)
js algorithms graph	prim.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.422 (small)
js algorithms graph	stronglyConnectedComponents.js	0.00	1.000 (large)	0.00	1.000 (large)	0.10	0.518 (negligible)
js algorithms knapsack	Knapsack.js	0.00	0.050 (large)	0.00	0.340 (small)	0.00	0.877 (large)
js algorithms matrix	Matrix.js	0.00	0.336 (small)	0.17	0.497 (negligible)	0.00	0.639 (small)
js algorithms sort	CountingSort.js	0.01	0.562 (negligible)	0.00	0.591 (small)	0.08	0.530 (negligible)
js algorithms tree	RedBlackTree.js	0.00	0.908 (large)	0.00	0.944 (large)	0.00	0.656 (small)
lodash	equalArrays.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.931 (large)
lodash	hasPath.js	0.81	0.520 (negligible)	0.00	0.584 (small)	0.00	0.575 (small)
lodash	random.js	0.00	0.370 (small)	1.00	0.500 (negligible)	0.00	0.630 (small)
lodash	result.js	0.00	0.063 (large)	0.32	0.510 (negligible)	0.00	0.938 (large)
lodash	slice.js	0.01	0.570 (negligible)	0.32	0.490 (negligible)	0.00	0.420 (small)
lodash	split.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
lodash	toNumber.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.610 (small)
lodash	transform.js	0.00	0.355 (small)	0.00	0.697 (medium)	0.00	0.778 (large)
lodash	truncate.js	0.00	1.000 (large)	0.00	1.000 (large)	0.18	0.531 (negligible)
lodash	unzip.js	0.32	0.510 (negligible)	0.08	0.530 (negligible)	0.16	0.520 (negligible)
moment	add-subtract.js	0.00	1.000 (large)	0.00	1.000 (large)	0.08	0.530 (negligible)
moment	calendar.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.414 (small)
moment	check-overflow.js	0.00	1.000 (large)	0.00	1.000 (large)	0.01	0.624 (small)
moment	compare.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	constructor.js	0.00	1.000 (large)	0.00	1.000 (large)	0.02	0.563 (negligible)
moment	date-from-array.js	1.00	0.500 (negligible)	0.32	0.510 (negligible)	0.32	0.510 (negligible)
moment	diff.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	format.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	from-anything.js	0.00	0.704 (medium)	0.00	0.848 (large)	0.00	0.715 (medium)
moment	from-array.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
moment	from-object.js	1.00	0.500 (negligible)	0.05	0.540 (negligible)	0.05	0.540 (negligible)
moment	from-string-and-array.js	0.00	1.000 (large)	0.00	1.000 (large)	0.16	0.520 (negligible)
moment	from-string-and-format.js	0.00	1.000 (large)	0.00	0.940 (large)	0.00	0.431 (negligible)
moment	from-string.js	0.00	1.000 (large)	0.00	1.000 (large)	1.00	0.500 (negligible)
moment	get-set.js	0.00	1.000 (large)	0.00	1.000 (large)	0.00	0.814 (large)
moment	locale.js	0.00	0.350 (small)	0.00	0.340 (small)	0.32	0.490 (negligible)
moment	min-max.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	start-end-of.js	1.00	0.500 (negligible)	1.00	0.500 (negligible)	1.00	0.500 (negligible)
moment	valid.js	1.00	0.500 (negligible)	0.32	0.510 (negligible)	0.32	0.510 (negligible)

Table 7.3: Pairwise comparison of the baseline, *SA*+*DA* *Rank-Based Sampling*, and *SA*+*DA* *Proportional Sampling* variant. The Wilcoxon signed-rank *p*-value and the Vargha-Delaney $\hat{A}_{12}$ statistic are reported. Significant *p*-values are marked gray

Benchmark	File	*SA-only* vs *SA+DA*	
		p-value	$\hat{A}_{12}$
commanderjs	help.js	0.00	0.681 (medium)
commanderjs	option.js	0.00	0.990 (large)
commanderjs	suggestSimilar.js	0.00	0.582 (small)
express	application.js	0.03	0.527 (negligible)
express	query.js	1.00	0.500 (negligible)
express	request.js	1.00	0.500 (negligible)
express	response.js	0.00	0.712 (medium)
express	utils.js	0.08	0.470 (negligible)
express	view.js	1.00	0.500 (negligible)
js algorithms graph	articulationPoints.js	0.00	1.000 (large)
js algorithms graph	bellmanFord.js	0.00	0.980 (large)
js algorithms graph	bfTravellingSalesman.js	0.00	0.690 (medium)
js algorithms graph	breadthFirstSearch.js	1.00	0.500 (negligible)
js algorithms graph	depthFirstSearch.js	0.05	0.540 (negligible)
js algorithms graph	detectDirectedCycle.js	0.00	0.990 (large)
js algorithms graph	detectUndirectedCycle.js	1.00	0.500 (negligible)
js algorithms graph	dijkstra.js	0.00	0.990 (large)
js algorithms graph	eulerianPath.js	0.00	0.905 (large)
js algorithms graph	floydWarshall.js	1.00	0.500 (negligible)
js algorithms graph	graphBridges.js	0.16	0.520 (negligible)
js algorithms graph	hamiltonianCycle.js	0.00	0.720 (medium)
js algorithms graph	kruskal.js	0.00	0.370 (small)
js algorithms graph	prim.js	0.00	0.690 (medium)
js algorithms graph	stronglyConnectedComponents.js	0.00	1.000 (large)
js algorithms knapsack	Knapsack.js	0.00	0.546 (negligible)
js algorithms matrix	Matrix.js	0.00	0.637 (small)
js algorithms sort	CountingSort.js	0.16	0.520 (negligible)
js algorithms tree	RedBlackTree.js	0.06	0.459 (negligible)
lodash	equalArrays.js	0.00	0.323 (medium)
lodash	hasPath.js	0.16	0.518 (negligible)
lodash	random.js	0.05	0.460 (negligible)
lodash	result.js	0.06	0.527 (negligible)
lodash	slice.js	1.00	0.500 (negligible)
lodash	split.js	1.00	0.500 (negligible)
lodash	toNumber.js	0.08	0.530 (negligible)
lodash	transform.js	0.00	0.423 (small)
lodash	truncate.js	0.01	0.567 (negligible)
lodash	unzip.js	0.05	0.460 (negligible)
moment	add-subtract.js	0.08	0.470 (negligible)
moment	calendar.js	0.01	0.567 (negligible)
moment	check-overflow.js	0.03	0.458 (negligible)
moment	compare.js	1.00	0.500 (negligible)
moment	constructor.js	0.02	0.459 (negligible)
moment	date-from-array.js	0.32	0.490 (negligible)
moment	diff.js	1.00	0.500 (negligible)
moment	format.js	1.00	0.500 (negligible)
moment	from-anything.js	0.10	0.475 (negligible)
moment	from-array.js	1.00	0.500 (negligible)
moment	from-object.js	1.00	0.500 (negligible)
moment	from-string-and-array.js	0.16	0.480 (negligible)
moment	from-string-and-format.js	0.00	0.212 (large)
moment	from-string.js	1.00	0.500 (negligible)
moment	get-set.js	0.00	1.000 (large)
moment	locale.js	0.32	0.510 (negligible)
moment	min-max.js	1.00	0.500 (negligible)
moment	start-end-of.js	1.00	0.500 (negligible)
moment	valid.js	1.00	0.500 (negligible)

(a) *Rank-Based Sampling*

Benchmark	File	*SA-only* vs *SA+DA*	
		p-value	$\hat{A}_{12}$
commanderjs	help.js	0.00	0.431 (negligible)
commanderjs	option.js	0.00	0.667 (medium)
commanderjs	suggestSimilar.js	0.32	0.476 (negligible)
express	application.js	0.71	0.498 (negligible)
express	query.js	1.00	0.500 (negligible)
express	request.js	1.00	0.500 (negligible)
express	response.js	0.00	0.590 (small)
express	utils.js	0.68	0.481 (negligible)
express	view.js	1.00	0.500 (negligible)
js algorithms graph	articulationPoints.js	0.16	0.520 (negligible)
js algorithms graph	bellmanFord.js	0.00	0.633 (small)
js algorithms graph	bfTravellingSalesman.js	0.00	0.719 (medium)
js algorithms graph	breadthFirstSearch.js	0.16	0.480 (negligible)
js algorithms graph	depthFirstSearch.js	0.08	0.470 (negligible)
js algorithms graph	detectDirectedCycle.js	0.05	0.512 (negligible)
js algorithms graph	detectUndirectedCycle.js	1.00	0.500 (negligible)
js algorithms graph	dijkstra.js	0.00	0.810 (large)
js algorithms graph	eulerianPath.js	0.05	0.540 (negligible)
js algorithms graph	floydWarshall.js	1.00	0.500 (negligible)
js algorithms graph	graphBridges.js	0.32	0.510 (negligible)
js algorithms graph	hamiltonianCycle.js	0.00	0.750 (large)
js algorithms graph	kruskal.js	0.03	0.550 (negligible)
js algorithms graph	prim.js	0.00	0.590 (small)
js algorithms graph	stronglyConnectedComponents.js	0.00	0.726 (medium)
js algorithms knapsack	Knapsack.js	0.04	0.476 (negligible)
js algorithms matrix	Matrix.js	0.00	0.340 (small)
js algorithms sort	CountingSort.js	0.16	0.520 (negligible)
js algorithms tree	RedBlackTree.js	0.16	0.524 (negligible)
lodash	equalArrays.js	0.00	0.643 (small)
lodash	hasPath.js	0.18	0.500 (negligible)
lodash	random.js	1.00	0.500 (negligible)
lodash	result.js	0.06	0.450 (negligible)
lodash	slice.js	0.32	0.490 (negligible)
lodash	split.js	1.00	0.500 (negligible)
lodash	toNumber.js	1.00	0.500 (negligible)
lodash	transform.js	0.32	0.510 (negligible)
lodash	truncate.js	0.32	0.490 (negligible)
lodash	unzip.js	0.08	0.530 (negligible)
moment	add-subtract.js	0.16	0.520 (negligible)
moment	calendar.js	0.58	0.485 (negligible)
moment	check-overflow.js	1.00	0.500 (negligible)
moment	compare.js	1.00	0.500 (negligible)
moment	constructor.js	0.05	0.476 (negligible)
moment	date-from-array.js	1.00	0.500 (negligible)
moment	diff.js	1.00	0.500 (negligible)
moment	format.js	1.00	0.500 (negligible)
moment	from-anything.js	0.05	0.562 (negligible)
moment	from-array.js	1.00	0.500 (negligible)
moment	from-object.js	0.05	0.460 (negligible)
moment	from-string-and-array.js	1.00	0.500 (negligible)
moment	from-string-and-format.js	0.00	0.451 (negligible)
moment	from-string.js	1.00	0.500 (negligible)
moment	get-set.js	0.06	0.527 (negligible)
moment	locale.js	0.16	0.480 (negligible)
moment	min-max.js	1.00	0.500 (negligible)
moment	start-end-of.js	1.00	0.500 (negligible)
moment	valid.js	0.16	0.480 (negligible)

(b) *Proportional Sampling*

Table 7.4: Comparison of *SA-only* and *SA+DA* variants

7.3 Time impact

The third and final research question, as stated in Chapter 6 is:

3. How significant is the amount of time used for the type inference?

We will look at several statistics to answer the third and final research question. The first is the mean time it takes per benchmark to perform the static analysis type inference. The mean and standard deviation can be found in Table 7.5a. Table 7.5b shows the number of elements, relations, and types per benchmark file.

Next, we will look at the mean number of evaluations the variants can perform given the time budget of 120 seconds. However, since the *Proportional Sampling* variant and *Rank-Based Sampling* variant do not differ in their type inference processes, except for how the types are being used, the difference in evaluations between the two is irrelevant and not interesting. Thus we will only be comparing the number of evaluations between the two *Proportional Sampling* variants and the *baseline*. These statistics are shown in Table 7.6.

Benchmark	Mean (seconds)	Standard deviation
Commander.js	3.00	0.0865
Express	2.26	0.1301
JavaScript Algorithms	4.66	0.1584
Lodash	2.28	0.1335
Moment	9.78	0.2701

(a) Mean static analysis type inference time and standard deviation per benchmark project in seconds

Benchmark	Elements	Relations	Types
Commander.js	4623	3566	40
Express	7606	5640	139
JavaScript Algorithms	11377	9606	96
Lodash	9094	6567	19
Moment	35727	19701	236

(b) Number of elements, relations, and types per benchmark project

Table 7.5: Type related statistics for all benchmarks

7.3.1 Results

To investigate the significance of the time used by the type inference, we will first look at Table 7.5a. This table shows that the mean number of seconds used to perform static analysis for type inference differs per benchmark. The *Moment* benchmark requires the most time with a mean of 9.78 seconds. The *JavaScript Algorithms* benchmark requires only half that time with a mean of 4.66 seconds. For the *Commander.js* benchmark the mean is 3.00 seconds. The *Express* and *Lodash* benchmark required a similar amount of time with 2.26 and 2.28 seconds respectively. Since the type inference is performed for the entire benchmark at once, this time is only required once per benchmark before the search process starts.

Evaluating how much extra time the type inference requires during the test case generation process is harder to assess. By looking at the number of evaluations per variant, we can try to assess the differences. However, there are several caveats to this. First, if for a certain UuT 100% branch coverage is reached the evaluation stops, and we move on to the next UuT. This gives incorrect results, and we can not derive conclusions from such data. There were three files in the *Lodash* benchmark where this problem was very clear since the results in Table 7.1 indicate that 100% branch coverage is reached. These benchmark files have been removed from Table 7.6. However, the table gives a strong suspicion that there are more such cases.

Table 7.6 shows that the baseline can do more evaluations than the two variants for 51 of the benchmark files. For 3 files it did significantly less evaluations. The *SA-only* variant did significantly more evaluations than the *SA+DA* variant on 30 files and significantly less on 17 files.

7.3.2 Discussion

The time impact of performing static analysis for type inference is benchmark specific. This seems to be correlated to the values given in Table 7.5b. For example, out of all the benchmarks, the *Moment* benchmark contains the most elements, relations, and types. It also has the highest mean static analysis type inference time. This correlation makes sense as more elements, relations, and types would mean that more resolving is required. A larger study is required with more benchmarks to determine if there is indeed a correlation between one or more of these values. However, in general, the time required to perform the static analysis for type inference is insignificant compared to the amount of time the search process uses. To give an example, the smallest benchmark *Commander.js* consisted of 3 files with 4 UuTs in the current experimental setup that means that the search process took $120 \times 4 = 480$ seconds. The static analysis for type inference only added 3 seconds to that. The added time becomes even less significant for larger benchmarks with more UuTs.

As mentioned in the results, several issues exist with using the number of evaluations to assess the time impact of using type inference during test case generation. These issues make it tough to draw a clear conclusion. However, the general result seems to be that using either *SA-only* or *SA+DA* type inference uses slightly more time, allowing for fewer evaluations compared to the baseline. Using *SA+DA* type inference does not clearly use more time than *SA-only* type inference, but it does lean that way as for 55% of the files, it does use more time.

Benchmark	Unit Of Test	*Baseline*		*Proportional Sampling*			
				SA-only		*SA+DA*	
		Median	IQR	Median	IQR	Median	IQR
commanderjs	help.js	1782.0	44.50	1488.5	59.50	1274.0	53.75
commanderjs	option.js	2299.5	79.25	2216.0	103.00	2260.0	109.75
commanderjs	suggestSimilar.js	2106.5	63.75	2079.5	56.00	2103.5	44.25
express	application.js	1351.0	26.75	1338.0	37.50	1344.5	31.75
express	query.js	1959.0	45.50	1913.5	37.50	1915.0	41.75
express	request.js	1534.5	19.75	1523.0	19.75	1527.0	27.00
express	response.js	391.0	12.50	388.0	11.00	389.0	13.00
express	utils.js	1858.0	60.00	1813.5	66.25	1819.5	69.75
express	view.js	1655.5	50.75	1515.0	33.25	1506.0	48.00
js algorithms graph	articulationPoints.js	2343.5	61.00	381.5	561.00	254.0	153.00
js algorithms graph	bellmanFord.js	2330.0	61.50	2114.5	88.50	1934.5	56.00
js algorithms graph	bfTravellingSalesman.js	2086.0	67.00	1927.0	54.25	1827.5	62.75
js algorithms graph	breadthFirstSearch.js	2187.5	64.00	2058.5	93.50	1983.5	74.25
js algorithms graph	depthFirstSearch.js	2153.0	66.75	2137.5	56.50	2139.5	62.75
js algorithms graph	detectDirectedCycle.js	2243.5	64.75	407.0	459.00	152.0	102.00
js algorithms graph	detectUndirectedCycle.js	50.0	0.00	50.0	0.00	50.0	0.00
js algorithms graph	dijkstra.js	2177.0	57.75	2013.0	63.00	1823.0	70.00
js algorithms graph	eulerianPath.js	2157.5	91.75	2038.5	64.50	1888.0	50.25
js algorithms graph	floydWarshall.js	2152.0	72.00	2068.5	61.00	1943.0	78.75
js algorithms graph	graphBridges.js	2269.0	88.00	305.0	573.75	203.0	204.00
js algorithms graph	hamiltonianCycle.js	2172.5	72.50	2041.0	66.25	1888.5	77.75
js algorithms graph	kruskal.js	2135.0	70.25	2023.5	58.25	1923.0	55.25
js algorithms graph	prim.js	2124.5	89.75	2025.0	55.75	1907.0	48.50
js algorithms graph	stronglyConnectedComponents.js	2104.0	73.50	1943.0	46.50	1836.5	51.00
js algorithms knapsack	Knapsack.js	2137.0	47.25	2081.0	57.25	2075.0	73.50
js algorithms matrix	Matrix.js	2204.0	64.25	2178.5	54.50	2175.0	46.50
js algorithms sort	CountingSort.js	203.0	153.00	101.0	191.25	152.0	102.00
js algorithms tree	RedBlackTree.js	2128.5	51.00	1932.0	71.50	1996.5	67.25
lodash	equalArrays.js	2293.0	49.50	2124.5	77.50	2036.5	65.50
lodash	hasPath.js	2513.5	73.25	2338.5	64.50	2331.5	81.00
lodash	random.js	-	-	-	-	-	
lodash	result.js	2501.0	88.75	2302.0	109.25	2320.0	81.50
lodash	slice.js	-	-	-	-	-	-
lodash	split.js	2487.0	65.75	2370.0	60.50	2369.0	75.75
lodash	toNumber.js	2538.0	48.00	2341.0	55.25	2344.5	77.75
lodash	transform.js	2496.5	72.75	50.0	51.00	75.5	51.00
lodash	truncate.js	2489.5	68.00	2303.0	71.00	2287.0	49.00
lodash	unzip.js	-	-	-	-	-	-
moment	add-subtract.js	2268.5	54.00	2075.0	62.00	2050.0	49.00
moment	calendar.js	2297.5	86.25	2263.0	66.50	2255.0	51.50
moment	check-overflow.js	2275.0	50.00	2222.0	69.75	2206.0	58.50
moment	compare.js	2210.0	51.75	2213.5	47.50	2217.5	65.50
moment	constructor.js	2261.5	47.00	2233.0	67.75	2237.5	74.25
moment	date-from-array.js	2316.5	55.75	2289.0	77.25	2270.5	62.00
moment	diff.js	2150.5	53.00	2150.0	50.50	2139.5	57.25
moment	format.js	2155.0	50.75	2138.5	50.50	2140.0	54.75
moment	from-anything.js	2022.0	44.50	2046.0	43.75	2027.0	42.25
moment	from-array.js	2088.0	62.50	2035.0	70.50	2010.0	41.50
moment	from-object.js	2051.5	52.25	1986.5	66.00	1971.5	48.00
moment	from-string-and-array.js	2144.0	66.75	2012.0	71.75	2011.5	50.00
moment	from-string-and-format.js	1948.0	45.25	1884.0	43.25	1886.0	72.00
moment	from-string.js	1819.5	44.50	1864.0	58.75	1844.0	48.25
moment	get-set.js	2071.0	61.00	2065.0	50.50	2050.5	83.75
moment	locale.js	2097.0	71.25	2063.5	50.75	2076.5	59.50
moment	min-max.js	2061.0	52.75	2027.0	44.50	2026.5	71.00
moment	start-end-of.js	2040.5	37.50	2034.0	47.00	2015.5	57.75
moment	valid.js	2081.0	47.25	2070.0	48.50	2083.5	48.75

Table 7.6: Median number of evaluations together with the Inter-Quartile-Range per benchmark file for the *Proportional Sampling* variant and the baseline. Superior values are marked gray

Chapter 8

Conclusions and Future Work

In this chapter, we will answer the research questions from Chapter 6 and draw conclusions using the results from Chapter 7. Afterward, we discuss some ideas for future work.

8.1 Conclusions

Given the results described in Chapter 7, we gain insights concerning the research questions posed in Chapter 6.

> **1. What is the performance impact of using inferred types versus random types on the test coverage generated by automated test case generation tools?**

As described in Chapter 4, we have implemented three variants of the approach to answer the first research question. The first variant (baseline) does not use any inference and uses randomly sampled types for the test case generation. The second variant is the Static Analysis only (*SA-only*) *Rank-Based Sampling* variant. The third is the *SA-only Rank-Based Sampling* variant. Both the second and third variant use static analysis to perform type inference. During the sampling of an argument for a test case, the *Rank-Based Sampling* variant always selects the argument's type to be the type with the highest likelihood. The *Proportional Sampling* variant selects a type proportional to the type's likelihood.

The *SA-only Rank-Based Sampling* variant outperformed the baseline on 42% of the 57 benchmarks. It was outperformed by the baseline on 11%. It performed equally well or without a significant difference on 47% of the benchmarks. The *SA-only Proportional Sampling* variant outperformed the baseline on 54% of the benchmarks. It performed equally well or without a significant difference on 46% of the benchmarks. The *SA-only Proportional Sampling* variant outperformed the *SA-only Rank-Based Sampling* on 28% of the benchmarks. It lost on 2%. It performed equally well or without a significant difference on 70% of the benchmarks.

On the benchmark, compared to the baseline, the *SA-only* variants are able to achieve equal or higher test coverage during test case generation. This shows that using inferred types over random types does provide an advantage during test case generation.

From the results, we learn that for some benchmark files, the type inference is more beneficial than for others. During the analysis of these results, we observed that using type inference is most beneficial when the UuTs in the benchmark make use of complex types, i.e., the arguments for the UuTs should not only consist of primitive types. Additionally, we found that some unsupported syntax prevents the variants from achieving higher coverage. This limitation has nothing to do with the type inference, meaning that resolving this limitation could improve the success rates of the *SA-only* variants.

2. How does the incorporation of execution information impact the performance of automated test generation tools when using inferred types?

We have implemented two additional approach variants to answer the second research question. These new variants use the execution information from the search process to improve the accuracy of the type inference. In other words, next to the SA they use DA to infer types. The variants are the *SA+DA Rank-Based Sampling* and *SA+DA Proportional Sampling* variant.

The *SA+DA Rank-Based Sampling* variant outperformed the baseline on 49% of the benchmarks. It was outperformed by the baseline on 7%. It performed equally well or without a significant difference on 44% of the benchmarks. The *SA+DA Proportional Sampling* variant outperformed the baseline on 56% of the benchmarks. It lost on 0%. It performed equally well or without a significant difference on 44% of the benchmarks. These results show that both the *Rank-Based Sampling* and *Proportional Sampling* variant benefit from the Dynamic Type Inference since the percentage of benchmarks where they outperformed the baseline grew compared to the *SA-only* variants.

To dive deeper in the effectiveness of Dynamic Type Inference the *SA-only* variants are compared to their *SA+DA* counterparts. The *SA+DA Rank-Based Sampling* variant outperformed its *SA-only* counterpart on 21% of the benchmarks. It lost on 4%. It performed equally well or without a significant difference on 75% of the benchmarks. The *SA+DA Proportional Sampling* variant outperformed the *SA-only Proportional Sampling* on 9% of the benchmarks. It lost on 5%. It performed equally well or without a significant difference on 86% of the benchmarks. From this, we can conclude that for the *Rank-Based Sampling* variant the Dynamic Type Inference is more beneficial than for the *Proportional Sampling*.

The *SA+DA Proportional Sampling* variant still outperformed the *SA+DA Rank-Based Sampling* variant on 23% of the benchmarks. It lost on 2%. It performed equally well or without a significant difference on 75% of the benchmarks. So we can conclude that the *SA+DA Proportional Sampling* is superior to the *SA+DA Rank-Based Sampling* variant for the given set of benchmarks.

To conclude, the incorporation of execution information (i.e. DA) is beneficial for the *Rank-Based Sampling* variant. For the *Proportional Sampling* variant, the execution information is less helpful. The nature of these sampling methods can explain this difference. The *Rank-Based Sampling* variant always selects the type with the highest likelihood, if this type is incorrect, i.e., the inferred likelihood is incorrect, then modifying the likelihood through DA is beneficial. However, the *Proportional Sampling* variant can already select one of the other types since it proportionally selects based on the likelihood of the types.

The DA can still provide an advantage for *Proportional Sampling* by reducing the likelihood of incorrect types, but the advantage is less prominent in the results.

3. How significant is the amount of time used for the type inference?

To answer the third and final research question, we have looked at the amount of time the SA type inference takes and how the SA and DA type inference influence the number of evaluations SYNTEST-JAVASCRIPT can make in the given time budget.

We learned that for the smallest benchmark, the Static Type Inference only took 0.6% of the total time used. Since the Static Type Inference only takes place once per benchmark, this number decreases as the benchmark becomes larger or the time budget per UuT increases.

By looking at the number of evaluations SYNTEST-JAVASCRIPT was able to make using the different approach variants, we concluded that the baseline can do more evaluations within a given time budget than the variants. Additionally, we found that the *SA+DA* variants use slightly more time than the *SA-only* variants resulting in fewer evaluations.

In conclusion, the amount of time required to perform the type inference is insignificant compared to the total time. Additionally, given the added benefit of using the type inference, the little extra time is worthwhile.

Generally, we conclude that the combination of unsupervised Static Type Inference and Dynamic Type Inference has a positive performance impact on test case generation with regard to achieved structural coverage. In addition, the unsupervised type inference is shown to be a time inexpensive process.

8.2 Future work

Although the usage of type inference is shown to be beneficial to the test case generating capabilities of SYNTEST-JAVASCRIPT, it is clear that there is still much work to be done. For starters, as mentioned in Chapter 7, in its current state, the tool does not fully support all JavaScript syntax (e.g., function definitions within functions). This creates the problem that the CFG is incorrect for some branch coverage objectives. This, in turn, disables the guidance for the search algorithm, making it significantly harder for the tool to cover all objectives. Since several of the benchmark files seem to have a common structural coverage limit, solving this issue would allow the tool to achieve higher coverage. It could also mean that, once this limitation is resolved, the type inference variants can actually outperform the baseline on the benchmarks for which they now perform equally.

Another improvement for SYNTEST-JAVASCRIPT can be found in the DA. In its current state, the *SA+DA* variants outperform the baseline more than their *SA-only* counterparts, but only slightly. Especially for the *Proportional Sampling* variant, the advantage is small. Currently, the Dynamic Type Inference only compares the variable names with the error message of the thrown TypeErrors. This can be improved in future work by investigating the stack trace of the TypeError and checking for which exact test case statement the type was wrongly inferred. This would ensure that the correct type map is modified and not another similarly named variable.

All of the experiments have been performed on a benchmark consisting of 5 benchmark projects with a total of 57 files. In the future, this benchmark can be extended with additional projects of varying sizes and syntax'. This extended benchmark can be used to perform a more extensive empirical study to confirm the findings of this study. In addition, the larger benchmark can confirm the suspected correlation between the required static analysis type inference time and the number of elements, relations, and types, as mentioned in Chapter 7.

Finally, it might be interesting to try to learn the type scores used for certain relations by mining git repositories. In the current state of SynTest-JavaScript, the type scores that are assigned to elements involved in certain relations have been hand-picked based on the JavaScript experience of the authors of this study. Instead of hand-picking the scores and types per relation, it could be interesting to use statistics from a large dataset of git repositories.

Bibliography

[1] Expressions and operators - javascript: Mdn. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators.

[2] Stack overflow developer survey 2022. URL https://survey.stackoverflow.co/2022/#most-popular-technologies-language.

[3] The most popular programming languages - 1965/2021 - new update, Aug 2021. URL https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021/.

[4] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, pages 143–154, United States, 2018. Association for Computing Machinery (ACM). doi: 10.1145/3238147.3238192. URL http://www.ase2018.com. ASE 2018 : 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2018 ; Conference date: 03-07-2018 Through 07-07-2018.

[5] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385997. URL https://doi.org/10.1145/3385412.3385997.

[6] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017.

[7] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, page 459–472, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926437. URL https://doi.org/10.1145/1926385.1926437.

[8] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2013.02.061. URL https://www.sciencedirect.com/science/article/pii/S0164121213000563.

[9] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 428–452, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

[10] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.05.003. URL https://www.sciencedirect.com/science/article/pii/S0950584917304822.

[11] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), jan 2019. ISSN 1049-331X. doi: 10.1145/3293455. URL https://doi.org/10.1145/3293455.

[12] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[13] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 571–580, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985871. URL https://doi.org/10.1145/1985793.1985871.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL https://doi.org/10.1145/3182657.

[15] Justus Bogner and Manuel Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github, 2022. URL https://arxiv.org/abs/2203.11115.

[16] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.08.010. URL https://www.sciencedirect.com/science/article/pii/S0950584917304858.

[17] George Candea and Patrice Godefroid. *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*, pages 505–531. Springer International Publishing, Cham, 2019. ISBN 978-3-319-91908-9. doi: 10.1007/978-3-319-91908-9_24. URL https://doi.org/10.1007/978-3-319-91908-9_24.

[18] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, pages 320–329, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30502-6.

[19] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976. doi: 10.1109/TSE.1976.233817.

[20] William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998.

[21] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Empirical comparison of black-box test case generation tools for restful apis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 226–236, 2021. doi: 10.1109/SCAM52516.2021.00035.

[22] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Generating class-level integration tests using call site information, 2020. URL https://arxiv.org/abs/2001.04221.

[23] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-aware concolic testing of javascript programs. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 168–179, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884859. URL https://doi.org/10.1145/2884781.2884859.

[24] Bishal Kumar Dubey. Difference between typescript and javascript, May 2022. URL https://www.geeksforgeeks.org/difference-between-typescript-and-javascript/.

[25] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20 (3):611 – 639, June 2015. URL https://eprints.whiterose.ac.uk/86826/. © Springer Science+Business Media New York 2013. This is an author produced version

of a paper subsequently published in Empirical Software Engineering. Uploaded in accordance with the publisher's self-archiving policy.

[26] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL https://doi.org/10.1145/2025113.2025179.

[27] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013. doi: 10.1109/TSE.2012.14.

[28] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542528. URL https://doi.org/10.1145/1542476.1542528.

[29] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 758–769, 2017. doi: 10.1109/ICSE.2017.75.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065036. URL https://doi.org/10.1145/1065010.1065036.

[31] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 72–82, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568278. URL https://doi.org/10.1145/2568225.2568278.

[32] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051.

[33] Renáta Hodován, Dániel Vince, and Ákos Kiss. Fuzzing javascript environment apis with interdependent function calls. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods*, pages 212–226, Cham, 2019. Springer International Publishing. ISBN 978-3-030-34968-4.

[34] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

[35] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. Java unit testing tool competition - seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 15–20, United States, 2019. IEEE. ISBN 978-1-7281-2234-2. doi: 10.1109/SBST.2019.00014. SBST '19: 12th International Workshop on Search-Based Software Testing , SBST '19 ; Conference date: 27-05-2019 Through 27-05-2019.

[36] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

[37] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015. doi: 10.1109/SANER.2015.7081877.

[38] John R. Koza and Riccardo Poli. *Genetic Programming*, pages 127–164. Springer US, Boston, MA, 2005. ISBN 978-0-387-28356-2. doi: 10.1007/0-387-28356-0_5. URL https://doi.org/10.1007/0-387-28356-0_5.

[39] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 449–459, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635913. URL https://doi.org/10.1145/2635868.2635913.

[40] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In Aldeida Aleti and Annibale Panichella, editors, *Search-Based Software Engineering*, pages 9–24, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59762-7.

[41] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315, 2019. doi: 10.1109/ICSE.2019.00045.

[42] Ahmed Mateen, Marriam Nazir, and Salman Afsar Awan. Optimization of test case generation using genetic algorithm (ga). *ArXiv*, abs/1612.08813, 2016.

[43] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification & Reliability*, 14:105–156, 2004.

[44] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Deep similarity learning-based type inference for python. *CoRR*, abs/2101.04470, 2021. URL `https://arxiv.org/abs/2101.04470`.

[45] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102595.

[46] Shayma Mustafa Mohi-Aldeen, Safaai Deris, and Radziah Mohamad. Systematic mapping study in automatic test case generation. *New Trends in Software Methodologies, Tools and Techniques*, pages 703–720, 2014.

[47] Esmaeel Nikravan, Farid Feyzi, and Saeed Parsa. Enhancing path-oriented test data generation using adaptive random testing techniques. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 510–513, 2015. doi: 10.1109/KBEI.2015.7436097.

[48] Mitchell Olsthoorn, D.M. Stallenberg, A. van Deursen, and A. Panichella. Syntest-solidity: Automated test case generation and fuzzing for smart contracts. In *The 44th International Conference on Software Engineering - Demonstration Track*, Proceedings - International Conference on Software Engineering, pages 202–206. IEEE / ACM, 2022. doi: 10.1109/ICSE-Companion55297.2022.9793754.

[49] M.J.G. Olsthoorn, A. van Deursen, and A. Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, pages 1224–1228. ACM/IEEE, 2020. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3418930. Virtual/online event due to COVID-19; 35th IEEE/ACM¡br/¿International Conference on Automated Software Engineering (ASE '20),, ASE '20 ; Conference date: 21-09-2020 Through 25-09-2020.

[50] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102604.

[51] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi: 10.1109/TSE.2017.2663435.

[52] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27(7):170, Sep 2022. ISSN 1573-7616. doi: 10.1007/s10664-022-10207-5. URL https://doi.org/10.1007/s10664-022-10207-5.

[53] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677009. URL https://doi.org/10.1145/2775051.2677009.

[54] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010. URL http://www.isoc.org/isoc/conferences/ndss/10/pdf/06.pdf.

[55] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491447. URL https://doi.org/10.1145/2491411.2491447.

[56] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: Ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125, 2019. doi: 10.1109/MSR.2019.00028.

[57] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46(12):1294–1317, 2020. ISSN 0098-5589. doi: 10.1109/TSE.2018.2877664.

[58] Hideo Tanida, Tadahiro Uehara, Guodong Li, and Indradeep Ghosh. Automated unit testing of javascript code through symbolic executor symjs. *International Journal on Advances in Software*, 8(1):146–155, 2015.

[59] Laurence Tratt. Chapter 5 dynamically typed languages. volume 77 of *Advances in Computers*, pages 149–184. Elsevier, 2009. doi: https://doi.org/10.1016/S0065-2458(09)01205-4. URL https://www.sciencedirect.com/science/article/pii/S0065245809012054.

[60] Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. *International Journal of Advance Research, Ideas and Innovations in Technology*, 5(6):32–40, 2019.

[61] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane

Ducasse. Deviation testing: A test case generation technique for graphql apis. In *11th International Workshop on Smalltalk Technologies (IWST)*, pages 1–9, 2018.

[62] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[63] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks, 2020. URL https://arxiv.org/abs/2005.02161.

[64] Dinuka R Wijendra and KP Hewagamage. Analysis of cognitive complexity with cyclomatic complexity metric of software. *International Journal of Computer Applications*, 975:8887.

[65] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 598–609, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367509. doi: 10.1145/3320269.3384736. URL https://doi.org/10.1145/3320269.3384736.

Appendix A

Glossary

Dynamically Typed Language A programming language where the types of the variables are checked during run-time only. 1

Dynamic Analysis The analysis of Computer programs performed by executing them. xi, 77

Dynamic Type Inference Type inference using only Dynamic Analysis techniques. 8, 17, 28, 33, 46, 55, 58, 66, 67

Execution Information Information gathered from executing a program. 2, 17

Proportional Sampling A sampling strategy in which one draws random samples based on their likelihood. xi, 25, 28, 39, 46, 49–53, 55–61, 63, 65–67

Rank-Based Sampling A sampling strategy in which one draws samples with the highest likelihood. xi, 25, 28, 39, 46, 49–53, 55–60, 65, 66

Statically Typed Language A programming language where the types of the variables are checked during compile-time. 1

Static Analysis The analysis of computer programs performed without executing them. xi, 77

Static Type Inference Type inference using only Static Analysis techniques. 8, 17, 28, 33, 46, 67

TypeScript TypeScript is a strict syntactical superset of JavaScript which adds optional static typing to the language. 14, 29, 30