NEURAL INFERENCE OF PROGRAM SPECIFICATIONS

Elizabeth A. Dinella

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2023

Supervisor of Dissertation

Mayur Naik, Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Rajeev Alur, Zisman Family Professor of Computer and Information Science
Chris Callison-Burch, Associate Professor of Computer and Information Science
Osbert Basanti, Assistant Professor of Computer and Information Science
Shuvendu Lahiri, Senior Principal Researcher at Microsoft Research

Dedicated To Those Who Came Before Me.

"I did not invent the transistor, the microprocessor, object oriented programming, or most of the technology I work with. I love and admire my species, living and dead, and am totally dependent on them for my life and well being." - Steve Jobs

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and appreciation to those who have supported and guided me throughout my PhD journey. I have been blessed with a group of incredible mentors who have helped me grow as a researcher, scientist, and person.

Firstly, I am profoundly thankful my advisor Mayur Naik for many years of excitement and humor. I appreciate his expertise, unwavering support, and for allowing me to pursue my unconventional interests. This encouragement and freedom paved the way for shaping my research agenda in my PhD and beyond. I also extend my gratitude to the members of my thesis committee for their valuable insights. Their feedback has greatly enriched the quality of my work over the years. I am deeply grateful to Shuvendu Lahiri for his mentorship during my internships at Microsoft Research. His patience, guidance in selecting meaningful research problems, and unwavering belief in my abilities have been incredibly helpful in setting me up for success in my PhD. Additionally, I would like to express my gratitude to Tal Rabin for her mentorship, support, and love. Lastly, I want to thank Ana Milanova, for instilling in me an excitement for program analysis.

I would also like to thank my collaborators whose expertise and friendship were invaluable to this research. I am particularly grateful to Gabriel Ryan, Hanjun Dai, Todd Mytkowicz, Ziyang Li, Saswat Anand, Aravind Machiry, and Shashank Sharma.

Next, I would like to thank my friends in the Penn Blockchain community and TA team. I extend my sincere thanks to my Web3 Security students for making the class a joyful and exciting learning experience for us all.

Lastly, I would like to thank my classmates and peers for engaging discussions, shared experiences, and collaborative efforts. Your diverse perspectives have broadened my understanding and enriched my research. Namely, my lab mates, the Saturday game night crew, my fellow Dads, and Jan Arne Sparka for sharing his expertise even across the world in different time zones. Most importantly, I cannot thank my inner circle enough for supporting me throughout this experience.

ABSTRACT

NEURAL INFERENCE OF PROGRAM SPECIFICATIONS

Elizabeth A. Dinella

Mayur Naik

Ensuring program correctness is a fundamental goal in the field of software engineering. Reliable functioning of computer programs is increasingly essential in today's digital world. However, productively writing correct code for complex systems remains a significant challenge. Decades of active research in program reasoning yielded many fruitful techniques grounded in rules and formal logic. Such symbolic techniques have achieved considerable successes, but they do come with some noteworthy limitations. Firstly, many techniques require a correctness property to check against. Without an explicitly provided specification these approaches fail to perform any level of reasoning. Secondly, many techniques struggle to scale in the presence of constructs widely seen in real world programs. This dissertation aims to address these challenges by inferring program specifications through statistical patterns from a large corpus of data. Recently, deep learning techniques have achieved remarkable breakthroughs in many domains including natural language processing and image generation. Inspired by these advances, this dissertation applies techniques from the field of deep learning to program reasoning. A data driven paradigm of program specification inference is appealing as it automatically provides a definition of program correctness for a reasoning tool to check against. Furthermore, such a system can be invoked in a quick and lightweight query. This dissertation demonstrates the promise of deep learning based specification inference techniques in a variety of program reasoning tasks including *static bug finding*, *merge conflict resolution*, and *automated testing*. For each domain, it provides datasets, methodologies, neural techniques, and comparative evaluations. It also includes a detailed analysis of the tradeoffs between data driven techniques and traditional symbolic methods as well as the benefits of combining these techniques. It concludes with a summary of the insights gained and lessons learned, offering guidance for the application of specification inference to additional program reasoning domains.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

1.1 Motivation: Automated Program Analysis

Ensuring program correctness is a fundamental goal in the field of software engineering, as reliable functioning of computer programs is essential in today's digital world. Programs are now integral to safety critical systems including self driving automobiles, aircrafts, medical devices, and the management of financial transactions. As a result, the size and complexity of codebases has significantly risen. The growth of software in terms of both scale and dependence in society stimulates a need for tools and techniques that assist developers in productively developing correct software. The pursuit of these goals has been a subject of continuous research in the field of program analysis over the course of several decades. Numerous approaches have been thoroughly investigated to facilitate the production of error-free software. For instance, static bug detection, program repair, and automated testing have been successfully integrated into industry tooling. Likewise, to expedite the software development cycle, various methods have emerged including strategies for automatically resolving merge conflicts, offering autocomplete suggestions, and automated continuous integration.

Broadly, these techniques share some noteworthy limitations to full automation. Perhaps the most fundamental limitation is the absence of the intended program behavior. The question of when an arbitrary program is considered *correct*, remains an open problem. It is unclear when a correctness checking tool should report an error. For instance, should a developer be notified of any potential divide-by-zero errors? In the case of C programs, should a developer should be notified of a potential segmentation fault? The aforementioned error types are considered general purpose *safety properties* and are often tied to the underlying language rather than the particular program being analyzed. Safety properties are broadly applicable, but must be manually curated and tend to be fragile, changing as programming languages and their features evolve. Furthermore, safety properties are not comprehensive. Programmers often wish to check for properties that are specific to the application which they are developing. For example, consider a program for an online forum in which developers want to ensure that only logged-in users can make posts. Similarly, a developer creating a desktop application may want to ensure it never crashes. A developer of a smart contract on the Ethereum blockchain may want to ensure that a hacker cannot arbitrarily withdraw funds. All of the aforementioned properties go beyond safety properties as they are specific to the application being analyzed. Given an arbitrary piece of code, there is ambiguity surrounding the properties that an analysis tool should check for. Typically, application specific properties must be conveyed from the developer to the program analysis tool through a *specification*: a predicate representing the expected correct behavior of a program.

Many techniques require specifications as input from the developer, either in the form of explicit logical predicates or as incomplete specifications presented as test suites. Unfortunately, developers lacking training in formal methods may struggle to craft accurate and comprehensive logical specifications for the properties they are wishing to check. This creates a significant barrier to achieving full tool automation. To address this, some tools use heuristics for likely specifications. For example, a Java program should *likely* not throw a `Null Pointer Exception` on a non-null input. Although likely, this is not always a valid assumption during modular analysis. Heuristics like these can introduce both false positives and false negatives. False positives leave the developer to sift through false alarms which indicate incorrect program behavior, but are not actually valid. On the other hand, false negatives can be concerning as the tool can fail to report a true bug. Without having knowledge of developer intent, the automation of program analysis tools is seriously hindered. In essence, the effectiveness of a program analysis technique is inherently tied to the quality of the specification it operates with. A tool cannot check against a specification which solely exists in the developer's mind.

1.2 My Approach: Learning Specifications Through Latent Developer Intent

Deep learning provides an exciting opportunity to enhance program analysis tools. Rather than relying on handwritten rules to specify correctness criteria, this dissertation asks: can these specifications be learned directly from data? This dissertation explores leveraging large corpora of both code and natural language data to reason about latent developer intent. Massive online code sharing and version control repositories such as Github host millions of repositories in a variety of languages with over 73Mil contributors around the world [Github (2019)]. Such largescale and diverse datasets in conjunction with advances in deep learning provides a compelling opportunity for probabilistic program analysis.

Recently, the deep learning revolution has spurred a shift in research focus across many domains from symbolic approaches to probabilistic reasoning. In computer vision, conventional techniques heavily depended on hand crafted features. For example, in 1980, experts in visual perception identified an edge detection property: a discontinuity in image brightness often corresponds to a discontinuity in physical properties [Marr and Hildreth (1980)]. However, the landscape radically changed with the advent of deep neural models such as AlexNet [Krizhevsky et al. (2012)], which learn such features directly from the data, leading to significant advancements in the field.

Likewise, in Natural Language Processing (NLP), research attention has shifted from logical rule based approaches to statistical methods [Eisenstein (2019); Jurafsky and Martin (2014)]. Traditionally, NLP has involved transforming text into logical linguistic representations of meaning. An iconic example is the Eliza [Weizenbaum (1966)] chatbot, developed in the 1960s, which employed rigid regular expression based rules to respond to user inputs. Since, the emergence of deep neural models has revolutionized the field by enabling the learning of such features from data [Collobert et al. (2011)]. A recent instance, ChatGPT [OpenAI; Brown et al. (2020); OpenAI (2023)] has garnered widespread attention due to its human-like responses and impressive performance on a wide range of tasks.

This dissertation explores a similar shift in the field of program analysis. Systems which had historically been entirely logical and rule based may benefit from an integration with statistical reasoning. This approach not only offers scalability but also has the potential to generalize across programming languages and contexts, providing a more adaptable and effective approach to program analysis in the dynamic landscape of software engineering.

In this dissertation I explore learning specifications of developer intent in the following program analysis domains:

1. Static Bug Finding
2. Program Merge
3. Test Generation

Static Bug Finding. Static bug finding refers to techniques which identify bugs without executing the code being analyzed. This broad approach has demonstrated its effectiveness in discovering bugs within various contexts. For instance, static analysis has been successful in finding bugs in Google's Java code [Sadowski et al. (2018)], Meta's Hack codebase [Distefano et al. (2019a)], and safety critical C code in the automotive and avionics sectors [Cousot et al. (2005)].

Meta's static analysis tool Zolocon was designed through a manual process. The development involved careful examination of severe bugs, followed by the formulation of hand crafted rules to generalize the pattern of these bugs [Distefano et al. (2019a)]. Another prominent example is TAJS [Jensen et al. (2009a)], a well-known static analysis tool for JavaScript programs. Notably, due to the absence of a precise and universally applicable definition for what constitutes an error [Jensen et al. (2009a)], TAJS incorporates nine meticulously curated rules to identify bugs. For example, one of the rules asserts that: a program should never read an absent variable. Another asserts that: accessing a property of a `null` or `undefined` object is not permissible. These language specific "safety" properties are handwritten by domain specific experts and are unique to the JavaScript language being analyzed. Its worth noting that these rules may not be relevant in other languages like Java where such errors will be rejected by the compiler. Consequently, static bug finding in JavaScript is an exciting domain for inferring developer correctness specifications through large corpora of JavaScript data.

Program Merge. Program merge is the mechanism to integrate changes from multiple developers in collaborative development environments. As in many other program analysis domains, domain specific rules handwritten by experts constitute when a merge can safely occur. If the rules indicate that the changes cannot be safely merged, the developer is presented with a *merge conflict* which requires manual intervention. Consider JDime [Apel et al. (2012)], a structured program merge tool for Java. JDime operates under the insight that nodes on the same level in a Java AST can be safely merged. This insight, discovered by experts in program analysis and program merging, improves the effectiveness of program merge substantially. However, it does not apply to other languages such as JavaScript. In fact, there is no such analogous insight for JavaScript, suggesting poor performance of structured merge tools when the underlying language is highly dynamic. As such, this dissertation explores learning latent developer merge patterns from large corpora of data.

Test Generation. Testing is widely recognized as an important stage of the software development lifecycle. Tests can provide documentation, find bugs, and prevent regressions. However, crafting high-quality unit tests is a time-intensive and challenging endeavor. As such, extensive work has been devoted to automated unit test generation [Fraser and Arcuri (2011); Pacheco et al. (2007); Lukasczyk et al. (2020); Zalewski (2015)]. However, like most other program analysis techniques, test generation tools have no definitive knowledge of the developer's intended program behavior. As such, domain specific experts have created correctness rules for the testing framework to evaluate against. Namely, program crashes and undesirable exceptions (e.g. null dereference or out of bound array accesses) are considered incorrect. These rules are potentially costly to manually identify and must be customized for the given language or domain. Furthermore, these cannot capture the functional, application specific properties. In modular testing, tools often require two types of specifications. First, is a *precondition* which specifies a predicate that must be true *prior* to execution. Second, is a *postcondition* which specifies a predicate that must be true following execution of a prefix and is commonly expressed as an assertion. This dissertation explores the potential for learning both pre and postconditions from a large corpus of data.

In this dissertation, I propose to explore the following question: *Can large amounts of data be leveraged to infer intended program behavior in software engineering domains?* Rather than requiring rigid handwritten rules for testing predicates, safe program merge transformations, and bug patterns, this dissertation explores learning them directly from large corpora of data. To explore learning developer intent from large corpora of data there are a number of challenges. Firstly, the acquisition of large datasets is imperative. In the context of supervised learning, these datasets must possess accurate labeling. Furthermore, the data must exhibit high quality, featuring minimal noise, and encompass a multitude of examples amounting to at least several thousand. Secondly, a learning approach requires designing an effective input representation for a model to consume. Thirdly, specifications can take many forms, an effective output representation is not clear. Lastly, the task of inferring specifications must be framed as a learning problem complete with a well-defined training routine and objective function.

1.3 Contributions and Organization

In summary, this dissertation makes the following contributions organized under three program analysis domains:

1. **Static Bug Finding:**

   (a) A formulation of static bug finding as a neural graph transformation problem.

   (b) A dataset of real world JavaScript bugs and repairs mined from Github projects.

   (c) A crawler framework to continuously collect commits from Github projects.

   (d) A graph transformation model HOPPITY for multi-step bug localization and repair.

   (e) An evaluation of HOPPITY, and comparison to a symbolic static bug finding technique.

2. **Merge Conflict Resolution:**

   (a) A formulation of merge conflict resolution as a neural translation problem.

   (b) A dataset of real world merge conflict resolutions.

(c) An algorithm for localizing merge conflict resolution regions.

(d) A tool, DeepMerge, an encoder-decoder model for merge conflict resolution with a novel edit-aware input representation.

(e) An evaluation of DeepMerge on real world conflicts and comparison to a structured merge approach.

(f) A follow up work, MergeBert which addresses limitations of DeepMerge.

3. **Automated Testing:**

(a) A formulation of test oracle inference as a neural classification and ranking problem.

(b) A dataset for precondition violation classification.

(c) A transformer based model for legal/illegal input classification.

(d) A transformer based model for synthesizing postconditions as assertions over predicates.

(e) A tool, `TOGA`, for end to end test-oracle synthesis including both precondition and postcondition inference.

(f) An evaluation of `TOGA`, compared to traditionally derived test oracles.

(g) An argument for naturalness as an important property of precondition predicates.

(h) A dataset of natural, human readable, preconditions.

(i) A methodology for inferring correct and natural preconditions through program transformation.

(j) A tool which can be applied at scale to infer natural preconditions.

(k) An evaluation and comparison to a state-of-the-art precondition generation approach.

In Chapter 2 I briefly provide necessary background on these three domains (static bug finding, program merge, and test generation) as well as a background on deep learning architectures and program representations. In Chapter 3 I explore learning developer intent for the domain of static bug localization and repair. I present a neural bug localization and repair tool, Hoppity as well as a real world dataset of bugs and developer fixes. In Chapter 4 I explore learning developer intent for program merge. I present a neural approach for merge conflict resolution, DeepMerge, as well as a dataset of real world merge tuples as well as an algorithm to localize resolution regions. In Chapter 5 I explore learning specifications for test generation. In Section 5.2 I present a transformer based approach for test oracle inference, `TOGA`. In Section 5.3, I present a technique for inferring natural preconditions through program transformations as well as a dataset which can later be used for integration with neural models. I conclude in Chapter 6 with a summary of the insights gained and lessons learned, offering valuable guidance for the application of specification inference to additional program reasoning domains.

CHAPTER 2

Background

2.1 Deep Learning

In this section, we provide a preliminary discussion of some machine learning architectures. The models used in this dissertation can be grouped into **encoder-decoder** models and **graph neural networks**.

2.1.1 Encoder-Decoder Architectures

Unique challenges in downstream tasks have led to innovations in model architectures. Many tasks in natural language processing prove challenging for traditional DNNs as they assume a fixed size input and output. In machine translation, for example, the length of the output sequence is not known until inference time. Seq2Seq models refer to a broad category of models which take a sequence as input and output a sequence. They are typically implemented with encoder-decoder architectures [Cho et al. (2014c)].

Encoder An encoder takes a variable length sequence and outputs a single fixed length vector representing the input sequence. The encoder is typically implemented using LSTMs [Hochreiter and Schmidhuber (1997)] or GRUs [Cho et al. (2014a)]. The encoder outputs a hidden fixed length vector z_n.

Decoder A decoder decodes z_n into a variable-length sequence. The decoder generates the output token by token. At each step of generation it generates a hidden state h_t conditioned on the previous hidden state h_{t-1}, the previous token y_{t-1}, and the output of the encoder z_n.

$$h_t = f(h_{t-1}, y_{t-1}, z_n)$$

To obtain the token in the output sequence y_t, a softmax is performed over each h_t and a decoding strategy is selected. Decoding strategies refer to the algorithm for selecting a y_t from $softmax(h_t)$.

A natural decoding strategy, *top-1*, is to select the maximum likelihood y_t at each timestep. However, this greedy approach is not always optimal. A popular decoding strategy *beam search* finds a sequence y which maximizes the conditional probability of the entire sequence rather than the token at each step [Graves (2012)]. More recently, non-deterministic decoding strategies to increase diversity, such as nucleus sampling [Holtzman et al. (2020)] have been proven to be effective.

Transformers

A transformer is a instance of an encoder-decoder architecture [Vaswani et al. (2017)]. Transformers have been extremely effective at natural language tasks. GPT-3 is a popular instance of a transformer [Brown et al. (2020)]. Rather than RNNs or convolution, transformers rely entirely on self-attention. Attention mechanisms allow the model to attend to some parts of the input sequence during decoding while diminishing attention on others. Transformers are more effective in modeling long range dependencies then traditional encoder decoder frameworks.

2.1.2 Graph Neural Networks

Just as textual sequences require architecture decisions for effective modeling, modeling graphs also requires choices in architecture design. Graph structures can be effectively modeled using graph neural networks [Scarselli et al. (2009)]. In this approach, nodes contain data that is embedded and shared among neighbors through a process known as message passing. During message passing, nodes iteratively update their embeddings by aggregating information from neighbor nodes. GNNs have applications in biology, chemistry, and other domains where natural graph representations are present. Notably, in program analysis literature, code is often represented as a graph representing hierarchical structure.

2.2 Program Representations

Program analysis requires selecting a suitable representation of the code to reason over. Different representations offer unique advantages and challenges, often varying drastically in terms of analysis performance and speed. This dissertation employs a range of program representations, broadly categorized as either text or graphs. Textual representations typically retain syntactic details such as semicolons or brackets, but fail to capture the hierarchical structure of code. Take, for instance, the composition of a Java class, comprised of methods and the statements within them. Intuitively, a method can be seen as a member or a "child" of the class. Similarly, the statements within the method can be viewed as children of that method, forming a tree-like structure. This relationship not easily conveyed in textual form but is well expressed using a graph structure. Program analysis techniques often represent programs as graphs where nodes are program entities and edges represent various relations (data flow, control flow, call graphs, etc). However, since different languages have different hierarchical structures, program graphs are typically language specific. Representing a program as input to a machine learning model is an active area of research. Popular representations include sequences of tokens [Chen et al. (2021); Feng et al. (2020)], relational databases [Pashakhanloo et al. (2022)], and graphs [Scarselli et al. (2009)].

2.3 Program Reduction

Program reduction refers to a family of techniques which transform a program P to a smaller variant P'. Program reduction reduces P such that its reduction P' satisfies some property. For example, in test case reduction techniques such as Delta Debugging [Zeller and Hildebrandt (2002)], the reduced test cases must trigger the same failure as the full test case.

Program reduction techniques run on various program representations. C-Reduce [Regehr et al. (2012)], a popular reduction technique for compiler bugs, represents the program textually. As such, it can be used out of the box in a variety of domains beyond C languages including JavaScript and Rust [Regehr]. Other program reduction techniques such as Perses [Sun et al. (2018)] exploit the program's structure by representing the program as an AST. However, it is highly tied to the Java language and does not generalize. Perses is, on average, faster than C-Reduce as it does not waste iterations creating variants which don't compile.

`DivideZero`	Check for division by zero.
`NullDereference`	Check for null pointers passed as arguments to a function whose arguments are marked with the non-null attribute.
`uninitialized.Branch`	Check for uninitialized values used as branch conditions.
`uninitialized.Assign`	Check for assigning uninitialized values.
`uninitialized.ArraySubscript`	Check for uninitialized values used as array subscripts.

Table 2.1: Five Clang Static Analyzer Checkers Selected For Illustration.

2.4 Program Reasoning Domains

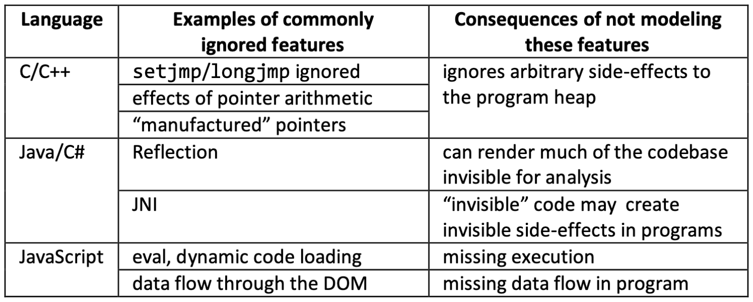
This section provides backgrounds on the three program reasoning domains explored in this dissertation: static bug finding, program merging, and test generation.

2.4.1 Static Bug Finding

Static program analysis attempts to reason about a program's possible behaviors without executing them. Only inspecting the source code, static analysis tools aim to automatically answer questions regarding program correctness, security, optimization etc. Although static analysis has been successful in many industry settings [Distefano et al. (2019a); Cousot et al. (2005); Sadowski et al. (2018)], effectively scaling static analysis has challenges. This dissertation discusses and attempts to address two major challenges. The first, is a lack of functional correctness specifications. Static analysis typically checks against a set of universal rigid handwritten rules. For example consider Clang static analyzer, the industrial strength static analysis framework for C languages built on LLVM and the Clang frontend [Lattner]. Clang static analyzer website states: *"Static analysis is not magic; a static analyzer can only find bugs that it has been specifically engineered to find."* As such, Clang static analyzer has hundreds of rules for incorrectness. A few of these "checkers" are shown in Table 2.1 Similarly, TAJS, a static analyzer for JavaScript has nine handwritten rules which their analysis checks for [Jensen et al. (2009a)].

The second challenge explored in this dissertation regards scaling static analysis. One of the main advantages of static analysis is *soundness*. Soundness implies that if a bug is present, the analysis *will* report it. However, this property rarely holds in practice. According to a group of static analysis experts: *"virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to real programming languages... we are not aware of a single realistic whole-program analysis tool that does not purposely make unsound choices"* [Livshits et al. (2015)]. Figure 2.1 details some commonly ignored program constructs in various languages.

Even scaling static analysis to handle function calls within a procedure is challenging. Google's static analysis tools check for simple property violations and do not support interprocedural analysis. According to their developer tooling experts,*"Google does not have the infrastructure support to run interprocedural or whole-program analysis."* [Sadowski et al. (2018)].



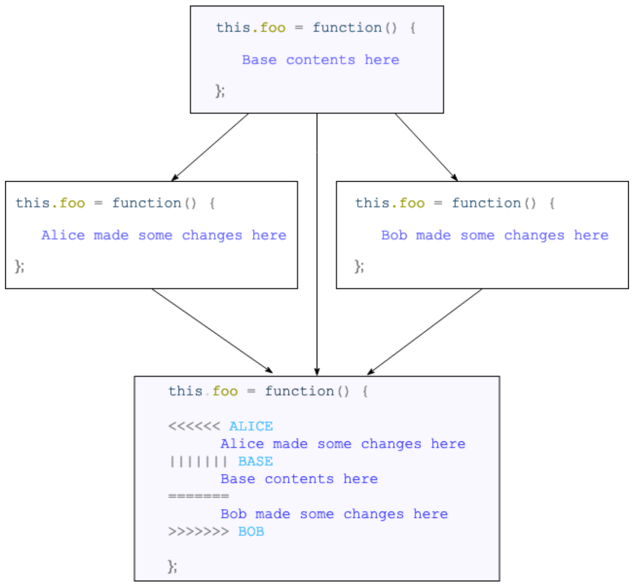
Language	Examples of commonly ignored features	Consequences of not modeling these features
C/C++	`setjmp/longjmp` ignored	ignores arbitrary side-effects to the program heap
	effects of pointer arithmetic	
	"manufactured" pointers	
Java/C#	Reflection	can render much of the codebase invisible for analysis
	JNI	"invisible" code may create invisible side-effects in programs
JavaScript	eval, dynamic code loading	missing execution
	data flow through the DOM	missing data flow in program

Figure 2.1: Program Constructs Commonly Ignored in Static Analysis. Figure from Livshits et al. (2015).

2.4.2 Program Merging

In nearly every largescale software project, many developers work on the same codebase often editing the same files. To orchestrate simultaneous editing, collaborative development environments are an essential practice. Collaborative development environments manage source code with a global shared copy of each file. Developers can edit a file by checking out a local version, making changes, and then later synchronizing the update the replica version online. Synchronization becomes challenging when multiple users make changes to the same file.

Consider a common situation (depicted in Figure 2.2) in which one developer Alice "checks out" a local copy of a global file and makes changes locally. Before she syncs her changes with the global



```
this.foo = function() {
    Base contents here
};
```

```
this.foo = function() {
    Alice made some changes here
};
```

```
this.foo = function() {
    Bob made some changes here
};
```

```
this.foo = function() {

<<<<<< ALICE
      Alice made some changes here
||||||| BASE
      Base contents here
=======
      Bob made some changes here
>>>>>>> BOB

};
```

Figure 2.2: A Three Way Merge.

version, another developer Bob, checks out a local copy of the unedited global file. Now, when both Alice and Bob attempt to sync with the global shared version, a **program merge** algorithm is invoked.

Program merge is the mechanism to integrate changes from multiple developers. The goal is to align copies such that both user's changes and the contents in the base program are respected. When the program merge algorithm cannot successfully merge changes, a *merge conflict* is reported to the user which they must manually resolve.

Current approaches for program merging can be broadly categorized by the program representation they operate over. **Unstructured merge**, [Smith (1988)], is the most widely used program merge algorithm. It is the underlying algorithm behind `git merge`. Unstructured merge represents the program as a sequence of tokens. It declares a conflict if revisions modify the same textual locations.

There are situations where changes, although occur in the same textual location, can be safely integrated (i.e. order of method declarations or import statements). Unstructured merge has no knowledge of the underlying programming language and cannot reason about higher level elements like method declarations. It simply reports a conflict if changes occur on the same line. [1]

Consider the example (Figure 2.3) where both developers revise the Java base program by adding a new function declaration. In Java, function declarations can be safely integrated in any order without affecting semantics. However, the unstructured merge algorithm declares a conflict as both revisions modify the same textual location (lines 7-9).

The benefits of an unstructured merge are its language agnostic nature and speed. An unstructured merge doesn't require language dependent handwritten rules for safe merge situations. However, by renouncing safe merge rules, unstructured merge loses knowledge of underlying semantics. Merges which don't conflict semantically can be reported to the user as a conflict.

Structured merge [Westfechtel (1991)] addresses the drawbacks of unstructured merge by incorporating knowledge of the underlying programming language. Structured merge algorithms treat the program as a graph (AST) and uses tree based merging algorithms. Structured merge algorithms exploit the insight that the order of child nodes with a common parent is arbitrary. The ordering may change without affecting the program semantics. The trees are merged using *superimposition* which merges trees recursively beginning from the root based on structural and node name similarities. Empirically, structured merge reduces number of conflicts by around 30% in Java as opposed to unstructured merge. However, structured merge still has limitations. Even with full knowledge of the underlying language, there are cases in which there is no rule for safe merging. For example, it is not clear how to merge arbitrary orders of statements within a method. In these cases, structured techniques often default back to unstructured merge, declaring a conflict if changes interfere textually. This is considered *Semi-Structured merge* [Apel et al. (2011b, 2012)]. (Semi)-Structured merge has achieved some success in static domains (i.e. JDime for Java). However, it hardly generalizes to dynamic languages. Dynamic languages such as JavaScript and Python have less semantic guarantees as compared to static languages. Additionally, the structure of modules in dynamic languages is typically more flexible. Statements can occur at the same

[1] Git merge is implemented by line-based unstructured merge, although this is not a key feature of an unstructured merge algorithm.

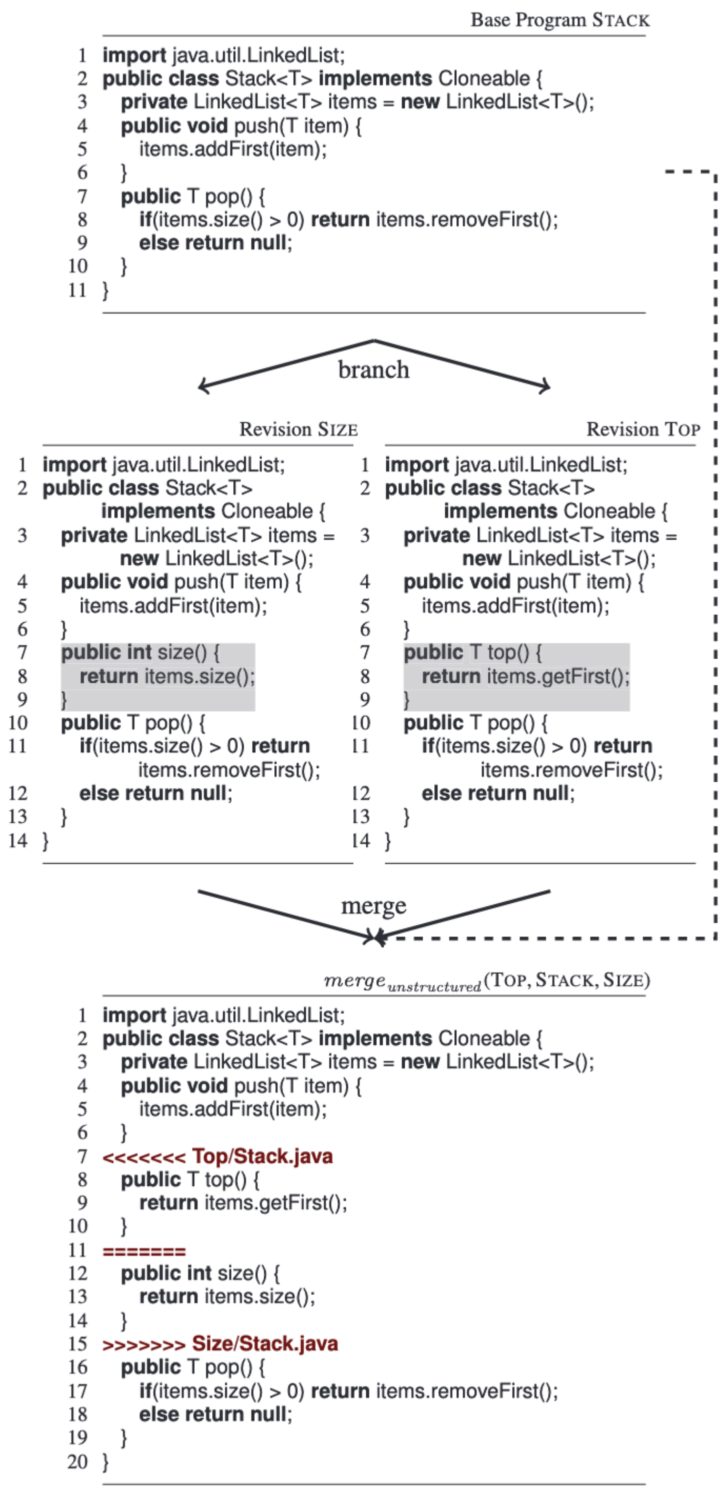
syntactic level as method declarations which causes challenges for superimposition. A semi-structured approach adapted to JavaScript, exploiting the same tree structure insight only reduces the number off conflicts by 6% as compared to unstructured merge.

2.4.3 Test Generation

Software testing is an important part of the software development lifecycle. Test generation attempts to automatically generate a test or suite of tests provide documentation, find bugs, and prevent regressions. Randoop, a well known test generation tool, generates test prefixes that drive the unit under test to an interesting state. For test oracles, Randoop uses a fixed set of rigid heuristic based rules. We show a small subset of these rules in Figure 2.4.

EvoSuite, [Fraser and Arcuri (2011)], is a similar test generation tool. However, its goal is to prevent regressions, and provide documentation, but does not target bug finding. For test oracles, EvoSuite assumes the current program behavior is correct. It generates *regression oracle* assertions that reflect the current program behavior.

In an automated bug finding scenario, it is likely that neither of these solutions suffice. As such, this dissertation explores automatically inferring test oracles.



```java
import java.util.LinkedList;
public class Stack<T> implements Cloneable {
  private LinkedList<T> items = new LinkedList<T>();
  public void push(T item) {
    items.addFirst(item);
  }
  public T pop() {
    if(items.size() > 0) return items.removeFirst();
    else return null;
  }
}
```

```java
import java.util.LinkedList;
public class Stack<T>
    implements Cloneable {
  private LinkedList<T> items =
      new LinkedList<T>();
  public void push(T item) {
    items.addFirst(item);
  }
  public int size() {
    return items.size();
  }
  public T pop() {
    if(items.size() > 0) return
        items.removeFirst();
    else return null;
  }
}
```

```java
import java.util.LinkedList;
public class Stack<T>
    implements Cloneable {
  private LinkedList<T> items =
      new LinkedList<T>();
  public void push(T item) {
    items.addFirst(item);
  }
  public T top() {
    return items.getFirst();
  }
  public T pop() {
    if(items.size() > 0) return
        items.removeFirst();
    else return null;
  }
}
```

```java
import java.util.LinkedList;
public class Stack<T> implements Cloneable {
  private LinkedList<T> items = new LinkedList<T>();
  public void push(T item) {
    items.addFirst(item);
  }
<<<<<<< Top/Stack.java
  public T top() {
    return items.getFirst();
  }
=======
  public int size() {
    return items.size();
  }
>>>>>>> Size/Stack.java
  public T pop() {
    if(items.size() > 0) return items.removeFirst();
    else return null;
  }
}
```

Figure 1: Merging the revisions Size and Top with unstructured merge

Figure 2.3: Merging the revisions `Size` and `Top` with unstructured merge [Apel et al. (2011b)]

By default, a thrown exception is not considered to indicate an error in the code under test. This is because a method may have a precondition that is unknown to Randoop. For example, a binary search implementation might require that its input is sorted. Randoop should not mark the method as buggy just because it throws an exception when Randoop passes it an ill-formed unsorted array. **Randoop has no way of knowing each method's precondition. In order to avoid outputting an "error-revealing test" that doesn't reveal an error but merely misuses the software under test, by default Randoop assumes that any exception thrown by a method is correct behavior in response to the values that it was passed.**

Checked exceptions: **default: expected**
NullPointerException when null is passed as an argument to any method in the test. **default: expected**
NullPointerException when only non-null values are provided as an argument. **default: invalid**
Unchecked exceptions other than NullPointerException, including AssertionError. **default: expected**

Figure 2.4: Randoop's Heuristics (as specified in the manual).

CHAPTER 3

Static Bug Finding

3.1 Introduction

The sheer size and complexity of modern codebases makes it impossible for them to be bug-free. As a result, a more reasonable and effective strategy has emerged, which aims to prevent bugs in production by applying automated tools to detect and even fix them early in the development process.

This trend has gained increasing popularity in recent years. Examples include Google's Tricorder [Sadowski et al. (2015)], Facebook's Getafix [Scott et al. (2019)] and Zoncolan, and Microsoft's Visual Studio IntelliCode. The techniques underlying these tools can be classified into broadly two categories: logical, rule-based techniques [Sadowski et al. (2015)] and statistical, data-driven techniques [Allamanis et al. (2018); Pradel and Sen (2018); Vasic et al. (2019)]. The former uses manually written rules capturing undesirable code patterns and scans the entire codebase for these classes of bugs. The latter learns to detect abnormal code from a large code corpus using deep neural networks. Despite great strides, however, both kinds of tools are limited in generality because they target error patterns in specific codebases or they target specific bug types. For instance, Zoncolan's rules are designed to be specifically applicable to Facebook's codebases, and deep learning models target specialized bugs in variable naming [Allamanis et al. (2018)] or binary expressions [Pradel and Sen (2018)]. Moreover, the patterns are relatively syntactic, allowing them to be specified by human experts using logical rules or learned from a corpus of programs.

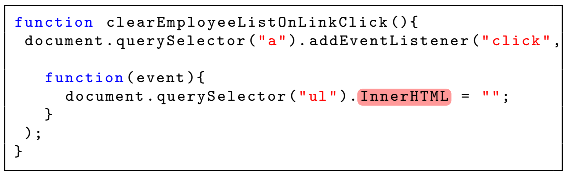
In this dissertation, we propose a novel learning-based approach for finding and fixing bugs in Javascript programs automatically. Javascript is a scripting language designed for web application development. It has been the most popular programming language on GitHub since 2014 [Github (2019)]. Repairing Javascript code presents a unique challenge as bugs manifest in diverse forms due to unusual language features and the lack of tooling support. Therefore, the primary goal of our approach is generality since it must be effective against a board spectrum of programming errors, such as using wrong operators or identifiers, accessing undefined properties, mishandling variable scopes, triggering type incompatibilities, among many others. Another important novel aspect concerns our approach's ability to deal with bugs that are more complex and semantic in nature, namely, bugs that require adding or removing statements from a program, which are not considered by prior works. Finally, compared to automated program repair techniques [Le Goues et al. (2019); Scott et al. (2019); Hua et al. (2018); Chen et al. (2018)] which require knowledge of bug location, this dissertation presents an end-to-end approach including localizing bugs, predicting the types of fixes, and generating patches.

We design our model architecture in a similar vein as a Neural Turing Machine (NTM) [Graves et al. (2014)]. It consists of an external memory (a Graph Neural Network) for embedding a buggy program and a central controller (an LSTM) that makes a sequence of primitive actions (e.g., predicting type, generating patch, etc.) to perform a fix. The multi-step decision process is implemented by an autoregressive model. Crucially, our model differs from the standard NTM in how the memory is manipulated: apart from the common read and write operations, the controller can also expand or shrink the memory when adding or deleting nodes in the original graph.

We have realized our approach in a tool called Hoppity. By training on 290,715 Javascript code change commits collected from Github, Hoppity correctly detects and fixes bugs in 9,490 out of 36,361 programs using a beam size of three.

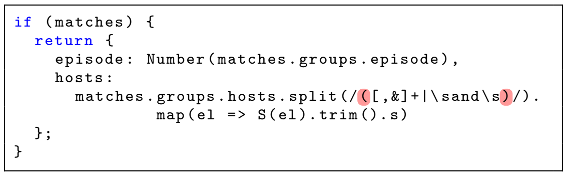
3.2 Motivating Examples

Javascript is quite different compared to traditional object-oriented languages (e.g. C++, Java, or C#). In addition to the weak, dynamic typing discipline of scripting languages, Javascript supports many peculiar features that do not exist in other languages. For example, it allows a property (i.e., a field) to be added to or removed from an object at runtime. As another example, Javascript did not support block-level scoping until recently, allowing a variable defined in a block structure such as a `for` loop to be exposed to the entire function in which the loop occurs. While the latest ES6 language standard incorporates block-level scoping, developers have been programming without it for decades, resulting in a large body of legacy code. Finally, Javascript's `eval` function, which interprets and executes a string as a code fragment, is widely regarded as a major source of bugs and vulnerabilities. All of these aspects make programming in Javascript a frustrating and error-prone experience.



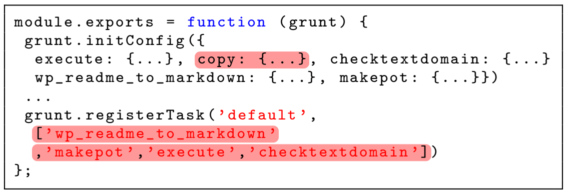
```
function clearEmployeeListOnLinkClick(){
 document.querySelector("a").addEventListener("click",

   function(event){
     document.querySelector("ul").InnerHTML = "";
   }
 );
}
```

(a) `InnerHTML` should have been `innerHTML`.



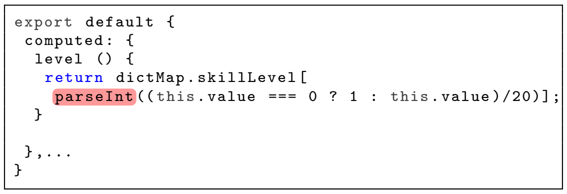
```
if (matches) {
  return {
    episode: Number(matches.groups.episode),
    hosts:
      matches.groups.hosts.split(/([,&]+|\sand\s)/).
              map(el => S(el).trim().s)
  };
}
```

(b) Highlighted parentheses should have been removed.



```
module.exports = function (grunt) {
 grunt.initConfig({
  execute: {...}, copy: {...}, checktextdomain: {...}
  wp_readme_to_markdown: {...}, makepot: {...}})
 ...
 grunt.registerTask('default',
  ['wp_readme_to_markdown'
  ,'makepot','execute','checktextdomain'])
};
```

(c) `copy` function should have also been included in the highlighted list.



```
export default {
 computed: {
  level () {
   return dictMap.skillLevel[
    parseInt((this.value === 0 ? 1 : this.value)/20)];
  }

 },...
}
```

(d) `parseInt` should have been removed because `===` implies `this.value` is an integer.

Figure 3.1: Example programs that illustrate limitations of existing approaches including both rule-based static analyzers and neural-based bug predictors.

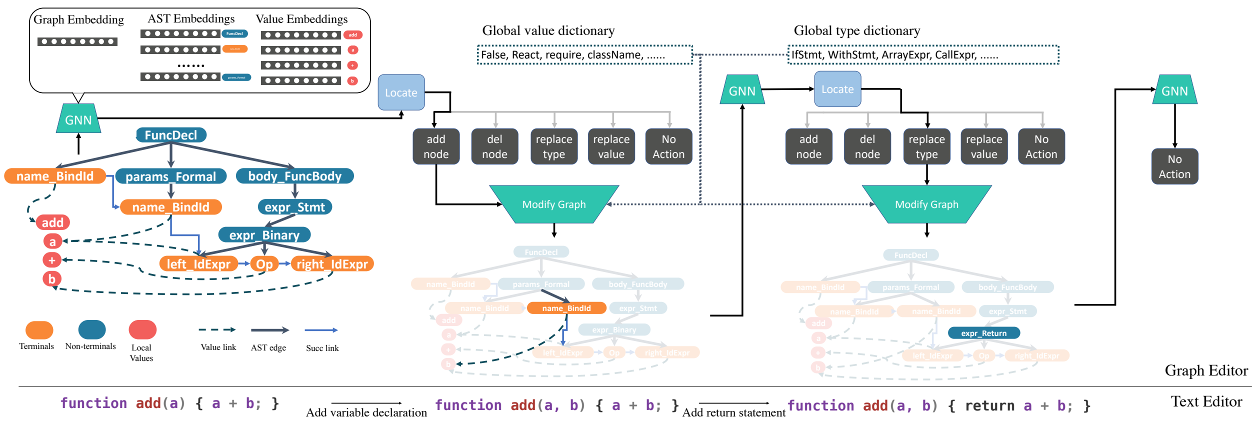
Static analyzers aim to detect common coding errors in Javascript programs by applying logical rule-based reasoning on source code. TAJS [Jensen et al. (2009b)] and ESLint [Zakas (2013)] are prominent examples. These tools face important challenges to be effective. We present several examples in Figure 3.1 to illustrate their limitations. Due to the complex nature of client-side web APIs, TAJS and ESLint choose to ignore analyzing built-in libraries for the sake of scalability. As depicted in Figure 3.1a, when developers mistakenly capitalize the first letter of `innerHTML`, a property of class `Element` in DOM (Document Object Model), both analyzers fail to catch the error. Javascript will then silently allow developers to set the previously non-existent property `InnerHTML` to the empty string. Later, when developers attempt to access the intended property, `innerHTML`, the program will crash and potentially cause a security vulnerability or incur a costly debugging experience. Additionally, static analyzers can never deal with functional bugs (i.e., errors that violate the program specification and yet conform to the coding rules). Figure 3.1b shows one such example. The goal is to split a string using regular expressions. However, the program incorrectly splits the input `' and '` into `['', ' and ', '']` instead of `['', '']`, which is what the developer intended. Since the error is simply a mismatch between the developer's implicit specification and implementation, static analyzers are incapable of catching it.

The bugs that static analyzers missed in both cases are in hindsight quite obvious to human programmers. The criteria they use is very simple: any code snippet that seems to deviate from common code patterns is likely to be buggy. This is precisely the observation that our approach seeks to mimic. In particular, if a model observes a property or an unusual way of splitting strings that never appeared in the training data, it is likely to recognize those abnormal code fragments as potential bugs. The main advantage of our approach over existing neural-based bug detectors [Allamanis et al. (2018); Pradel and Sen (2018); Vasic et al. (2019)] is its generality. Unlike prior works that target specific classes of bugs (e.g., variable naming issues or binary expression bugs), we train a single model to deal with a wide range of bug types, encompassing all previously proposed ones. Compared to past approaches that leverage a graph-based neural network model [Allamanis et al. (2018)], our model is capable of more sophisticated transformations such as adding or removing nodes, as shown in Figure 3.1c and 3.1d. Finally, our model not only locates but also fixes bugs, whereas program repair [Le Goues et al. (2019); Scott et al. (2019); Hua et al. (2018); Chen et al. (2018)] or bug localization [Ball et al. (2003); Jose and Majumdar (2011); Wang et al. (2019)] techniques only solve a single task.

3.3 Model

We model the problem of detecting and repairing bugs in programs as a structured prediction problem on a graph-based representation of programs. Given a graph g_{bug} that represents a buggy program, we wish to predict a graph g_{fix} that represents the fixed program. Our model aims to capture the structured prediction by a sequence of up to T steps of graph transformations:

$$p(g_{fix}|g_{bug};\theta) = p(g_1|g_{bug};\theta)p(g_2|g_1;\theta)\dots p(g_{fix}|g_{T-1};\theta) \tag{3.1}$$



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

Figure 3.2: Code repair as graph transformation. Each step the source code graph is edited via one of the operator module until STOP is triggered by controller.

The high-level overview of the graph sequence transformation is shown in Figure 3.2. Different programs may need a different number of steps $T(g_{bug})$ which is also determined by the model.

We first introduce our representation module for programs in Sec 3.3.1. We then elucidate each step of the above transformation in Sec 3.3.2. Finally, we summarize and present the full model in Sec 3.3.3.

3.3.1 Program Representation

Programs written in a high-level language have rich structure. Researchers have proposed graph-based representations to capture this structure [Allamanis et al. (2018)]. We start with this approach of representing programs using graphs with certain modifications for our task.

As shown in the left part of Figure 3.2, we first parse the program's source code into an abstract syntax tree (AST) form that captures the program's syntactic structure. We then connect the leaf nodes with `SuccToken` edges. Unlike previous approaches, we additionally add value nodes that store the actual content of the leaf nodes, with special `ValueLink` edges connecting them together. The purpose of introducing this additional set of nodes is to provide a name-independent strategy for code representation and modification, which we elucidate in the next section. Hereafter, we use g_{fix}, g_{bug} or g in general to represent either the source code or the corresponding graph structure.

After representing the program as a graph, we use a Graph Neural Network (GNN) [Scarselli et al.

(2008)] to map the graph into a representation in a fixed dimensional vector space. Specifically, given a graph $g = (V, E)$ with set of nodes V and edges E, we need a function $f(g) \mapsto (\mathbb{R}^d, \mathbb{R}^{|V| \times d})$ to obtain the d-dimensional representation of graph g (denoted as $\vec{g}$), as well as representations of individual nodes $v \in V$ (denoted as $\vec{v}$). To parameterize $f(\cdot)$, we employ the form in GIN [Xu et al. (2018)], with our adaptation to our multigraph for program representation in the following manner:

$$\begin{aligned} h_v^{(l+1),k} &= \sigma(\textstyle\sum_{u \in \mathcal{N}^k(v)} \mathbf{W}_1^{l,k} h_u^{(l)}), \forall k \in \{1, 2, \ldots, K\} \\ h_v^{(l+1)} &= \sigma(\mathbf{W}_2^l [h_v^{(l+1),1}, h_v^{(l+1),2}, \ldots, h_v^{(l+1),K}] + h_v^{(l)}) \end{aligned} \tag{3.2}$$

where $\mathbf{W}_1^{l,k} \in \mathbb{R}^{d \times d}, \mathbf{W}_2^l \in \mathbb{R}^{dK \times d}$ are model parameters and $\sigma(\cdot)$ is `tanh` in this implementation. K is the total number of edge types in this multi-graph representation. In the end, the node embedding is $\vec{v} = h_v^{(L)}$, where L is the total number of propagations in the GNN. $\mathcal{N}^k(v)$ is the set of neighbors of node v that are connected by edge with type k. Following GIN, the graph representation $\vec{g}$ is the aggregation of $h_v^l, \forall l \in 0, 1, \ldots, L$. We use max pooling to aggregate h_v^l for each l, and then take the average of these $L + 1$ vectors to obtain $\vec{g}$.

Initially, we use the node type as one-hot features as a starting value for $h_v^{(0)}$, where the types are either obtained from the AST representation, or from the *local value table* as shown in Figure 3.2. Note that we don't use features like variable names or function names in this graph representation, as different programs may follow different naming conventions. Instead, we focus on the syntactic structure of the source code, so as to enable naming-agnostic representation across different programs.

3.3.2 One-Step Graph Edit

There are five types of operators to choose from for a single step graph edit, namely, adding a node (`ADD`), deleting a node (`DEL`), replacing a node value (`REP_VAL`), replacing a node type (`REP_TYPE`) and stop (`NO_OP`). When combined with multi-step edits, these operators suffice to capture a rich variety of code modifications. These operators share some common low-level primitives, such as finding the location, predicting value, etc. We first introduce the individual low-level primitives and then present how to assemble these for each type of graph edit operator.

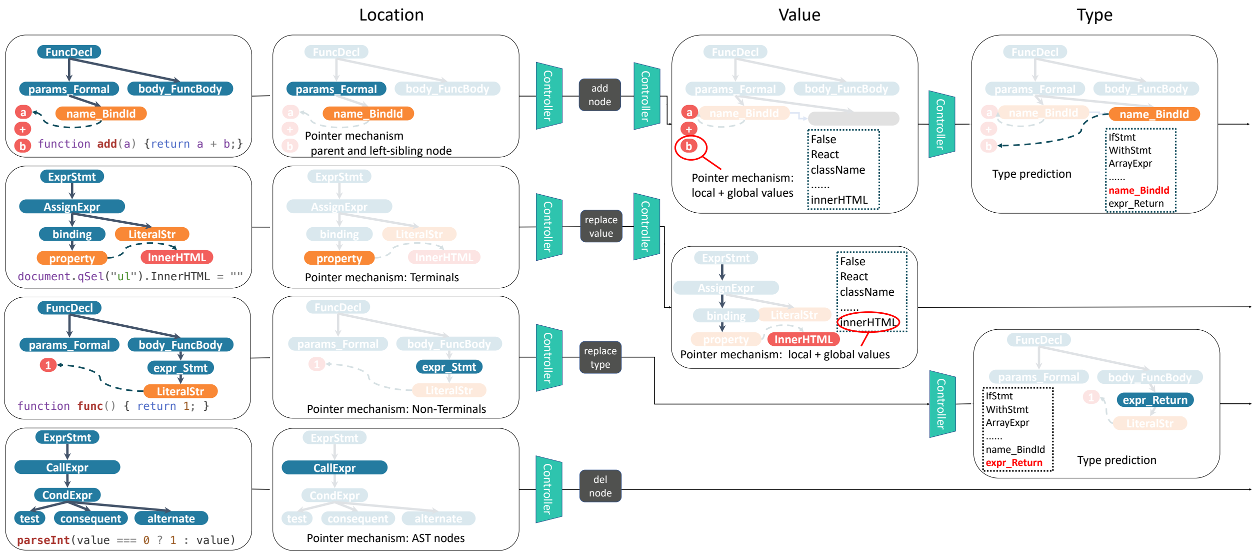


Figure 3.3: Graph edit operators with low-level primitives.

Low-level primitives

Our low-level primitives contain location, type, and value prediction. These primitives can be combined for different operators later on. In this section, we assume the availability of a controller, represented as $\vec{c} \in \mathbb{R}^d$. It keeps track of the global state, including the original source code, as well as the edits made so far. We will elaborate this when we assemble different primitives together.

Location The *location* primitive locates a specific position in the source code. While it corresponds to region selection in the original text representation, with the graph representation, we can easily treat it as a node selection step. As different programs have different numbers of nodes, we employ a pointer network [Vinyals et al. (2015b)] into the graph structure. Specifically, after obtaining the node embeddings $\{\vec{v}\}_{v \in V}$, we select the node via $loc(\vec{c}, g) = \arg\max_{v \in V} \vec{v}^\top \vec{c}$ for simplicity.

Value The *value* primitive assigns a value for a leaf node in the AST. Instead of predicting the replacement value using a language generative model [Chen et al. (2018)] or GNN score function [Allamanis et al. (2018)], we adopt the attention mechanism to let the model to choose from either the values appearing in the current file (local value table), or a collection of global values that are common for the specific language. Let D_{val} be the global dictionary of commonly used leaf-node values in the language, where each item $i_v \in D_{val}$ is associated with a vector representation $\vec{i_v} \in \mathbb{R}^d$.

The local value table is denoted as V_{val} which is a subset of the nodes in current graph. Then, the value is predicted via $val(\vec{c}, g) = \text{argmax}_{t \in D_{val} \cup V_{val}} \vec{t}^\top \vec{c}$. Again we use inner product simply for efficiency, while more expressive score functions can also be used.

Type The *type* primitive assigns the type for non-terminal nodes in an AST. As the total possible number of types is finite and fixed for a given language, the type prediction is simply a multi-class classification problem. However, we can utilize the AST grammar checker with contextual information to prune the output space. To predict the type of a given non-terminal node, we can obtain its parent node and current children. Then, by looping over the valid production rules at the current location, we can obtain a list of all valid types. The final type is only chosen from this set.

Graph edit operators

The t-th round of edit starts with the current graph g_{t-1}, the corresponding graph embedding, and the 'macro-context' embedding $\overrightarrow{c_M}_{t-1}$ that captures the edit history so far. Every type of edit operation (excluding `NO_OP`), requires prediction of the buggy location. So, in each round, the *location* primitive is invoked to determine the node to target. Then the edit type e that is feasible at this location v is predicted out of the five operators. A 'micro-context' embedding $\overrightarrow{c_{m_t}}$ is obtained from the macro embedding updated by two LSTM calls with location node embedding $\vec{v}$ and operator embedding $\vec{e}$. To summarize:

$$\overrightarrow{c_{M_t}}' = \text{LSTM}(\overrightarrow{g_{t-1}} | \overrightarrow{c_{M_{t-1}}}), \overrightarrow{c_{m_t}} = \text{LSTM}(\overrightarrow{e_t} | \text{LSTM}(\overrightarrow{v_t} | \overrightarrow{c_{M_t}})), \tag{3.3}$$

The micro-context embedding is used as the controller throughout the process of each operator. In the following content, we present these operators in detail.

`ADD` This operation adds a new node to the graph. Unlike in Li et al. (2018), where the node and corresponding edges are added in separate stages, which would introduce extra complexity, we introduce a simple mechanism that can uniquely add a node and corresponding edges. As is shown in Figure 3.3, this process invokes one *location* primitive, one *value* primitive, and one *type* primitive. The *location* primitive invoked before the edit (*i.e.*, node v) determines the parent of the node to be added, while the *location* primitive called during the edit chooses the left sibling of the node. In a special case where the parent node does not have any children, then such left sibling node is set to the parent node itself. With this information, we can uniquely determine the position to insert into the AST. Finally, the corresponding edges—`SuccToken`, `ValueLink`, and AST edges—can automatically be inferred with the location of new node to be added.

As this process is autoregressive, the micro-context embedding is kept updated with all the primitive calls. For this specific operator, the context is updated in the order of: $\vec{c_{m1}} = \text{LSTM}(\overrightarrow{v_{sibling}}|\vec{c_m})$, $\vec{c_{m2}} = \text{LSTM}(val(\vec{c_{m1}}, g)|\vec{c_{m1}})$ and $\vec{c_{m3}} = \text{LSTM}(type(\vec{c_{m2}}, g)|\vec{c_{m2}})$. In the end, $\vec{c}_{ADD} = \vec{c_{m3}}$ summarizes the process.

`DEL` This operator deletes a node and corresponding edges in the graph. If it is a non-terminal node in the AST, then the corresponding subtree is removed as well. The micro-context embedding is updated by the LSTM via the embedding of the node being deleted.

`REP_VAL` This operator replaces the value of a leaf (terminal) node in the AST. This procedure requires the prediction the value. The leaf node is linked to the new value node in the internal value table via a `ValueLink` edge. Also, the micro-context embedding is updated by the LSTM via the embedding of the corresponding node and value.

`REP_TYPE` This operator changes the type of a non-terminal node, which involves *type* primitive steps. The micro-context embedding is updated by the LSTM via the embedding of the corresponding node and type.

`NO_OP` This op does not change the graph. It simply denotes the end of the sequence of graph edits.

3.3.3 Graph Transformation

Our end-to-end model for graph transformation inference is shown in Alg 1. We denote the buggy graph g_{bug} as g_0 for simplicity. Then, for the t-th graph edit, the following steps are performed:

1. Obtain the graph representation $\overrightarrow{g_{t-1}}$ and node embeddings. Update the macro-context embedding using $\overrightarrow{g_{t-1}}$;

2. Choose edit location v_t by performing *location* primitive and update the context embedding;

3. Pick the graph edit operator e_t that is compatible with v_t; Use both v_t and e_t to obtain the micro-context embedding.

4. Perform the edit, obtain the corresponding micro-context summary $\overrightarrow{c_{e_t}}$ and update the macro-context embedding.

5. If the edit is not `NO_OP`, then go back to step 1; otherwise return the graph.

This process repeats until it reaches the maximum steps T or the `NO_OP` operator is selected. Note that our framework can capture the situation when the input program is bug-free. In this case, the `NO_OP` operator is supposed to be triggered at the first step. Also, each edit step is not limited to a single node level operation. It can be extended to modify a certain substructure (e.g., replace a tree node with one of its children). This in turn allows program repair to be performed in fewer edit steps.

3.4 Learning

Given the dataset $\mathcal{D} = \{(g_{bug}^{(i)}, g_{fix}^{(i)})\}_{i=1}^{|\mathcal{D}|}$ which consists of pairs of buggy code and the fixed code, the learning objective $\max_\theta \mathbb{E}_{(g_{bug}, g_{fix}) \sim \mathcal{D}} p(g_{fix}|g_{bug}; \theta)$ maximizes the likelihood of fixes.

Since the probability is factorized according to Eq 3.1 where a sequence of transformations is performed, we parse the source code using the SHIFT AST format, and utilize a JSON diff toolbox to compile the code differences into a sequence of AST edits. This serves as the fine-grained supervision mechanism for our graph transformation formulation. Thus, the MLE objective above is realized with the sum of cross entropy loss at each step of graph edits. During training, we jointly optimize the graph representation module $\{f_t(\cdot)\}_{t=1}^{T}$, each of the operator module and the controller module which is parameterized by LSTM. We use the Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.99$ and initial learning rate of 10^{-3}. Due to the large size of each sample, we use a small batch size of 10 during training. Furthermore, to stabilize the training, we apply the gradient clip with the maximum norm of 5.

3.5 Inference

The inference procedure involves searching for the maximum in the combinatorial space: $\arg\max_{g_{fix}} p(g_{fix}|g_{bug};\theta)$. Since the search space is very large, however, we use beam-search to approximately find the fixes with highest probabilities.

Specifically, we maintain a pool of partially fixed programs $\{\tilde{g}\}$, which starts with simply the single buggy program g_{bug}. The pool size is limited by the beam-search size B. For each $\tilde{g}$, we propose the top B locations to be modified, top B operators or top B primitives (*location*, *type*, *value*), depending on the current stage of the edit $\tilde{g}$. Then the total joint one-step graph transformation solutions are ranked together based on the joint log-likelihood, and the top B solutions with the largest likelihood are kept in the pool for the next round of beam search.

Unlike beam search for language models where the vocabulary size is fixed, in our setting, the available choices or even the steps of inference may vary (e.g., the `ADD` operator has more steps of primitive calls than the `DEL` operator). Our implementation is based on PyTorch with customized GPU kernels to enable efficient inference on GPUs.

3.6 Evaluation

Dataset Our model is trained and evaluated on a corpus of nearly half a million data points. We have created a robust system to continuously collect small changes in Javascript programs from Github. Given a commit, we download the Javascript file before and after the change: $(src_{buggy}, src_{fixed})$. Commits can contain many types of changes such as feature additions, refactorings, bug fixes, etc. In an attempt to filter our dataset to only include bug fixes, we use a heuristic based on the number of changes to the AST. Our insight is that a commit with a smaller number of AST differences is more likely to be a bug fix than a commit containing large changes. Thus for the experiments, we use three different datasets: `OneDiff` with precisely one edit; `ZeroOneDiff` with zero and one edit and `ZeroOneTwoDiff` with zero, one or two edits. We additionally filter out data points with ASTs larger than 500 nodes as a parameter in our system. A

	`ADD`	`REP_TYPE`	`REP_VAL`	`DEL`	total
train	6,473	1,864	251,097	31,281	290,715
validate	790	245	31,357	3,957	36,349
test	796	233	31,387	3,945	36,361

Table 3.1: Statistics of the `OneDiff` Dataset.

detailed overview of our corpus crawler is available in Appendix A.1.2.

Experiments. We train the model for 3 epochs on the training set until the validation loss converges. We tried different configurations of our model with different number of layers and different graph embedding methods besides the generic one in Eq 3.2. We report on these ablation studies in Appendix A.1.3.

Table 3.2 shows the evaluation results of our model on a held out test set consisting of samples from our `OneDiff` dataset. Additional experiments on `ZeroOneDiff` and `ZeroOneTwoDiff` datasets are available in Appendix A.1.1. We also provide experimental results with respect to different configurations.

Accuracy is shown for each graph edit operation type. Accuracy is measured in a complete discrete graph edit operation step. For example consider Figure 3.1a, in which we edit an object property name with the `REP_VAL` operation. If the model incorrectly predicts the edit operation to be of type `DEL`, then it will not go on to predict a *value*. In this case, the model will be penalized twice in the operation accuracy as well as the value accuracy. A prediction is considered totally correct only if the entire sequence of graph edit primitives is correct. Note that top-1 greedy prediction is not always among top-3 when beam search is used. Additionally, operator prediction is only evaluated on the top prediction as the search space only includes four operators.

To demonstrate the magnitude of the search space, we compare Hoppity to a model that selects uniformly at random, in each step of the graph edit process. The random model performs well at operation type selection since the search space only has four options (`ADD`, `REP_VAL`, `REP_TYPE`, `DEL`). However, after the operation type is predicted, the random model's accuracy drops, as there are up to 500 nodes in the buggy AST. When it predicts value, the accuracy drops even further as our vocabulary contains 5,000 values. Lastly, type prediction has slightly better accuracy than value

	Total		*Location*		*Operator*	*Value*		*Type*	
	Top-3	Top-1	Top-3	Top-1	Top-1	Top-3	Top-1	Top-3	Top-1
TOTAL	**26.1**	14.2	35.5	20.4	34.4	52.3	29.1	76.1	66.7
`ADD`	52.9	39.2	69.6	51.4	70.6	65.7	55.1	76.8	68.5
`REP_VAL`	23.4	11.9	33.3	18.5	31.7	53.0	28.8	-	-
`REP_TYPE`	71.7	52.4	73.0	52.8	79.4	-	-	74.7	61.0
`DEL`	39.6	24.8	44.0	27.5	45.8	-	-	-	-
Random	.08	.07	2.28	1.4	27.7	.01	.01	.27	0

Table 3.2: Evaluation on the `OneDiff` Dataset: Accuracy (%).

prediction because the number of the types of AST nodes in total is smaller than our vocabulary.

Baselines

As existing approaches cannot be applied for comparison in Table A.1, we adapt the baselines to some restricted settings in this section. We report the results on the `OneDiff` dataset as most of the baselines target repair of a single bug. Note that for all comparisons we provide equal amounts of information to Hoppity and the baseline without retraining our model.

GGNN: Allamanis et al. (2018), uses Gated Graph Neural Networks (GGNN) for two specific bug repair tasks: VARMISUSE, in which the model learns to select the correct variable that should be used at a given location, and VARNAMING, in which the model predicts a variable name based on its usage. We adapt these tasks to compare with Hoppity on the `REP_TYPE` and `REP_VAL` tasks. Specifically, for `REP_TYPE` prediction we have

- GGNN-Rep: we adopt VARMISUSE to replace with candidate node type and modify the graph structure correspondingly; we use their proposed max-margin formulation for training.
- GGNN-Cls: we perform multi-class classification using the target node and graph embedding.

For `REP_VAL` prediction, we also made two versions of adaptations:

- GGNN-Rep: similar to above, here the candidate set is from values in the current graphs plus the top-100 frequent values used for repair in the training set.

Type	GGNN-Rep	GGNN-Cls	Hoppity
Top-1	53.2%	**99.6%**	90.0%
Top-3	85.8%	**99.6%**	94.8%

Table 3.3: Accuracy of `REP_TYPE` with location and operation type given.

	Top-1	Top-3
Hoppity	**67.7%**	**73.3%**
SequenceR	64.2%	68.6%

Table 3.4: OneDiff accuracy with location.

Value	GGNN-Rep	GGNN-RNN	Hoppity
Top-1	63.8%	60.3%	**69.1%**
Top-3	67.6%	63.6%	**73.4%**

Table 3.5: Accuracy of `REP_VAL` with location and operation type given.

Bug Type	Amount	TAJS	Hoppity
Undefined Property	7	0	1
Functional Bug	11	0	3
Refactoring	12	0	1
Total	30	0	5

Table 3.6: Comparison with TAJS.

- GGNN-RNN: we adopt VARNAMING approach to predict value directly. Due to the huge vocabulary size, we use char-level language model for predicting the replacement.

Table 3.3 and 3.5 show the comparison when buggy node is known. Regarding the type prediction, as the number of types is large, the likelihood formulation with classification objective outperforms the max-margin loss based one (*i.e.*, GGNN-Rep). As in this limited case GGNN-Cls and Hoppity are quite similar except for graph representation, the performance is expected to be comparable. As Hoppity is not trained to predict type fix only, it performs slightly worse than GGNN-Cls. Also for the value prediction, our formulation of pointer on graph is more effective. We found when the space of decisions is large, it is hard to apply structured prediction method like GGNN-Rep in this setting. Since real-world programs are noisy, the sentences used in different programs vary greatly, making it difficult for language models to predict the exact accurate value. A possible extension is to combine the language model with the graph pointer, which we will explore in future work.

SequenceR: The model proposed by Chen et al. (2018) is a translation based model that predicts a fixed sequence of tokens when given a buggy line in the source code. We compare with our model by providing location information to both approaches.

Table 3.4 summarizes the total accuracy for fixing a single bug. In order to provide a fair comparison, we allow SequenceR to predict the same information as our model (*i.e.*, predict op, value *etc.*in a sequential way), rather than an entire sequence of raw textual tokens. This experiment shows the benefit of formulating code repair with graphs over text tokens.

With the above two baselines, we can see that in the restricted case our model can still yield comparable or even better performances. Given that our model can go for more edits without location information, we believe this tool is more generic and effective for code repair.

TAJS: We also compare the bug detection ability of Hoppity against TAJS [Jensen et al. (2009b)] which is a well-known static analysis tool for Javascript programs. Automating the comparison for our entire test set proved to be infeasible. For example, TAJS only accepts JavaScript ES5 programs, while the vast majority of current JavaScript projects use ES6 or other variants like React JSX. Another problem is that TAJS does not analyze code that is not invoked, e.g., a library function that is not called by client code. Moreover, determining the right command-line options of TAJS is non-trivial since it provides many options targeting different JavaScript runtime environments.

Due to these issues, we forgo a large-scale comparison, and instead pick 30 random points in our test set to manually analyze using TAJS. Table 3.6 depicts the results (Appendix A.1.4 provides further details).

We restrict the chosen test points to satisfy a necessary condition for undefined property bugs since TAJS claims to be proficient in detecting this class of bugs. In the process, we also pick some functional bugs, as well as cases of refactoring modifications. By resolving the numerous issues that prevented us from automating the comparison, we were able to run TAJS manually. TAJS failed to detect any real bugs in the 30 test points. While functional bugs and refactoring modifications are beyond TAJS, however, TAJS also raises many unrelated false alarms due to its failures in locating NodeJS libraries, importing JSON files, or recognizing built-in global variables. These warnings are detrimental because TAJS suspends the analysis as soon as it detects what it perceives to be a bug. To further aid TAJS, we omitted parts of each program that are unrelated to the bug, in the hope of driving TAJS's analysis as deep as possible. After all these measures, TAJS managed to detect two of the undefined property bugs (Bug IDs 4 and 6 in Appendix A.1.4).

3.7 Related Work

Static analysis for bug detection. Static analyzers such as FindBugs, Error-Prone, and Semmle use syntactic pattern-matching and dataflow analysis to find common bugs. Typically, detecting even a single class of bugs can require dozens or even hundreds of patterns. Coverity [Bessey et al. (2010)], SonarQube, and Clang Static Analyzer check for semantic inconsistencies in code based on more sophisticated path analyses. Infer [Calcagno et al. (2015)] is built upon sound principles and can prove the absence of certain classes of bugs. TAJS belongs to this category as well. Due to the undecidability of the problem, however, approximations are inevitable which voids the guarantees in practice. Compared to all static analysis tools, Hoppity offers the following advantages: (1) it targets a board range of programming errors; (2) it not only localizes bugs but also fixes them; and (3) it has significantly higher signal-to-noise ratio (i.e., detects more bugs with less false alarms).

Learning-based bug detection. Allamanis et al. (2018) target variable-misuse errors and present a solution based on a gated graph neural network model to predict the correct variable name given a buggy location. Vasic et al. (2019) present a pointer network on top of a RNN which outperforms Allamanis et al. (2018) on the same task. DeepBugs [Pradel and Sen (2018)] proposes a name-based bug detection scheme. Their model is trained to predict three classes of bugs: swapped function arguments, wrong binary operator, and wrong operand in a binary operation. Compared to these models, our approach is capable of detecting and fixing a wide range of errors in Javascript. SequenceR [Chen et al. (2018)] uses sequence-to-sequence model to translate a buggy code segment into correct one; Getafix [Scott et al. (2019)] produces human-like bug fixes by learning from past fixes. It employs a hierarchical clustering algorithm that sorts fix patterns according to their generality. While these approaches are general against different types of bugs, they still need the bug location as input.

Graph learning and optimization. Our work is closely related to the literature in graph representation learning and optimization. Our model uses a variant of GNN that is inspired by many representative works, [Li et al. (2015); Xu et al. (2018); Si et al. (2018)], with the adaptation of local value table and pointer mechanism. Our work is also related to auto-regressive graph modeling [Johnson (2016); Li et al. (2018); Brockschmidt et al. (2018); Dai et al. (2018)], but with more generic operations such as subtree deletion and attribute modifications. Some other works model the graph modification in latent space [Jin et al. (2018); Yin et al. (2019)], but such frameworks lack fine-grained control over the generative process, and thus are not very suitable for performing code repair.

3.8 Conclusion

We proposed an end-to-end learning-based approach to detect and fix bugs in Javascript programs. We realized the approach in a tool HOPPITY and demonstrated that it correctly predicts 9,490 out of 36,361 code changes in real programs on Github. In the future, we plan to expand the targeted bugs to include those that are caused by the interdependence among multiple files or that require multiple steps to fix. We will also deploy HOPPITY in an IDE to further evaluate its accuracy and utility. Finally, we plan to extend our learning framework to support other languages. Due to its language-independence, we believe HOPPITY will benefit developers beyond Javascript as well.

Algorithm 1 Transformation inference of $p(g_{fix}|g_{bug})$

```
1:  Input $g_{bug} \sim \mathcal{D}$ and model parameters $\theta$.
2:  Obtain $\vec{g_0}, \{\vec{v}_{v \in g_{bug}}\} = f_0(g_{bug})$, let $c_{\vec{M}_0}$ be null.
3:  for $t = 1$ to $T$ do
4:      Obtain $\overrightarrow{c'_{M_t}} = \text{LSTM}(\overrightarrow{g_{t-1}}|\overrightarrow{c_{M_{t-1}}})$.
5:      Choose location $v_t$, then edit type $e_t$;
6:      if $e_t =$ NO_OP then
7:          set $g_T = g_{t-1}$ and exit the loop.
8:      end if
9:      Perform operator $e_t$ with $\overrightarrow{c_{m_t}}$ obtained by Eq 3.3.
10:     Get new graph $g_t$, update $\overrightarrow{c_{M_t}}$ with $\overrightarrow{c_{e_t}}$.
11: end for
12: Return $g_T$
```

CHAPTER 4

Program Merging

4.1 Introduction

In this section, we explore implicit specification inference in the domain of merge conflict resolution. Our approach presents a significant departure from conventional program merging techniques. Traditional merging techniques attempt to safely incorporate program changes from all parties, resorting to developer intervention when necessary. In contrast, our approach targets *merge conflict resolution*. Our goal is to eliminate the necessity for manual developer intervention by employing a neural model. In essence, performing a merge conflict resolution boils down to providing a program specification. During conflict resolution, the developer's task is to modify the code such that it matches the intended behavior of all parties. In this section, we extract implicit specification data for merge conflict resolution. We present the first data-driven approach to resolve merge conflicts with deep neural models. We perform an evaluation over a symbolic merging tool and find a 10x improvement on merging JavaScript programs. Moreover, we engage in a discussion regarding the limitations of such an approach and introduce follow-up work aimed at addressing these limitations.

4.2 DeepMerge

4.2.1 Motivation

In collaborative software development settings, version control systems such as "git" are commonplace. Such version control systems allow developers to simultaneously edit code through features called branches. Branches are a growing trend in version control as they allow developers to work in their own isolated workspace, making changes independently, and only integrating their work into the main line of development when it is complete. Integrating these changes frequently involves merging multiple copies of the source code. In fact, according to a large-scale empirical study of Java projects on GitHub [Ghiotto et al. (2020)], nearly 12% of all commits are related to a merge.

	Base $\mathcal{O}$	Variant $\mathcal{A}$	Variant $\mathcal{B}$	Resolution?
	`(base.js)`	`(a.js)`	`(b.js)`	`(m.js)`
(1).	`y = 42;`	`x = 1;`	`y = 42;`	`x = 1;`
		`y = 42;`	`z = 43;`	`y = 42;`
				`z = 43;`
(2).	`y = 42;`	`x = 1;`	`z = 43;`	`CONFLICT`
		`y = 42;`	`y = 42;`	

Figure 4.1: Two examples of unstructured merges.

To integrate changes by multiple developers across branches, version control systems utilize merge algorithms. Textual three-way file merge (e.g. present in *"git merge"*) is the prevailing merge algorithm. As the name suggests, three-way merge takes three files as input: the common *base* file $\mathcal{O}$, and its corresponding modified files, $\mathcal{A}$ and $\mathcal{B}$. The algorithm either:

1. declares a "conflict" if the two changes interfere with each other, or

2. provides a merged file $\mathcal{M}$ that incorporates changes made in $\mathcal{A}$ and $\mathcal{B}$.

Under the hood, three-way merge typically employs the `diff3` algorithm, which performs an *unstructured* (line-based) merge [Smith (1998)]. Intuitively, the algorithm *aligns* the two-way diffs of $\mathcal{A}$ (resp. $\mathcal{B}$) over the common base $\mathcal{O}$ into a sequence of diff slots. At each slot, a change from either $\mathcal{A}$ or $\mathcal{B}$ is incorporated. If both programs change a common slot, a *merge conflict* is produced, and requires manual resolution of the conflicting modifications.

Figure 4.1 shows two simple code snippets to illustrate examples of three-way merge inputs and outputs. The figure shows the base program file $\mathcal{O}$ along with the two variants $\mathcal{A}$ and $\mathcal{B}$. Example (1) shows a case where `diff3` successfully provides a merged file $\mathcal{M}$ incorporating changes made in both $\mathcal{A}$ and $\mathcal{B}$. On the other hand, Example (2) shows a case where `diff3` declares a conflict because two independent changes (updates to `x` and `z`) occur in the same diff slot.

When `diff3` declares a conflict, a developer must intervene. Consequently, merge conflicts are consistently ranked as one of the most taxing issues in collaborative, open-source software development, "especially for seemingly less experienced developers" [Gousios et al. (2016)]. Merge conflicts impact developer productivity, resulting in costly broken builds that stall the continuous integration (CI) pipelines for several hours to days. The fraction of merge conflicts as a percentage of merges range from 10% — 20% for most collaborative projects. In several large projects, merge conflicts account for up to 50% of merges (see Ghiotto et al. (2020) for details of prior studies).

Merge conflicts often arise due to the popular unstructured `diff3` algorithm that simply checks if two changes occur in the same diff slot. This insight has inspired research to incorporate program structure and semantics while performing a merge. *Structured merge* approaches [Apel et al. (2011a); Leßenich et al. (2015); Tavares et al. (2019)] and their variants treat merge inputs as abstract syntax trees (ASTs), and use tree-structured merge algorithms. However, such approaches still struggle in some situations as they do not model program semantics and cannot safely reorder statements that have side effects. To make matters worse, the gains from structured approaches hardly transfer to dynamic languages, namely JavaScript [Tavares et al. (2019)], due to the absence of static types. Semantics-based approaches [Yang et al. (1992); Sousa et al. (2018)] can, in theory, employ program analysis and verifiers to detect and synthesize the resolutions. However, there are no semantics-based tools for synthesizing merges for any real-world programming language, reflecting the intractable nature of the problem. Current automatic approaches fall short, suggesting that merge conflict resolution is a non-trivial problem.

Inspired by the abundance of data in open-source projects, we present a dataset of merge conflict resolutions and an approach to learn from them. This dataset drives the work's key insight: a vast majority (80%) of resolutions do not introduce new lines. Instead, they consist of (potentially rearranged) lines from the conflicting region. This observation is confirmed by a prior independent large-scale study of Java projects from GitHub [Gousios et al. (2016)], in which 87% of resolutions are comprised exclusively from lines in the input. In other words, a typical resolution consists of re-arranging conflicting lines without writing any new code. Our observation naturally begs the question: *Are there latent patterns of rearrangement? Can these patterns be learned?*

We investigate the potential for learning latent patterns of rearrangement. Effectively, this boils

{ unchanged lines (prefix) }
<<<<<<
{ lines edited by $\mathcal{A}$ }
|||||||
{ affected lines of base $\mathcal{O}$ }
=======
{ lines edited by $\mathcal{B}$ }
>>>>>>
{ unchanged lines (suffix) }

(a) Format of a conflict.

```
<<<<<< a.js
x = 1;
|||||| base.js
=======
z = 43;
>>>>>> b.js
y = 42;
```

(b) Instance of a conflict.

Figure 4.2: Conflict format and an instance reported by `diff3` on Example (2) from Figure 4.1.

down to the question:

Can we learn to synthesize merge conflict resolutions?

Specifically, the work frames merging as a sequence-to-sequence task akin to machine translation.

To formulate program merging as a sequence-to-sequence problem, we consider the text of programs $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$ as the input sequence, and the text of the resolved program $\mathcal{M}$ as the output sequence. However, this seemingly simple formulation does not come without challenges. Section 4.2.5 demonstrates an out of the box sequence-to-sequence model trained on merge conflicts yields very low accuracy. In order to effectively learn a merge algorithm, one must:

1. represent merge inputs in a concise yet sufficiently expressive sequence;

2. create a mechanism to output tokens at the line granularity; and

3. localize the merge conflicts and the resolutions in a given file.

To represent the input in a concise yet expressive embedding, we present an edit aware sequence to be consumed by DeepMerge. These edits are provided in the format of `diff3` which is depicted in Figure 4.2(a) in the portion between markers “<<<<<<<“ and “>>>>>>>“. The input embedding is extracted from parsing the conflicting markers and represents $\mathcal{A}$'s and $\mathcal{B}$'s edits over the common base $\mathcal{O}$.

To represent the output at the line granularity, DeepMerge's design is a form of a pointer network [Vinyals et al. (2015a)]. As such, DeepMerge constructs resolutions by copying input lines, rather than learning to generate them token by token. Guided by our key insight that a large majority of resolutions are entirely comprised of lines from the input, such an output vocabulary is sufficiently expressive.

Lastly, we show how to localize merge conflicts and the corresponding user resolutions in a given file. This is necessary as our approach exclusively aims to resolve locations in which `diff3` has declared a conflict. As such, our algorithm only needs to generate the conflict resolution and not the entire merged file. Thus, to extract ground truth, we must localize the resolution for a given conflict in a resolved file. Localizing such a resolution region unambiguously is a non-trivial task. The presence of extraneous changes unrelated to conflict resolution makes resolution localization challenging. We present the first algorithm to localize the resolution region for a conflict. This ground truth is essential for training such a deep learning model.

This section demonstrates an instance of DeepMerge trained to resolve unstructured merge conflicts in JavaScript programs. Besides its popularity, JavaScript is notorious for its rich dynamic features, and lacks tooling support. Existing structured approaches struggle with JavaScript [Tavares et al. (2019)], providing a strong motivation for a technique suitable for dynamic languages. This dissertation contributes a real-world dataset of 8,719 merge tuples that require non-trivial resolutions from nearly twenty thousand repositories in GitHub. Our evaluation shows that, on a held out test set, DeepMerge can predict correct resolutions for 37% of non-trivial merges. DeepMerge's accuracy is a 9x improvement over a recent semistructured approach [Tavares et al. (2019)], evaluated on the same dataset. Furthermore, on the subset of merges with upto 3 lines (comprising 24% of the total dataset), DeepMerge can predict correct resolutions with 78% accuracy.

Contributions. In summary, this work:

1. is the first to define merge conflict resolution as a machine learning problem and identify a set

of challenges for encoding it as a sequence-to-sequence supervised learning problem (§ 4.2.2).

2. presents a data-driven merge tool DeepMerge that uses edit-aware embedding to represent merge inputs and a variation of pointer networks to construct the resolved program (§ 4.2.3).

3. derives a real-world merge datasetfor supervised learning by proposing an algorithm for localizing resolution regions (§ 4.2.4).

4. performs an extensive evaluation of DeepMerge on merge conflicts in real-world JavaScript programs. And, demonstrates that it can correctly resolve a significant fraction of unstructured merge conflicts with high precision and 9x higher accuracy than a structured approach.

4.2.2 Data-Driven Merge

We formulate program merging as a sequence-to-sequence supervised learning problem and discuss the challenges we must address in solving the resulting formulation.

Problem Formulation

A merge consists of a 4-tuple of programs $(\mathcal{A}, \mathcal{B}, \mathcal{O}, \mathcal{M})$ where $\mathcal{A}$ and $\mathcal{B}$ are both derived from a common $\mathcal{O}$, and $\mathcal{M}$ is the developer resolved program.

A merge may consist of one or more regions. We define a *merge tuple* $((A, B, O), R)$ such that A, B, O are (sub) programs that correspond to regions in $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$, respectively, and R denotes the result of merging those regions.

Although we refer to (A, B, O, R) as a merge tuple, we assume that the tuples also implicitly contain the programs that they came from as additional contexts (namely $\mathcal{A}, \mathcal{B}, \mathcal{O}, \mathcal{M}$).

Definition 1 (Data-driven Merge)**.** *Given a dataset of M merge tuples,*

$$D = \{(A^i, B^i, O^i, R^i)\}_{i=1}^{M}$$

```
<<<<<< a.js
    let b = x + 5.7
    var y = floor(b)
    console.log(y)
|||||| base.js
    var b = 5.7
    var y = floor(b)
======
    var y = floor(x + 5.7)
>>>>>> b.js
```

(a) A merge instance.

```
A = | let b = x + 5.7
    | var y = floor(b)
    | console.log(y)

O = | var b = 5.7
    | var y = floor(b)

B = | var y = floor(x + 5.7)

R = | var y = floor(x + 5.7)
    | console.log(y)
```

(b) Corresponding merge tuple.

Figure 4.3: Formulation of a merge instance in our setting.

a data-driven merge algorithm merge is a function that maximizes:

$$\sum_{i=1}^{M} \mathit{merge}(A^i, B^i, O^i) = R^i$$

treating Boolean outcomes of the equality comparison as integer constants 1 (for `true`*) and 0 (for* `false`*).*

In other words, *merge* aims to maximize the number of merges from D. Rather than constraining merge to exactly satisfy *all* merge tuples in D, we relax the objective to maximization. A perfectly satisfying merge function may not exist in the presence of a real-world *noisy* dataset D. For instance, there may be $(A^i, B^i, O^i, R^i) \in D$ and $(A^j, B^j, O^j, R^j) \in D$ for $i \neq j$, $A^i = A^j$, $B^i = B^j$, $O^i = O^j$ but $R^i \neq R^j$. In other words, two merge tuples consist of the same edits but different resolutions.

Example 1. Figure 4.3(a) shows a merge instance that we will use as our running example throughout. This instance is formulated in our setting as the merge tuple (A, B, O, R) depicted in Figure 4.3(b).

R contains only lines occurring in the input. The two lines in R correspond to the first line of B and the third line of A. For this example, the R also incorporates the intents from both A and B intuitively, assuming `b` does not appear in the rest of the programs. □

One possible way to learn a *merge* algorithm is by modeling the conditional probability

$$p(R|A, B, O) \tag{4.1}$$

In other words, a model that generates the output program R given the three input programs.

Because programs are sequences, we further decompose Eq 4.1 by applying the chain rule [Sutskever et al. (2014a)]:

$$p(R|A, B, O) = \prod_{j=1}^{N} p(R_j|R_{<j}, A, B, O)$$

This models the probability of generating the $j-$th element of the program, given the elements generated so far. There are many possible ways to model a three-way merge. However, the above formulation suggests one obvious approach is to use a maximum likelihood estimate of a sequence-to-sequence model.

Challenges

Applying a sequence-to-sequence (Seq2seq) model to merge conflict resolution poses unique challenges. We discuss three key challenges, concerning input representation, output construction, and dataset extraction.

Representing the Merge Inputs as a Sequence. In a traditional sequence-to-sequence task such as machine translation, there is a single input sequence that maps to a single output sequence. However, in our case, we have three input sequences of varying sizes, corresponding to the three versions of a program involved in a merge conflict. It is not immediately evident how to determine a suitable token granularity and encode these sequences in a manner that is amenable to learning. One obvious solution is to concatenate the tokens of the three sequences to obtain a single sequence. However, the order of concatenation is unclear. Furthermore, as we show in Section 4.2.3, such a naive representation not only suffers from information loss and truncation, but also poor precision by being unaware of A and B's edits over common base O. In summary, we have:

CH1: *Encode programs A, B, and O as the input to a Seq2Seq model.*

Constructing the Output Resolution

Our key insight that a majority of resolutions do not introduce new lines leads us to construct the output resolution directly from lines in the conflicting region. This naturally suggests the use of pointer networks [Vinyals et al. (2015a)], an encoder-decoder architecture capable of producing outputs explicitly pointing to tokens in the input sequence. However, a pointer network formulation suggests an equivalent input and output granularity. In Section 4.2.3, we show that the input is best represented at a granularity far smaller than lines.

Thus, the challenge is:

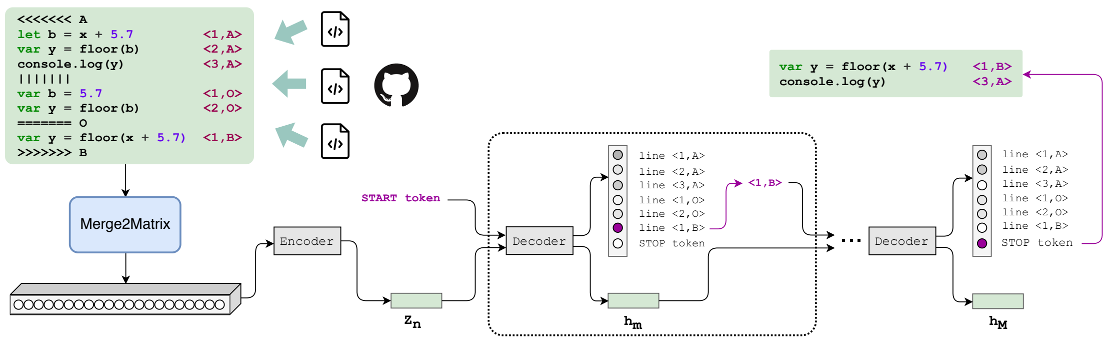
CH2: *Output R at the line granularity given a non-line granularity input.*

Extracting Ground Truth from Raw Merge Data. Finally, to learn a data-driven merge algorithm, we need real-world data that serves as ground truth. Creating this dataset poses non-trivial challenges. First, we need to localize the resolution region and corresponding conflicting region. In some cases, developers performing a manual merge resolution made changes unrelated to the merge. Localizing resolution regions unambiguously from input programs is challenging due to the presence of these unrelated changes. Second, we need to be able to recognize and subsequently filter merge resolutions that do not incorporate both the changes. In summary, we have:

CH3: *Identify merge tuples $\{(A^i, B^i, O^i, R^i)\}_{i=1}^{M}$ given $(\mathcal{A}, \mathcal{B}, \mathcal{O}, \mathcal{M})$.*

4.2.3 The DeepMerge Architecture

Section 4.2.2 suggested one way to learn a three-way merge is through a maximum likelihood estimate of a sequence-to-sequence model. In this section we describe DeepMerge, the first data-driven merge framework, and discuss how it addresses challenges **CH1** and **CH2**. We motivate the design of DeepMerge by comparing it to a standard sequence-to-sequence model, the encoder-decoder architecture.



```
<<<<<<< A
let b = x + 5.7           <1,A>
var y = floor(b)          <2,A>
console.log(y)            <3,A>
|||||||
var b = 5.7               <1,O>
var y = floor(b)          <2,O>
======= O
var y = floor(x + 5.7)    <1,B>
>>>>>>> B
```

```
var y = floor(x + 5.7)    <1,B>
console.log(y)            <3,A>
```

Figure 4.4: Overall DeepMerge framework. The dotted box represents repetition of *decode* until $m = M$ i.e. the $\langle \texttt{STOP} \rangle$ token is predicted. In this example, we have omitted $m = 2$ in which the call to *decode* outputs $y_2 = \langle 3, A \rangle$.

Encoder Decoder Architectures

Sequence-to-sequence models aim to map a fixed-length input $((X_N)_{N \in \mathbb{N}})$, to a fixed-length output, $((Y_M)_{M \in \mathbb{N}})$. [2] The standard sequence-to-sequence model consists of three components: an **input embedding**, an **encoder**, and a **decoder**.

Input embedding: An embedding maps a discrete input from an input vocabulary V $(x_n \in \mathbb{N}^{|V|})$, to a continuous D dimensional vector space representation $(\overline{x}_n \in \mathbb{R}^D)$. Such a mapping is obtained by multiplication over an embedding matrix $E \in \mathbb{R}^{D \times |V|}$. Applying this for each element of X_N gives $\overline{X}_N$.

Encoder: An encoder *encode*, processes each $\overline{x}_n$ and produces a hidden state, z_n which summarizes the sequence upto the n-th element. At each iteration, the encoder takes as input the current sequence element x_n, and the previous hidden state z_{n-1}. After processing the entire input sequence, the final hidden state, z_N, is passed to the decoder.

Decoder: A decoder *decode*, produces the output sequence Y_M from an encoder hidden state Z_n. Similar to encoders, decoders work in an iterative fashion. At each iteration, the decoder produces a

[2]Note that M is not necessary equal to N.

single output token y_m along with a hidden summarization state h_m. The current hidden state and the previous predicted token y_m are then used in the following iteration to produce y_{m+1} and h_{m+1}. Each y_m the model predicts is selected through a softmax over the hidden state:

$$p(y_m|y_1, ..., y_{m-1}, X) = \mathit{softmax}(h_m)$$

DeepMerge is based on this encoder-decoder architecture with two significant differences.

First, rather than a standard embedding followed by encoder, we introduce a novel embedding method called *Merge2Matrix*. *Merge2Matrix* addresses **CH1** by summarizing input programs (A, B, O) into a single embedding fed to the encoder. We discuss our *Merge2Matrix* solution as well as less effective alternatives in Section 4.2.3.

Second, rather than using a standard decoder to generate output tokens in some output token vocabulary, we augment the decoder to function as a variant of pointer networks. The decoder outputs line tuples (i, W) where $W \in \{A, B\}$ and i is the i-th line in W. We discuss this in detail in Section 4.2.3.

Example 2. Figure 4.4 illustrates the flow of DeepMerge as it processes the inputs of a merge tuple. First, the raw text of A, B, and O is fed to *Merge2Matrix*. As the name suggests, *Merge2Matrix* summarizes the tokenized inputs as a matrix. That matrix is then fed to an encoder which computes the encoder hidden state z_N. Along with the start token for the decoder hidden state, the decoder takes z_N and iteratively (denoted by the $\cdots$) generates as output the lines to copy from A and B. The final resolution is shown in the green box. □

Merge2Matrix

An encoder takes a single sequence as input. As discussed in Section 4.2.2, a merge tuple consists of three sequences. This section introduces *Merge2Matrix*, an input representation that expresses the tuple as a single sequence. It consists of embedding, transformations to summarize embeddings, and finally, edit-aware alignment.

Tokenization and Embedding This section discusses our relatively straightforward application of both tokenization and embedding.

Tokenization. Working with textual data requires tokenization whereby we split a sequence of text into smaller units referred to as *tokens*. Tokens can be defined at varying granularities such as characters, words, or sub-words. These units form a *vocabulary* which maps input tokens to integer indices. Thus, a vocabulary is a mapping from a sequence of text to a sequence of integers. This work uses byte-pair encoding (BPE) as it has been shown to work well with source code, where tokens can be formed by combining different words via casing conventions (e.g. `snake_case` or `camelCase`) causing a blowup in vocabulary size [Karampatsis et al. (2020)]. Byte-pair encoding is an unsupervised sub-word tokenization that draws inspiration from information theory and data compression wherein frequently occurring sub-word pairs are recursively merged and stored in the vocabulary. We found that the performance of BPE was empirically superior to other tokenization schemes.

Embedding. Given an input sequence X_N, and a hyperparameter (embedding dimension) D, an embedding transformation creates $\overline{X}_N$. As described in Section 4.2.3, the output of this embedding is then fed to an encoder. Because a merge tuple consists of three inputs (A, B, and O), the following sections introduce novel transformations that *summarize* these three inputs into a format suitable for the encoder.

Merge Tuple Summarization

In this section, we describe summarization techniques that are employed after embedding. Before we delve into details, we first introduce two functions used in summarization.

Suppose a function that concatenates embedded representations:

$$concat_s : (\mathbb{R}^{D\times N} \times \cdots \times \mathbb{R}^{D\times N}) \to \mathbb{R}^{D\times sN}$$

that takes s similarly shaped tensors as arguments and concatenates them along their last dimension.

Concatenating these s embeddings increases the size of the encoder's input by a factor of s.

Suppose a function *linearize* that linearly combines s embedded representations. We parameterize this function with learnable parameters $\theta \in \mathbb{R}^{s+1}$. As input, *linearize* takes an embedding $\overline{x}_i \in \mathbb{R}^D$ for $i \in 1..S$. Thus, we define

$$linearize_\theta(\overline{x}_1, \ldots, \overline{x}_s) = \theta_1 \cdot \overline{x}_1 + \cdots + \theta_s \cdot \overline{x}_s + \theta_{s+1}$$

where all operations on the inputs $\overline{x}_1, \ldots, \overline{x}_s$ are pointwise. *linearize* reduces the size of the embeddings fed to the encoder by a factor of s.

Now that we have defined two helper functions, we describe two summarization methods.

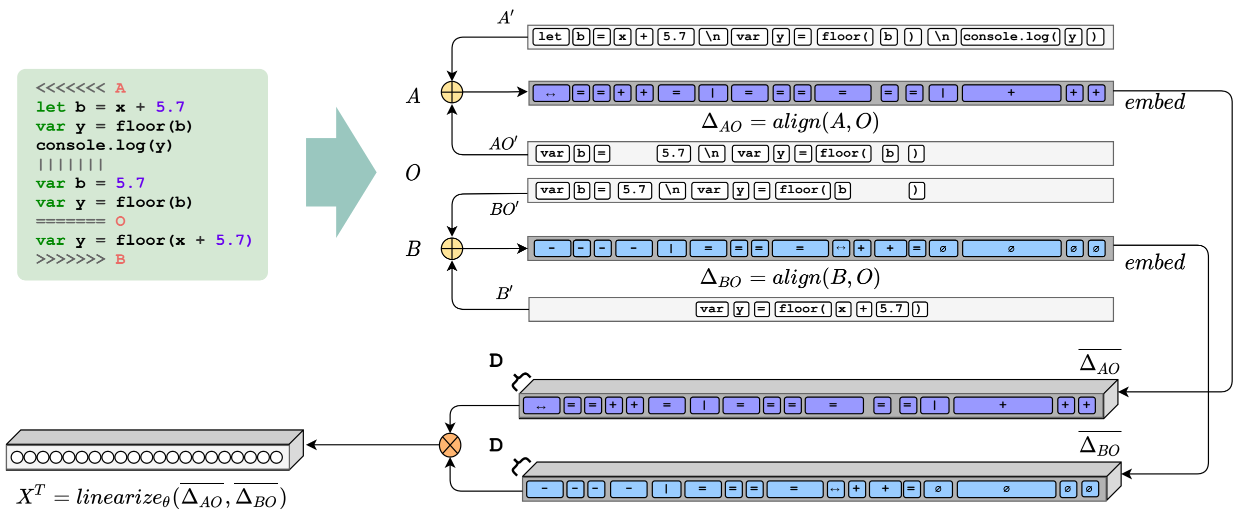
Naïve. Given a merge tuple's inputs (A, B, O), a *naïve* implementation of *Merge2Matrix* is to simply concatenate the embedded representations (i.e., $concat_3(\overline{A}, \overline{B}, \overline{O})$) Traditional sequence-to-sequence models often suffer from information forgetting; as the input grows longer, it becomes harder for *encode* to capture long-range correlations in that input. A solution that addresses **CH1**, must be concise while retaining the information in the input programs.

Linearized. As an attempt at a more concise representation, we introduce a summarization we call *linearized*. This method linearly combines each of the embeddings through our helper function: $linearize_\theta(\overline{A}, \overline{B}, \overline{O})$. In Section 4.2.5 we empirically demonstrate better model accuracy when we summarize with $linearize_\theta$ rather than $concat_s$.

Edit-Aware Alignment

In addition to input length, **CH1** also alludes that an effective input representation needs to be "edit aware". The aforementioned representations do not provide any indication that A and B are edits from O.

Prior work, *Learning to Represent Edits* (LTRE) [Yin et al. (2019)] introduces a representation to succinctly encode 2 two-way diffs. The method uses a standard deterministic diffing algorithm and



```
<<<<<<< A
let b = x + 5.7
var y = floor(b)
console.log(y)
|||||||
var b = 5.7
var y = floor(b)
======= O
var y = floor(x + 5.7)
>>>>>>> B
```

A′
A
O
AO′
BO′
B
B′
$\Delta_{AO} = align(A, O)$
$\Delta_{BO} = align(B, O)$
embed
embed
D
D
$\overline{\Delta_{AO}}$
$\overline{\Delta_{BO}}$
$X^T = linearize_\theta(\overline{\Delta_{AO}}, \overline{\Delta_{BO}})$

Figure 4.5: *Merge2Matrix*: implemented with the Aligned Linearized input representation used in DeepMerge.

represents the resulting pair-wise alignment as an auto-encoded fixed dimension vector.

A two-way alignment produces an "edit sequence". This series of edits, if applied to the second sequence, would produce the first. An edit sequence, Δ_{AO}, is comprised of the following editing actions: $=$ representing equivalent tokens, $+$ representing insertions, $-$ representing deletions, $\leftrightarrow$ representing a replacement. Two special tokens $\emptyset$ and $|$ are used as a padding token and a newline marker, respectively. Note that these Δs only capture information about the *kinds* of edits and *ignore* the the tokens that make up the edit itself (with the exception of the newline token). Prior to the creation of Δ, a preprocessing step adds padding tokens such that equivalent tokens in A (resp. B) and O are in the same position. These sequences, shown in Figure 4.5 are denoted as A' and AO' (resp. B' and BO').

Example 3. Consider B's edit to O in Figure 4.5 via its preprocessed sequences B', BO', and its edit sequence Δ_{BO}. One intuitive view of Δ_{BO} is that it is a set of instructions that describe how to turn B' into BO' with the aforementioned semantics. Note the padding token $\emptyset$ introduced into Δ_{BO} represents padding out to the length of the longer edit sequence Δ_{AO}. $\square$

We now describe two edit-aware summarization methods based on this edit-aware representation.

However, our setting differs from the original *LTRE* setting as we assume three input sequences and a three-way diff. In the following summarization methods, we assume that A, B, O are tokenized, but *not* embedded before invoking *Merge2Matrix*.

Aligned *naïve*. Given Δ_{AO} and Δ_{BO}, we embed each to produce $\overline{\Delta_{AO}}$ and $\overline{\Delta_{BO}}$, respectively. Then we combine these embeddings through concatenation and thus $concat_2(\overline{\Delta_{AO}}, \overline{\Delta_{BO}})$ is fed to the encoder.

Aligned linearized. This summarization method is depicted in Figure 4.5, invoking *linearize* to construct an input representation over edit sequences. First, we apply alignment to create Δ_{AO} and Δ_{BO}. This is portrayed through the $\bigoplus$ operator. Following construction of the Δs, we apply embedding and subsequently apply our edit-aware linearize operation via the $\bigotimes$ operator. Thus, we summarize embeddings with $linearize_\theta(\overline{\Delta_{AO}}, \overline{\Delta_{BO}})$ and feed its output to the encoder. As we demonstrate in Section 4.2.5, this edit-aware input representation significantly increases the model's accuracy.

***LTRE*.** Finally, for completeness, we also include the original *LTRE* representation. We modify this to our setting by creating *two* 2-way diffs. The original *LTRE* has a second key difference from our summarization methods. *LTRE* includes all tokens from from the input sequences in addition to the edit sequences That is, *LTRE* summarizes A' AO', Δ_{AO}, B', BO', and Δ_{BO}. Let $\overline{A'}$, $\overline{AO'}$ and $\overline{\Delta_{AO}}$, (resp $\overline{B'}$, $\overline{BO'}$, and $\overline{\Delta_{BO}}$) be the embedding of a two-way diff. Then, the following summarization combines all embeddings:

$$concat_6(\overline{\Delta_{AO}}, \overline{A'}, \overline{AO'}, \overline{\Delta_{BO}}, \overline{B'}, \overline{BO'})$$

The Encoder

The prior sections described *Merge2Matrix* which embeds a merge into a continuous space which is then summarized by an encoder. DeepMerge uses a bi-directional gated recurrent unit [Cho et al. (2014b)] (GRU) to summarize the embedded input sequence. We empirically found that a bi-directional GRU was more effective than a uni-directional GRU.

Synthesizing Merge Resolutions

This section summarizes DeepMerge's approach to solving **CH2**.

Given a sequence of hidden vectors Z_N produced by an encoder, a decoder generates output sequence Y_M. We introduce an extension of a traditional decoder to copy lines of code from those input programs.

Denote the number of lines in A and B as Li_A and Li_B, respectively. Suppose that $L = 1..(Li_A + Li_B)$; then, a value $i \in L$ corresponds to the i-th line from A if $i <= Li_A$, and the $i - Li_A$-th line from B, otherwise.

Given merge inputs (A, B, O), DeepMerge's decoder computes a sequence of hidden states H_M, and models the conditional probability of *lines* copied from the input programs A, B, and O by predicting a value in $y_m \in Y_M$:

$$p(y_m | y_1, ..., y_{m-1}, A, B, O) = \mathit{softmax}(h_m)$$

where $h_m \in H_M$ is the decoder hidden state at the m-th element of the output sequence and the $\mathit{argmax}(y_m)$ yields an index into L.

In practice, we add an additional $\langle$`STOP`$\rangle$ token to L. The $\langle$`STOP`$\rangle$ token signifies that the decoder has completed the sequence. The $\langle$`STOP`$\rangle$ token is necessary as the decoder may output a variable number of lines conditioned on the inputs.

This formulation is inspired by pointer networks [Vinyals et al. (2015a)], an encoder-decoder architecture that outputs an index that explicitly points to an input token. Such networks are designed to solve combinatorial problems like sorting. Because the size of the output varies as a function of the input, a pointer network requires a novel attention mechanism that applies attention weights directly to the input sequence. This differs from traditional attention networks which are applied to the outputs of the encoder Z_N. In contrast, DeepMerge requires no change to attention. Our architecture outputs an index that points to the abstract concept of a line, rather than an explicit token in the input. Thus, attention applied to Z_N, a summarization of the input, is sufficient.

Training and Inference with DeepMerge

The prior sections discussed the overall model architecture of DeepMerge. This section describes hyperparameters that control model size and how we trained the model. We use a embedding dimension $D = 1024$ and 1024 hidden units in the single layer GRU encoder. Assume the model parameters are contained in θ; training seeks to find the values of θ that maximize the log-likelihood

$$\underset{\theta}{\operatorname{argmax}} \log p_\theta(R|A, B, O)$$

over all merge tuples $((A, B, O), R)$ in its training dataset. We use standard cross-entropy loss with the Adam optimizer. Training takes roughly 18 hours on a NVIDIA P100 GPU and we pick the model with the highest validation accuracy, which occurred after 29 epochs.

Finally, during inference time, we augment DeepMerge to use standard beam search methods during decoding to produce the most likely k top merge resolutions. DeepMerge predicts merge resolutions up to C lines. We set $C = 30$ to tackle implementation constraints and because most resolutions are less than 30 lines long. However, we evaluate DeepMerge on a full test dataset including samples where the number of lines in M is $\geq C$.

4.2.4 Real-World Labeled Dataset

This section describes our solution to **CH3**: localizing merge instances $(A, B, O, R)_i$ from $(\mathcal{A}, \mathcal{B}, \mathcal{O}, \mathcal{M})$. Since a program may have several merge conflicts, we decompose the overall merge problem into merging individual instances. As shown in Figure 4.3, A, B, and O regions can be easily extracted given the `diff3` conflict markers. However, reliably localizing a resolution R involves two sub-challenges:

1. How do we localize individual regions R unambiguously?

2. How do we deal with trivial resolutions?

In this section, we elaborate on each of these sub-challenges and discuss our solutions. We conclude with a discussion of our final dataset and its characteristics.

Algorithm 2 Localizing Merge Tuples from Files for Dataset

```
procedure LocalizeMergeTuples($\mathcal{C}$, $\mathcal{M}$)
    $MT \leftarrow \emptyset$                                          ▷ Merge Tuples
    for $i \in [1,$ NumConflicts($\mathcal{C}$)$]$ do
        $R \leftarrow$ LocalizeResRegion($\mathcal{C}, \mathcal{M}, i$)
        if $R ==$ nil then
            continue                                                   ▷ Could not find resolution
        end if
        $(A, B, O) \leftarrow$GetConflictComponents($\mathcal{C}, i$)
        if $R \in \{A, B, O\}$ then
            continue                                                   ▷ Filter trivial resolutions
        end if
        if Lines($R$) $\subseteq$ Lines($A$) $\cup$ Lines($B$) then
            $MT \leftarrow MT \cup \{(A, B, O, R)\}$
        end if
    end for
    return $MT$
end procedure

procedure LocalizeResRegion($\mathcal{C}$, $\mathcal{M}$, $i$)
    $n \leftarrow Length(\mathcal{M})$                                 ▷ Length of $\mathcal{M}$ in chars
    $m \leftarrow Length(\mathcal{C})$                                 ▷ Length of $\mathcal{C}$ in chars
    $(spos, epos) \leftarrow$ GetConflictStartEnd($\mathcal{C}, i$)
    $prfx \leftarrow \langle BOF \rangle + \mathcal{C}[0 : spos]$
    $sffx \leftarrow \mathcal{C}[epos : m] + \langle EOF \rangle$
    s $\leftarrow$ MinimalUniquePrefix($reverse(prfx), reverse(\mathcal{M})$)
    e $\leftarrow$ MinimalUniquePrefix($sffx, \mathcal{M}$)
    if $s \geq 0$ and $e \geq 0$ then
        return $\mathcal{M}[n - s : e]$
    else
        return nil
    end if
end procedure

procedure MinimalUniquePrefix($x$, $y$)
    Output:Returns the start position of the minimal non-empty prefix of $x$ that appears uniquely in $y$, else -1
end procedure

procedure lines($p$)
    Output:Returns the set of lines comprising the program $p$
end procedure
```

Algorithm 2 denotes a method to localize merge tuples from a corpus of merge conflict and resolution files. The top-level procedure ExtractMergeTuples takes $\mathcal{C}$, the `diff3` conflict file with markers, along with $\mathcal{M}$, the resolved file. From those inputs, it extracts merge tuples into MT. The algorithm loops over each of the conflicted regions in $\mathcal{C}$, and identifies the input (A, B, O) and output (R) of the tuple using GetConflictComponents and LocalizeResRegion respectively. Finally, it applies a filter on the extracted tuple (lines 5 – 11). We explain each of these components in the next few subsections.

Localization of Resolution Regions

Creating a real-world merge conflict labeled dataset requires identifying the "exact" code region that constitutes a resolution. However, doing so can be challenging; Figure 4.6 demonstrates an example. The developer chooses to perform a resolution `baz();` that does not correspond to anything from the A or B edits, and the surrounding context also undergoes changes (e.g. changing var with let which restricts the scope in the prefix). To the best of our knowledge, there is no known algorithm to localize R for such cases.

LocalizeResRegion is our method that tries to localize the i^{th} resolution region R, or returns `nil` when unsuccessful. Intuitively, we find a prefix and suffix in a merge instance and use this prefix and suffix to bookend a resolution. If we cannot uniquely find those bookends, we say the resolution is *ambiguous*.

The method first obtains the prefix *prfx* (resp. suffix *sffx*) of the i^{th} conflict region in $\mathcal{C}$ in line 22 (resp. line 23). We add the start of file $\langle BOF \rangle$ and end of file $\langle EOF \rangle$ tokens to the prefix and suffix respectively. The next few lines try to match the prefix *prfx* (resp. suffix *sffx*) in the resolved file $\mathcal{M}$ unambiguously. Let us first focus on finding the suffix of the resolution region in $\mathcal{M}$ in line 25. The procedure MinimalUniquePrefix takes two strings x and y and finds the start position of the minimal non-empty prefix of x that appears uniquely in y, or returns -1. For example, MinimalUniquePrefix("abc", "acdabacc") is 3 since "ab" is the minimal prefix of x that appears uniquely in y starting in position 3 (0-based indexing).

To find the prefix of the resolution, we reverse the *prfx* string and search for matches in reversed $\mathcal{M}$, and then finally find the offset from the start of $\mathcal{M}$ by subtracting s from the length n of $\mathcal{M}$. The unique occurrence of both the prefix and suffix in $\mathcal{M}$ allows us to map the conflicted region to the resolved region.

```
<BOF>
...
var time = new Date();
print_time(time);
<<<<<<< a.js
x = foo();
||||||| base.js
=======
x = bar();
>>>>>>> b.js
print_time(time);
<EOF>
```

(a) A merge instance.

```
<BOF>
...
let time = new Date();
print_time(time);
baz();
print_time(time);
<EOF>
```

(b) Resolution.

Figure 4.6: Challenging example for localizing resolution.

For our example, even though the line "`print_time(time);`" that encloses the conflicted region appears twice in $\mathcal{M}$, extending it by "`time = new Date();`" in the prefix and $\langle EOF \rangle$ in the suffix provides a unique match in $\mathcal{M}$. Thus, the algorithm successfully localizes the desired region "`baz();`" as the resolution region.

After localizing the resolution regions, we have a set of merge instances of the form (A, B, O, R). We can use our definition from Section 4.2.2 to label a merge tuple (A, B, O, R).

Filtering Trivial Resolutions

Upon examining our dataset, we found a large set of merges in which A was taken as the resolution and B was entirely ignored (or vice versa). These trivial samples, in large, were the product of running *git merge* with "ours" or "theirs" command-line options. Using these merge options indicates that the developer did not resolve the conflict after careful consideration of both branches, but instead relied on the git interface to completely drop one set of changes. The aforementioned command-line merge options are typically used the commit is the first of many fix-up commits to perform the full resolution.

We appeal to the notion of a "valid merge" that tries to incorporate both the syntactic and semantic changes from both A and B. Thus, these samples are not valid as they disregard the changes from B (resp. A) entirely. Furthermore, these trivial samples comprised 70% of our "pre-filtering" dataset. Previous work confirmed our observation that a majority of merge resolutions in GitHub Java projects (75% in Table 13 [Ghiotto et al. (2020)]) correspond to taking just A or B. To avoid polluting our dataset, we filter such merges (A, B, O, R) where $R \in \{A, B, O\}$ (line 9 in Algorithm 2). Our motivation to filter the dataset of trivial labels is based on both dataset bias and the notion of a valid merge.

Final Dataset

We crawled repositories in GitHub containing primarily JavaScript files, looking at merge commits. To avoid noise and bias, we select projects that were active in the past one year (at the time of writing), and received at least 100 stars (positive sentiment). We also verified that the dataset did not contain duplicate merges. We ignore *minified* JavaScript files that compress an entire JavaScript file to a few long lines. Finally, note that Algorithm 2 filters away any resolution that consists of new segments (lines) outside of A and B as our technique targets resolutions that do not involve writing any new code. After applying filters, we obtained 8,719 merge tuples. We divided these into a 80/10/10 percent training/validation/test split. Our dataset contains the following distribution in terms of total number of lines in A and B: 45.08% ([0,5]), 20.57% ([6,10]), 26.42% ([11,50]), 4.22% ([51,100]) and 3.70% (100+).

4.2.5 Evaluation

In this section, we empirically evaluate DeepMerge to answer the following questions:

RQ1 How effective is DeepMerge at synthesizing resolutions?

RQ2 How effective is DeepMerge at suppressing incorrect resolutions?

RQ3 On which samples is DeepMerge most effective?

RQ4 How do different choices of input representation impact the performance of DeepMerge?

	Top-1	Top-3
DeepMerge	**36.50%**	**43.23%**
ScanMerge	4.20%	7.43%
Seq2Seq	2.3%	3.3%
jsFSTMerge	3.7%	N/A

Table 4.1: Evaluation of DeepMerge and baselines: resolution synthesis accuracy (%).

RQ1: Effectiveness of Resolution Synthesis

In this section, we perform an evaluation to assess DeepMerge's effectiveness of synthesizing resolutions. Our prediction, $\hat{R}$, is considered correct if it is an exact (line for line, token for token) match with R.

Evaluation metrics. DeepMerge produces a ranked list of predictions; we define top-1 (resp. top-3) accuracy if the R is present in first (resp. top 3) predictions. This is a lower bound, as multiple resolutions may be "correct" with respect to the semantics of the changes being merged (e.g., in some cases, switching two declarations or unrelated statements has no impact on semantics).

Quantitative Results. Table 4.1 shows the performance of DeepMerge on a held out test set. DeepMerge has an overall top-1 accuracy of 36.5%, correctly generating more than one in three resolutions as its first ranked choice. When we consider the top-3 ranked resolutions, DeepMerge achieves a slightly improved accuracy of 43.23%.

Baselines. Table 4.1 also includes a comparison of DeepMerge to three baselines. We compare to a heuristic based approach (ScanMerge), an off-the-shelf sequence-to-sequence model (Seq2Seq), and a structured AST based approach (jsFSTMerge).

Our first baseline ScanMerge, is a heuristic based approach designed by manually observing patterns in our dataset. ScanMerge randomly samples from the space of sub-sequences over lines from A and B that are: (i) syntactically valid and parse, (ii) include each line from A and B, and (iii) preserve the order of lines within A and B.

These heuristic restrictions are based on manual observations that a large fraction of resolutions satisfy these conditions.

Table 4.1 shows ScanMerge's performance averaged over 10 trials. DeepMerge performs significantly better in terms of top-1 resolution accuracy (36.50% vs 4.20%). ScanMerge only synthesizes one in 20 resolutions correctly. In contrast, DeepMerge correctly predicts one in 3 resolutions. On inputs of 3 lines or less, ScanMerge only achieves 12% accuracy suggesting that the problem space is large even for small merges.

We also compared DeepMerge to an out of the box sequence-to-sequence encoder-decoder model [Sutskever et al. (2014b)] (Seq2Seq) implemented with Fairseq [3] natural language processing library. Using a *naïve* input (i.e., $concat_3(A, B, O)$), tokenized with a standard byte-pair encoding, and Fairseq's default parameters, we trained on the same dataset as DeepMerge. DeepMerge outperforms the sequence-to-sequence model in terms of both top-1 (36.5% vs. 2.3%) and top-3 accuracy (43.2% vs. 3.3%). This is perhaps not surprising given the precise notion of accuracy that does not tolerate even a single token mismatch. We therefore also considered a more relaxed measure, the BLEU-4 score [Papineni et al. (2002)], a metric that compares two sentences for "closeness" using an n-gram model. The sequence-to-sequence model achieves a respectable score of 27%, however DeepMerge still outperforms with a BLEU-4 score of 50%. This demonstrates that our novel embedding of the merge inputs and pointer network style output technique aid DeepMerge significantly and outperform a state of the art sequence-to-sequence baseline model.

Lastly, we compared DeepMerge to jsFSTMerge[Tavares et al. (2019)], a recent semistructured AST based approach. jsFSTMerge leverages syntactic information by representing input programs as ASTs. With this format, algorithms are invoked to safely merge nodes and subtrees. Structured approaches do not model semantics and can only safely merge program elements that do not have side effects. Structured approaches have been proven to work well for statically typed languages such as Java [Apel et al. (2011a); Leßenich et al. (2015)]. However, the benefits of semistructured merge hardly translate to dynamic languages such as JavaScript. JavaScript provides less static

[3]https://github.com/pytorch/fairseq

Threshold	[1,3] lines	[4,5] lines	[6,7] lines	[8,10] lines	[>10] lines
0	78.40%	56.50%	37.04%	10.87%	2.93%

Table 4.2: Evaluation of DeepMerge: accuracy vs input size (%).

information than Java and allows statements (with potential side effects) at the same syntactic level as commutative elements such as function declarations.

As a baseline to compare to DeepMerge, we ran jsFSTMerge with a timeout of 5 minutes. Since jsFSTMerge is a semistructured approach we apply a looser evaluation metric. A resolution is considered correct if it is an exact syntactic match with R *or* if it is semantically equivalent. We determine semantic equivalence manually. jsFSTMerge produces a correct resolution on 3.7% of samples which is significantly lower than DeepMerge. Furthermore, jsFSTMerge does not have support for predicting Top-k resolutions and only outputs a single resolution. The remaining 96.3% of cases failed as follows. In 92.1% of samples, jsFSTMerge was not able to produce a resolution and reported a conflict. In 3.3% of samples, jsFSTMerge took greater than 5 minutes to execute and was terminated. In the remaining 0.8% jsFSTMerge produced a resolution that was both syntactically and semantically different than the user's resolution. In addition to effectiveness, DeepMerge is superior to jsFSTMerge in terms of execution time. Performing inference with deep neural approaches is much quicker than (semi) structured approaches. In our experiments, jsFSTMerge had an average execution time of 18 seconds per sample. In contrast, sequence-to-sequence models such as DeepMerge perform inference in under a second.

Sensitivity to Input Merge Conflict Size. We observe that there is a diverse range in the size of merge conflicts (lines in A plus lines in B). However, as shown in Figure 4.7, most (58% of our test set) merges are small, consisting of 7 or less lines. As a product of the dataset distribution and problem space size, DeepMerge performs better for smaller merges. We present aggregate Top-1 accuracy for the input ranges in Table 4.2. DeepMerge achieves over 78% synthesis accuracy on merge inputs consisting of 3 lines or less. On merge inputs consisting of 7 lines or less (58% of our test set) DeepMerge achieves over 61% synthesis accuracy.

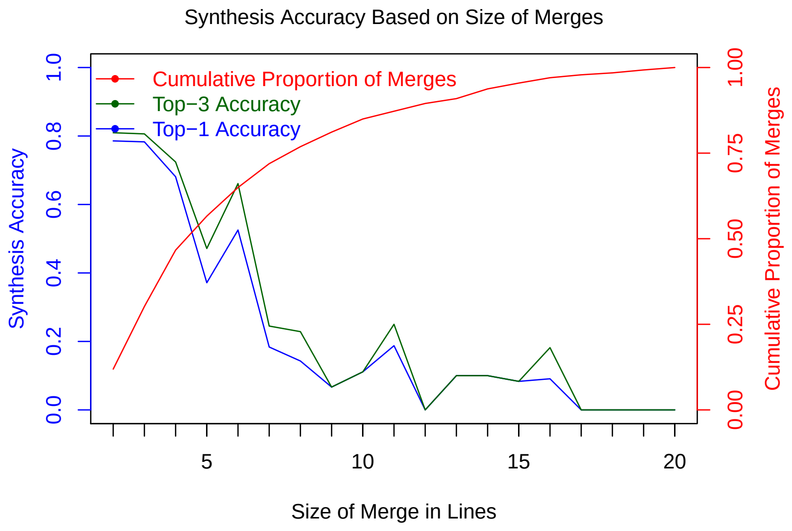


Figure 4.7: DeepMerge's performance vs merge input size. Cumulative distribution of merge sizes in red.

RQ2: Effectiveness of Suppressing Incorrect Resolutions

The probabilistic nature of DeepMerge allows for accommodating a spectrum of users with different tolerance for incorrect suggestions. "Confidence" metrics can be associated with each output sequence to suppress unlikely suggestions. In this section, we study the effectiveness of DeepMerge's confidence intervals.

In the scenario where DeepMerge cannot confidently synthesize a resolution, it declares a conflict and remains silent without reporting a resolution. This enables DeepMerge to provide a higher percentage of correct resolutions (higher precision) at the cost of not providing a resolution for every merge (lower recall). This is critical for practical use, as prior work has shown that tools with a high false positive rate are unlikely to be used by developers [Johnson et al. (2013)]. Figure 4.8 depicts the precision, recall, and F1 score values, for various confidence thresholds (with 95% confidence intervals). We aim to find a threshold that achieves high precision without sacrificing too much recall. In Figure 4.8, the highest F1-Score of 0.46 is achieved at 0.4 and 0.5. At threshold of 0.5, DeepMerge's top-1 precision is 0.72 with a recall of 0.34. Thus, while DeepMerge only produces a resolution one third of the time, that resolution is correct three out of four times. Compared to

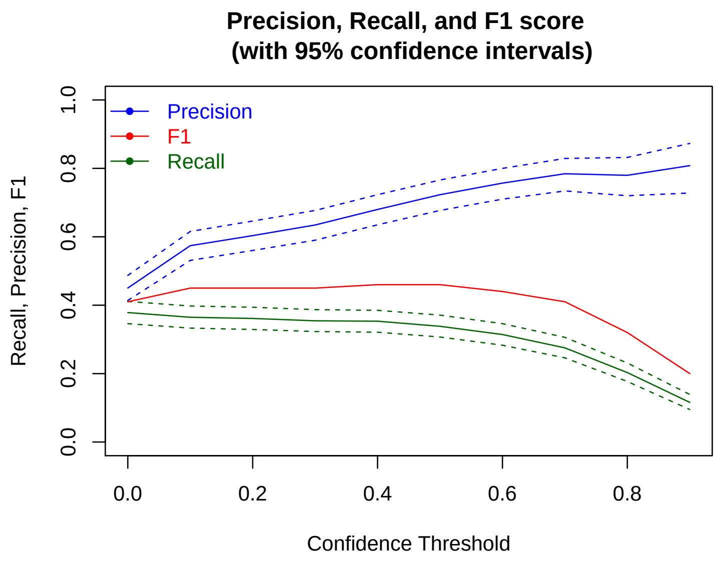


Figure 4.8: Top-1 precision and recall by confidence threshold.

DeepMerge with no thresholding, at a threshold of .5 DeepMerge achieves a 2x improvement in precision while only sacrificing a 10% drop in recall. Thresholds of 0.4 and 0.5 were identified as best performing on a held out validation set. We then confirmed that these thresholds were optimal on the held out test set reported in Figure 4.8.

RQ3: Categorical Analysis of Effectiveness

We now provide an analysis of DeepMerge's performance. To understand which samples DeepMerge is most effective at resolving, we classify the dataset into two classes: `CONCAT` and `OTHER`. The classes are defined as follows:

1. `CONCAT` - resolutions of the form AB or BA. Specifically:
    - R contains all lines in A and all lines in B.
    - There is no interleaving between A's lines and B's lines.
    - The order of lines within A and B is preserved.
2. `OTHER` - resolutions not classified as `CONCAT`.

Class	Top-1 Accuracy	Percent of Dataset
`CONCAT`	44.40%	26.88%
`OTHER`	29.03%	73.12%

Table 4.3: Accuracy and Distribution of classes.

`OTHER` samples can be any interleaving of any subset of lines.

Table 4.3 shows the performance of DeepMerge on each class. DeepMerge performs comparably well on each category suggesting that DeepMerge is effective at resolving conflicts beyond concatenation.

RQ4: Impact of Input Representation

We now evaluate the use of *Merge2Matrix* and show the benefit of the **Aligned Linearized** implementation used in DeepMerge.

We evaluate DeepMerge on each combination of summarization and edit aware alignment described in Section 3.2: *Naïve*, *Linearized*, *LTRE*, *Aligned naïve*, and *Aligned Linearized.* Table 4.4 shows the performance of each input representation on detection and synthesis. The edit-aware input formats: *LTRE*, *Aligned Naïve*, and *Aligned Linearized* attain an improvement over the edit-unaware formats. Our *Aligned* representations are more succinct and contribute to a large increase in accuracy over the edit-unaware formats. *Aligned Naïve* increases accuracy over our best edit-unaware format by 12.16% for top-1 and 12.27% for top-3. We believe this is due to the verbosity of including the underlying tokens as well as the Δ edit sequence. The combination of our edit-aware and summarization insights (*Aligned Linearized*) yields the highest accuracy.

Summary of Results

Our evaluation and baselines indicate that the problem of synthesizing resolutions is a non-trivial task, even when restricted to resolutions that rearrange lines from the conflict. DeepMerge not only can synthesize resolutions for more than a third of times, but can also use its internal confidence to

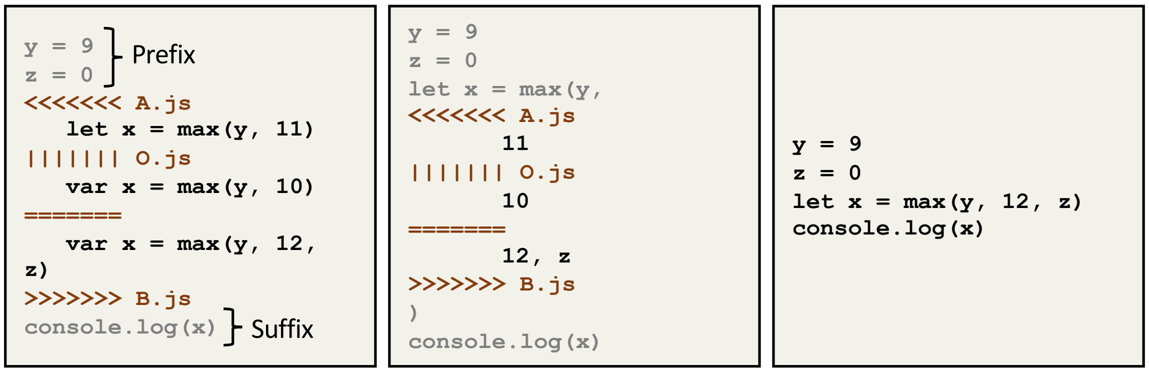
	Top-1	Top-3
Naïve	9.62%	14.09%
Linearized	15.25%	19.95%
LTRE	23.37%	29.21%
Aligned Naïve	27.41%	32.22%
Aligned Linearized	**36.50%**	**43.23%**

Table 4.4: Accuracy of different input representation choices.

achieve high precision (72%). DeepMerge can synthesize resolutions significantly more accurately than heuristic based, neural, and structured approaches. We also illustrate the need for edit-aware aligned encoding of merge inputs to help deep learning be more effective synthesizing non-trivial resolutions.

4.2.6 Limitations

We motivated the problem of *data-driven merge* and highlighted the main challenges in applying machine learning. We proposed DeepMerge, a data-driven merge framework, and demonstrated its effectiveness in resolving unstructured merge conflicts in JavaScript. We chose JavaScript as the language of focus in this work due to its importance and growing popularity and the fact that analysis of JavaScript is challenging due at least in part to its weak, dynamic type system and permissive nature [Jensen et al. (2009c); Kashyap et al. (2014)]. DeepMerge was more effective than other merge techniques for JavaScript, outperforming a structured approach by a wide margin. However, DeepMerge has a fundamental limitation that it is only capable of synthesizing resolutions at the line level. We plan to generalize our approach beyond line level output granularity. Lastly, we believe that DeepMerge can be easily extended to other languages and perhaps to any list-structured data format such as JSON and configuration files.



```
y = 9
z = 0
<<<<<<< A.js
   let x = max(y, 11)
||||||| O.js
   var x = max(y, 10)
=======
   var x = max(y, 12,
z)
>>>>>>> B.js
console.log(x)
```

```
y = 9
z = 0
let x = max(y,
<<<<<<< A.js
       11
||||||| O.js
       10
=======
       12, z
>>>>>>> B.js
)
console.log(x)
```

```
y = 9
z = 0
let x = max(y, 12, z)
console.log(x)
```

(a) Line-level conflict (b) Token-level conflict (c) Resolved merge

Figure 4.9: Example merge conflict represented through standard `diff3` (left) and token-level `diff3` (center), and the user resolution (right). The merge conflict resolution takes the token-level edit b.

4.3 MergeBERT

In this section, we introduce a follow up work to DeepMerge which presents multiple innovations to improve performance.

4.3.1 Motivation

Perhaps the largest limitation of DeepMerge is the restricted nature of its output granularity. DeepMerge synthesizes resolutions by selecting *lines* from the input merge tuple. This choice of granularity was motivated by empirical evidence that nearly 90% of JavaScript merge conflict resolutions exclusively consist of lines from the input. It also led to an improvement in accuracy as it drastically limits the search space of possible merge resolutions. As such, this also limits the expressiveness of its outputs. Although only a small subset of the dataset consisted of resolutions that could not be expressed by DeepMerge, a robust method ideally should be expressive enough to handle all possible resolutions. Consider the example in Figure 4.9. The resolution contains the `let` and `11` tokens from A and the `z` token from B.

4.3.2 Token Level Merge

MergeBERT, [Svyatkovskiy et al. (2021)], addresses DeepMerge's output granularity limitation by formulating merge conflict resolution as a token level classification task. Recall that the DeepMerge system first performs a line level `git merge` and then intervenes if a conflict occurs. In contrast, the MergeBERT system first performs a token level `diff3`. Starting with a token level unstructured merge has multiple advantages. First, it helps localize the merge conflicts to small program segments. For example, in Figure 4.9, although A and B edited the same lines, A's edit from `var` to `let` will be safely merged using a token level merged. Second, it allows for a natural formulation of resolution synthesis as a classification task.

The classification task is over nine resolution classes: A, B, O, $A + B$, $B + A$, A, B, $A + B$, $B + A$ where denotes excluding tokens that also exist in O. Over 99% of resolutions can be represented by using these labels. This formulation addresses DeepMerge's limitations by providing a higher level of expressiveness as well as exploiting the ease of classification over generation.

4.3.3 Architecture

MergeBERT also improves upon DeepMerge by taking advantage of advances in deep learning. DeepMerge employs a standard encoder-decoder framework which was state of the art at the time of writing. Since then, transformers such as BERT [Devlin et al. (2018)] and GPT [Brown et al. (2020)] have radically advanced the field of deep learning. MergeBERT is built on the pretrained CodeBERT model [Feng et al. (2020)] and finetuned on a large multi-lingual dataset of merge tuples.

4.3.4 Evaluation

To evaluate MergeBERT it was first compared to other neural approaches and to `diff3`. To be comprehensive, it was evaluated at both the token level and the line level. MergeBERT was also compared to existing state of the art structured and semi-structured merge language-specific merge approaches.

As seen in Tab. 4.5, language model baselines' performance on merge resolution synthesis is relatively low, suggesting that the naturalness hypothesis is insufficient to capture the developer intent when merging programs. This is perhaps not surprising given the notion of precision that does not tolerate even a single token mismatch.

Table 4.5: MergeBERT Precision and Accuracy vs Neural Baselines.

Approach	**Granularity**	**Precision**	**Accuracy**
LM	Line	3.6	3.1
DeepMerge	Line	55.0	35.1
`diff3`	Token	82.4	36.1
LM	Token	49.7	48.1
DeepMerge	Token	64.5	42.7
MergeBERT	Token	**69.1**	**68.2**

MergeBERT is based on two core components: token-level `diff3` and a multi-input neural transformer model. The token-level differencing algorithm alone gives a high top-1 precision of 82.4%, with a relatively low accuracy of only 36.1% (i.e., it doesn't always generate a resolution suggestion, but when it does, it is very often correct). Combined with the neural transformer model, the accuracy is increased to a total of 68.2%. Note, as a deterministic algorithm token-level `diff3` can only provide a single suggestion.

DeepMerge precision of merge resolution synthesis is quite admirable, showing 55.0% top-1 precision. However, it fails to generate predictions for merge conflicts which are not representable as a line interleaving. This type of merge conflict comprises only roughly one third of the test set, resulting in an accuracy of only 35.1% which is significantly lower than MergeBERT.

The DeepMerge model was also evaluated in combination with the token-level `diff3`. This enables DeepMerge to overcome the limitation of providing only resolutions comprised of interleavings of lines from the conflict region by interleaving tokens instead. As seen in Tab. 4.5 (DeepMerge with Token granularity) overall accuracy improves from 35.1% to 42.7%. However this still falls short of MergeBERT with precision that is 5% less (64.5% vs. 69.1%) and accuracy that is 25% less (42.7% vs 68.2%).

MergeBERT was also compared to state of the art structured and semi-structured merge tools. Since both JDime and jsFSTMerge are language-specific, to compare against MergeBERT, the evaluation uses a language-specific subset of conflicts (leading to slightly different results for MergeBERT on Java and JavaScript).

Table 4.6: MergeBERT Precision and Accuracy vs Semi-Structured Baselines.

Approach	**Language**	**% conf. w/ res.**	**Precision**	**Accuracy**
jDime	Java	82.1	26.3	21.6
MergeBERT	Java	**98.9**	**63.9**	**63.2**
jsFSTMerge	JavaScript	22.8	15.8	3.6
MergeBERT	JavaScript	**98.1**	**66.9**	**65.6**

As can be seen from Tab. 4.6, jsFSTMerge only produces a resolution for 22.8% of conflicts and when a resolution is produced by jsFSTMerge, it is only correct 15.8% of the time, yielding a total accuracy of 3.6%. This is in line with the conclusions of the creators of jsFSTMerge that semi-structured merge approaches may not be as advantageous for dynamic scripting languages. Because jsFSTMerge may produce reformatted code, cases where a resolution was produced but did not match the user resolution were manually examined. If the produced resolution was semantically equivalent to the user resolution, it was classified as correct.

A fair comparison of JDime to MergeBERT is challenging as scenarios occur where `diff3` reports a conflict, but JDime produces a non-conflicted merge. When comparing the JDime output to the actual historical user-performed merge conflict resolution, a simple syntactic match does not suffice. As a result of its AST matching approach, code generated by JDime is reformatted, and the original order of statements and other constructs are not always preserved. In an effort to accurately and fairly identify semantically equivalent merges the evaluation is based on, GumTree [Falleri et al. (2014)], an AST differencing tool, to identify and ignore semantically equivalent differences between JDime output and the user resolution, such as reordered method declarations. When JDime produces a resolution, it generates a semantically equivalent match 26.3% of the time, with an accuracy of 21.6%.

MergeBERT addresses the limitations of DeepMerge in three key ways.

1. Token level classification rather than line level resolution synthesis

2. Larger training dataset of ∼200,000 merge tuples (as compared to ∼8,000 in DeepMerge)

3. Larger state of the art model that takes advantage of a pretrained model of code

These key innovations led to a 2x improvement in accuracy over DeepMerge and a 3x performance improvement over existing semi-structured approaches.

4.3.5 Limitations and Threats to Validity

The choice of hyper-parameters in our model is based on prior work of others and generally accepted norms [Devlin et al. (2018)]. It's possible that exploring the hyper-parameter space could yield different results. The sample of conflicts and projects used in the study may pose a threat to the external validity of our work. We only considered public open-source projects hosted on GitHub, therefore, results may not generalize to closed source projects or repositories hosted on other platforms. To mitigate this threat, a diverse set of projects varying in size and language were selected.

4.4 Conclusion

In this section, we motivated the problem of data-driven merge and highlighted the main challenges in applying machine learning. We proposed DeepMerge, a data-driven merge framework, and demonstrated its effectiveness in resolving unstructured merge conflicts in JavaScript. We chose JavaScript as the language of focus in this paper due to its importance and growing popularity and the fact that analysis of JavaScript is challenging due at least in part to its weak, dynamic type system and permissive nature. Furthermore, we presented MergeBERT, a transformer-based program merge framework that leverages token-level differencing and reformulates the task of generating the resolution sequence as a classification task over a set of primitive merge patterns extracted from real-world merge commit data. MergeBERT exploits pretraining over massive amounts of code and then finetuning on specific programming languages, achieving 64–69% precision and 63–68% recall of merge resolution synthesis. Lastly, MergeBERT is flexible and effective, capable of resolving more conflicts than the existing tools in multiple programming languages. In conclusion, this section demonstrates the promise of implicitly learning program specifications through resolving merge conflicts.

CHAPTER 5

Test Generation

5.1 Introduction

In this section, we explore explicit specification inference in the domain of automated testing. We present approaches for both precondition and postcondition inference. In conventional automated testing, these specifications are typically provided by the developer or estimated using heuristics. Our goal in this work is to eliminate the necessity for developer interaction and heuristics by inferring specifications. For postcondition inference, we present a grammar based approach for ranking likely assertions based on a given a test prefix. For precondition inference, we present two distinct approaches. The first approach involves classifying an input as legal or illegal. However, this specification is not ideal due to its lack of interpretability, tendency to introduce errors, and ultimately trustworthiness. The second approach infers a precondition as a segment of code which returns `true` if the input is legal and `false` otherwise. This approach has several advantages, including naturalness, human readability, and statistical similarity to traditional programs. We perform evaluations of our first approach `TOGA` in conjunction with the automated testing tool EvoSuite [Fraser and Arcuri (2011)]. Moreover, we engage in a discussion regarding the limitations of such an approach and directions for future research.

5.2 Test Oracle Generation

5.2.1 Motivation

Unit testing is a critical aspect of software development. Effective unit tests for a component (a method, class, or module) can provide documentation, find bugs, and prevent regressions. In terms of documentation, unit tests express the unit's *intended* functionality, as conceived by the developer. Documenting the unit's functionality through a test conveys the unit's intended usage. The test also serves as a mechanism for detecting functional bugs during development. When executed, a test checks for mismatches between intended and *implemented* functionality. Such a mismatch causes a test failure, indicating a bug in the implementation. Furthermore, unit tests can alert the developer when future code changes introduce bugs. Effective (unit) testing during development can prevent release of buggy software and reduce costs by billions of dollars [Tassey (2002)].

A unit test is composed of two parts: a *prefix*, which drives the unit under test to an interesting state, and an *oracle*, which specifies a condition that the resultant state should satisfy. A sufficiently expressive test suite should document functionality under both normal invocations where the precondition is met, and *exceptional* behaviors where the precondition is violated. Figure 5.1 shows two examples of unit tests for a `stack` class. The tests document a normal invocation (Figure 5.1a) and an exceptional invocation (Figure 5.1b). Figure 5.1a shows a normal invocation of the unit where the test prefix instantiates a `stack` and makes sequential calls to `push` and `pop`. The test oracle, highlighted in red, asserts that the stack's `isEmpty` method should return `true` at the resultant state. If the unit contains a bug related to the tested behavior *e.g.*if pop always fails to remove an item from the stack, this test can aid in detecting the bug. On the other hand, Figure 5.1b shows the unit's expected behavior when the precondition of `pop` is not satisfied. In this case, the intended behavior of calling `pop` on an empty `stack` is to raise an exception. As such, the test oracle is the expected exception. The try-catch structure ensures that the unit does indeed raise an exception. If the unit contains a bug and does not raise an exception, the test will fail by executing `Assert.fail()`.

```
public void testPop() {
  Stack<int> s = new Stack<int>();
  int a = 2;

  s.push(a);
  s.pop();

  bool empty = s.isEmpty();
  assertTrue(empty);
}
```

(a) Normal invocation of `pop`

```
public void testPop() {
 try {
  Stack<int> s = new Stack<int>();
  s.pop();

  Assert.fail(); //fail
 } catch (Exception e) {
  //pass
 }
}
```

(b) Exceptional invocation of `pop`

Figure 5.1: Unit tests of a `Stack` class. The test oracles are highlighted in red. A correct implementation of `Stack` will be empty after a sequential push and pop and must raise an exception if pop is called on an empty stack.

It is clear that testing has immense benefits. However, authoring high quality unit tests is time consuming. On average, developers spend 15% of their time writing tests [Daka and Fraser (2014)]. As such, extensive work has been devoted to automated unit test generation [Fraser and Arcuri (2011); Pacheco et al. (2007); Lukasczyk et al. (2020); Zalewski (2015)]. However, test generation tools have no definitive knowledge of the developer's intended program behavior. This creates a challenge for generating functional test oracles. Instead, these tools consider program crashes and undesirable exceptions (e.g. null dereference or out of bound array accesses) as the test oracles. These tests are capable of finding numerous *safety* bugs in the unit's implementation, but are not sufficient to find violations of intended functionality and thus do not replace the need for manual unit tests.

Complimentary to automated test generation tools, extensive work has been devoted to test oracle creation from documentation and comments [Pandita et al. (2012); Tan et al. (2012); Blasi et al. (2018); Zhai et al. (2020); Goffi et al. (2016)]. We refer to these techniques as *specification mining* methods for test oracle generation. These methods rely on a restricted structure of documentation and a set of handcrafted rules to infer exceptions and assertions for a unit. However, given that users do not follow a prescribed format for writing documentation, or omit them altogether, these methods fail to extract interesting oracles on most real-world software components. In our evaluation, we show that these methods cannot infer bug-finding assertions for a benchmark of real world Java projects.

Recently, neural generative models have shown promise in generating functional test oracles [Tufano et al. (2021); Watson et al. (2020); Tufano et al. (2020)]. Neural methods are more flexible than specification mining approaches as they do not rely on fixed patterns. This flexibility makes neural generative models robust to imprecise or even missing documentation. However, we find in our evaluation that these methods struggle to generate accurate oracles due to the large space of possible assertions.

In summary, an effective test generation approach must infer both exception and assertion oracles that accurately reflect developer intent, and find bugs in real world programs. Additionally, such an approach must gracefully handle cases with ambiguous or missing documentation, or even missing implementations.

We propose a neural approach to infer both exceptional and assertion bug finding test oracles: TOGA. To address the limitations of existing neural generative methods, we propose a new approach that reformulates the oracle generation problem as a ranking over a small set of highly likely, possible oracles. We base our approach on the empirical observation that oracles in developer-written unit tests typically follow a small number of common patterns. We describe a taxonomy on these patterns and define a simple grammar that expresses this taxonomy. We use this grammar along with type-based constraints to restrict the space of candidate oracles and produce well-formed test oracles satisfying syntactic and type correctness. To perform ranking, we develop a two-step neural ranking procedure using pretrained transformers finetuned to score candidate oracles.

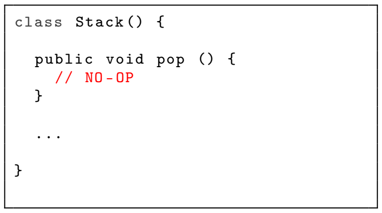
We evaluate our approach on both test oracle inference and bug-finding. Our technique improves accuracy by 33% over existing oracle inference approaches, achieving 96% accuracy on a held out test dataset that fits our grammar and constraints, and 69% accuracy on an overall assertion benchmark, a relative improvement of 11% over existing methods. Furthermore, we show that when integrated with a randomized test generation tool (EvoSuite), our approach finds 57 real world bugs in Java benchmark, `Defects4J` [Just et al. (2014)]. Our approach finds 30 bugs that are not found by any other automated testing method in our evaluation. We provide an open source implementation of TOGA at https://github.com/microsoft/toga.

Contributions. In summary, this dissertation:

1. Introduces a transformer (neural network) based approach to generating both exceptional and assertion oracles without relying on the unit's implementation.

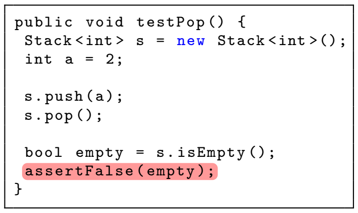
2. Derives adapted datasets for exceptional and assertion oracle training that incorporate method signatures and docstrings. These datasets are included in our open source release.

3. Implements TOGA, an end-to-end test generation technique that integrates neural test oracle generation with the automated test generation tool, EvoSuite.



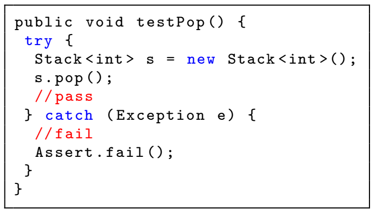
```
class Stack() {

  public void pop () {
    // NO-OP
  }

  ...
}
```

(a) Buggy implementation.



```
public void testPop() {
 Stack<int> s = new Stack<int>();
 int a = 2;

 s.push(a);
 s.pop();

 bool empty = s.isEmpty();
 assertFalse(empty);
}
```

(b) Regression oracle test.



```
public void testPop() {
 try {
  Stack<int> s = new Stack<int>();
  s.pop();
  //pass
 } catch (Exception e) {
  //fail
  Assert.fail();
 }
}
```

(c) Safety oracle test.

Figure 5.2: Regression and safety oracles for a buggy pop method. The regression oracle (employed by EvoSuite) assumes that the current behavior is correct, resulting in an incorrect oracle asserting that the stack is non-empty. The safety oracle (employed by Randoop) assumes that any non-crashing behavior is correct. As such, it results in an incorrect oracle asserting that an exception should *not* be raised when calling pop on an empty stack. Correct oracles for pop are shown in Figure 5.1.

4. Performs an extensive evaluation on test oracle inference. We demonstrate that our approach improves oracle inference accuracy by 33% and finds 57 real world bugs, including 30 bugs that are not found by any other method in our evaluation.

5.2.2 Related Work

We broadly categorize related work on unit test generation into (i) automated test generation methods, (ii) specification mining methods, and (iii) neural methods.

Automated Test Generation Tools

Automated unit test generation techniques use a combination of black-box or white-box techniques to generate interesting *test prefixes* for a unit. For example, tools such as Randoop, [Pacheco et al. (2007); Pacheco and Ernst (2007)], use random fuzzing of APIs of a unit to construct test prefixes that drives the unit to interesting states. Fuzzers such as AFL, [Zalewski (2015)], use fuzzing on the data inputs of a method to derive interesting values to drive a method. Korat, [Milicevic et al. (2007)], performs test generation for data structure inputs based on lazy unfolding of the type structure. PeX, [Tillmann and de Halleux (2008)], performs concolic execution [Sen et al. (2005); Godefroid et al. (2005)] to enumerate paths in a program and synthesize inputs using a constraint solver to derive inputs.

However, none of these tools explore the generation of *test oracles* to find functional bugs in a unit.

They rely on program crashes (from implicit or explicit assertions present in the code), or use exception type heuristics to distinguish between desirable and undesirable behavior. For example, null dereferences or out of bounds exceptions may be considered as undesirable. *Regression Oracles*, used by tools such as EvoSuite [Fraser and Arcuri (2011, 2014)], are intended to find *future* bugs and assume the unit under test is correctly implemented. This assumption allows for generating assertions from observed execution behavior. However, expecting a correct implementation is not always a safe assumption. When the implementation is buggy, the regression oracles are incorrect with respect to the intended behavior. That is, regression oracles are incapable of catching non-exceptional bugs, introducing false negatives.

Consider the example in Figure 5.2a that shows a buggy no-op implementation of stack pop. Figure 5.2b shows a generated unit test with a regression oracle. The test creates a stack and makes sequential push and pop calls. Since the pop method has a buggy no-op implementation, the stack will have one element after executing pop. Thus, the regression oracle is an incorrect assertion: the stack should *not* be empty.

Similarly, qualifying any exceptional output as a bug (*Safety Oracle*) can fail on correctly implemented methods, causing false positives, *e.g.*the intended behavior of calling `pop()` on an empty method is to throw an exception. Figure 5.2c shows a generated unit test with a safety oracle. A method that relies on safety oracles will also generate a passing test on the buggy pop implementation. Since pop is implemented as a no-op, an exception will not be raised when calling pop on an empty stack. In this case, the test oracle is implicit and asserts that an exception will not be thrown.

Therefore although automated test generation techniques find numerous non-functional bugs, and are useful for detecting regression bugs for future code changes, they are not a substitute for manually written unit tests documenting intended functionality.

Specification Mining Methods

Specification mining works [Pandita et al. (2012); Tan et al. (2012); Blasi et al. (2018); Zhai et al. (2020); Goffi et al. (2016)] aim to generate test oracles that accurately reflect the intended behavior (as in Figure 5.1). Unlike randomized test generation methods, specification mining approaches do not have any knowledge of the unit's implementation and as such, do not require execution. Instead, they rely on docstring documentation. Specification mining methods typically define a set of natural language docstring patterns. These patterns cannot capture all docstrings as program comments can be written flexibly without any necessary syntax or structure.

@Tcomment, [Tan et al. (2012)] defines natural language patterns along with heuristics to infer nullness properties. However, it cannot generalize to other property or exception types. An example heuristic @Tcomment employs is: generate an "expected NullPointerException" oracle if the keyword `@param` has the words `null` and `not` within 3 words of each other. ToraDocu, [Goffi et al. (2016)], uses a combination of pattern, lexical, and semantic similarity matching. Unlike @TComment, ToraDocu is not limited to nullness properties. However, ToraDocu can only generate oracles for exceptional behavior. JDoctor, [Blasi et al. (2018)], is an extension of ToraDocu that can also generate assertion oracles. More recently, MeMo [Blasi et al. (2021)] uses equivalence phrases in javadoc comments to infer metamorphic relations *e.g.*`sum(x,y) == sum(y,x)`, which are also used as test oracles. These methods can precisely determine oracles when code comments fit their expected patterns, but do not generalize when comments fall outside these patterns.

Lastly, C2S [Zhai et al. (2020)], generates JML specifications from docstrings. C2S does not manually define patterns, but instead performs a search over JML tokens. However, C2S relies on a developer written test prefix to filter candidate assertions. C2S has performance improvements over JDoctor in terms of specification synthesis accuracy, but does not improve performance in bug finding.

On average, real-world Java projects lack precisely structured docstring documentation. In our evaluation, we show that specification mining methods struggle to infer bug-finding oracles for a benchmark of real world Java projects.

Invariant Mining. There is a long line of work in deriving program invariants for the observed execution behavior of the program. These include systems such as Daikon [Ernst et al. (2007b)] and DySy [Csallner et al. (2008)], which extends the derived program invariants with symbolic execution. Recently, EvoSpex [Molina et al. (2021); Molina (2020)] combines observed executions with mutations to generate samples of both valid and likely invalid program states and applies a genetic algorithm to infer invariants for method postconditions. GAssert, [Terragni et al. (2020)] also utilizes an evolutionary approach to make inferred program invariants more accurate and compact. These approaches can be used to generate specifications and associated test oracles from the inferred invariants, but because they are based on the execution/symbolic behavior of the current implementation they will generate regression oracles, and cannot detect if bugs are already present in the unit under test.

Neural Methods

Recently, neural models have shown promise in generating test oracles and even entire unit tests. In contrast to specification mining methods, neural methods are not tied to hard coded patterns and can generalize to flexibly written docstrings. Furthermore, unlike randomized test generation tools, neural methods do not necessarily require knowledge or execution of the unit under test.

We refer the reader to CodeBERT [Feng et al. (2020)], for a discussion on the transformer architectures as applied to code. A transformer, like a recurrent neural network, maps a sequence of text into a high dimensional representation, which can then be decoded to solve downstream tasks. While not originally designed for code, transformers have found many applications in software engineering [Svyatkovskiy et al. (2021, 2020); Clement et al. (2020); Kanade et al. (2020)].

ATLAS is a neural-network-based approach to generate assertion oracles. Given a test prefix and the unit under test, ATLAS, [Watson et al. (2020)], generates assertions using a recurrent neural network. ATLAS relies on the unit's implementation and does not have any knowledge of the docstring documentation. ATLAS exclusively targets assertion oracle generation and does not attempt to infer any exceptional oracles.

Subsequent methods [Tufano et al. (2020); White and Krinke (2020); Mastropaolo et al. (2021)] have improved upon ATLAS by using a transformer-based seq2seq architecture pretrained on natural language and code. A transformer seq2seq model outperforms ATLAS in terms of inference accuracy. However, in section 5.2.5, we show that in combination with a test prefix generator, it struggles to find real world bugs in Java projects.

Lastly, AthenaTest, [Tufano et al. (2021)], is a transformer model approach to generate entire unit tests including both prefixes and oracles. AthenaTest takes as input the unit's context *e.g.*surrounding class, method signatures, etc., and implementation. Like the previous neural methods, it does not have any knowledge of the docstring documentation and relies on the implementation for inferring intended behavior.

5.2.3 Structure of an Oracle

Our approach addresses the limitations of existing neural methods by employing a ranking architecture over a set of candidate test oracles, rather than a generative model. In this section we develop a grammar for describing this set of test oracles. We first describe a taxonomy of commonly occurring oracle structures based on a qualitative investigation of a unit test dataset, and then use this taxonomy to inform the construction of our oracle grammar.

We develop a taxonomy of common oracle structures based on unit tests from methods2test [met], a dataset of Java unit tests collected from GitHub. We describe methods2test in Section 5.2.4

Unit test oracles typically test either exceptional behavior *i.e.*verifying an expected exception is raised or return behavior (assertion oracles). Additionally, an implicit exception oracle is usually present in tests with assertion oracles. That is, a test with an assertion oracle is not expected to raise an exception.

Taxonomy: We develop the following taxonomy of oracle usage, drawn from our observations of almost 200K developer-written tests. To develop this taxonomy, we manually inspected 100 random samples and categorized the most frequently occurring types of oracles we observed. To ensure that our grammar generalized well and did not overfit to our 100 inspected samples, we evaluated the proportion of tests in the dataset that fit the grammar (Section 5.2.5).

1. **Expected Exception Oracles.** Expected exception oracles verify that executing the test prefix with some invalid usage raises an exception. They are most frequently expressed with the following structure:

```
try {
    Unit.methodcall(invalidInput);
    Assert.fail();
} catch (Exception e) {
    verifyException(e, ExceptionType);
}
```

2. **Assertion Oracles.** Assertion oracles verify correct return behavior, although they will also fail if any exception is thrown. We observe several common assertion patterns:

   (a) **Boolean Assertions.** Boolean assertions are used to check some property of the unit under test is `true`/`false`. They are typically asserted directly on method return values:

   ```
   Unit.methodcall(input);
   assertTrue(Unit.getStatus());
   ```

   (b) **Nullness Assertions.** Nullness assertions usually check the return value of a method call that processes some input.

   ```
   assertNotNull(Unit.processInput(input));
   assertNull(Unit.processInput(invalidInput));
   ```

   (c) **Equality Assertions.** Developers typically write equality assertions to check the return value of a single method call. The return value is usually checked against a constant or literal representing the expected value. In many cases, especially when the unit under test incorporates some data structures, the expected value was previously passed as an argument to some method in the test prefix.

   ```
   String msg = "foo";
   Unit.sendMessage(msg);
   assertEqual(Unit.getLastMessage(), msg);
   ```

As we demonstrate in Section 5.2.5 this taxonomy captures a majority of tests (82% of a large dataset of developer written tests). This coverage could potentially be expanded by including other assertion types *e.g.*`AssertSame`, however, in developing the oracle taxonomy, our goal is not to express the entire grammar of Java test oracles. Instead, we aim to identify a minimal syntactic subset which represents many semantically equivalent oracles. Such a grammar greatly restricts the output space for the oracle generator to consider.

Uncommon oracles. We note several other patterns that occur more rarely, including equality assertions on arrays or assertions on multiple method calls (as opposed to a method call and a constant). We also note that there are some assertion patterns that we did not observe in *any* unit test, although they are often used to express invariants within programs. These include assertions with logical connectives and assertions with inequality constraints.

Test oracle grammar. Based on the taxonomy of common oracle structures, we develop a restricted grammar that expresses commonly used test oracles.

```
Test                 T   := O(P)
Prefix               P   := statement | P; P
Oracle            O(P)   := E(P) | R(P)
Except Oracle     E(P)   := try{P; fail();} catch(Exception e){}
Return Oracle     R(P)   := P; A
Assertion            A   := assertEquals(const|var,expr) |
                            assertTrue(expr) | assertFalse(expr) |
                            assertNull(expr) | assertNotNull(expr)
```

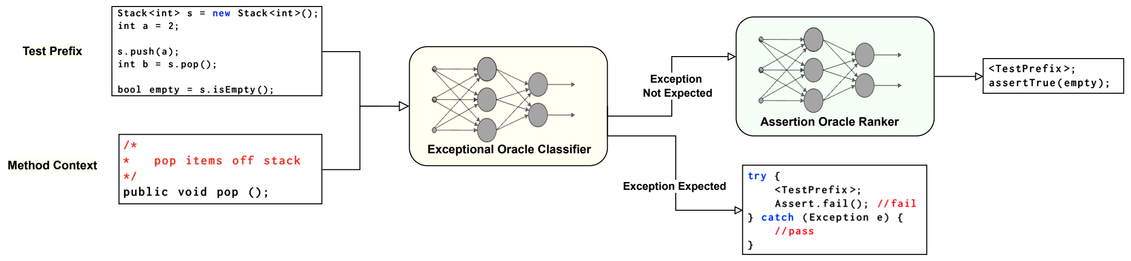
Intuitively, `TOGA` is a code-generation model for tests that is explicitly designed to exploit the structure of a unit test. This grammar succinctly describes a set of test oracles that are possible candidates for generation. In particular, given a test prefix P, we can synthesize either an exceptional oracle $E(P)$ or an assertion oracle on the return value of a method $R(P)$. Further the assertion oracle can be constructed using one of the five `assert*` constructs when instantiated with the return value and other constants and variables.

In the sections that follow, we demonstrate how to (i) prune this set, using type constraints, and (ii) rank the resulting possible test oracles using neural models.

5.2.4 TOGA: Neural Test Oracle Generation

In this section we present our approach for inferring test oracles that reflect developer intent. Unlike previous works, TOGA is capable of inferring both exception and assertion oracles. Furthermore, TOGA can handle units with vaguely written or absent docstrings, or even absent implementation. Our approach infers test oracles from only a given test prefix and unit context. Unit context may refer to method signature(s), or a docstring (if present). Notably, the unit context need not include the unit's implementation.

Method Overview



```
Stack<int> s = new Stack<int>();
int a = 2;

s.push(a);
int b = s.pop();

bool empty = s.isEmpty();
```

```
/*
*   pop items off stack
*/
public void pop ();
```

```
<TestPrefix>;
assertTrue(empty);
```

```
try {
    <TestPrefix>;
    Assert.fail(); //fail
} catch (Exception e) {
    //pass
}
```

Figure 5.3: Overall TOGA framework. The system takes as input a test prefix and a unit context. The unit context contains method signature(s) and docstrings, but not the implementation. It outputs a unit test with an inferred test oracle. The system has two main components: the Exceptional Oracle Classifier and the Assertion Oracle Ranker.

TOGA depicted in Figure 5.3 contains two key components: the Exceptional Oracle Classifier and the Assertion Oracle Ranker.

The Exceptional Oracle Classifier, described further in Section 5.2.4, is a pretrained transformer model fine-tuned on a binary decision task. The model decides if an exception should be thrown according to the developer intent conveyed through the unit context. If the classifier infers that the given test prefix should raise an exception, TOGA has found an exceptional oracle and can now generate a complete test. The resulting test has the *Expected Exception Oracle* format shown in Section 5.2.3. Otherwise, the classifier predicts that the input should not raise an exception and TOGA continues in the test generation process by invoking the Assertion Oracle Ranker.

The Assertion Oracle Ranker, described in Section 5.2.4, similarly uses a pretrained transformer model backbone. To address the limitations of existing neural assertion generation methods, our approach treats oracle inference as a ranking over a small set of possible common oracles. We base our approach on our observed taxonomy and defined grammar described in Section 5.2.3. We use this grammar along with type-based constraints to restrict the space of candidate oracles and enforce syntactic and type correctness. The model is is fine-tuned on ranking the set of candidate assertions given the test prefix and unit context. Each assertion in the set is ranked, and the highest ranked candidate is selected as the assertion oracle. Lastly, `TOGA` generates a test with the given test prefix and the inferred assertion oracle.

Exceptional Oracle Classifier

As mentioned previously, the Exceptional Oracle Classifier is based on a pretrained BERT transformer model. In particular, we use the CodeBERT [Feng et al. (2020)] model trained on both natural language and code masked language modeling. To train the Exceptional Oracle Classifier we fine-tune the pretrained model on the task of exceptional oracle inference. The fine-tuning is performed using a supervised dataset $D = ((p, c), l)_1, ...(p, c), l)_n)$ where p is a test prefix, c is a unit context, and l is a binary label ($l \in 0, 1$). A label of 1 indicates that the sample should raise an exception while a label of 0 indicates that it should not raise an exception.

Methods2Test* dataset. Our training dataset D is variation of the Methods2Test dataset [Tufano et al. (2021)], we call Methods2Test*. As the name suggests, Methods2Test is a corpus of unit methods and corresponding developer written unit tests extracted from over 91K open source Java projects. Originally created to train AthenaTest, Methods2Test is structured for the translation task from methods to tests. We adapt Methods2Test to our setting of exception oracle inference. Our adapted dataset, Methods2Test*, has modifications in both the input methods and developer written tests. The input method's implementation is removed, and the method docstring (if present) is added. The tests are modified to remove any exception or assertion oracles. These stripped oracles are used to create binary labels for expected exceptions. Lastly, we normalize the test method name to prevent potential information leakage. For example, a test method named `testThrowsException` would leak label information to the model. To remedy this, we rename all tests to follow the format:

`testN` where N is a positive integer. In summary, Methods2Test* is a supervised dataset for exception oracle inference. It excludes unit implementation and includes docstrings if present. Our resulting dataset contains a training set of more than 432,000 labeled samples.

Assertion Oracle Ranker

The Assertion Oracle Ranker is also based on the pretrained CodeBERT [Feng et al. (2020)] model. To train the Assertion Oracle Ranker we fine-tune the pretrained model on the task of assertion oracle inference. The fine-tuning is performed using a supervised dataset $D = ((p, c, a), l)_1, ...(p, c, a), l)_n$ where p is a test prefix, c is a unit context, a is a candidate assertion and l is a binary label ($l \in 0, 1$). A label of 1 indicates that the given candidate assertion accurately reflects developer intent. For a given p and c only one a can have a label of 1. The other assertions in the candidate set will have a negative label.

Atlas* dataset. Our training dataset D is a variant of the Atlas dataset [Watson et al. (2020)]. Atlas is a corpus of test case prefixes, corresponding method units, and assertions. Atlas was collected from 9K open source Java projects on GitHub. We modify Atlas to create our variant dataset Atlas*. Similar to our construction of Methods2Test*, we remove the method implementation, normalize the test method name, and remove the assertion from the test case. Then, we generate a set of assertion candidates for each sample and construct our labels to indicate the correct assertion in the set. Our negative samples are also taken from the candidate set of assertions. In total the resulting Atlas* dataset contains over 170,000 labeled (p, c, l) samples for supervised training.

Candidate Assertion Set Generation

To generate a candidate set of assertions, we use our grammar along with type-based constraints to restrict the space of candidate oracles and enforce syntactic and type correctness. Based on the return value of the unit under test, we iteratively construct a set of candidate assertions. Our assertion candidate generation algorithm is shown in Algorithm 3. If the assertion that is being added requires an additional value (`assertEquals`), our approach draws likely candidates from

Global and Local Dictionaries.

Global Constant Dictionary. The Global Constant Dictionary contains the most frequently occurring constant values in the training data. Our global dictionary contains the top K values of each type. The use of a global dictionary is inspired by our observation that the vast majority of constants in test asserts are a few common values. For example, over 90% of the integer constants in asserts in the ATLAS dataset are one of the top 10 most frequently occurring integer values.

Local Dictionary. In addition to the global constant dictionary, we also build a local dictionary based on values that appear in the test prefix. Note that these values are not necessarily constants. Variables that appear in the test prefix are also valid local dictionary entries. The use of a local dictionary is based on the observation that many assertions check against values that were previously passed as arguments to methods called in the test prefix.

At inference time, our method makes calls to the Assertion Oracle Ranker for each assertion in the set of candidates. The model outputs a predicted label along with a confidence score. We use this confidence score in post-processing to select the highest ranked assertion. The test prefix along with the selected assertion oracle is output as the generated test.

End-to-End EvoSuite integration

We have described a method, `TOGA`, to infer functional test oracles given a test prefix and unit context. However, in order to catch bugs, a test prefix that exercises the buggy behavior is necessary. To obtain a high quality test prefix, we use the randomized test generation tool EvoSuite. As mentioned in Section 5.2.2 EvoSuite generates a set of tests guided by coverage. We extract test prefixes by stripping EvoSuite's oracles from each test. In cases where a test contains multiple assertions, we extract the test prefix for each assertion individually. For each of the generated test prefixes, we invoke `TOGA` to infer a test oracle. In combination with a large set of prefixes that attempt to cover the entirety of the unit, our approach is able to generate functional test oracles that find real world bugs.

Algorithm 3 Assertion Template Creation

```
procedure CreateCandidateTemplates(GlobalDict, k, test)
    $cs \leftarrow \emptyset$                                          ▷ Template Candidates
    retVal = extractRetVal(test)
    t = type(retVal)
    LocalDict = createLocalDict(test)
    if retVal is an object then
        $cs \leftarrow cs \cup$ assertNull(retVal)
        $cs \leftarrow cs \cup$ assertNotNull(retVal)
    else if retVal is a boolean then
        $cs \leftarrow cs \cup$ assertTrue(retVal)
        $cs \leftarrow cs \cup$ assertFalse(retVal)
    end if
    for globalVal $\in$ GlobalDict.get(t) do
        $cs \leftarrow cs \cup$ assertEquals(globalVal, retVal)
    end for
    for localVal $\in$ LocalDict.get(t) do
        $cs \leftarrow cs \cup$ assertEquals(localVal, retVal)
    end for
    return $cs$
end procedure

procedure CreateLocalDict(test)
    LocalDict = { }
    for val in getValue(test) do                      ▷ Loop over all values in prefix
        LocalDict[type(val)] += {val}
    end for
    return LocalDict
end procedure

procedure extractRetVal(test)
    assign = getLastLine(test)                        ▷ last line will be an assignment
    retVal = getLHS(assign)
    return retVal
end procedure
```

When we obtain prefixes from EvoSuite, we assume that prefixes will be written in EvoSuite's standardized format. This allows us to identify the variables on which EvoSuite generated assertions in the `extractRetVal` method (Algorithm 3).

Lastly, we apply a confidence threshold to the assertion oracle ranker to suppress low confidence assertions. In these cases, only the exception oracle is applied to the test. Conceptually, this allows the model to avoid generating incorrect assertions in cases where the model believes all the candidate assertions are incorrect.

5.2.5 Evaluation

Research Questions. We consider the following research questions in our evaluation:

RQ1 Is `TOGA`'s grammar representative of most developer-written assertions?

RQ2 Can `TOGA` infer assertions and exceptional behavior with high accuracy?

RQ3 Can `TOGA` catch bugs with low false alarms?

Evaluation Setup

Datasets. Our evaluation uses the Atlas* and Methods2Test* datasets described in sections 5.2.4 and 5.2.4 respectively. For exceptional oracle inference, we evaluate on a Methods2Test* held-out test set of size 53,705. For assertion oracle inference, we evaluate on an Atlas* held-out test set of size 8,024.

Bug Benchmark. We evaluate real-world bug finding on the Defects4J [Just et al. (2014)] benchmark. Defects4J is a benchmark of 835 bugs from 17 real world Java projects. Each sample in the benchmark includes both buggy and fixed code versions. Each fixed program version is based on a minimal patch to fix the bug, and passes all the project tests, while each buggy program version fails at least one test. Each bug is based on an error that was logged in the project's issue tracker, involves source code changes, and is reproducible *i.e.*with a deterministic test. The benchmark also includes utilities for generating and evaluating test suites on the programs to determine if generated tests pass on the fixed versions and catch bugs.

Test environment. The evaluation was conducted on a Linux machine with Intel(R) Xeon(R) E5-2690 v3 CPU (2.60GHz) and 112GB main memory. As in the Defects4J environment, we use JDK 8.

RQ1: Oracle Grammar

We evaluate RQ1 on the original ATLAS dataset, which contains a total of 188,157 assertions mined from Java projects. To answer RQ1, we parse each assertion and check if it can be expressed in the grammar based on the assertion method name and structure of the AST. After excluding 695 samples that fail to parse, we find that 154,523 (82%) can be expressed by our grammar.

Of the 32,938 (18%) of assertions that cannot be expressed in our grammar, the majority (23,913, 13%) use assertion methods that we do not include *e.g.*`assertThat`, `assertSame`. In many cases (74% based on a manual inspection of 50 samples), the non-matching assertions appear to be symbolically equivalent to assertions expressible in our grammar (Figure 5.4).

```
assertThat(counter.get(),CoreMatchers.equalTo(2))
vs.
assertEquals(counter.get(),2)
```

Figure 5.4: The first assertion highlighted in red cannot be expressed in our grammar. However, the equivalent assertion highlighted in green, does fit our grammar.

Other assertions that did not match our grammar (5%) include equality assertions on expressions rather than variables or literals. For example:

```
assertEquals(id1.hashCode(),id2.hashCode())
```

Although we deliberately exclude generic assertions like these from our grammar, we note for a test executing in a deterministic environment, an equivalent property could be enforced through a syntactic rewrite.

Result 1: 82% of the developer-written assertions in the ATLAS dataset are in our grammar, and many other assertions are semantically equivalent to assertions expressed in our grammar.

RQ2: Oracle Inference Accuracy

To answer RQ2, Tables 5.1 and 5.2 reports accuracy results on a held-out test set. We include results for both exceptional and return test oracle inference.

For exceptional oracle inference (Table 5.1), our experimental setup involves the Methods2Test* dataset described in Section 5.2.4. There are no neural techniques for exceptional oracle inference that we are aware of. Instead, we include a random baseline (weighted coin) to illustrate the complexity of the problem space. The coin performs a random choice weighted on the distribution in our training set. In our training set, we observed that 80% of samples are non-exceptional. As such, the coin predicts negative labels frequently (and usually correctly), but rarely predicts a positive. The coin performs similarly to our approach in terms of accuracy, but significantly worse in terms of F1 score, as it rarely predicts a positive label correctly.

For assertion oracle inference (Table 5.2), our experimental setup involves the Atlas* dataset described in Section 5.2.4. The accuracy metric is syntactic: a suggestion is considered correct if it is an exact lexical match. As a baseline, we compare to a sequence-to-sequence (seq2seq) return test oracle model [Tufano et al. (2020)]. The seq2seq model is a transformer pre-trained on natural language and code with a beam search decoder. In contrast to our approach which performs ranking over a set of template assertions, the seq2seq model generates a test oracle token by token. As such, the model suffers due to the large space of possible oracles. We report results on two held out test sets: an *Overall* set and an *In-Vocab* set. The in-vocab set is the subset of the overall set that can be expressed by our grammar and vocabulary based on the local and global dictionaries. Our model achieves 96% accuracy on the in-vocab set compared to 63% by the seq2seq model, and 69% overall accuracy, an 11% relative improvement over the seq2seq model.

Result 2: Our assertion oracle inference model achieves over 69% accuracy compared to 62% accuracy from existing approaches. Our exceptional inference model achieves 86% accuracy with an F1 score of .39 relative to a weighted coin baseline's .15 F1 score.

Approach	Accuracy	Precision	Recall	F1-Score
TOGA Model	86%	.55	.30	.39
Weighted Coin	76%	.15	.13	.15

Table 5.1: RQ2: Evaluation of Exceptional Oracle Inference

Approach	In-Vocab Accuracy	Overall Accuracy
TOGA Model	96%	69%
Seq2Seq	63%	62%

Table 5.2: RQ2: Evaluation of Assertion Oracle Inference

Vocabulary size ablation. We perform an study on K, the vocabulary size of our global dictionary, to examine the tradeoff between generating a larger number of assertion candidates and ranking the assertion candidates accurately. Figure 5.5 shows the overall model accuracy, percent of samples supported by the vocabulary, and accuracy on samples supported by the vocabulary evaluated on the ATLAS* test set.

For K=0, the global dictionary is unused and only variables and constants in the local dictionary are considered the assertion generation. Using only the local dictionary can still generate correct assertion candidates for approximately 50% of the samples in the test set. Increasing K causes the model accuracy to decline slightly, but causes overall accuracy to improve because more correct assertion candidates are generated using the global dictionary. Once the vocabulary becomes too large however, the model accuracy starts to drop off, and setting higher Ks reduces overall accuracy.

In RQ2, we set K=8 based on tuning on the ATLAS* validation set. This setting achieves the best tradeoff between high model accuracy on the candidate set, and supporting a large set of likely assertions.

RQ3: Bug Detection

To answer RQ3, we run our end-to-end test generation system, integrated with EvoSuite. As described in section 5.2.4, the system uses EvoSuite to generate test prefixes guided by coverage.

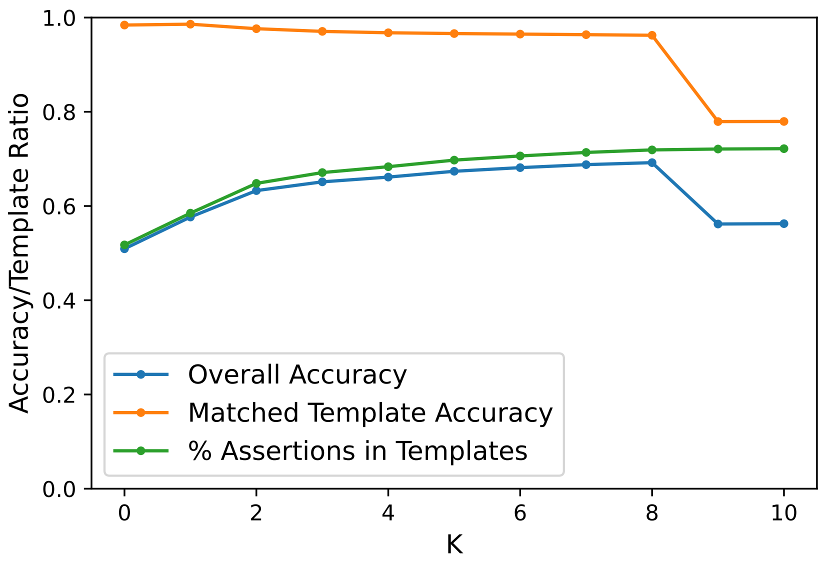


Figure 5.5: Evaluation of global dictionary size `K` on overall accuracy. Matched Template Accuracy indicates model accuracy when the candidate assertion set included the correct assertion. % Assertions in Templates indicates the percentage of dataset assertions that appear in the candidate assertion set for a given `K`.

Approach	Bugs Found (TPs)	FPR
EvoSuite + Ground Truth	120	0%
EvoSuite + `TOGA` (Ours)	57	25%
Randoop	20	87%
EvoSuite + seq2seq	6	46%
AthenaTest	0	15%*
EvoSuite + JDoctor	1	0.4%

Table 5.3: RQ3: Overall Bug Finding. *AthenaTest FPR based on 5 projects

Approach	Exception Raised	Exception Not Raised	Assertion Failure
EvoSuite + Ground Truth	45	27	51
EvoSuite + `TOGA` (Ours)	39	5	14
Randoop	20	0	0
EvoSuite + seq2seq	0	0	6
AthenaTest	0	0	0
EvoSuite + JDoctor	1	0	0

Table 5.4: RQ3: Number of bugs found by oracle type.

Our models are invoked to generate the test oracles.

Baselines. We consider the following baselines in this evaluation:

1. **Randomized Test Generation.** To represent randomized test generation we run Randoop [Pacheco et al. (2007)], which is a widely used and actively maintained test generation tool used for bug finding. We also run EvoSuite [Fraser and Arcuri (2011)] as a baseline, although EvoSuite's intended use case for regression testing limits its ability to find bugs present in the program. We run both Randoop and EvoSuite for 3 minutes per tested program, following the procedure used in Shamshiri et al. (2015).

2. **Neural Test/Oracle Generation.** To test neural methods, we compare with a seq2seq transformer finetuned to generate assertions [Tufano et al. (2020)]. We also evaluate against a whole-test generation model, AthenaTest [Tufano et al. (2021)].

3. **Specification Mining.** We use JDoctor's open source implementation to evaluate specification mining approaches. JDoctor supports exception oracle generation by parsing specific patterns in docstrings [Blasi et al. (2018)]. We integrate the generated oracles with the same EvoSuite-generated tests used by `TOGA` in this evaluation. Note that we do not evaluate on C2S [Zhai et al. (2020)] because the implementation is not publicly available.

Evaluation setting. We evaluate RQ 3 on the Defects4J [Just et al. (2014)] benchmark. To evaluate the effectiveness of oracles in detecting bugs present in the program, the generated tests are run on a buggy version of the unit under test. We consider a bug is found if a generated test both fails on the buggy program and passes on the fixed program. Since each fixed program is distinguished from the buggy program by a minimal patch fixing the specific bug, a test must be failing due to the specific bug if it only fails on the buggy version.

For the oracle generation methods in the evaluation that require a test prefix (`TOGA`, seq2seq, JDoctor), we evaluate on a set of *bug-reaching* EvoSuite test prefixes that exercise buggy behavior (and therefore can detect a bug given the right test oracle). We obtain this bug-reaching test prefix set by running EvoSuite with default settings *i.e.*coverage-guided on the fixed program versions to generate regression tests, and then selecting tests that fail the buggy program version, indicating they exercise buggy behavior. We extract these tests' prefixes as an evaluation set. Methods evaluated on this set are denoted "EvoSuite + <method>" in Tables 5.3 and 5.4.

It is important to note that our evaluation setting is fundamentally different from the regression evaluation setting in which the Defects4J benchmark is most often used. In a regression evaluation, tests are generated on the *fixed* program version and evaluated on the buggy version. Regression studies of randomized test generation tools report finding larger numbers of bugs than in our setting as they use regression assumptions to generate higher quality oracles [Shamshiri et al. (2015)]. In our setting where tests are generated on the buggy program version, regression test oracles will not find bugs as they assume the observed buggy behavior is correct.

In addition to evaluating the number of bugs found, we use per-test metrics as defined in Blasi et al. (2018). These metrics include false positives to evaluate the performance of an oracle generation method from a usage perspective. A method that generates many erroneously failing tests will not usable in a realistic application setting where a developer must inspect each failure to determine if they represent real bugs or false alarms.

A failing test is considered a "positive" while a passing test is a "negative". However, a "positive" does not necessarily indicate that the oracle caught the bug. A failing test can indicate one of two things:

1. **True Positive** - The test has a correct oracle and fails due to the buggy implementation.
2. **False Positive** - The test has an incorrect oracle and fails on the correct functionality of the unit in the fixed version.

To distinguish between these cases, we run the same test on the unit's fixed version. If the test fails on the fixed version, we can safely assume the test has an incorrect oracle, and is a FP.

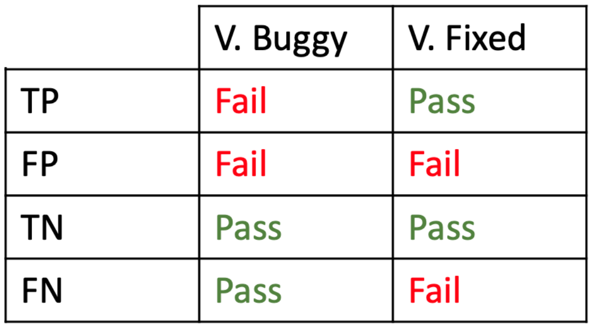
Similarly, a passing test can indicate one of two things:

1. **True Negative** - The test has a correct oracle and is testing correct functionality.

2. **False Negative** - The test has an incorrect oracle and is testing buggy functionality.

Again, to distinguish between these cases, we run the same test on the unit's fixed version. If the test fails on the fixed version, we can safely assume the test has an incorrect oracle, and is a FN.

We summarize the meaning of these metrics in Figure 5.6. In our evaluation, we summarize these metrics in the False Positive (FPR), which represents the rate of incorrectly failing tests on non-buggy code. A high FPR implies that a developer will need to validate many tests that have no utility and thus is a good metric for a bug-finding tool.



	V. Buggy	V. Fixed
TP	Fail	Pass
FP	Fail	Fail
TN	Pass	Pass
FN	Pass	Fail

Figure 5.6: Bug Finding Metrics

Discussion of RQ3 Results. Table 5.3 reports overall bug finding performance. EvoSuite + Ground Truth is a measure of EvoSuite's ability to generate bug-reaching tests. These tests were generated from the fixed program versions with regression oracles to obtain ground truth. We use this to distinguish between EvoSuite prefix generation performance from test oracle generation performance. EvoSuite + Ground Truth detects 120 bugs, indicating the best possible performance that any of the oracle generation methods can achieve on the EvoSuite test prefixes.

`TOGA` finds 57 total bugs, including 30 that are not found by any other method in our evaluation. The next best performing method, Randoop, finds 20 bugs but with a much higher false positive rate. Of the two tested neural methods, AthenaTest does not generate any bug-finding tests. The seq2seq model run on EvoSuite-generated test prefixes finds 6 bugs, but incurs a higher error rate. The specification mining tool, JDoctor, only finds one bug, but is the most precise oracle generation method in the evaluation.

```
class KeyedValues() {
  public void removeValue(int i){
    this.keys.remove(i);
    this.values.remove(i);

    // Bug, misses update
    if (i < this.keys.size()) {
      rebuildIndex();
    }
  }
  public int itemCount() {
    return this.index.size();
  }
}
```

(a) Buggy implementation.

```
public void testKeyedValues() {

  KeyedValues kv;
  kv = new KeyedValues();

  Short short = new Short(2);
  kv.insertValue(0, short0, 2);
  kv.removeValue(0);

  // Asserts buggy itemCount 1
  // is correct and misses bug
  assertEquals(1,kv.itemCount());
}
```

(b) Regression oracle test.

```
public void testKeyedValues() {
  try {
    KeyedValues kv;
    kv = new KeyedValues();

    Short short = new Short(2);
    kv.insertValue(0,short0,2);
    kv.removeValue(0);

  // No exception raised
  } catch (Exception e) {
    fail(); // misses bug
  }
}
```

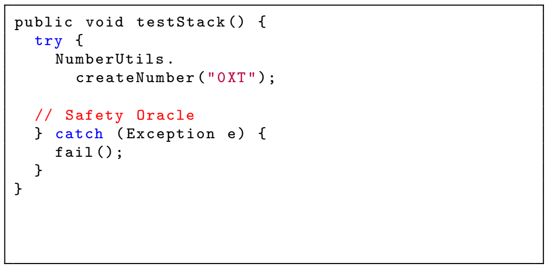
(c) Safety oracle test.

```
public void testKeyedValues() {

  KeyedValues kv;
  kv = new KeyedValues();

  Short short = new Short(2);
  kv.insertValue(0, short0, 2);
  kv.removeValue(0);

  // Asserts itemCount should be 0
  // Test fails and identifies bug
  assertEquals(0, kv.itemCount());
}
```

(d) `TOGA` generated oracle.

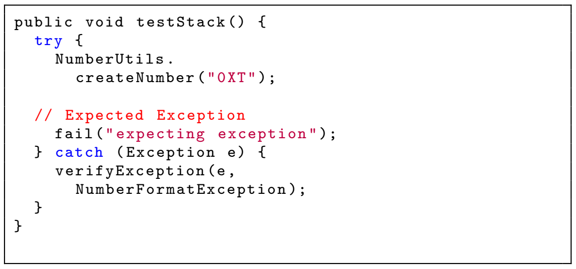
Figure 5.7: Different types of test oracles for a bug in the `removeValue` method from the Java `Chart` project. The bug causes a data structure to return an incorrect item count when the most recently added item is removed. Although the test input exposes this behavior, regression and safety oracles will generate a false negative by passing the buggy behavior, either by generating an incorrect assert statement or because the bug does not cause any exceptions to be thrown. Only the oracle generated by `TOGA` correctly asserts that itemcount should be 0 after an item is inserted and removed detects the bug. `TOGA` is the only system in our evaluation that correctly identifies this bug.

Table 5.4 reports a breakdown of bug finding performance on three different bug types: unexpected exception raised, expected exception not raised, and assertion failures. `TOGA`'s ability to infer exception oracles correctly is critical to its bug finding performance. Overall 44 of the bugs it finds are exceptional, and 5 involve expected exceptions not being raised. None of the other methods in the evaluation detect any expected exception not raised bugs. Of the other evaluated methods, AthenaTest and JDoctor are both capable of generating expected exception bugs in principle but in practice do not generate any in the evaluation. For raised (unexpected) exceptions, `TOGA` exception model correctly identifies 39/45 of them are unexpected exceptions. This demonstrates the value of using a neural model for exception oracle generation, which is more flexible than the fixed rules used by a tool like Randoop.



```java
public void testStack() {
  try {
    NumberUtils.
      createNumber("0XT");

  // Safety Oracle
  } catch (Exception e) {
    fail();
  }
}
```

(a) Safety oracle test.



```java
public void testStack() {
  try {
    NumberUtils.
      createNumber("0XT");

  // Expected Exception
    fail("expecting exception");
  } catch (Exception e) {
    verifyException(e,
      NumberFormatException);
  }
}
```

(b) `TOGA` generated oracle.

Figure 5.8: Generated oracles testing a buggy `createNumber` method in the Java `Lang` project. The bug prevents a `NumberFormatException` from being raised on an invalid input. The oracle generated by `TOGA` correctly checks that an exception should be raised on the invalid input, and fails when no exception is raised due to the bug. A safety oracle cannot detect the absence of an exception. `TOGA` is the only system in our evaluation that detects this bug.

`TOGA` also identifies 14 assertion bugs. The only other method in the evaluation to generate assertion oracles that catch bugs is the seq2seq generative model, which catches 6 bugs. This shows that while `TOGA` ranking-based oracle generation procedure is effective for bug finding, its overall performance in bug finding comes from providing a unified method for oracle generation that can detect all three types of bugs. In contrast, none of the methods in the evaluation are successful in generating oracles for more than one type of bug, although JDoctor and AthenaTest can in theory generate oracles for all three classes of bugs.

The AthenaTest and seq2seq assertion generation models do not effectively find bugs. This evaluation illustrates the challenges in neural oracle generation. In practice we found that both AthenaTest's whole test generation and the seq2seq assertion model generated many tests and oracles that were not executable. The AthenaTest authors noted this issue in their evaluation, where they found that only 16% of the generated test cases were executable without errors and actively tested the unit under test [Tufano et al. (2021)]. The oracle generation model generated 34% executable oracles, and of these we observed that a further 5% were tautologies, resulting in an overall yield of 29% potentially meaningful oracles. In contrast, the ranked oracle generation used by `TOGA` always generates oracles that are executable and exercise the unit under test. Note that due to the large volume of generated test candidates (30 per tested method) that must be individually compiled and run when following the procedure in Tufano et al. (2021), we estimate the false positive rate of AthenaTest on five projects and otherwise only generate tests specifically on methods exercising buggy code.

The specification mining method, JDoctor also does not effectively find bugs, but it generates oracles precisely. JDoctor only produces an exceptional test oracle if there is a docstring comment indicating specific behavior. However, on the projects in the Defects4J benchmark, this approach only succeeds in generating test oracles to catch a single bug. We observed that in practice, many buggy methods either had vaguely worded docstrings or lacked docstrings entirely, and JDoctor created few test oracles as a result. JDoctor's inability to generate sufficient oracles to effectively find bugs illustrates why robustness to vague or missing docstrings is a important requirement for effective oracle generation. In many cases, the bugs detected by `TOGA` occurred on methods that lacked docstrings entirely, where any specification mining approach would not be able to identify them.

EvoSuite vs. `TOGA` Performance: Finding bugs requires both test prefixes that reach buggy behavior and oracles that correctly identify the bug. For the oracle generation methods in this evaluation, we distinguish the performance of the test prefix generator (EvoSuite) by evaluating the generated test prefixes with the ground truth oracles. Out of the 835 bugs in the Defects4J benchmark, the EvoSuite generated tests reach 120 bugs. That is, overall EvoSuite + `TOGA` misses 715 Defect4J bugs due to EvoSuite not generating reaching test prefixes, and 63 bugs due to `TOGA` not generating correct oracles. This result highlights that generating test prefixes to reach buggy code remains a challenging open problem, and improving the test prefix generator used with `TOGA` could have large impact on bug detection performance.

`TOGA` Exception Oracle Error Analysis: For a single focal method, EvoSuite often generates multiple test cases. For some focal methods (~10%), EvoSuite generates both exceptional and non-exceptional input states. However, `TOGA` rarely predicts (4%) differing exception oracles for the same focal method, regardless of input state. This observation suggests that `TOGA` is conditioning primarily on the focal method signature rather than particular input states.

`TOGA` Assertion Oracle Error Analysis: We performed a manual analysis of ground truth oracles and found that of 229 total assertion oracles, 31 were predicted correctly. The remaining 198 predictions can be broken down as follows: 106 of the ground truth assertions could not be expressed with the given vocabulary, 13 could not be expressed with the grammar, and 9 were not predicted because `TOGA` incorrectly predicted an exceptional oracle. In the remaining 70 samples, the ground truth oracle could be expressed by the vocab and grammar, but the model made the wrong prediction, resulting in an in-vocab accuracy of 31% on the bug-reaching EvoSuite tests. This is significantly lower than `TOGA`'s 96% in-vocab accuracy on ATLAS*. The difference in performance suggests that the distribution of tests in ATLAS* is very different from EvoSuite's generated tests. A model trained specifically on EvoSuite generated (test, oracle) pairs instead of ATLAS* pairs may result in better performance.

Result 3: Our approach finds 57 bugs in real world Java projects, 30 of which are not found by any other method in the evaluation.

Case Studies

We consider two case studies of bugs that are detected by `TOGA` in our evaluation but not by other methods.

Assertion bug case study. The first case study, shown in Figure 5.7 involves a bug in a key-value store used in the Chart Java project. The buggy method, shown in Figure 5.7a, contains incorrect logic that prevents the data structure from updating its index when the most recently added item is removed. This causes the `itemCount()` method to return an incorrect count, because it bases the item count on the index.

The EvoSuite-generated test for this method shown in Figure 5.7b uses a regression oracle and generates an assertion based on the observed execution behavior. Because the method is buggy, this results in an incorrect assertion being generated, which not only fails to catch the bug but also could potentially make future detection of the bug more difficult. Figure 5.7c shows a simplified version of an unexpected exception oracle, which is the approach used by Randoop in the evaluation.

In contrast to these two approaches, `TOGA` generates the correct oracle by performing a ranking over a small number of assertions on integers and the return value of `kv.itemCount()`. This identifies that after calling `removeValue(0)`, the most likely assertion is `assertEquals(0, kv.itemCount())`.

Expected exception case study. Figure 5.8 illustrates how `TOGA` is able to catch an expected exception bug detected in our evaluation. The bug in the `NumberUtils.createNumber` method of the Java `Lang` project prevents the method from correctly detecting invalid inputs and raising an exception. The exception ranking model predicts that the `createNumber("OXT")` call should raise an exception based on the method signature and context, and `TOGA` generates an oracle based on this prediction to pass the test if an exception is raised on fail otherwise. In contrast, a safety oracle that checks for unexpected exceptions cannot detect this type of bug where a raised expected is desired behavior. 5 of the bugs found the `TOGA` in the evaluation are expected exception bugs, and no other tool in evaluation finds any expected exception bugs.

Threats to Validity

We consider three potential sources of bias that could conceivably threaten the validity of our results: (i) test dataset bias, (ii) bug dataset bias, and (iii) bias from EvoSuite performance. Both of the unit test datasets ATLAS and Methods2Test sourced tests from publicly available Java projects, and filtered their results using heuristics such as GitHub star count and presence of matching focal methods to select tests for inclusion. Bias in these datasets towards specific applications or types of tests may effect the validity of RQ1 and RQ2. However, we note that these datasets are large (sourced from 91K open source Java projects in the case of Methods2Test), and therefore likely to be representative of common patterns in Java unit testing.

Our RQ3 bug evaluation dataset, Defects4J, is much smaller with 835 samples from 17 projects due to difficulty in constructing minimal bug samples, so bias towards specific applications or bug types is possible. However, Defects4J only contains large, widely used projects and difficult real-world bugs, so evaluations on this benchmark are likely to be indicative of real world performance on large software projects.

Finally, bias in EvoSuite's test prefix generation is also a potential threat to validity for RQ3. EvoSuite can only generate bug-reaching tests for a fraction of the Defects4J bugs (120 out of 835), and may be biased towards classes of bugs that are easier to reach with coverage guided exploration.

5.2.6 Limitations

Grammar and Vocabulary: `TOGA` makes the tradeoff of supporting a restricted set of commonly used oracles, but predicting oracles in that set accurately. A limitation of this approach is that `TOGA` can only generate oracles that can be expressed by the grammar and exclusively contain values that appear in the vocabulary. We conducted a manual analysis of `TOGA` predictions in RQ3. When TOGA did not correctly predict a bug-finding assertion, in 54% of the cases the assertion value did not appear in the vocabulary, and in 8.5% of cases the assertion could not be expressed in the grammar. For example, `TOGA` could not predict the following ground truth assertion as the string literal is not contained in either the global or local dictionaries:

```
assertEquals("\"qDxD_5>q,)‘dEgM", string0)
```

While our grammar limitation is strict, we found that approaches with unlimited vocabularies also did not correctly predict these oracles.

Out of distribution training: `TOGA` is also limited by its dependence on datasets of (primarily) developer-written unit tests for both training and vocabulary learning. However, the RQ3 test set is taken from EvoSuite, an out of distribution sample set. As a future direction, `TOGA` could be trained on an EvoSuite generated dataset for a model that more closely fits an end-to-end automated testing distribution.

Dependencies on EvoSuite: `TOGA` assumes a particular structure of the test prefixes generated by EvoSuite to select the variable to assert on. However, as long as the assertion variable is specified to TOGA and defined somewhere in the test prefix, the test prefix could conceivably have any format.

Therefore, integrating TOGA with another test generation method might require integrating a suitable mutation analysis tool such as PIT [Coles et al. (2016)] to select variables on which to generate assertions.

Acknowledgements

We would like to thank Michele Tufano and Alexey Svyatkovskiy for their help with the ATLAS and Methods2Test datasets and running AthenaTest, and helpful discussions and feedback.

5.3 Precondition Generation

In TOGA, preconditions were inferred as a binary classification task. That is, given an input to a method, is the input legal or illegal? However, preconditions are typically expressed as predicates, which can be evaluated over the space of all inputs. Ideally, a technique which classifies an input as legal / illegal should be interpretable. This dissertation explores inferring interpretable preconditions through a predicate. This is challenging as precondition predicates are rarely written in practice. There is rich non-neural program analysis literature in interpretable precondition inference. However, the predicates inferred by such approaches are often unnatural and not easily consumed by a human or neural model.

5.3.1 Motivation

This dissertation introduces a novel approach for inferring *natural* preconditions from code. Our technique produces preconditions of high quality in terms of both correctness (modulo a test generator) and naturalness. Methods in programming languages are partial functions that map a subset of inputs to an output value. Unlike mathematical functions, a method is expected (according to the designer of the method) to be invoked on only a subset of all possible input values. Invoking a method on illegal inputs can result in an exception, crash, or undefined behavior. A boolean function which accepts all legal inputs and rejects all illegal inputs is a *precondition.*

Preconditions serve a valuable purpose for both software engineering tools and programmers.

Preconditions assist programmers in comprehension and correct usage of an API [Yun et al. (2016)]. Reasoning about which inputs are legal vs. illegal is critical to software correctness [Cousot et al. (2011); Sutton (1995)]. Preconditions also enable effective application of automated testing [Pacheco et al. (2007); Godefroid et al. (2005); Tillmann and de Halleux (2008); Fraser and Arcuri (2011)], static analysis [Lattner; Xie and Aiken (2005)] and verification [Leino (2010)]. In general, preconditions are essential for making modular analyses more precise. Ultimately, they play a significant role for both tools and programmers alike.

Various approaches have been proposed to automatically infer preconditions. Static approaches are conservative and lead to false positives [Chandra et al. (2009); Cousot et al. (2013); Seghir and Kröning (2013)]. Dynamic approaches which employ a test generator [Padhi et al. (2016); Sankaranarayanan et al. (2008); Ernst et al. (2007a); Astorga et al. (2018)], have been successful in generating correct preconditions. However, the inferred preconditions are often unnatural and difficult for humans to reason over. Such approaches involve combining "features" to construct a boolean predicate. A correct, dynamically inferred, predicate accepts legal inputs and rejects illegal ones, modulo a test generator. Features can be defined as predicates over a method's inputs (e.g. `x > 0`) [Padhi et al. (2016)] or object state (e.g. `foo.size() > 0`) [Astorga et al. (2019)]. By combining these features, we show that the resulting predicate can become unnecessarily complex and difficult to comprehend, and ultimately, unnatural. Natural preconditions are desirable for a variety of reasons. Firstly, they are easier for programmers to consume and comprehend.

Natural preconditions are better suited for human-in-the-loop settings such as integrated development environments and interactive program verifiers. In general, natural preconditions are easier to understand and therefore easier to specify and refine, contributing to a more effective and accurate development or analysis [Fava et al. (2016)]. Likewise, natural preconditions phrased in a similar language to the original method are likely to be preferred by statistical models for code that increasingly underlie software engineering tools [Hindle et al. (2012)]. Leveraging natural preconditions aligns well with the evolving landscape of software engineering tools and statistical modeling and enhances the efficacy of human-involved processes [Sutton et al. (2023);

Hellendoorn et al. (2019); Zhai et al. (2020)].

In this dissertation, I propose a dynamic approach for inferring natural and correct preconditions. In contrast to existing approaches, we infer a precondition by performing transformations to the target method. Rather than constructing preconditions from scratch through feature combination, we leverage the structure of the method to seed and guide inference. We introduce a pragmatic and efficient framework that tackles the intricacies of real-world constructs found in modern programming languages including complex control flow, heap manipulations, interprocedural calls, and lengthy call chains. Furthermore, we discuss the challenges involved in developing an algorithm to infer preconditions through program transformation. To illustrate this, we present alternate naive approaches and illustrate pitfalls of these algorithms. Through this, we arrive upon an algorithm which iterates between input generation and refined program transformation, ultimately converging on a correct and natural precondition.

We evaluate our approach in terms of both naturalness and correctness by performing a comparative study to prior work. We find that our preconditions are comparably correct yet strongly preferred by humans in terms of ability to reason over. By conducting a user study, we find that human participants were able to more accurately reason over our preconditions in a shorter time span vs the state of the art approach. Consumers of our preconditions completed reasoning tasks more accurately (95.56%) with an average total duration of 144 seconds. Consumers of the preconditions inferred by prior work took nearly twice as long (273 seconds) to finish the study and answered with lower accuracy (83.89%).

Finally, we present the implementation of our approach as a tool and demonstrate its application on a significant scale across 87 real-world Java projects. This yields ∼18k (method, precondition) pairs, obtaining the first large-scale dataset of preconditions [Zhai et al. (2020); Fava et al. (2016)]. The resulting dataset contains 3,547 specifications. This dataset not only underscores the viability and effectiveness of applying our tool at scale, but also provides a benchmark for evaluation. In this dissertation, we use this benchmark to conduct a large-scale evaluation of the design decisions made to enhance effectiveness of our approach. We also present characteristics of the dataset which in

```java
public BigInteger[] DivideAndRemainder(BigInteger val) {
    if (val.m_sign == 0)
      throw new ArithmeticException("Division by zero error");
    BigInteger[] biggies = new BigInteger[2];
    if (m_sign == 0) {
      biggies[0] = Zero;
      biggies[1] = Zero;
    } else if (val.QuickPow2Check()) {
      int e = val.Abs().getBitLength() - 1;
      NetBigInteger quotient = Abs().ShiftRight(e);
      int[] remainder = LastNBits(e);
      biggies[0] =
        val.m_sign == m_sign ? quotient : quotient.Negate();
      biggies[1] =
        new BigInteger(m_sign, remainder, true);
    } else {
      int[] remainder = (int[]) m_magnitude.clone();
      int[] quotient = Divide(remainder, val.m_magnitude);
      biggies[0] =
        new BigInteger(m_sign * val.m_sign, quotient, true);
      biggies[1] =
        new BigInteger(m_sign, remainder, true);
    }
    return biggies;
}
```

```java
boolean M_pre(NetBigInteger val) {
  if (val == null)
    return false;
  if (val.m_sign == 0)
    return false;
  return true;
}
```

(a) Generated by our approach.

```
(not (val == null))
and (
    (val.getIntValue() == val.m_sign and (
       (val.getIntValue() <= -1) or
       ((not (val.getIntValue() <= -1)) and
           (not (val.getIntValue() <= 0))
       ))
    ) or
    (not (val.getIntValue() == val.m_sign))
)
```

(b) Generated by Proviso.

Figure 5.9: A method and its corresponding preconditions.

turn offers valuable insights to guide future research efforts in the field of precondition inference.

Contributions In summary, our work makes the following contributions:

1. We introduce a novel methodology for inferring correct and natural preconditions through program transformation.
2. We present an instantiation of our approach as an open-source tool which can be applied at scale.
3. We perform a comparative evaluation to the state-of-the-art precondition generation approach and show that ours are more easily reasoned over by humans through a user study.
4. We curate a substantial dataset of ∼18k as a benchmark for large-scale evaluation, accompanied by an exploratory study of its characteristics.
5. We perform a large-scale evaluation of the decisions made to design our approach, showing they contribute to higher quality preconditions.

In this section, we motivate and illustrate our approach using the example in Figure 5.9. Consider a Java method `DivideAndRemainder`, for which we aim to infer a precondition. The target method implementation for dividing arbitrary-precision integers is shown on the left of the figure. A correct precondition for this method does not allow inputs where the denominator argument `val` is either the `BigInteger` Zero or `null`. Indeed, the method *explicitly* throws an `ArithmeticException` when the denominator is zero, which is captured by the check (`val.m_sign == 0`) on line 3. Additionally, the method *implicitly* throws a `NullPointerException` upon execution of line 2 when `val` is null.

Without an explicit precondition, it may be difficult to reason about which inputs are illegal. From a cursory analysis of a method's implementation it may be evident from the explicit `throw` expression that an input `val` with `m_sign` equal to 0 is an illegal input. However, the second exceptional input, a `val` equal to `null`, is less obvious. This exception is checked for and thrown implicitly by the JVM during the evaluation of expression `val.m_sign`. The challenge of inspecting the implementation is further compounded by the need for interprocedural reasoning since exceptions could be thrown by callees of the target method.

Our approach is based on the observation that methods usually contain sufficient structure of the precondition to seed and guide the inference process. By leveraging this insight, our approach is able to generate the precondition shown in Figure 5.9a through a series of program transformations. The precondition is not only correct (modulo a test generator) but also natural. In contrast, existing approaches attempt to synthesize the precondition from scratch through boolean combinations of predicates. Figure 5.9b shows such a precondition synthesized by a state-of-the-art approach Proviso [Astorga et al. (2019)], which is unnatural albeit semantically equivalent.

Proviso generates the precondition using a combination of predicates and inequalities over numeric primitives as well as boolean and integer returning observer methods. By combining such predicates, the precondition can become unnecessarily complex. In particular, the numeric inequalities and negations on lines 4-6 can be simplified to the equivalent clause: `val.getIntValue() != 0`. Further simplification results in the semantically equivalent predicate `val != null and val.m_sign != 0`.

In contrast, we find that leveraging the program structure and performing program transformations often leads to more natural preconditions. Such an approach gives rise to two subproblems: (1)

lifting implicit JVM checks to explicit source code checks and (2) removing code irrelevant to the precondition. In Figure 5.9a, this amounted to inserting the nullness check on line 2 and removing lines 4-24 (in the target method) which are unrelated to the precondition. In Section 5.3.2 we discuss the challenges of these two subproblems as well as a discussion of our design decisions in Section 5.3.3.

5.3.2 Problem Formulation

In this section, we formulate the problem of inferring preconditions through program transformation. Given a deterministic method `M`, we infer a precondition, `M_pre`. Following the definition of correctness presented in Astorga et al. (2019), we aim to generate a precondition which is *safe* and *maximal*. An `M_pre` which is safe, rejects any illegal input while an `M_pre` which is maximal, accepts any legal input.

Given an exhaustive tester `T` which generates a set of input environments `E`, we aim to infer an `M_pre` with the following correctness properties for every $e \in$ `E`:

`M_pre`$(e) \rightarrow True$ iff `M` exits normally on e and

`M_pre`$(e) \rightarrow False$ iff `M` throws an exception on e.

Sub-Problems

Creating a correct `M_pre` through program transformation does not come without challenges.

Lifting Implicit Exception Checks In order for `M_pre` to be safe, it must include a check for all feasible crashes. It is not obvious how to design program transformations to guard against a variety of crash types. For instance, it is unclear how to insert source checks for interprocedural crashes. Furthermore, it is not clear when and where to add explicit checks for an arbitrary crash type. One obvious approach is to insert checks exhaustively, prior to execution. Presumably, it is possible for a static exploration to find all JVM level exception checks which can then be translated to source level checks. Another approach would be to generate `E` up-front, and insert checks to create `M_pre` in a single step, non-iterative, process. However, neither of these approaches are effective in practice: the former does not guarantee correctness, whereas the latter can result in a poor-quality test suite `E`. We illustrate these issues in Section 5.3.3 and propose to address them by inserting exception checks both iteratively and dynamically. In particular, our approach interleaves program transformation and test generation in an iterative process until convergence.

Removing Irrelevant Computation Upon lifting exception checks, our transformed precondition is correct modulo the test generator. However, it contains many program segments which are unrelated to the precondition. For illustration, consider Figure 5.9. Lines 4-24 are unrelated to the precondition. We present a dual rationale for eliminating these segments. Firstly, we emphasize the importance of removal on naturalness. To ensure that `M_pre` is conducive to human reasoning it should not include program segments which are unrelated to the precondition and incur additional cognitive load [Astorga et al. (2018)]. Secondly, these extraneous segments might encompass statements or expressions with side effects. A pure precondition is desirable in both ease of understanding and in tool usage. In tool usage, a precondition with side effects must be run in a sandbox environment as to preserve the state of the program being analyzed. The challenge of maintaining purity is unique to our approach. Prior works have the benefit of ensuring purity by construction. That is, their precondition features are designed to only include observer methods which do not change the program state. The challenge of removing irrelevant computation can be tackled in a number of ways, both statically and dynamically. In section 5.3.3 we describe a set of program transformations we employ for efficient removal of irrelevant computation.

5.3.3 Technique

In this section, we present the details of our general approach as well as our instantiated framework. We describe the phases involved in inferring a precondition through program transformation. We address sub-problems (1) lifting implicit exception checks and (2) removing code irrelevant to the precondition.

Our overall technique, which comprises three main phases, is illustrated in Figure 5.10. The initial source transformation phase, described in Section 5.3.3 creates a seed for inference. This up-front

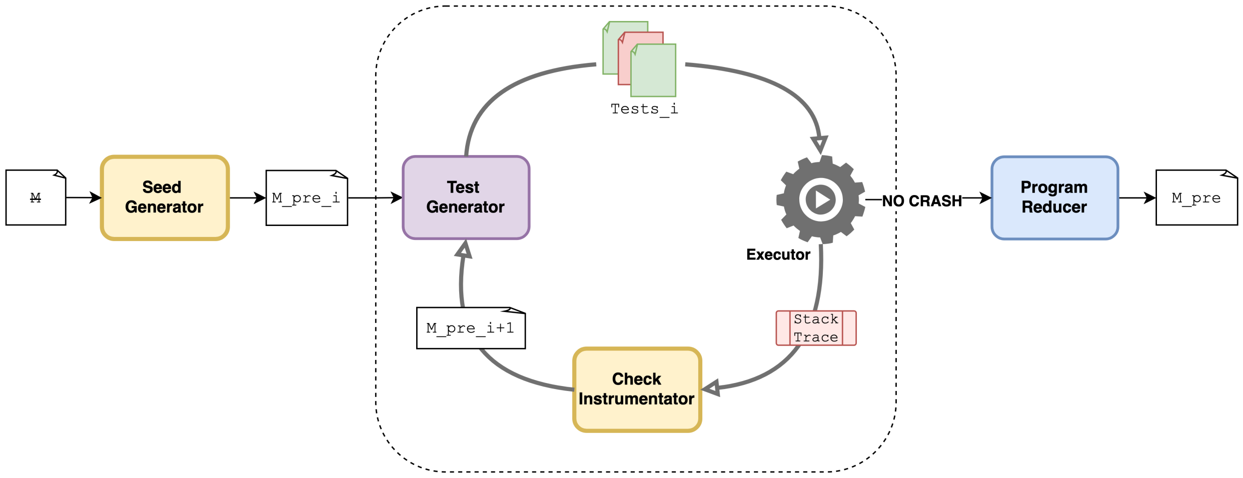


Figure 5.10: Precondition Inference Technique

transformation retains the structure of `M` creating an initial boolean returning function `M_pre_i`. The second phase, described in Section 5.3.3 is the main phase of inference. It tackles sub-problem one by adding explicit checks in the source of `M_pre_i` through an iterative transformation process. We present an algorithm for program transformation as well as detailing the process of iterative test generation and transformation. In our framework, we instantiate the test generator T as EvoSuite [Fraser and Arcuri (2011)]. The last phase, described in 5.3.3 tackles sub-problem two by leveraging an off-the-shelf program reducer. The final transformed program `M_pre` contains explicit exception checks and does not contain irrelevant computation. Each phase works in synergy benefiting from design choices made in other phases. The seed generation phase localizes expressions with both exception check insertion and syntax guided reduction in mind. The second phase, benefits from the first, and relies on the reducer's (third phase) properties to remove conservative checks. The reducer benefits from the previous phases as it receives a `M_pre` with a reducer-aware structure such that exception checks are localized. Ultimately, this results in a natural and correct precondition. Lastly, in Section 5.3.3 we motivate the need for an iterative transformation process by illustrating the incorrectness incurred through alternate approaches presented in Section 5.3.2.

```
BigInteger Sqrt() {
   if (m_sign < 0)
      throw new ArithmeticException();

   return Round(Sqrt(this));
}
```

(a) Target Method `M`.

```
boolean Sqrt_pre_init() {
   if (m_sign < 0)
        return false;

   BigInteger var_a = Sqrt(this);
   Round(var_a);
   return true;
}
```

(b) `M` after Seed Transformation.

Figure 5.11: Seed Generation.

Seed Generation

Our overall technique begins by creating a seed through an up-front source transformation on `M`. Since our approach leverages the structure of the method, it is apparent that we begin with the original method body. However, our problem formulation requires `M_pre` to be a non-exceptional, boolean returning function.

Figure 5.11 illustrates the source transformation on a `Sqrt` method. The source transformation is designed such that the seed has the following desirable qualities:

1. The `M_pre` must be boolean returning:

   We transform M to be a boolean returning function by modifying the method signature and adding the expression `return true;` at all exit points. If `M` returns a type other than boolean, we maintain the original expression return value by lifting it to an expression statement prior to the newly inserting return statement. This is essential as the return expression in `M` itself may be exceptional.

2. The `M_pre` must be non-exceptional:

   Our problem formulation requires `M_pre` to either return true or false, implying that it exits normally on all inputs. At times, developers will explicitly guard against precondition violations, by explicitly throwing an exception. For example, in Figure 5.11a the developers `throw new ArithmeticException()` for a `Zero` input. To satisfy this requirement, we replace any `throw` expression with a `return false;` indicating an illegal input. Any implicitly thrown exceptions are handled in phase two described in Section 5.3.3 and illustrated in the

```
public boolean Sqrt_pre() {
    if (m_sign < 0)
        return false;
    try {
        Round(Sqrt(this));
    } catch (Exception e) {
        return false;
    }
    return true;
}
```

(a) `M_pre` Without Call Normalization.

```
public boolean Sqrt_pre() {
    if (m_sign < 0)
        return false;
    try {
        Sqrt(this);
    } catch (Exception e) {
        return false;
    }
    return true;
}
```

(b) `M_pre` With Call Normalization.

Figure 5.12: Call Normalization in Seed Generation.

dotted box in Figure 5.10.

3. The `M_pre` localizes crashes:

   One challenge described in Section 5.3.2 sub-problem 1 is in regards to interprocedural crashes. In Section 5.3.3 we describe and motivate our approach to tackle this through a predefined program transformation. In short, this transformation amounts to wrapping an exceptional call in a `try-catch` block. As a product of this transformation we perform *call normalization* in our seed. The process of call normalization is essential for crash localization during the next phase. The seed generating transformation normalizes each call by putting it on its own source line. Call normalization ultimately results in a more readable precondition. This is illustrated in Figure 5.12. Suppose the `Sqrt` throws a general type Exception on an input e. Without call normalization, the final `M_pre` is shown in Figure 5.12a. It is not clear whether the exception is occurring in the method `Sqrt` or the method `Round`. On the other hand, performing call normalization (Figure 5.12b) localizes the exception in `Sqrt`. By localizing the crash we reduce the cognitive load of interprocedural inspection of the exceptional callee.

We perform this initial source transformation up-front to create a seed `M_pre` for our forthcoming phases. All transformations beyond the boolean returning behavior are semantic preserving transformations. [4]

[4]We don't currently support some Java constructs in our framework due to engineering work.

Explicit Exception Check Insertion

In this section we describe our solution to sub-problem 1, lifting implicit JVM exception checks to source level. We do this by iteratively interleaving program transformation and test generation. This phase of our approach is shown in the dotted box in Figure 5.10. It begins by operating on the seed program M_pre_i. In order to discover feasible implicit crashes, we invoke the **Test Generator** T to obtain a set of tests, $Tests\_i$. We then **execute** the tests and observe the stack trace(s) of all crashing inputs. If crashing inputs do indeed exist in $Tests\_i$, then the M_pre_i should be transformed such that any crash instead exits normally with a return value of `false`. To do so, we invoke the **Check Instrumentor** which performs program transformations to satisfy this formulation. The check instrumentor, modifies the semantics of M_pre_i to produce an $M\_pre\_i + 1$, by adding checks (false returning if-statement guards) prior to any crash found by the test generator. After transformation, we again invoke the test generator on the newly produced $M\_pre\_i + 1$ to yield a new set of tests $Tests\_i + 1$. If this new set of tests contains new crashing inputs which are not already guarded against, we continue in the loop and invoke the check instrumentor to guard against any new crashes. We continue this iterative process until convergence. That is, when we no longer observe crashes during execution of the current $Tests\_i$. Upon convergence, we are guaranteed to have a safe and maximal precondition, modulo the test generator. In Section 5.3.3 we illustrate the need for such an iterative process to produce correct and high quality preconditions.

Check Instrumentor. Here, we describe the process for inserting `false` returning guards prior to crashes found by the test generator. These are inserted such that the `M_pre` will exit normally on an illegal input rather than throwing an exception. The check instrumentor parses a stack trace produced from the execution of the current tests $Tests\_i$. The stack trace provides a crash type and location, which allows us to make precise AST transformations. By only guarding against the given crash type at the given location, we maintain maximality and do not reject any legal inputs. We categorize and design our program transformations in a data-driven fashion. We define six transformations which guard against 99% of the crashes EvoSuite found during test generation on

87 real world Java projects. Using the Javaparser [jav] library, our technique performs AST transformations according to the Algorithm 4. This algorithm works in synergy with our seed generation as it expects localized statements to match the given line number. The first transformation, on Lines 4-7, is performed when the crash occurs in a callee of `M`. This amounts to wrapping the crashing call in a `try-catch` block. The seed generation process guarantees that S is of the form `method_call_expr(arg_expr_list)` to ensure precise localization. If the crash occurs directly in the target method, the statement on Line L can contain multiple child expressions. Using a post-order traversal, the parser collects the expressions in order of execution. Our algorithm iterates over the child expressions and similarly transform the program segment based on crash and expression types. For instance, given an `ArrayOutOfBoundsException` crash type, the transformation on line 20 is performed for each `ArrayIndexExpression` on line L. If multiple index expressions are present as children of S, guards are conservatively added for each. Working in synergy with the reducer, if the checks are unnecessary, they will be removed in a later stage of the technique. The other four transformation types follow similarly.

A Note on Interprocedural Wraps. It is not obvious how to add explicit checks for crashes that occur within callees in `M`. One possibility is to inline any calls. We experimented with inlining method calls, but found it to lead to lower quality or equivalent preconditions for two reasons. Firstly, inlining can quickly hit a library or framework boundary where the source code is not included. Secondly, even when all source is available, inlining often leads to potentially complex precondition logic due to long call chains with large method bodies. Through a manual inspection of 10 random samples from our dataset, we found that only one sample benefited from inlining. Furthermore, we found that over 85% of the exceptional callees in our dataset are at a source boundary and cannot be inlined. For these reasons, we choose to handle exceptions in the callee by wrapping the crashing call in a `try-catch` block. Lastly, we note that our catch expression captures the Exception type seen in the stack trace rather than the general Exception type in order to further localize crashes.

Algorithm 4 Transformation to replace a crashing program fragment on Line L with a new program fragment that guards the crashing exception type.

```
 1: procedure CheckInstrumentorAlgorithm(Exception Type T, Line L, bool isInCallee)
 2:     S ← statement on line L
 3:     P ← S
 4:     if isInCallee then                                              ▷ Interprocedural Exception
 5:         P ← try { S } catch (T exc) { return false; }
 6:         return;
 7:     end if
 8:     Let (expr_1, ..., expr_n) be the list of all expressions in S in order of execution
 9:     for $i = 1 : n$ do
10:         switch T do
11:             case java.lang.NullPointerException
12:                 if expr_i is a field access expression of the form object_expr.field then
13:                     P ←  if (object_expr == null) return false;  P
14:                 end if
15:                 if expr_i is an array access expression of the form array_expr[index_expr] then
16:                     P ←  if (array_expr == null) return false;  P
17:                 end if
18:             case java.lang.ArrayIndexOutOfBoundsException
19:                 if expr_i is an array access expression of the form array_expr[index_expr] then
20:                     P ←  if (index_expr < 0 || index_expr >= array_expr.length) return false;  P
21:                 end if
22:             case java.lang.ClassCastException
23:                 if expr_i is a class cast expression of the form (cast_type) expr_to_cast then
24:                     P ←  if !(expr_to_cast instanceof cast_type) return false;  P
25:                 end if
26:             case java.lang.NegativeArraySizeException
27:                 if expr_i is an array creation expression of the form new array_expr[index_expr] then
28:                     P ←  if (index_expr < 0) return false; P
29:                 end if
30:             case java.lang.ArithmeticException
31:                 if expr_i is a binary operator DIVIDE expression of the form numerator/denominator then
32:                     P ←  if (denominator == 0) return false; P
33:                 end if
34:     end for
35: end procedure
```

Removing Irrelevant Computation

Lastly, we wish to remove program components which are unrelated to the precondition. This is an important step as it reduces cognitive load of reasoning over the precondition and removes unrelated statements which potentially have side effects. We consider a program segment to be *unrelated* to the precondition if, when removed, the output value (true/false) of `M_pre` executed on e for all e in E, does not change. In our technique, the inputs E come from our test suite. After the iterative process has converged, the test suite only includes assertions on non-exceptional behavior. That is, we have executable input and output examples, asserting true on legal inputs and false otherwise. We exploit these generated tests as an oracle for correct behavior. If a program segment is removed and the tests still pass, it is unrelated to the precondition.

We formulate this sub-problem as *program reduction* and leverage an off-the-shelf program reducer, Perses [Sun et al. (2018)]. A program reducer takes as input a program to reduce and a set of constraints. It then outputs the smallest sub-program which satisfies the given constraints. In our setting, the constraints are that every EvoSuite generated test must pass. We also experimented with the c-reduce [Regehr et al. (2012)] program reducer but found it to be time-inefficient as it considers syntactically invalid sub-programs in the search space. After the reduction phase, we have inferred a natural and correct `M_pre`. We evaluate the impact of this reduction on the resulting preconditions in Section 5.3.5.

Alternate Check Insertion Approaches

Lifting implicit exception checks requires much thought. It is not obvious when and where to insert source checks. In this section, we describe two alternate approaches and illustrate why an iterative, dynamic transformation process is necessary for correctness and efficiency. We assume an equivalent technique in terms of seed generation and transformation types in order to draw attention to the necessity of iteration.

Exhaustive Up-Front Transformation. One naive approach would be to exhaustively insert source checks at each point which *may* throw an exception. All implicit JVM checks would be lifted

```
BigInteger Decrement(BigInteger val) {

  BigInteger One = BigInteger(1);

  try {
    return val.Sub(One);
  } catch (NullPointerException e) {
    return BigInteger(-1);
  }

}
```

(a) Target `M`, `Decrement`.

```
boolean M_pre_i(BigInteger val) {
 BigInteger One = BigInteger(1);
 try {
    if (val == null)
      return false;
    val.Sub(One);
 } catch (NullPointerException e) {
    BigInteger(-1);
    return true;
 }
}
```

(b) M_pre_i: exhaustive checks.

```
void test() {
  boolean b1 = M_pre_i(null);
  assertFalse(b1); //legal input!

  boolean b2 = M_pre_i(BigInteger(1));
  assertTrue(b2);
}
```

(c) Tests for M_pre_i.

```
boolean M_pre(BigInt val) {
  if (val == null)
    return false;
  return true;
}
```

(d) Resulting `M_pre`.

Figure 5.13: Alternate Approach 1: Up-Front Exhaustive Guard Insertion

to explicit source checks regardless of if it is a feasible crash (i.e. add a null pointer check before every field access). In this section, we will illustrate that this approach is inefficient and can even lead to incorrect preconditions. Consider the `M`, `Decrement` in Figure 5.13a. Intuitively, it takes a `val` and subtracts one. The developers of the method guarded against the case in which `val` is `null` through a `try-catch` block. In the case of a `null` input, the program returns the `BigInteger` representing -1. Assuming `Sub` (called on a non-null target) and the `BigInteger` constructors always exit normally, `Decrement` is non-crashing and has a precondition of `true`.

An exhaustive approach would analyze the field access on the call to `Sub` as a potential `Null Pointer Exception` and insert a nullness check before the access (Figure 5.13b even though the exception is not reachable. After the check is inserted, T is invoked. Since the M_pre_i is non-exceptional, EvoSuite generates a test suite with only assertions on true or false return behavior. Suppose EvoSuite generates the tests in Figure 5.13c. Program reduction is invoked with these EvoSuite generated test suite as a correctness oracle. Due to the eagerly inserted null check, the tests do not match our formulation of `M_pre` as an input of `null` is *legal* but has a return value of `false`. So, the program reducer removes segments to satisfy the incorrect behavior described in the test suite. The resulting `M_pre` (Figure 5.13d) inferred through this exhaustive up-front check insertion approach, is incorrect (non-maximal) compared to our manually derived precondition as it rejects the non-exceptional input `val == null`.

Single Step Non-Iterative Test Generation and Transformation. Here we detail the second alternate approach presented in Section 5.3.2. This approach involves a single step non-iterative process. In this approach, T is invoked once on the seed program to obtain a suite of tests. The tests are executed and the Check Instrumentor is invoked to insert guards parsed from the stack trace, creating M_pre_1. In contrast to our approach, the reducer is invoked on M_pre_1 and is not fed to the test generator. Due to coverage guided nature of EvoSuite's test generation, we show that an iterative process obtains a higher coverage, more robust test suite.

To illustrate the problems with a two-step process, we use a sample from our real-world dataset (Figure 5.14) which required multiple iterations to find new exceptional inputs. In a single step process, we begin with the seed program in Figure 5.14a and invoke T to find the crashing input `y == null`. After executing the test and parsing the stack trace, we insert a null checking guard prior to the field access on `y` to create M_pre_1 (Figure 5.14b). In this alternate non-iterative process, no additional rounds of test generation would occur, and M_pre_1 would be fed to the program reducer. The following two Sub-figures 5.14c and 5.14d illustrate the additional checks that are inserted when we continue to invoke EvoSuite until convergence. By only invoking T as a single non-active step, we do not find the crashes in `diff` and `redateBy`, leading to a lower quality precondition which does not guard against illegal inputs. In Section 5.3.5 we evaluate the empirical advantage of multiple iterative steps.

In summary, inserting exception checks both iteratively and dynamically is necessary to guarantee correctness. Furthermore, inserting exception checks on demand creates higher quality preconditions by driving the test generator to progressively higher coverage inputs. This allows for the discovery of additional crashes and insertion of corresponding explicit checks. We quantify the number of iterations before convergence in section 5.3.5. An iteration number greater than 1 is evidence that a single-step approach is not sufficient for inferring high quality preconditions.

```
boolean M_pre(Year y) {
    int var_a = y.diff(this.end);
    redateBy(var_a);
    return false;
}
```

(a) Seed Program.

```
boolean M_pre(Year y) {
    if (y == null) return false;
    int var_a = y.diff(this.end);
    redateBy(var_a);
    return false;
}
```

(b) `M_pre_1`.

```
boolean M_pre(Year y) {
  if (y == null) return false;
  int var_a;
  try {
      var_a = y.diff(this.end);
  } catch (NullPointerException e) {
      return true;
  }

  redateBy(var_a);



  return false;
}
```

(c) `M_pre_2` requiring 2 steps.

```
boolean M_pre(Year y) {
  if (y == null) return false;
  int var_a;
  try {
    var_a = y.diff(this.end);
  } catch (NullPointerException e) {
    return true;
  }
  try {
    redateBy(var_a);
  } catch (NullPointerException e) {
    return true;
  }
  return false;
}
```

(d) `M_pre_3` requiring 3 steps.

Figure 5.14: Alternate Approach 2: Single Step Non-Iterative Test Generation

5.3.4 Comparative Case Study

In this section, we perform an in-depth comparative evaluation to the state-of-the-art approach on a single real-world project. Here, we evaluate and compare the resulting preconditions inferred by both approaches on two aspects. We aim to answer the following research questions:

RQ1 Correctness: Are the preconditions safe and maximal?

RQ2 Naturalness: How natural are the preconditions?

Experimental Setup

Baseline. We evaluate in comparison to Proviso [Astorga et al. (2019)], as it is the state-of-the-art and instantiated in C# which is most similar to our target language.

Benchmark. We evaluate on 39 (M, PRE) pairs from the `NetBigInteger` C# project. Although our framework is designed for Java, we choose `NetBigInteger` as a common benchmark to compare to Proviso [Astorga et al. (2019)]. In order to evaluate our technique, we manually translate the `NetBigInteger` class to a semantically equivalent class in Java. Although Proviso evaluates on 5 projects, 3 are synthetically constructed benchmarks, and 2 are real-world. We select a single project to evaluate on as manual effort is required in inspection and semantically equivalent conversion to Java. As such, we perform an in-depth evaluation on this single project.

RQ1: Correctness

Following the definition of correctness in Astorga et al. (2019), we evaluate the safety and maximally of our inferred preconditions on the benchmark. In order to evaluate, we use another test generator, differing from the T used in our technique, EvoSuite. Here, we use Pex [Tillmann and de Halleux (2008)], and industrial test generator. We execute Pex on `M` to generate crashing and non-crashing inputs. If Pex can generate an invalid input for which `M_pre` returns `true`, we consider `M_pre` to be *unsafe*. If Pex generates a valid input for which `M_pre` returns `false`, we consider `M_pre` to be *non-maximal*. If a precondition is both safe and maximal, we consider it to be correct. We perform this evaluation for the 39 `M_pre`s inferred by our approach. For Proviso, we use the correctness results reported [Astorga et al.]. We find that our inferred preconditions are safe for 33 methods, maximal for 29 methods, and **both safe and maximal for 29 of the 39 methods**. In comparison, Proviso's preconditions are safe for 37 methods, maximal for 34 methods, and both safe and maximal for 34 methods. Through manual inspection, we find that the 10 incorrect `M_pre` our approach infers are due to EvoSuite incompleteness.

Result 1: Our approach infers correct (both safe and maximal) preconditions for 29 of the 39 methods in the benchmark. The 10 incorrect preconditions were due to EvoSuite incompleteness.

RQ2: Naturalness

In this section, we evaluate the naturalness of our preconditions. To better understand if our inferred preconditions are natural, we study a human's ability to reason over its behavior. We evaluate and compare to Proviso, by conducting a user evaluation. Each user is asked to review a given precondition and three inputs. We ask the user to classify each input as legal or illegal. The accuracy of their answers as well as the time taken to derive the answer are metrics of how natural or easy the precondition is to reason over.

User Study Design We identify 5 preconditions from our comparative evaluation to use filtered by the following criteria:

```
Input:

value: NetBigInteger                         target: NetBigInteger
---------------------                        ----------------------------------------
m_sign: int = -25                            m_sign: int = 25
m_magnitude: int[] =  {0, 0, 0, 0, 0, 0}     m_magnitude: int[] =  {0, 0, 0, 0, 0, 0}
----------------------------------------     ----------------------------------------
```

```
boolean M_pre(NetBigInteger value) {
  if (m_sign == 0)
    return true;
  if (value == null)
     return false;
   return true;
}
```

Figure 5.15: Precondition 3, Input 2, Group A.

```
Input:

value: NetBigInteger                         target: NetBigInteger
---------------------                        ----------------------------------------
m_sign: int = -25                            m_sign: int = 25
m_magnitude: int[] =  {0, 0, 0, 0, 0, 0}     m_magnitude: int[] =  {0, 0, 0, 0, 0, 0}
----------------------------------------     ----------------------------------------
```

```
Precondition:

(
  (value == null)
  and (
        (not (target.getSignValue() <= -1))
        and
        ((target.getSignValue() <= 0))
  )
) or (
   (not (value == null))
)
```

```
Context:


int getSignValue() {
 return m_sign;
}



.
```

Figure 5.16: Precondition 3, Input 2, Group B.

1. Both our approach and Proviso produce a correct precondition

2. The preconditions produced by our tools syntactically differ

3. The precondition is non-trivial (there exists at least 1 illegal and 1 legal input).

We split participants into two groups. Group A was presented with our preconditions while Group B was presented with the preconditions inferred by Proviso. The inputs for each precondition remained the same across both groups. For illustration, we include on of the (precondition, input) pairs from our study. Figure 5.15 shows an input given to Group A, while Figure 5.16 shows one of the inputs given to Group B. We include any necessary code context for completing the task. For brevity, we only show a single input, but each user was presented with three inputs for each precondition. The user was asked to classify each input as True or False (legal or illegal).

Protocol The user study was conducted by a qualtrics survey. We used fine grained timing for each participant. To reduce cognitive load, participants were randomly presented with three out of the five preconditions, using qualtrics randomization feature. The participants comprised of 10 users including computer science PhD students and industry software engineers split evenly between the two groups.

User Study Results On average, we found that users were able to more accurately reason over our preconditions in a shorter amount of time. Participants of Group A had an average accuracy of 96% while participants in Group B made more mistakes leading to a lower average accuracy of 84%. In terms of speed of reasoning, participants in Group A took an average total time of 144 seconds. Participants in Group B took nearly double the amount of time to complete the study. Our results show that Preconditions 1 and 3 inferred by our approach were significantly easier to reason over in terms of both accuracy and speed. Precondition 4 had comparable results from both groups. However, on Preconditions 2 and 5, our approach had weaker results. On Precondition 2, Group A achieved higher accuracy, but took 3x the average time as Group B (254.81 seconds vs 83.83 seconds). On Precondition 5, Group A again achieved higher accuracy, but took a comparable time to Group B. Preconditions 2 and 5 were the only samples in our study which included `try-catch` blocks, and thus, interprocedural reasoning. This indicates that although our preconditions are on average, easier to reason over, future research should address the additional cognitive load and lower level of naturalness due to interprocedural crashes. We include all five preconditions from both groups A and B in this dissertation. Precondition 1 is shown in Figure 5.9, Precondition 3 is shown in Figures 5.15 and 5.16, and we include Precondition 2 (Figure A.3), Precondition 4 (Figure A.4), and Precondition 5 (Figure A.5) in the appendix.

	Accuracy		Time Taken (Sec)	
	ours	PROVISO	ours	PROVISO
`Precondition 1`	100%	88.89%	64.83	354.96
`Precondition 2`	88.89%	66.67%	254.81	83.38
`Precondition 3`	100%	88.89%	83.71	551.44
`Precondition 4`	100%	100%	45.89	69.51
`Precondition 5`	88.89%	75%	272.86	307.76
Overall	**95.56%**	83.89%	**144.42**	273.41

Table 5.5: User Study Results.

Result 2: On average, users were able to more accurately reason over our preconditions in a shorter time span. Participants of Group A had an average accuracy of **96%** while participants in Group B committed more errors, leading to a lower average accuracy of **84%**. In terms of speed of reasoning, participants in Group A completed the task with an average total duration of **144 seconds**. Participants in Group B took nearly twice as long (**273 seconds**) to finish the study. Our results were weakest on preconditions which included interprocedural try-catch blocks.

5.3.5 Design Decision Evaluation On a Large Scale Dataset

In this section, we evaluate the design decisions of our technique at scale. We aim to evaluate our technique's design decisions in both sub-problems. First, we evaluate our choice of iterative, dynamic, exception check insertion over the two-step approach presented in Section 5.3.3. It is difficult to evaluate exhaustive up-front transformation approach presented in Section 5.3.3 since our definition of correctness is modulo a test generator. For this, we default to an illustrative example by contradiction of correctness in Figure 5.13. Secondly, we motivate and evaluate our solution to sub-problem two, removing irrelevant program segments in terms of complexity. To do so, we gather a substantial dataset of real-world Java preconditions and answer the following research questions:

RQ1 How do different choices of the explicit exception insertion algorithm impact the quality of test suite and resulting precondition?

RQ2 How does removing irrelevant program segments affect the resulting precondition in terms of complexity?

	Number	Percent
non-trivial	6,790	37.99%
true	10,791	60.38%
false	290	1.62%
Total	17,871	

Table 5.6: Overall Dataset Precondition Categorization.

Experimental Setup: Benchmark

We evaluate on a large scale dataset we've constructed of Java methods from popular, stable, projects. Our dataset consists of 17,871 (`M`, `M_pre`) pairs inferred from 87 projects. The projects are from the SF100 [Fraser and Arcuri (2013)] with an additional 13 projects popular projects from the Maven repository [5]. In order to provide further insight into our benchmark, we present the following characteristics of our dataset:

Crash Occurrence: Since our dataset is real-world and diverse, it contains methods which accept all inputs, reject all inputs, or have a non-trivial precondition which accepts some inputs and rejects others. We characterize the dataset by the precondition "type". In our dataset, more than half of the methods have a precondition of `true` indicating that any input is legal. A small percentage, (∼1%) of samples are always exceptional and have a precondition of `false`. Through manual inspection, we find these methods typically contain a single explicit `throws` expression. For example, a method with a body of: `throw new UnsupportedOperationException` would always be exceptional and have a precondition of `false`. The remaining samples, (∼40%), are non-trivial in that there exists at least one legal input and one illegal input.

Crash Types: EvoSuite found a diverse set of crashes which fit into the 6 crash type categories presented in Section 5.3.3. In this section, we present the total number of explicit exception checks inserted and the occurrence of each type. These checks were dynamic lifted to explicit guards by the Check Instrumentor and were not present in the source of `M`. Nearly 15,000 checks were added through program transformation. This emphasizes the need for a dynamic tester to find crashing

[5]The projects were collected in order of number of downloads as of April 2022.

Crash Type	Number of Checks
Overall	14,932
Intraprocedural Exception	8,079
`NullPointerException`	5,535
`ArrayIndexOutOfBoundsException`	793
`ClassCastException`	443
`NegativeArraySizeException`	65
`ArithmeticException`	17

Table 5.7: Dataset Breakdown by Crash Type

inputs rather than being able to parse out explicit checks as in Figure 5.9 line 3. We also note that over half of crashes found by EvoSuite occurred in a callee, indicating a potential fruitful research direction in natural precondition inference.

Defensive Checks: Here, we ask, how often do developers explicitly guard for precondition violations? This is in converse to the previous characterization in that we study only the checks which were included in the untransformed `M`. Through this study, we motivate the necessity of a dynamic tester to find legal and illegal inputs beyond the explicitly written exceptions. Previous approaches have attempted to statically create such a dataset of preconditions through parsing. One static parsing approach [Hellendoorn et al. (2020)] attempts to generate a dataset of invariants. The authors do this by considering an if-statement's guard condition to be an invariant of the guarded block. Although they were able to collect these invariants on a large scale (2.5 million invariants), such a dataset cannot capture implicit specifications that are not expressed directly in `if` conditions. Although our setting is slightly different in that we are collecting preconditions rather than invariants, the approach still stands as a naive evaluation.

Through an AST walk over the target methods in our dataset we find 1,451 explicitly coded illegal input guards. This, in conjunction with the nearly 15,000 implicit crashes found shows that high quality preconditions cannot be constructed by simple scraping of if conditions. It also indicates that developers do not sufficiently guard against illegal inputs

We find that only **1,451** guards were explicitly written by developers. The remaining **14,932** crashes were not explicitly guarded against and were found by EvoSuite.

RQ1: Impact of Iterative Dynamic Check Insertion

Here, we aim to answer RQ1 using the dataset presented above. In Section 5.3.3 we illustrated the necessity for an iterative approach to produce a high quality test suite. Here, we quantitatively motivate it on a large scale. We measure the number of samples which required iteration. That is, the number of samples for which EvoSuite finds new crashing inputs when it is invoked on M_pre_1 which includes inserted checks. We find that 2,198 samples required additional iterations beyond a single step process. This amounts to 32.37% of our non-trivial samples. During the inference of these 2k samples, they required on average, 1.73 additional iterations (beyond the initial step). In total, 4,517 additional crashes were found due to additional iterations.

Result 1: We find that over 1/3rd of non-trivial samples require additional iterations beyond a single step (test and transform) approach. A total of 4,517 additional crashes were found due to the iterative approach.

RQ2: Impact of Program Reduction

In this section, we evaluate the effect of removing irrelevant computation. The goal of removing irrelevant program segments is to reduce cognitive load and remove unrelated statements which potentially have side effects. In short, program reduction should reduce complexity. We evaluate this along three axis: number of program paths, precondition size, and purity. For each axis, we compare the unreduced `M_pre` (post-instrumentation) to the reduced `M_pre`.

Cyclomatic Complexity. In order to quantify the complexity of a precondition's control flow, we measure the cyclomatic complexity [McCabe (1976)], using checkstyle [che]. The average cyclomatic complexity of an unreduced sample is 2.86. After reduction, the complexity drops to 1.77. This shows the computational load is reduced by around one path through program reduction. For non-trivial samples with at least one illegal and one legal input, the average complexity is 4.99 and drops to 2.46 after reduction. For this sub-class of samples, complexity is more significantly lowered through reduction.

Precondition Size. In order to quantify a precondition's size, we measure the number of AST nodes. The average number of unreduced nodes is 56.27 and the average number of AST nodes in the reduced sample is 15.41 This shows that we removed around 1/3 of the AST nodes through program reduction, greatly reducing computational load of a consumer of the precondition.

Purity. Ideally, a precondition should not have side effects. Impure statements increase cognitive load, forcing the consumer to reason about state, and add complications for precondition execution during software testing. We measure the number of samples which are impure pre-reduction, but pure post-reduction to quantify impact of program reduction on side effects. To do so, we use the Facebook Infer [Distefano et al. (2019b)] purity checker. We find that only 50.63% of pre-reduction samples are pure, while 81.3% post-reduction samples do not contain side effects. This demonstrates the impact of reduction on the purity of inferred preconditions.

Result 2: We find that by employing a program reducer we produce preconditions which are of **lower complexity, smaller, and display a higher level of purity**. Cyclometic complexity drops by around 1 program path after reduction. For non-trivial samples, this is more significant, with reduction lowering complexity by half. The number of AST nodes is reduced by 1/3rd. Post reduction, there is a 60% increase in number of pure samples. This motivates the use of program reduction across three axis of complexity.

5.3.6 Limitations

Our approach, like many other precondition inference techniques, is limited by classic drawbacks of program analysis tools. Firstly, our approach is strictly tied to and heavily relies on the target method implementation. If the source code is not available or does not compile and execute then our technique cannot infer a precondition. Secondly, while instantiating our technique as a tool, we found that program analysis tools are very brittle. We applied our technique at scale on a total of 126,699 methods and only obtained 17,868 (M, M_pre) pairs (∼14% of the target methods). The methods we could not extract preconditions for failed due to limitations of the program analysis tools in our pipeline. Failures largely occurred due to automated testing challenges in EvoSuite. For example, EvoSuite often could not generate a well formed object as a target for M. EvoSuite struggled to generate test suites in some cases due to resource limits, I/O and file system mocking, concurrency, and missing environment dependencies [Fraser and Arcuri (2013)]. Furthermore, program analysis tools used in our pipeline are strictly tied to and assume correctness of the target method. If the method has a bug related to the precondition, it may be inherited in the resulting `M_pre`. Lastly, the inherit nature of program analysis limits us to an instantiation in a single language (Java). A limitation unique to our approach, is the possibility of side effects. Since our approach relies on the structure and program segments that appear in the original `M`, there is no guarantee that the precondition we produce is pure. In practice, we find that most preconditions infer are pure in nature. Although, if purity is critical, a purity checker such as FBInfer [Distefano et al. (2019b)], can be invoked to avoid the rare case where the method's precondition has side effects.

5.4 Conclusion

In this section, we motivated the problem of neural test oracle generation. We proposed `TOGA`, a neural technique to infer both exception and assertion test oracles from a given test prefix and unit context. `TOGA` is a two step transformer based architecture that is capable of generating oracles for units without implementation or docstrings. It improves upon generative neural assertion oracle inference techniques by ranking a small set of likely candidate assertions. When integrated with a random test generation tool (EvoSuite) to obtain prefixes, `TOGA` finds 57 real world bugs, out-performing existing test oracle inference techniques. This section also presented two datasets for future work in neural exception and assertion test oracle inference.

Furthermore, this section aims to address limitations of `TOGA` by inferring interpretable preconditions. We presented a novel approach for inferring natural preconditions through program transformations over the target method. Our findings revealed that the preconditions inferred by our approach are more easily reasoned over by humans than template based tools which build the precondition from scratch. Looking ahead, the potential of natural preconditions extends to diverse applications such as serving as training data or integrating with large language models. This exciting direction is reinforced by the promising advancements in the realm of program specification representations as evidenced by Odena and Sutton (2020), along with the high accuracy demonstrated by large language models in inferring Daikon invariants, as showcased by Sutton et al. (2023). The concept of natural preconditions is in alignment with the evolving landscape of statistical software engineering tools, especially within the domain of Large Language Models.

CHAPTER 6

Conclusion

In conclusion, this dissertation explores methods to learn developer intent from corpora of code. It addresses common challenges associated with program analysis tools concerning scalability and precision. By inferring program specifications, the research presented in this dissertation simultaneously addresses both aspects. One of the significant issues with program analysis tooling is their inability to handle large codebases with real-world programming constructs. This limitation, commonly referred to as scalability, poses a substantial challenge. Program analysis tools frequently falter when tasked with analyzing complex code. This dissertation proposes a neural approach to mitigate this problem. Instead of trying to analyze intricate methods entirely symbolically, it suggests inferring program specifications as a viable approach for integration. In addition to scalability, precision in program analysis is a critical concern. Program analysis tools often rely on heuristic methods for interpreting developer intent, which can lead to both false positives and false negatives. These inaccuracies undermine the reliability of the tools and ultimately lead to a lower level of precision. To address this, tooling often provides the option for developers to replace heuristics with their own manually written specifications. By avoiding heuristics, a higher level of precision is achieved, but a human in the loop is required. This dissertation proposes inferring specifications to remove the human effort but maintain a high level of precision. By adopting this perspective, program analysis tools stand to improve their precision, ultimately enhancing their effectiveness. In addition to these theoretical contributions, this dissertation presents a set of datasets and neural modeling techniques across three domains: static analysis, program merge, and automated test generation. These techniques are proficient in finding functional bugs, have lightweight inference, and can handle real-world programming constructs gracefully. They are a departure from purely symbolic programming language techniques, but can be tightly integrated to offer a promising avenue for enhancing software correctness. In summary, this dissertation addresses inherit issues in program analysis tools, through neural inference of program specifications.

Limitations There is great promise in neural techniques for program analysis. However, they do come with some noteworthy limitations. Firstly, the neural techniques presented in this dissertation lack guarantees for correctness. This is in contrast to the traditional program analysis literature that often provide soundness or completeness guarantees. Another notable limitation is interpretability. The outputs of neural techniques, including the ones presented in this dissertation, are typically not interpretable. This lack of interpretability becomes particularly evident in cases where *implicit* specifications are inferred. In Section 3 and Section 4, techniques for implicit specification inference were presented. To contrast purely symbolic to neural techniques consider the static bug finding setting. A conventional symbolic tool such as TAJS [Jensen et al. (2009a)] not only identifies bugs, but reports which program safety rule was violated. In contrast, when employing neural techniques for the same purpose, developers can be left wondering why a bug was reported. In summary, while neural techniques in program analysis hold great promise, they also introduce challenges related to correctness guarantees and interpretability.

Future Work The aforementioned limitation regarding interpretability naturally leads to an agenda in neural inference of explicit program specifications. Explicit specification inference holds promise as it offers interpretability, ultimately allowing for a higher level of trust. Unfortunately, explicit formal specifications are rarely written in real-world codebases. Since they cannot be scraped syntactically, we propose utilizing specifications generated by program analysis tools as training data for neural models. This approach will bring us closer to our ultimate goal of removing the human in the loop from program analysis tools without compromising precision.

Furthermore, this work can be wrapped into a general specification inference library with implications beyond the domains presented in this dissertation. For instance, when a static checker encounters a source boundary or needs to limit inlining, it could benefit from a program specification or a summary of the method it cannot analyze. Providing a specification at this point in analysis can alleviate scalability issues. In fuzzing, specifications can serve as postcondition assertions to detect bugs beyond general purpose safety properties. Similarly, in modular testing, preconditions can be provided to mitigate false alarms where bugs occur on illegal inputs. In fuzzing specifications can be employed beyond pre and postconditions. A fuzz driver embodies a specification for the structure of inputs to a method. In symbolic execution, specifications and function summaries can be leveraged much like in static analysis, given the involvement of static modeling. In verification, specifications are a prerequisite at various program points to achieve effectiveness. In summary, specification inference holds the potential to enhance scalability, precision, and reduce manual effort across many program analysis domains.

APPENDIX A

Appendix

A.1 Static Bug Finding

A.1.1 Additional Experiments

	Total		*Location*		*Operator*	*Value*		*Type*	
	Top-3	Top-1	Top-3	Top-1	Top-1	Top-3	Top-1	Top-3	Top-1
`ZeroOneTwoDiff`	40.8	29.7	18.9	3.9	30.3	35.0	6.5	38.6	3.4
`ZeroOneDiff`	51.6	34.5	27.1	5.5	35.6	45.4	10.4	73.9	58.9
`OneDiff`	**26.1**	14.2	35.5	20.4	34.4	52.3	29.1	76.1	66.7
Random	.08	.07	2.28	1.4	27.7	.01	.01	.27	0

Table A.1: Static Bug Finding: Accuracy Results on Extended Datasets.

Full experiment results

In addition to the evaluation of samples with one edit Table A.1, we also evaluate DeepMerge on the following datasets:

- `ZeroOneDiff` - Includes samples with labels of zero or one edit
- `ZeroOneTwoDiff` - Includes samples with labels of zero, one, or two edits.

We trained models on each dataset for roughly 12 hours on a single GTX 2080Ti GPU. Accuracy on the `ZeroOneDiff` is the highest as predicting that an AST is not buggy does not consist of any low level primitive predictions. This makes it a much easier prediction for the model than say, an `ADD` operation which the parent location, left sibling, value, and type must all be predicted correctly in order to be considered accurate.

False positive/negative study

An evaluation of false positives and false negatives is available in Table A.2. In this setting, we

		TRUE LABEL	
		BUGGY	NOT BUGGY
PREDICTED	ALARM	10,293	7,210
	NO ALARM	26,517	20,605

Table A.2: Static Bug Finding: Classification Results.

treat the problem as a classification problem on our `ZeroOneDiff` dataset and our model attempts to predict if a given AST is BUGGY / NOT BUGGY. If the model predicts `ADD`, `REP_VAL`, `REP_TYPE`, or `DEL`, we consider this a prediction of "BUGGY." Accordingly, if the model predicts `NO_OP`, we consider this to be a prediction of "NOT BUGGY."

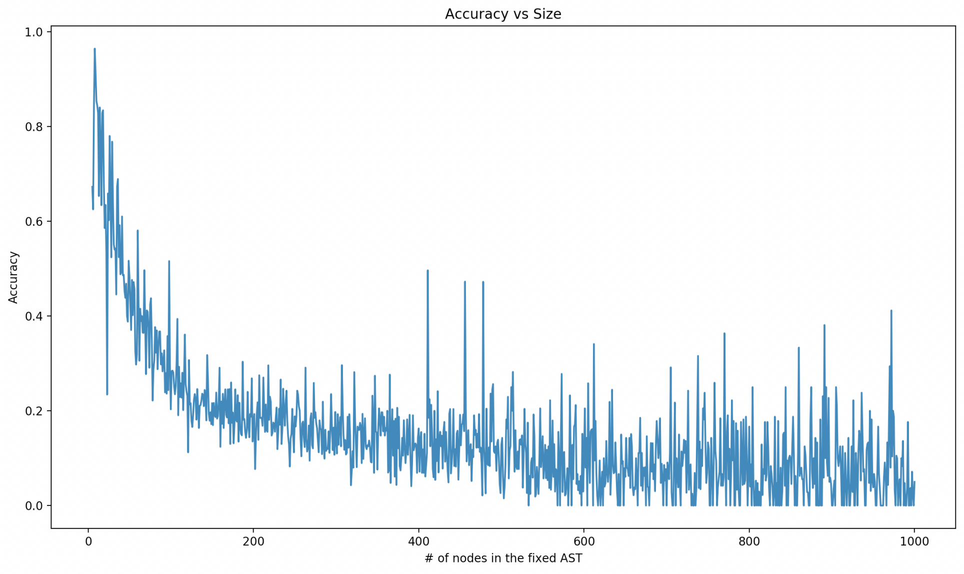


Figure A.1: Static Bug Finding: AST Size Sensitivity.

Accuracy v.s. size of graph

To demonstrate the affect of AST size on DeepMerge's prediction accuracy on the `OneDiff` dataset, we include Figure A.1. As expected, AST size and accuracy are inversely related.

Beam Size Effect On Accuracy.

In Table A.3 we compare the performance with different beam sizes on the `OneDiff` dataset. As we can see, the top-3 accuracy with beam size 3 is significantly better than top-1 accuracy with just

Beam Size (k)	Top-k Accuracy (%)
1	14.37%
2	21.10%
3	26.14%
4	30.12%
5	33.58%

Table A.3: Static Bug Finding: Beam Size Effect On Accuracy.

greedy prediction. This is expected, as in the decision process there are 'bottleneck' stages with only a few predictions (e.g., the op prediction). Thus from beam-1 to beam-3 there's huge improvement, but further beyond the performance maxed out.

A.1.2 Data Collection

We have built a robust system to automatically collect millions of bug-fixes in Javascript programs from Github. Our system continuously crawls Github for commits containing Javascript files and creates a label consisting of the change to the AST corresponding to each such file.

Our system consists of three entirely automated parallel steps:

1. **Collect Commits:** Our system uses the GH Archive API to easily access Github event data for a specific hour in time. After obtaining all data for the hour, we filter this using the Github API to only include commits that consist of edits to Javascript files.

2. **Download Files:** As we are obtaining a list of valid commits from step 1, we begin downloading the pair: $(src_{buggy}, src_{fixed})$ where src_{buggy} is the file prior to the commit, and src_{fixed} is the file following the commit that contains the changes made.

3. **Create Label:** For each Javascript file downloaded, we parse the source code into a JSON format of the AST. Our system uses the SHIFT AST [6]. Abstract Syntax Tree representations are designed to naturally and intuitively represent the structure of the source code. Because of this design goal, small changes in the source code can often lead to very large changes in the AST. We chose the SHIFT AST representation with consideration to our goal of maximizing the number of

[6]https://shift-ast.org/

commits with only one difference between the ASTs. This component produces a pair of ASTs: $(AST_{buggy}, AST_{fixed})$, at which point a JSON differencing algorithm, fast-json-patch [7] is applied to create a label. The label includes the operation type and node edited for each difference between AST_{buggy} and AST_{fixed}.

Each step of this process is parallelized in order to grow our corpus as quickly as possible. Our dataset has the advantage that it is continuously growing without human input.

Our system is language independent and highly extensible and modular. For example, it can handle any language so long as it can be parsed into a JSON AST.

Total Files Downloaded:	52,719,402
Total Labelled Data Points:	15,225,347
# AST differences:	# data points:
0	3,473,391
1	1,863,193
2-10	3,247,437
11-20	2,117,977
21-50	2,047,998
51-100	858,981
101+	921,754

Table A.4: Static Bug Finding: Data Collection Crawler Statistics.

For each label, we must download two files src_{buggy} and src_{fixed}. Additionally, if source files cannot be parsed into a SHIFT AST, a label cannot be created. For our learning corpus, we limit the dataset to only include labels with one AST difference. Additionally, in an attempt to limit graph size, we only include data points in which the AST_{buggy} and AST_{fixed} have less than 500 nodes.

Figure A.2 plots the distribution of number of edits that are recorded in Table A.4. We can see the distribution is long tail, with majority of edits as 1 or 2.

[7] https://www.npmjs.com/package/fast-json-patch

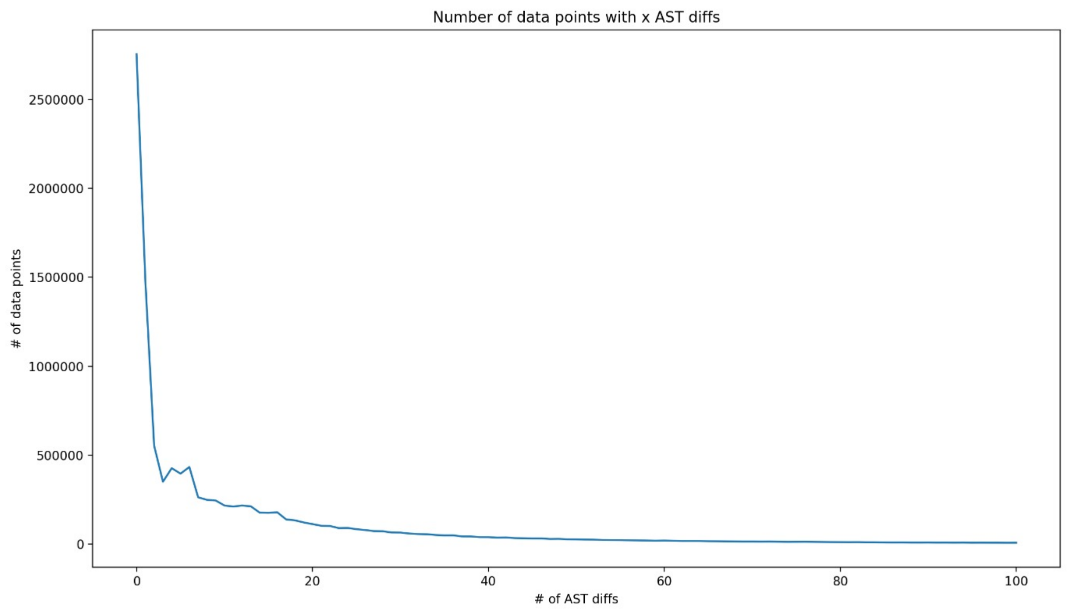


Figure A.2: Static Bug Finding: Edit Number Distribution.

A.1.3 Ablation Study

We tried different graph representations with corresponding graph embedding methods. The *multi* represents the multi-graph defined by different edge types, with the parameterization of message passing function mentioned in Eq 3.2; the *code2inv* is the parameterization used in Si et al. (2018); the *single* instead uses a single graph with edge types as one-hot edge features. We found that more layers does not lead to better generalization in our setting, and it becomes slower in terms of convergence. So we report the results with 4 layers in our main evaluation section.

model	max_lv	Total	*Operator*	*Location*	*Value*	*Type*
multi	20	7.63	30.0	13.1	22.6	54.5
multi	14	11.05	48.0	17.9	38.6	61.6
multi	**4**	**13.33**	53.4	36.2	38.6	56.4
code2inv	20	10.3	18.1	25.7	38.8	57.7
code2inv	14	8.92	40.0	18.1	36.0	55.9
code2inv	4	13.29	30.8	18.9	28.2	68.21
single	20	5.00	20.2	10.3	14.2	44.8
single	14	10.69	67.7	18.6	49.6	38.7
single	4	12.88	55.8	20.8	43.2	55.8

Table A.5: Static Bug Finding: Top-1 Ablation Study on Graph Embedding Parameterizations and Number of Layers.

A.1.4 TAJS Baseline Study

ID	Github Link to Diff	File	Buggy Code	Fixed Code
1	js-ajax-hitting-apis-lab-v-000	index.js	`login`	`name`
2	roma2hira	convert.js	`(elem).value`	`(elem).innerHTML`
3	LetsRoll	router.js	`username`	`userName`
4	MEAN	articles.server.route.js	`app.params`	`app.param`
5	js-dom-and-events-acting-on-events-lab-v-000	index.js	`(elem).InnerHTML`	`(elem).innerHTML`
6	rn-mitrais-mb	ListAlbums.js	`info.type`	`info.album_type`
7	React-QuizComponent	QuizQuestion.js	`(obj).instruction_texts`	`(obj).instruction_text`
8	musketeer-shop	order.js	`DECIMAL`	`INTEGER`
9	ALFACharts	crosshairs.js	`Math.floor`	`Math.round`
10	hackcincinnati/site	Advisors.js	`color.primary`	`color.accent`
11	react-native-with-redux-react-navigation-v2-boilerplate	splash.js	`COLOR.DARK`	`COLOR.PANTOME`
12	AloChat	Container.js	`PropTypes.string`	`PropTypes.object`
13	pandora-validation	index.js	`addAsyncSetup`	`addSyncSetup`
14	Orca	z.js	`erase()`	`explode()`
15	iotdb-mongodb	count.js	`then`	`make`
16	flom-react	question.js	`DataTypes.STRING`	`DataTypes.TEXT`
17	j5-leds	blink.js	`blink`	`strobe`
18	Craxi	display.js	`game.draw()`	`game.run()`
19	graph-js	point.js	`data`	`dataManager`
20	matic.js	getETHFromFaucet.js	`ETHFaucetAddress`	`ETHFAUCET_ADDRESS`
21	zulip	stream_muting.js	`add_messages`	`add_old_messages`
22	WTF-Adventure	mana.js	`this._super`	`this.super`
23	simple-crafting	gather.js	`getBehavior`	`getMeta`
24	geekTalks	index.js	`checkTalkOwnership`	`checkCommentOwnership`
25	pizza-totally-rocks	Form.js	`getRecipe`	`getRecipes`
26	koot	before_router_match.js	`origin`	`originTrue`
27	fo_rest	index.js	`testRecipes`	`preference`
28	cryptii	ROT13.js	`registerSetting()`	`addSetting()`
29	DiscordWithDatabase	help.js	`print()`	`printSplit()`
30	org.civicrm.civicase	CaseDetailsFileTab.js	`areAvailable()`	`isAllowed()`

ID	Bug Type	TAJS	DeepMerge			
		False Alarm	Prediction	Top 1	Top 2	Top 3
1	undefined property					
2	undefined property	Failed `doc.getElem()`				
3	undefined property	Failed importing library `util`				
4	undefined property	Undefined `process`	`app.params('...', ...)`			✓
5	undefined property	Failed `doc.getElem()`				
6	undefined property	Undefined `process`				
7	undefined property					
8	functional bug					
9	functional bug					
10	functional bug					
11	functional bug					
12	functional bug					
13	functional bug		`serviceProvider.addAsyncSetup(...)`		✓	
14	functional bug					
15	functional bug	Failed importing library `events`				
16	functional bug					
17	functional bug		`led.strobe(750)`	✓		
18	functional bug		`this.game.draw()`	✓		
19	refactoring					
20	refactoring					
21	refactoring					
22	refactoring					
23	refactoring	Failed importing library				
24	refactoring	Failed importing library				
25	refactoring					
26	refactoring					
27	refactoring		`exports.testRecipes = ...`		✓	
28	refactoring					
29	refactoring					
30	refactoring	Failed importing library				

A.2 Precondition User Study Samples

```
Precondition:

boolean M_pre(NetBigInteger value) {
  try {
    CompareTo(value);
  } catch (NullPointerException e) {
    return false;
  }
  return true;
}
```

(a) Precondition 2, Group A

```
Context:

int CompareTo(NetBigInteger value) {
  return m_sign < value.m_sign ? -1 :
      m_sign > value.m_sign ? 1 :
      m_sign == 0 ? 0 :
      m_sign * CompareNoLeadingZeroes(0, m_magnitude,
                                      0, value.m_magnitude);
}
.
```

(b) Context 2, Group A

```
((not (value == null)))
```

(c) Precondition 2, Group B.

Figure A.3: Test Generation: Precondition 2.

```
boolean M_pre(NetBigInteger m) {
  if (m == null)
    return false;
  if (m.m_sign < 1)
      return false;
  return true;
}
```

(a) Group A.

```
(not (m == null)) and (not (m.m_sign <= 0))
```

(b) Group B.

Figure A.4: Test Generation: Precondition 4.

```
Precondition:

boolean M_pre(NetBigInteger value) {
  if (value == null)
    return false;
  if (m_sign == 0)
    return true;
  NetBigInteger r;
  NetBigInteger u = this;
  NetBigInteger v = value;
  while (v.m_sign != 0) {
    try {
      r = u.Mod(v);
    } catch (ArithmeticException e) {
      return false;
    }
    v = r;
  }
  return true;
}
```

(a) Precondition 5, Group A.

```
Context:






NetBigInteger Mod(NetBigInteger m) {
  if (m.m_sign < 1)
    throw new ArithmeticException("Modulus must be positive");
  NetBigInteger biggie = Remainder(m);
  return (biggie.m_sign >= 0 ? biggie : biggie.Add(m));
}






.
```

(b) Context 5, Group A.

```
((not (value == null)) and (not (target == null)))
and (
      (-target.m_sign + -value.m_sign <= 1) and (
        (not (-target.m_sign + value.m_sign <= -2))
      )
)
```

(c) Precondition 5, Group B.

Figure A.5: Test Generation: Precondition 5.

BIBLIOGRAPHY

checkstyle cyclomaticcomplexity. https://checkstyle.sourceforge.io/checks/metrics/cyclomaticcomplexity.html.

Javaparser. https://github.com/javaparser/javaparser.

Methods2test. https://github.com/microsoft/methods2test.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *International Conference on Learning Representations*, 2018.

S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: Rethinking merge in revision control systems. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 190–200, 2011a.

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 190–200, New York, NY, USA, 2011b. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025141. URL https://doi.org/10.1145/2025113.2025141.

Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 120–129, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312042. doi: 10.1145/2351676.2351694. URL https://doi.org/10.1145/2351676.2351694.

Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. Proviso preconditions results. https://sites.google.com/site/learnprecond/. Accessed: 2023-08-23.

Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. Preinfer: Automatic inference of preconditions via symbolic analysis. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 678–689, 2018. doi: 10.1109/DSN.2018.00074.

Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 775–787, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314641. URL https://doi.org/10.1145/3314221.3314641.

Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya

Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), February 2010.

Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213872. URL https://doi.org/10.1145/3213846.3213872.

Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation. *Journal of Systems and Software*, 181:111041, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111041. URL https://www.sciencedirect.com/science/article/pii/S0164121221001382.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Method Symposium*, 2015.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. *SIGPLAN Not.*, 44(6):363–374, jun 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542517. URL https://doi.org/10.1145/1543135.1542517.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808*, 2018.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014a. URL http://arxiv.org/abs/1409.1259.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014b.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014c. URL https://arxiv.org/abs/1406.1078.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers, 2020.

Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12(null):2493–2537, nov 2011. ISSN 1532-4435.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, page 150–168, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642182747.

Patrick Cousot, Radhia Cousot, Manuel Fahndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *in Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*. Springer Verlag, January 2013. URL https://www.microsoft.com/en-us/research/publication/automatic-inference-of-necessary-preconditions/.

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy. In *2008 ACM/IEEE 30th*

International Conference on Software Engineering, pages 281–290. IEEE, 2008.

Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. *arXiv preprint arXiv:1806.02371*, 2018.

Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014. doi: 10.1109/ISSRE.2014.11.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, jul 2019a. ISSN 0001-0782. doi: 10.1145/3338112. URL https://doi.org/10.1145/3338112.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, jul 2019b. ISSN 0001-0782. doi: 10.1145/3338112. URL https://doi.org/10.1145/3338112.

Jacob Eisenstein. *Introduction to natural language processing*. The MIT Press, 2019.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007a.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007b.

J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 313–324, 2014.

Daniel Fava, Dan Shapiro, Joseph Osborn, Martin Schäef, and E. James Whitehead. Crowdsourcing program preconditions via a classification game. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1086–1096, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884865. URL https://doi.org/10.1145/2884781.2884865.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, Los Alamitos, CA, USA, 2011. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/QSIC.2011.19.

Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013. doi: 10.1109/ICST.2013.51.

Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–42, 2014.

Gleiph Ghiotto, Leonardo Murta, Márcio de Oliveira Barros, and André van der Hoek. On the nature of merge conflicts: A study of 2, 731 open source java projects hosted by github. *IEEE Trans. Software Eng.*, 46(8):892–915, 2020. doi: 10.1109/TSE.2018.2871083. URL https://doi.org/10.1109/TSE.2018.2871083.

Github. State of the Octoverse. https://octoverse.github.com/#top-languages, 2019.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005. doi: 10.1145/1065010.1065036. URL https://doi.org/10.1145/1065010.1065036.

Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 213–224, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931061. URL https://doi.org/10.1145/2931037.2931061.

Georgios Gousios, Margaret-Anne D. Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 285–296. ACM, 2016.

Alex Graves. Sequence transduction with recurrent neural networks. *CoRR*, abs/1211.3711, 2012. URL http://arxiv.org/abs/1211.3711.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Vincent J. Hellendoorn, Premkumar T. Devanbu, Oleksandr Polozov, and Mark Marron. Are my invariants valid? A learning approach. *CoRR*, abs/1903.06089, 2019. URL http://arxiv.org/abs/1903.06089.

Vincent Josua Hellendoorn, Premkumar Devanbu, Alex Polozov, and mark marron. Learning to infer run-time invariants from source code. In *NeurIPS 2020 Workshop on Computer-Assisted Programming*, 2020. URL https://openreview.net/forum?id=AiJ2UxYE4cQ.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press, 2012. ISBN 9781467310673.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020.

Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

Simon Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. volume 5673, pages 238–255, 01 2009a. ISBN 978-3-642-03236-3. doi: 10.1007/978-3-642-03237-0_17.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009b.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009c.

Wengong Jin, Kevin Yang, Regina Barzilay, and Tommi Jaakkola. Learning multimodal graph-to-graph translation for molecular optimization. *arXiv preprint arXiv:1812.01070*, 2018.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

Daniel D Johnson. Learning graphical state transitions. 2016.

Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

Daniel Jurafsky and James H. Martin. *Speech and language processing.* Pearson, 2014.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International*

Symposium on Software Testing and Analysis, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1073 — 1085, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380342. URL https://doi.org/10.1145/3377811.3380342.

Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 121–132, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Chris Lattner. Clang static analyzer - cross translation unit (ctu) analysis. https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 2019.

Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010. URL https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/.

O. Leßenich, S. Apel, and C. Lengauer. Balancing precision and performance in structured merge. *Automated Software Engineering*, 22(3):367–397, 2015.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, jan 2015. ISSN 0001-0782.

doi: 10.1145/2644805. URL https://doi.org/10.1145/2644805.

Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8)*, volume 12420 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2020. doi: 10.1007/978-3-030-59762-7_2.

D. Marr and Ellen C. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207:187 – 217, 1980.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 771–774. IEEE Computer Society, 2007. doi: 10.1109/ICSE.2007.48. URL https://doi.org/10.1109/ICSE.2007.48.

Facundo Molina. Applying learning techniques to oracle synthesis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1153–1157. IEEE, 2020.

Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. Evospex: An evolutionary algorithm for learning postconditions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1223–1235. IEEE, 2021.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. *CoRR*, abs/2002.09030, 2020. URL https://arxiv.org/abs/2002.09030.

OpenAI. Chatgpt. https://chat.openai.com/chat.

OpenAI. Gpt-4 technical report, 2023.

Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 2007.

Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *SIGPLAN Not.*, 51(6):42–56, jun 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908099. URL https://doi.org/10.1145/2980983.2908099.

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 815–825, 2012. doi: 10.1109/ICSE.2012.6227137.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=WQc075jmBmf.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

John Regehr. C-reduce. https://github.com/csmith-project/creduce.

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312059. doi: 10.1145/2254064.2254104. URL https://doi.org/10.1145/2254064.2254104.

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, mar 2018. ISSN 0001-0782. doi: 10.1145/3188720. URL https://doi.org/10.1145/3188720.

Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 295–306. ACM, 2008. doi: 10.1145/1390630.1390666. URL https://doi.org/10.1145/1390630.1390666.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Andrew Scott, Johannes Bader, and Satish Chandra. Getafix: Learning to fix bugs automatically. volume 2, 2019.

Mohamed Nassim Seghir and Daniel Kröning. Counterexample-Guided Precondition Inference. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 451–471. Springer International Publishing, 2013. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_25.

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005. doi: 10.1145/1081706.1081750. URL https://doi.org/10.1145/1081706.1081750.

Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211. IEEE, 2015.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pages 7751–7762, 2018.

R. Smith. Gnu diff3. distributed with GNU diffutils package, April 1998.

Randy Smith. diff3. 1988.

M. Sousa, I. Dillig, and S. K. Lahiri. Verified three-way program merge. *Proc. ACM Program. Lang.*, 2:165:1–165:29, 2018.

Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, page 361–371. Association for Computing Machinery, 2018. doi: 10.1145/3180155.3180236.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112. Curran Associates, Inc., 2014a. URL https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks.

In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014b. MIT Press.

Charles Sutton, David Bieber, Kensen Shi, Kexin Pei, and Pengcheng Yin. Can large language models reason about program invariants? 2023.

S. M. J Sutton. Preconditions, postconditions, and provisional execution in software processes. Technical report, USA, 1995.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417058. URL https://doi.org/10.1145/3368089.3417058.

Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. Mergebert: Program merge conflict resolution via neural transformers. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2021.

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 260–269, Montreal, Canada, April 2012.

Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. 05 2002.

Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. *Semistructured Merge in JavaScript Systems*, page 1014–1025. IEEE Press, 2019. ISBN 9781728125084. URL https://doi.org/10.1109/ASE.2019.00098.

Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. ESEC/FSE 2020, page 1178–1189, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409758. URL https://doi.org/10.1145/3368089.3409758.

Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008. URL https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers, 2020.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *International Conference on Learning Representations*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 2692–2700. Curran Associates, Inc., 2015a. URL https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015b.

Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. Learning a static bug finder from data. *arXiv preprint arXiv:1907.05579*, 2019.

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun 2020. doi: 10.1145/3377811.3380429. URL http://dx.doi.org/10.1145/3377811.3380429.

Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, jan 1966. ISSN 0001-0782. doi: 10.1145/365153.365168. URL https://doi.org/10.1145/365153.365168.

Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM '91, page 68–79, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914295. doi: 10.1145/111062.111071. URL https://doi.org/10.1145/111062.111071.

Robert White and Jens Krinke. Reassert: Deep learning for assert generation. *arXiv preprint arXiv:2011.09784*, 2020.

Yichen Xie and Alex Aiken. Saturn: A sat-based tool for bug detection. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, page 139–143, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540272313.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, 1992.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJl6AjC5F7.

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun.

Nicholas C. Zakas. ESLint. https://eslint.org/, 2013.

Michal Zalewski. American fuzzy lop (afl), 2015. URL http://lcamtuf.coredump.cx/afl/.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, feb 2002. ISSN 0098-5589. doi: 10.1109/32.988498. URL https://doi.org/10.1109/32.988498.

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: Translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 25–37, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409716. URL https://doi.org/10.1145/3368089.3409716.